Automatic Retargeting of Binary Utilities for Embedded Code Generation

Alexandro Baldassin, Paulo Centoducatte, Sandro Rigo
State University of Campinas – P.O. Box 6176 - 13084-971, Campinas, Brazil
{alebal, ducatte, sandro}@ic.unicamp.br

Daniel Casarotto, Luiz C. V. Santos, Max Schultz, Olinto Furtado
Federal University of Santa Catarina – P.O. Box 476 - 88010-970, Florianopolis, Brazil
{casaroto, santos, max, olinto}@inf.ufsc.br

Abstract

Contemporary SoC design involves the proper selection of cores from a reference platform. Such selection implies the design exploration of alternative CPUs, which requires the generation of binary code for each possible target. However, the embedded computing market shows a broad spectrum of instruction-set architectures, ranging from micro-controllers to RISCs and ASIPs. As a consequence, binary utilities cannot always rely on pre-existent tools within standard packages. Besides, the task of manually retargeting every binary utility is not acceptable under time-to-market pressure. This paper describes a technique for the automatic generation of binary utilities from an abstract model of the target CPU, which can be synthesized from an arbitrary ADL. The technique is based upon two key mechanisms: model provision for tool generation (at the front-end) and automatic library modification (at the back-end). To illustrate the technique's automation effectiveness, we describe the generation of assemblers, linkers and disassemblers. We have successfully compared the files produced by the generated tools to those produced by conventional tools. Moreover, to give proper evidence of retargetability, we present results for MIPS, SPARC, PowerPC and i8051.

1 Introduction

The huge supply of hardware made available by state-of-the-art VLSI technologies, combined with the increasing demand on embedded system applications, gave rise to Systems-on-Chip (SoCs) [3]. SoCs can be built with general purpose processors or Application-Specific Instruction-set Processors (ASIPs). Platform-based SoC design [13] is the methodological response to the high non-recurring engineering costs of deep submicron VLSI technologies. Given a chosen platform, design exploration is crucial to fulfill not only functional requirements, but also real-time, low-power and code-size constraints.

Design exploration may require code generation for several alternative CPUs, possibly including ASIPs. In this context, the availability of binary utilities for the most popular general purpose processors does not help much. Besides, the task of developing a new binary utility for each new explored CPU would not be affordable under the time-to-market pressure. A widely explored approach consists in modelling the processor in an Architecture Description Language (ADL) which can drive the design space exploration and automatically generate software tools. Figure 1 shows the process of exploring alternative solutions with the help of an ADL. Given an ADL description of the CPU, the generated tool chain allows the execution of the application code (on a CPU simulation model) and its iterative evaluation until the requirements are met.

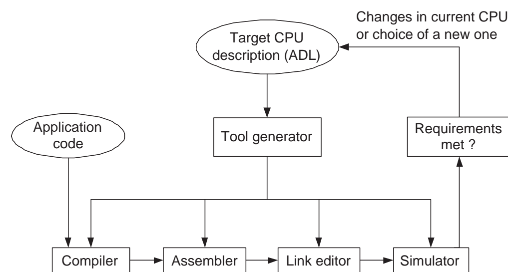

Figure 1. ADL-based CPU exploration flow

This paper addresses the automatic generation of binary utilities (assemblers, linkers and disassemblers) from an abstract model of the target CPU, which can be generated from its description written in an arbitrary ADL. A new relocation handling mechanism is proposed along with a memory layout model. The remainder of this paper is organized as follows. Section 2 addresses related work. Section 3 defines the proposed abstract model. Section 4 describes our technique for automatically retargeting binary utilities. Section 5 summarizes experimental results. Our main conclusions are drawn in Section 6 along with our perspectives of future work.

2 Related Work

While simulator and compiler generation have been extensively discussed in the context of ADL toolkit generation [8], the same attention has not been given to the generation of binary utilities. ISDL [5] and nML [6] are among the first ADLs to mention assembler and disassembler generation, but neither a description of the underlying technique nor experimental results are shown. Sim-nML [9] assembler generator uses the GNU `lex` and `yacc` tools to produce the assembly language's lexer and parser. Relocation handling is driven by an external configuration file describing the inner working of the resolution mechanism. The LISA tool-suite [7] provides also a link editor. The linking process is guided by a command file which specifies the target memory model.

The SLED language [11] defines constructs to describe instruction set architectures. From a SLED description, the New Jersey Machine-Code Toolkit [10] generates encoding and decoding routines which can be used by binary utility programs. However, no front-end support (such as the parsing of source files) is provided.

Abbaspour et al. [1] present systematic techniques to retarget the GNU Binary Utilities. A formal notation is used to capture the Instruction Set Architecture (ISA) and relocation information. A brief description of the retargeting algorithms is given. Experimental results are reported for the SPARC architecture. Although an Intel 386 specification is mentioned, no supporting results are shown. It is not possible to foresee if the proposed framework is generic enough to achieve the same results for a CPU which has not yet been ported to the GNU Binutils package.

Like in [1], our work is built upon an ADL-independent notation and relies on GNU Binutils as implementation infrastructure. However, our approach differs from theirs in the following aspects: our Application Binary Interface (ABI) model allows specifying the memory layout (see Section 3.2), an important feature for embedded system applications; our automatic generation of relocation actions keeps the user from getting involved in low-level details; our relocation handling mechanism is more general (it handles ISA allowing multiple choices for operand encoding, as will be seen in Section 4.1). Besides, as opposed to [1], our technique was able to retarget an ISA with no previous port in the GNU Binutils package: the i8051.

3 Abstract model

Our retargeting algorithms rely on an abstract model that captures both the target ISA and the ABI. The model deliberately abstracts from ADL syntax details, thereby releasing our tools from being tied to a specific ADL. To make the model synthesizable from an arbitrary ADL, it is formally defined in the well-known BNF notation. Besides, to ease its interpretation, it is also illustrated by means of examples. For the sake of self-containment, the model description is limited to the aspects required by the generation of binary utilities. For instance, instruction behavior and timing are not addressed in this paper. Moreover, the ABI description focuses on relocation information and memory layout.

3.1 ISA description

In our model, the ISA description captures information commonly available in most CPU manuals. The model is based upon notions like instruction, operand and modifier. The following subset of syntax rules (in BNF format) specifies these notions:

```
⟨operand-def⟩       ::= operand oper-id { ‘‘mapping definition’’ }
⟨modifier-def⟩      ::= modifier modifier-id { ‘‘modifier code’’ }
⟨instruction-def⟩   ::= instruction insn-id { ⟨format-desc⟩ ;
                        ( ⟨syntax-desc⟩ ) : ( ⟨operand-encoding⟩ ) ;
                        ⟨opcode-encoding⟩ }
⟨format-desc⟩       ::= field-id : constant , ⟨format-desc⟩
                        | field-id : constant
⟨syntax-desc⟩       ::= mnemonic-id ⟨oper-type-list⟩
⟨oper-type-list⟩    ::= ⟨qualifier⟩ ⟨oper-type⟩ , ⟨oper-type-list⟩
                        | ⟨qualifier⟩ ⟨oper-type⟩
⟨oper-type⟩         ::= oper-id | imm | addr ⟨modifier⟩ | exp ⟨modifier⟩
⟨modifier⟩          ::= << modifier-id ( constant ) | empty
⟨operand-encoding⟩  ::= field-id , ⟨operand-encoding⟩ | field-id
⟨opcode-encoding⟩   ::= field-id = constant , ⟨opcode-encoding⟩
                        | field-id = constant
⟨qualifier⟩         ::= # | $ | empty
```

A *modifier* is a function that transforms a given operand value. Within the modifier code (written in C language), four variables are pre-defined to specify the transformation: `input` is the original operand, `address` stores the instruction location (computed at assembly or linking time), `parm` is an optional parameter, and `output` returns the transformed operand.

An *operand type* specifies the nature of a given instruction field. At assembly or linking time, an operand type is tied to the binary value stored in a field and may undergo the action of a modifier, yielding a substitute binary value to be edited in the field.

Modifier and operand type are key notions to the process of replacing symbolic references with actual addresses. This process, known as *relocation*, is usually carried out by linkers. In our model, the necessary relocation information for automatic generation of binary utilities is captured by modifiers and operand types.

Since some operand types are common to a large number of architectures, they are provided in our model as *built-in types*: `imm` for immediate values, `addr` for symbolic addresses, and `exp` for expressions involving immediates and symbols.

Let's illustrate the model with the example in Figure 2, written according to the specified syntax. Line 1 describes the mapping for operand type `reg`, where the symbols `r0`, `r1`, ..., `r7` are mapped to the values `0`, `1`, ..., `7`. Line 3 defines the modifier `R`, which computes PC-relative transformations. The modifier's result (`output`) is computed by subtracting the current location (`address`) from the operand value (`input`) and by adding an offset (`parm`). Lines 5 to 9

```
1. operand reg { r[0..7] = [0..7]; }
2.
3. modifier R { output = input - address + parm; }
4.
5. instruction cjne {
6.   op:5, rd:3, dat:8, targ:8;
7.   (cjne reg, #imm, addr << R(3)):(rd, dat, targ);
8.   op=0x17
9. }
```

Figure 2. Fragment of an i8051 model

define the instruction `cjne`. Line 6 defines the instruction format as a list of fields and associated bit sizes. Line 7 defines the assembly syntax and operand encoding (where `cjne` is the instruction mnemonic). In line 7, operands `reg`, `imm` and `addr` are bound to the instruction fields `rd`, `dat` and `targ`, respectively. (The character `#` is a qualifier required to precede an immediate field in the i8051 assembly). Note that the modifier `R` (with an offset of 3) is applied to operand type `addr`, since the value encoded must be relative to `PC+3`. Finally, in line 8, the constant value `0x17` is assigned to field `op` of instruction `cjne`.

3.2 Memory layout

Programmers usually split software into modules which can be compiled individually. The resulting binary files (generated by the assembler) are called *relocatable object files*. In addition to instructions and data, they also supply relocation information (to be used by the linker). An *executable object file* is created by linking relocatable modules. During the process, the linker must determine how code and data sections must be grouped and which are their final addresses. This results in a memory mapping, which is captured in our model by the notion of *segment*, as specified by the following subset of syntax rules (in BNF format):

⟨*segment-def*⟩	::=	`segment` segment-id `{` `(` ⟨ *section-list*⟩ `)` `;` ⟨*mapping-properties*⟩ `:` ⟨*attributes*⟩ `}`
⟨*section-list*⟩	::=	section-id + ⟨*section-list*⟩ | section-id
⟨*mapping-properties*⟩	::=	load-addr-const `:` start-addr-const `:` align-const
⟨*attributes*⟩	::=	`(` constant `,` constant `,` constant `)` `:` length-const ⟨*unit*⟩
⟨*unit*⟩	::=	`b` | `Kb` | `Mb` | `Gb` | empty

Essentially, a segment consists of a list of sections, mapping properties (load address, start address and alignment) and attributes describing the nature of the segment (whether it is allocatable, read-only or executable) and its maximum length.

Figure 3 shows an illustrative example, where `rom` and `ram` segments are described within our i8051 model. Note that the segment `rom` consists of sections `CSEG`, `GSINIT0`, `GSINIT1` and `GSFINAL` (line 2), while the segment `ram` consists of sections `DATA` and `STACK` (line 7). In the `rom` segment, line 3 specifies the location where it is loaded (`0x0000`), states the value effectively used in the relocation process (in this case also `0x0000`) and defines byte alignment (value `1`). Then, it specifies the segment `rom` as not allocatable (value `0`), read-only (value `1`) and executable (value `1`). In the end, line 3 defines the maximum segment length (`64Kb`).

```
1. segment rom {
2.   (CSEG + GSINIT0 + GSINIT1 + GSFINAL);
3.   0x0000 : 0x0000 : 1 : (0, 1, 1) : 64Kb
4. }
5.
6. segment ram {
7.   (DATA + STACK);
8.   0x0008 : 0x0008 : 1 : (1, 0, 0) : 128b
9. }
```

Figure 3. Fragment of a i8051 memory layout

4 Retargeting of binary utilities

This section proposes a binary utility retargeting technique based upon the abstract model described in Section 3. We first describe our novel relocation handling mechanism and then we show how binary utilities are generated from our abstract model. The implementation infrastructure relies on the popular GNU Binutils package whose machine-dependent modules are automatically modified by our technique, while machine-independent ones are reused.

4.1 A new relocation handling mechanism

Given an instruction, its relocation information (if any) is stored within a tuple called *relocation action*, as follows:

```
reloc-action = (uid, format-id, field-id, modifier)
```

Algorithm 1 shows how a `reloc-action` is automatically generated from our ISA model for each instruction requiring relocation. Since Algorithm 1 makes our abstract model simpler, it keeps the user from handling low-level relocation details, as opposed to the approach in [1].

Algorithm 1 Automatic generation of relocation actions

1: *uid* $\leftarrow 1$
2: *fid* $\leftarrow 1$
3: **for each** *instruction* **do**
4:     **for each** *oper-type-id* **do**
5:         **if** *oper-type-id* **is** 'addr' or 'exp' **then**
6:             let *ra* = (*uid*, *fid*, *field-id*, *modifier-id*) be a new *reloc-id*
7:             *uid* $\leftarrow$ *uid*$+1$
8:         **end if**
9:     **end for**
10:     *fid* $\leftarrow$ *fid*$+1$
11: **end for**

Algorithm 2 defines a routine `apply-reloc` that performs relocation at linking time. The routine `encode-insn`, which patches the instruction under relocation, is described in Section 4.3.2.

Algorithm 2 Relocation handling mechanism

Routine: `apply-reloc`(*rid*, *value*, *addend*, *addr*)

1: let *ra* = (*uid*, *format-id*, *field-id*, *modifier-id*) be a *reloc-action* such that *uid* = *rid*
2: let *insn* be the instruction at address *addr*
3: let *f* be the function tied to *modifier-id*
4: *calc* $\leftarrow$ *f*(*value*, *addend*, *addr*)
5: *new-insn* $\leftarrow$ `encode-insn`(*format-id*, *field-id*, *insn*, *calc*)
6: store *new-insn* at address *addr*

As it will be discussed in Section 4.4, conventional approaches cannot easily handle ISAs admitting multiple choice for operand encoding (such as ARM's data-processing immediate operands). Since our mechanism employs the notion of modifier (whose code can be written as an arbitrary C function), virtually any calculation could be performed during relocation. Therefore, the proposed mechanism can handle those ISAs.

4.2 GNU Binutils overview

The GNU Binutils package [12] is a collection of tools aimed at binary file manipulation. Its main tools are an assembler (`gas`) and a link editor (`ld`). In addition, the package contains library managers (`ar` and `ranlib`), object file inspectors (`objdump` and `readelf`) and some other minor tools.

In general, a GNU Binutils tool has a machine-independent module (the core), which provides the main tool operation and control flow, and a machine-dependent module, which implements CPU-specific operations. Most of the retargeting effort is spent on the package's main libraries, namely *Opcodes* and *Binary File Descriptor* (BFD). In order to retarget a tool to a new CPU, the machine-dependent code must be implemented and the package's main libraries extended.

The Opcodes library encapsulates the target ISA. Although an API is defined for decoding purposes, there is no standard on how encoding routines and data structures are represented. Therefore, it is up to the developer to choose a convenient format for each new CPU to be ported.

The BFD library provides a set of generic routines to operate on object files regardless of the adopted binary format. Application programs interface with the library's front-end, which provides a set of format-independent routines to manipulate object files. The front-end is responsible for calling the proper back-end routine which in turn executes format-dependent operations on the object file. The main advantage of the framework provided by the BFD library is that, once an object file format is implemented in the backend, it can be reused by all CPU ports. Therefore, although CPU-specific information (such as relocation) is still required, the overall porting effort is substantially reduced.

4.3 Automatic Opcodes retargeting

Since our approach deliberately hides detailed information from the user, the information originally captured by our abstract model must be kept in such a way that it can fit the adopted implementation infrastructure. Since the Opcodes library does not specify any standard way to keep the information properly, we have created a generic framework consisting of a so-called *instruction table* and an *encoding routine*. This framework, which is automatically generated, bridges the gap between the abstract model and the binary utility tools.

4.3.1 Instruction table

The instruction table is a compact low-level description of the instructions declared in the abstract model, which is specially tailored for binary utility use. Each table entry is a tuple defined as:

```
table-entry = (mnemonic, opinfo, image, mask, format-id)
```

Let's illustrate its interpretation by means of an example. The fragment model in Figure 2 results in the following table entry:

```
{"cjne", "%reg:1:0:,#%imm:2:0:,%addrR3:3:5:",
 0xB80000, 0xF80000, 1}
```

The first element is the instruction mnemonic (`cjne`). The second element stores operand information such as types, instruction fields and relocation identifiers. For instance, the first operand, whose type is `reg`, is bound to field identifier `1` and has relocation identifier `0` (meaning none). The third element stores the instruction partial binary image (`0xB80000`). The fourth element stores a mask (`0xF80000`) used by the disassembly algorithm for instruction identification. The last element stores the format identifier (`1`, in this case).

4.3.2 Encoding and Decoding routines

Recall that Algorithm 2 invokes the patching routine `encode-insn(format-id, field-id, img, value)` to relocate an instruction. Given the format type (`format-id`) and its binary image (`img`), it edits a given instruction field (`field-id`) by replacing its contents with `value` and returning the resulting instruction.

Encoding routines are automatically generated from the instruction formats specified in the abstract model. The encoding routine consists of two nested switch-case constructs. The outer construct selects the format, while the inner construct selects the field to encode.

A decoding routine is also generated as part of the Opcodes API. Upon receiving a base address, the decoding routine first figures out the correct instruction size and decodes its binary image based on the `mask` and `image` fields of a `table-entry`. Once the instruction has been identified, its string representation is built up through the `mnemonic` and `opinfo` fields.

4.4 Automatic BFD retargeting

One of the most difficult aspects in porting the BFD library to a new CPU is certainly its relocation mechanism. Quoting the BFD internal documentation [15]: *"Clearly the current BFD relocation support is in bad shape"*.

BFD's relocation handling mechanism is based on a relocation table structure, named `howto`, which stores relocation attributes such as the instruction size, the shift values to guide the relocation process and whether the instruction is PC-relative or not. A generic function in the library uses the table properties to actually apply the relocation at linking time. The `howto` structure imposes severe restrictions on some of its fields. For instance, 24-bit instructions cannot be specified (as required by the i8051).

Besides, its table-driven nature overly constraints the calculation expressiveness required to perform relocation for ISAs with complex encoding schemes, where operands may admit multiple encoding alternatives (such as the so-called data-processing immediate operands in the ARM CPU). To work around such limitations, BFD requires the writing of machine-dependent code, an expedient that hampers automatic retargeting.

The model described in [1] uses the same table-oriented approach. It overcomes some of the Binutils difficulties by providing an additional field to describe the relocation calculation expression. Although this improves the relocation handling, it is still unable to generically address the abovementioned complex encoding schemes. Besides, it is up to the user to specify the relocation parameters, as opposed to our approach, where the automatic generation of relocation actions (Algorithm 1) raises the abstraction level seen by the user.

To automatically retarget the BFD library, instead of reusing the `howto` structure, our approach relies on the *relocation actions* described in Section 4.1. The virtually unconstrained expressiveness captured by its modifiers grants our approach with a generic relocation mechanism. It is able, for instance, to handle multiple operand encoding alternatives. At linking time, Algorithm 2 successfully uses a relocation action to patch binary code via modifiers and an encoding routine.

4.5 Automatic tools retargeting

To retarget the assembler, a series of machine-dependent routines must be implemented. These routines deal mainly with instruction parsing, instruction encoding and, if necessary, relocation generation. We have developed a single parsing routine that handles all target CPUs. The parsing of an assembly source file is guided by the information stored in the Opcodes library. Once an instruction is syntactically validated, binary code is emitted and a relocation entry is created for each relocatable operand (according to its type) using the BFD library.

Most of the linker code is machine-independent since it relies on the BFD library and the so-called linker command language [12]. For each relocation found in the relocatable object code, the linker calls the `apply-reloc` routine, specified in Algorithm 2. The layout of the resulting executable object file is controlled by our memory model, defined in Section 3.2.

Retargeting the disassembler (`objdump` tool) is just a matter of retargeting the decoding API of the Opcodes library. For each memory address whose instruction must be disassembled, the decoding routine (briefly described in Section 4.3.2) returns the corresponding instruction string.

5 Experimental Results

To verify our retargeting techniques, we first adopted the ArchC ADL [2]. Then, we wrote ArchC models for targets MIPS, SPARC, PowerPC and i8051. Finally, each ADL model was automatically translated into an abstract model, according to the syntax rules described in Section 3.

The GNU Binutils package supplies assemblers, linkers and disassemblers for several target CPUs. Let's call them *conventional tools*. We employ conventional tools to produce executable code, which has served as a reference during validation (since there is no i8051 reference tool in GNU Binutils, we have used SDCC [14] instead). From each generated abstract model, our retargeting tool, henceforth called `bingen`, automatically retargets the conventional assembler, linker and disassembler (`objdump`) for the mentioned target architectures. The files produced by `bingen` are then inserted into the Binutils source tree, where the binary tools are built up like any other tools within that package. The automatically retargeted tools are henceforth called *generated tools*.

In order to validate the generated tools, two different procedures were employed. On the one hand, to validate the tools producing object code (assembler and linker), we performed the following procedure: given a target CPU and a benchmark program in assembly form, conventional tools were used to produce the reference code. Then, the code produced by the generated tools was compared to the reference code, allowing us to check its correctness. On the other hand, to validate the disassembler, we employed the following procedure: given a benchmark program, its assembly code was used as input to the assembler and linker resulting in executable code. Then, the executable code was fed to the disassembler. Finally, the resulting disassembler output was compared to the assembler input to check for matching.

In order to validate the `bingen` tool itself, we had to check for proper automatic retargetability. Therefore, the procedures described in last paragraph were repeated for MIPS, SPARC, PowerPC and i8051 targets. For the experiments, we adopted two well-known benchmark suites, MiBench [4] and Dalton [16].

5.1 Validation flow

Figure 4 shows the validation flow for the generated assembler and linker. Given a program consisting of n `C` source files, the cross compiler GNU `gcc` produces n assembly source files. These files pass through a filter which normalizes the section names of the assembly sources, giving rise to a set of n filtered assembly source files. Then the flow forks: one path uses the conventional tools; the other path uses the generated tools. In each path, the assembled object files are then linked, resulting in two executable files, one serving as reference, and another being the file under validation. In the end, they are compared to check if they match.

5.2 Result analysis

The validation flow shown in Figure 4 was repeated for each benchmark program and for each target CPU, giving rise to the results summarized in Table 1. For compactness, results are computed by adding up the figures result-

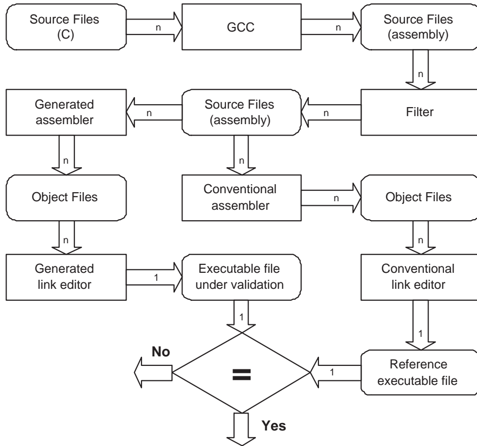

Figure 4. Tool generator validation flow

ing from each individual program. The first row identifies the adopted benchmarks, while the second row shows how many programs were used. The third row shows the overall number of source files handled by our tool. Rows four and five display, respectively, the number of relocations generated and the number of relocations actually performed. The last two rows show the added-up sizes (expressed in bytes) for the code and data sections of the generated binary files. We have grouped the instruction sections under the `.text` segment and the data sections under the `.data` one. Results do not include run-time library code (except for the i8051).

Table 1. A summary of experimental results

	MIPS	SPARC	PowerPC	i8051
benchmark	MiBench	MiBench	MiBench	Dalton
programs	19	19	19	9
files	109	109	109	18
relocs (gen)	7	8	12	12
relocs (perf)	17895	15951	14926	480
`.text` size	451600	398988	369000	3327
`.data` size	467840	470368	466868	107

We also observed that, for both conventional and generated tools, the resulting number of relocation types is the same. This shows the effectiveness of our relocation handling mechanism.

The fact of getting correct results for four distinct target CPUs and for such a diversity of real programs (with a huge variation on the number of relocations, number of files and code size) is a strong evidence of the robustness of the generated binary utilities.

6 Conclusions and Future Work

This paper has shown a general and pragmatic approach for binary utility generation. Generality results from an abstract description of the target CPU and from the tool structure breakdown in machine-dependent (generated) and machine-independent (invariant) modules. Implementation pragmatism results from the adoption of a well-accepted binary utility package.

Experimental results have provided strong evidences of correction (by comparing results produced by both generated and conventional tools), robustness (by observing results for a varied set of benchmark programs) and retargetability (by testing for distinct RISC and CISC targets).

On the one hand, our new relocation handling mechanism grants our approach with the generality required to face the broad spectrum of ISAs in the embedded system arena. On the other hand, its underlying abstract model, was designed to keep our tools from being tied to a specific ADL.

As future work, we plan to extend the range of generated binary utilities. For instance, an implementation is in progress for the generation of debuggers. To further probe our abstract model, we intend to make use of other ADLs.

References

[1] M. Abbaspour and J. Zhu. Retargetable binary utilities. In *Proc. of 39th DAC*, pages 331–336, June 2002.

[2] R. Azevedo and S. Rigo and M. Bartholomeu and G. Araújo and C. Araújo and E. Barros The ArchC Architecture Description Language. In *IJPP*, 53(5):453–484, October 2005.

[3] R. A. Bergamaschi and J. Cohn. The A to Z of SoCs. In *Proc. of ICCAD*, pages 790–798, November 2002.

[4] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC-4*, pages 3–14, December 2001.

[5] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In *Proc. of 34th DAC*, pages 299–302, June 1997.

[6] M. R. Hartoog, J. A. Rowson, P. D. Reddy, S. Desai, D. D. Dunlop, E. A. Harcourt, and N. Khullar. Generation of software tools from processor descriptions for hardware/software codesign. In *Proc. of 34th DAC*, pages 303–306, June 1997.

[7] A. Hoffmann, A. Nohl, G. Braun, and H. Meyr. A survey on modeling issues using the machine description language LISA. In *Proc. ICASSP'01*, volume 2, pages 1137–1140, May 2001.

[8] P. Mishra and N. Dutt. Architecture description languages for programmable embedded systems. *IEE Proceedings – Computers and Digital Techniques*, 152(3):285–297, May 2005.

[9] R. Moona. Processor models for retargetable tools. In *Proc. of 11th IWRSP*, pages 34–39, June 2000.

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

[11] N. Ramsey and M. F. Fernandez. Specifying representations of machine instructions. *ACM Trans. Program. Lang. Syst.*, 19(3):492–524, May 1997.

[12] R. H. Pesch and J. M. Osier. *The GNU Binary Utilities*. Free Software Foundation Inc, May 1993.

[13] A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *IEEE Design & Test of Computers*, 18(6):23–33, November–December 2001.

[14] Small Device C Compiler. `http://sdcc.sourceforge.net` (August 2006).

[15] I. L. Taylor. *BFD Internals*. Free Software Foundation

[16] The UCR Dalton project. `http://www.cs.ucr.edu/~dalton` (August 2006).