Retargeting GCC: Do We Reinvent the Wheel Every Time?

Saravana Perumal P
Department of CSE, IIT Kanpur
saravanani1986@gmail.com

Amey Karkare
Department of CSE, IIT Kanpur
karkare@cse.iitk.ac.in

Abstract
Porting GCC to new architecture requires writing a Machine Description (MD) file that contains mapping from GCC’s intermediate form to the target assembly code. Constructing an MD file is a difficult task because it requires the user to understand both (a) the internals of GCC, and (b) the intricacies of the target architecture. Instruction sets of different architectures exhibit significant amount of semantic similarities across a large class (for example, the instruction sets for RISC architectures) and differ only in syntax. Therefore, it is expected that MD files of machines with similar architectures should also have similarities. To confirm our hypothesis, we created mdcompare, a tool to (a) extract RTL patterns (machine independent abstraction of RTL templates) from MD files of well known architectures and (b) compare the similarity of patterns across architectures. The results are encouraging; we found that 28% – 70% RTL expressions are similar across pairs of MD files, the similarity percentage being on the higher side for pairs of similar architectures.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—Code generation, Compilers, Retargetable compilers

General Terms GCC Machine Description

Keywords Compiler, Retargetable Compilers, GCC, Machine Descriptions, Code Generation

1. Introduction
The GNU Compiler Collection (GCC) is an integrated distribution of compilers for several programming languages. GCC is the most widely used compiler collection for developing applications that run across several different architectures and operating systems. One of the strengths of GCC is that it is highly portable, owing to the fact that the core compiler part of GCC does not have any machine-specific code, but has parameters which depend on the target machine’s features. The information about the target machine is obtained from Machine Description (MD) files. An MD file is a text file containing mappings from GCC’s intermediate representation to the instruction set of a target architecture. This is an elegant way of alienating the machine-specific details from the core compiler.

Figure 1 shows a high level view of translation of source code to target machine code by GCC. The front end parses the source language, converts it to an intermediate representation called GIMPLE. After a few internal transformations, GIMPLE representation is converted to another intermediate representation called RTL (Register Transfer Language). The RTL representation is converted to the target assembly code. The conversion from GIMPLE to RTL and from RTL to assembly is guided by the templates present in MD file. These are called RTL expressions and encode all machine-specific information. The MD file is used by the GCC framework during the building of the GCC compiler.

Constructing the MD file for a target architecture forms the most important step in porting GCC. Writing an MD file for a new architecture needs a good understanding of GCC’s intermediate representations, the RTL expressions, and the instruction set of the target architecture. Given the complexity of the modern architectures and the variety of architectures available, this is a huge ask. So, in practice, an MD file for a new architecture is constructed from an MD file of a similar architecture by making modifications to suit the needs. This is a method of trial-and-error, and construction of MD file in this way is observed to be complex, verbose and repetitive. Any mistake in MD file may result in the compiler producing wrong or worse, inefficient code, without the user detecting it quickly. This is because the MD files themselves are huge (running into few thousand lines for typical architectures). For example, MD file for ARM has 30,943 lines, for i386 has 38,817 lines, and MIPS has 15,534 lines for GCC version 4.6.1.

Retargeting GCC could be simplified significantly if the process of writing MD files is fully automated. However, due
The RTL expressions in this paper are modified for ease of explanation and to avoid referring to complex concepts that are out of scope of this work.


define expand "addi3" operand:SI 0 operand:GPR 0 "register operand:GPR 2 "arith
   (match
   plus:
   register Value
   int
   GPR operand:SI 0 "s operand:GPR 0 operand:SI 2 $arg1 operand:GPR 1 "register operand:GPR 1
   (match
   SI Tree Representation
   (match

   (define
   (match
   (set
   (define
   (match
\[(set ...(plus ...))\]
to avoid referring to complex concepts that are out of scope of this work.

In this paper, we describe our experiments to verify this hypothesis and report our findings. We first present an example to motivate the problem and to explain the notion of similarity used by us.

1.1 A Motivating Example

Consider the problem of adding two numbers on a given architecture. The assembly instruction to be used depends on the type of numbers to add (SI integer, DI integers, floating points, sign extension required etc.), the type of storage (register, memory, immediate, etc.) etc. We give an example of the similarity that exists across MIPS and ARM architectures for addition instruction\[1\].

\textbf{Example 1.} Figure 2(a) shows RTL expressions to add two numbers on MIPS and ARM architectures respectively. Without going into the details of the semantics, we notice that the form of the RTL expression ([(set ...(plus ...))]) to select the appropriate instruction is identical for these architectures. We call this common form, that is obtained by abstracting out the machine-specific parts of an RTL expression, an RTL Pattern.

Figure 2(b) shows the corresponding RTL template, the parameters to the templates, and the instantiation of parameters to recover the original expression. It also shows the tree representation of the RTL pattern to help visualize it.

The pattern in this case is extracted automatically by our tool by looking at RTL expressions present in the MD files. It is a part of the set of arithmetic patterns required for similar architectures. While retargeting GCC for a similar architecture, this is one of the arithmetic patterns that needs to be filled with machine-specific values.

We generalize this observation as follows: if we can find a set of minimal RTL patterns, that are common across machines having similar architecture, the process of retargeting can be made simpler and systematic where parts of MD files for new architectures can be generated automatically. Our hypothesis simply says that the RTL expressions are alike for two instructions that behave alike.

To verify our hypothesis, we experimented with five well known architectures (ARM, i386, MIPS, SPARC and VAX) for whom the MD files are present in the GCC's source tree. We developed a tool \textit{mdcompare} to compare similarities across these MD files. To do so, we identified patterns that are common across machines and measured the percentage of RTL expressions that can be generated by these patterns by supplying the machine-specific information. The results show up to 70% similarity for similar architectures. The results largely confirm our hypothesis and justify the need for an automatic tool to generate MD files for new architectures from existing MD files for similar architectures.

1.2 Contribution of This Paper

The main contributions of our work are as follows:

- We describe what are RTL patterns and how these are useful (Section 2).
- We describe the tool \textit{mdcompare} that is used to extract RTL patterns (machine independent abstraction of RTL expressions) from MD files (Section 3.1).
- The tool is used to compare similarity across pairs of 5 well known architectures (Section 3.2 3.3).

2. RTL patterns and their usage

GCC compiler follows the model proposed by Davidson and Fraser [1] for code generation. Even though the RTL expressions in an MD file represent machine instructions for a specific machine, their form is machine-independent. We call this machine independent form \textit{RTL pattern} and use it for comparing similarity across MD files for different architectures.

\textbf{Definition 1.} An \textit{RTL pattern} is an RTL expression whose machine-specific details are replaced by named parameters. An RTL pattern can correspond to more than one RTL expressions that differ only in machine-specific details. These
RTL expressions could be for same machine or for different machine. Figure 2(b) shows an RTL pattern that can give rise to RTL expressions in Figure 2(a) by providing suitable values to parameters $\text{mode}$, $\text{arg0}$, $\text{arg1}$ and $\text{arg2}$.

**RTL Patterns** Machine-specific parts occur at well defined places in various RTL expressions, hence it is easy to extract RTL patterns from expressions. For example, the mode of an arithmetic operator (e.g., plus) is not part of a pattern because the modes supported are machine specific. Similarly, the match_operand expressions are machine-specific as they have machine-specific fields like predicate and constraint. Such details are removed from RTL expression and replaced by named parameters to obtain an RTL pattern. It should be obvious that an RTL pattern, when instantiated with suitable machine-specific parameters like mode or match_operand expressions, will result in an RTL expression.

Our tool *mdcompare* parses MD files and stores the RTL expressions as expression trees. It generates RTL patterns also in tree form. To extract the pattern from an expression, the tool traverses the expression tree in a bottom up fashion and replaces machine-specific subexpression trees by parameterized subpattern trees. It also maintains a mapping between the parameter variables and the corresponding values to allow reuse of parameters. This is important to keep the number of parameters small as some machine-specific features like mode (SI, DI, GPR etc.) are used several times in a single RTL expression. The tool maintains a list of patterns, already identified, sorted in the increasing order of their heights. When a new (sub)pattern is encountered, it is compared with patterns of same height for equality and, if not already present, is added to the list. We use the height of the patterns to avoid unnecessary comparison between patterns of different heights.

**Usefulness of RTL Patterns** We found that RTL patterns are useful in two ways: (a) RTL patterns help in understanding the structure of MD files by allowing the user to focus on the basic forms of instructions, without worrying about such machine-specific subexpression trees. It also maintains a mapping between the parameter variables and the corresponding values to allow reuse of parameters. This is important to keep the number of parameters small as some machine-specific features like mode (SI, DI, GPR etc.) are used several times in a single RTL expression. The tool maintains a list of patterns, already identified, sorted in the increasing order of their heights. When a new (sub)pattern is encountered, it is compared with patterns of same height for equality and, if not already present, is added to the list. We use the height of the patterns to avoid unnecessary comparison between patterns of different heights.

### 3. Experiments and Results

We have developed a tool *mdcompare* to experiment with MD files. We now describe our experience with the tool and explain the results obtained.

#### 3.1 Analyzing MD Files with mdcompare

The tool *mdcompare* parses MD files and stores the RTL expressions in tree form. The components of MD files that are not of interest to create RTL pattern, for e.g. Output Patterns [4], are ignored.

Once the expression tree is generated the tool can systematically remove machine independent parts of the expression to compute RTL patterns and the number of occurrences. These RTL patterns can be used to understand the contents of MD files or to compute similarity of two MD files. We have added some auxiliary features to the tool to test its correctness and to enhance its usability. The details of the tool and its features are given in details in Saravana’s thesis [7].

### Table 1. Number of RTL Expressions and Patterns.

| Arch. | #Exprs (E) | #Patterns (P) | Average (E/P) |
|-------|------------|---------------|---------------|
| ARM   | 382        | 1381          | 4.57          |
| MIPS  | 209        | 750           | 3.52          |
| SPARC | 187        | 701           | 3.74          |
| i386  | 54         | 2238          | 4.97          |
| VAX   | 64         | 1581          | 9.95          |

### Table 2. Number and percent of common patterns between pairs of MD files.

| Arch. | MIPS | SPARC | i386 | VAX |
|-------|------|-------|------|-----|
| ARM   | 79 (28.78%) | 101 (22.22%) | 35 (16.43%) |
| MIPS  | 73 (19.31%) | 73 (19.31%) | 29 (21.25%) |
| SPARC | 63 (17.17%) | 63 (17.17%) | 30 (23.90%) |
| i386  | 34 (11.13%) | 34 (11.13%) | 34 (11.13%) |

Number in each cell denote identical pattern. Numbers in parentheses denote the percentage similarity based on patterns.

To verify our hypothesis that there is a lot of similarity across MD files, we used the MD files of five architectures, namely ARM, i386, MIPS, SPARC, VAX taken from the back-end of GCC version 4.6.1 [3]. Note that ARM, MIPS and SPARC are RISC architectures, while i386 and VAX are CISC architectures.

### 3.2 Expressions and Patterns

Table 1 lists the number of RTL expressions considered in each of the architecture’s machine description file and the number of patterns that form the basis of it.

From Table 1, it can be seen that on average a pattern is used in 3–4 expressions for an architecture. This shows that patterns can help tell us about the level of redundancy that is present within machine description files. VAX seems to be an anomaly. This can be attributed to the CISC nature of VAX architecture because it has a small number of instructions with little or no variations.

### 3.3 Similarity Between Machine Descriptions

We now describe the results for MD file similarities. Table 2 lists the number of patterns that are common between the architectures and their percentage. The percentage is computed as follows: Let architecture $a_1$ have $p_1$ patterns and $a_2$ have $p_2$ patterns. Let $p$ be the number of identical patterns. Then, the % similarity of patterns is $\frac{2p}{p_1 + p_2} \times 100$.

From Table 2 we can see that 11–29% of patterns are common across architecture. These numbers are not very encouraging to justify our hypothesis. However, on further investigation we found that the number of expressions generated by common patterns form a significant percentage of the MD files. We measures the percentage of the expressions generated by common patterns as follows: Let architecture $a_1$ has $e_1$ expressions and $a_2$ has $e_2$ expressions. Let $p$ be the number of identical patterns, $e_1'$ be the number of expressions in $a_1$ generated by the $p$ common patterns, and $e_2'$ be the same number for $a_2$. Then, the % similarity of MD files $= \frac{e_1'+e_2'}{e_1+e_2} \times 100$. Table 3 lists the numbers for similarity based on RTL expressions derived from common patterns.

From Table 3, it can be seen that 30–65% of the expressions share common patterns. It is interesting to note that VAX architecture does not show much similarity to others, and derives only a small percentage of expressions from common patterns. We believe the reason is that VAX is the smallest architecture in terms of the number of RTL expres-
Table 3. Number and percent of expressions generated by common patterns for pairs of MD files.

| Arch. | MIPS (410) | SPARC (49.63%) | i386 (2281) | VAX (59.72%) |
|-------|-------------|-----------------|-------------|--------------|
| ARM   | 1486 (64.13%) | 1475 (64.63%) | 2281 (59.72%) | 799 (46.83%) |
| MIPS  | 838 (58.31%) | 1584 (53.26%) | 355 (41.23%) | 1196 (53.44%) |
| SPARC | 1441 (49.03%) | 410 (49.63%) | 719 (30.42%) |

Number in each cell denote identical patterns. Numbers in parentheses denote the percentage similarity based on patterns.

Table 4. RTL expressions for Target architectures as generated from RTL patterns of Source Architectures.

| Target (Total #Exprs) | ARM (1581) | MIPS (736) | SPARC (701) | i386 (2238) | VAX (125) |
|-----------------------|------------|------------|-------------|-------------|-----------|
| ARM                   | 504 (68.48%) | 483 (68.91%) | 1196 (53.44%) | 88 (70.40%) |
| MIPS                  | 982 (62.11%) | 391 (55.77%) | 1075 (48.03%) | 80 (64.00%) |
| SPARC                 | 992 (62.75%) | 447 (60.73%) | 994 (44.41%) | 74 (59.20%) |
| i386                  | 1085 (68.63%) | 509 (69.16%) | 447 (63.76%) | 87 (69.60%) |
| VAX                   | 711 (44.97%) | 215 (37.36%) | 336 (47.93%) | 122 (28.23%) |

Number in each cell denote number of expressions for Target that can be generated from RTL patterns common with Source. Numbers in parentheses denote corresponding percentage for expressions: VAX has just 125 RTL expressions resulting in 64 patterns. Further, being a CISC architecture, the instruction set differs considerably from others. In contrast, i386 has 2238 expressions and 547 patterns. Many i386 instructions are similar to those present in RISC architectures. This is the reason why i386 has large number of common expressions with RISC architectures, despite itself being a CISC architecture. If we look at architectures of relatively similar size (in terms of count of RTL expressions), the pair MIPS-SPARC has about 60% of the RTL expressions that can be instantiated from their common patterns.

Another interesting study that we performed was, given MD files for two architectures, say s (source) and t (target), how many RTL expressions of t can be generated by providing machine-specific information of t to the RTL patterns of s. This study gives us an idea about the percentage of MD file for t that can be generated automatically from s, provided we have an automatic tool as powerful as human developers.

Table 3 lists our findings. Due to its small size, the percentage for VAX are very high if it is used as a target, and very low if it is used as a source. Still, it is interesting to see that only 125 RTL expressions of VAX span 771 RTL expressions of ARM. If we ignore VAX, we see that 44–69% of the RTL expressions in the target architecture have a similar RTL expression in the source architecture, numbers being on the higher side for pairs of RISC architectures. These numbers strengthen our belief that there is lot of rework going on in retargeting GCC and it is worth investing efforts to build an automated tool to allow reuse of RTL patterns.

5. Conclusions

Retargeting GCC compiler to a new architecture is a challenging job. It can be made simpler by automating the generation of parts of MD files, by reusing information from existing MD files. In this paper we showed, through empirical studies, that popular architectures show a lot of similarities in their MD files. This supports the observation of Davidson and Fraser [1] that it is possible to generate machine-dependent code from a low level machine-independent form that is common across multiple architectures.

As part of our work, we implemented mdcompare to study and compare MD files. We chose to concentrate on RTL expressions in MD files, which are used to describe machine-specific instructions in machine-independent form in GCC. We devised a method to extract RTL patterns from the RTL expressions to ease the understanding and comparison. With this tool, we were able to measure similarities between machine description files based on the common RTL patterns. The results obtained are promising and can be used to justify efforts to build an automated tool to generate parts of MD file for a new architecture using a similar architecture.

One of the immediate goal is to create a Graphical User Interface (GUI) that can help the user to visualize RTL patterns and experiment with it by supplying machine-specific parameters. This will help in understanding MD files of existing architecture. The long term goal of this work is to build a tool which can automatically generate parts of MD files from existing MD files with user assistance. While it seems impossible to generate a complete MD file automatically, we believe that a partially generated MD file will help the user by reducing the efforts to populate the MD file and by reducing the chances of errors which occur during the process.

References

[1] J. W. Davidson and C. W. Fraser. Code selection through object code optimization. ACM Trans. Program. Lang. Syst., 6(4):505–526, October 1984.
[2] S. Deshpande and U. P. Khedker. Incremental machine descriptions for GCC. In GREPS: International Workshop on GCC for Research in Embedded and Parallel Systems, 2007.
[3] GCC. The GNU compiler collection. http://gcc.gnu.org
[4] GCC. GNU compiler collection internals. http://gcc.gnu.org/onlinedocs/gccint/
[5] U. P. Khedker and A. Mathur. specRTL: A language for GCC machine descriptions. In 3rd International Workshop on GCC Research Opportunities (GROW 2011), Chamonix, France, April 2011.
[6] K.-W. Lin and P.-S. . Chen. An assistance tool employing a systematic methodology for GCC retargeting. Journal of Software: Practice and Experience, 42(1):19–36, 2012.
[7] P. S. Perumal. Improving GCC retargetability. Master’s thesis, IIT Kanpur, 2012. [http://goo.gl/T3Io4](http://goo.gl/T3Io4)

[8] R. M. Stallman and the GCC Developer Community. Using the GNU compiler collection, (for GCC version 4.3.6).