Regular Paper

A Retargetable Code Generator for the Generic Intermediate Language in COINS

Seika Abe,† Masami Hagiya† and Ikuo Nakata††

This paper describes a generic intermediate language, called LIR, and a retargetable code generator for LIR, both of which were developed as part of the compiler for the COINS (COmpiler INfraStructure) project. Although the purpose and basic concepts of LIR are similar to those of RTL, the intermediate language of GCC, LIR is defined as an independent and self-contained programming language that has the constructs of a high-level language, such as variable declarations, and their formal semantics. As a result, LIR has several advantages over other representations currently in use. We can describe all compiler back-end passes as program transformations. Consequently, LIR provides a concise interface for interaction with the compiler for users who want to replace part of the compiler with their code. Most of the recently developed, retargetable code generators, such as Burg, IBurg, etc., are based on the DP matching method. Their machine description language consists of rewriting rules with code generation actions. However, the rules that are used to describe a machine do not correspond directly to any of the existing instructions of the target machine. On the other hand, the description language of GCC consists of descriptive statements that correspond to each of the target's existing machine instructions. However, the GCC code generator does not produce optimal code because it is not based on the DP method. Our code generator makes use of both the DP and GCC methods by translating the GCC descriptive statements into rewriting rules that are suitable for use by the DP matching method. Furthermore, DP matching is also implemented as a kind of transformation of an LIR program, and later transformations such as register allocation are applied to the resulting LIR program.

1. Introduction

This paper describes a generic intermediate language, called LIR, and a retargetable code generator for LIR, both of which were developed as part of the compiler for the COINS (COmpiler INfraStructure) project [1]. COINS is an ongoing project to develop a compiler infrastructure that can be used as a base for constructing various compilers such as research compilers, educational compilers, and production compilers. COINS has two levels of intermediate representation; high-level intermediate representation (HIR), and low-level intermediate representation (LIR). HIR is used as a language for higher-level optimizations such as parallelization, whereas LIR is used for conventional code optimizations and for final assembler output, commonly known as compiler back-end passes. In this paper, we focus on the structure of LIR and on the retargetable code generator that we developed for LIR.

A compiler intermediate language is usually introduced as a private language that is specific to a certain compiler, and is known only to the compiler writers. However, many compiler bugs are caused by ambiguities in the specification of a compiler's intermediate language. Moreover, the COINS compiler is an open source program that can be freely modified and extended by any users. Thus, we have defined the formal semantics of LIR based on the ordinary denotational semantics. All compiler back-end passes, including instruction selection by our code generator and register allocation, are regarded as program transformations in LIR, which preserve the formal semantics of LIR programs. Consequently, LIR provides a concise interface for interaction with the compiler for users who want to replace part of the compiler with their own code. They can even write their own code in a language other than JAVA, which is the implementation language of the COINS compiler, and in fact the first implementation of our code generator was written in Scheme.

A retargetable code generator is one that can produce object code for various target machines without any modification to the generator. In such a code generator, the characteristics of a target machine are described in the machine description language of the generator, which is independent of any target machine. Most re-

† Graduate School of Information Science and Technology, the University of Tokyo
†† Faculty of Computer and Information Sciences, Housei University

cently developed retargetable code generators, such as Burg, IBurg, etc., are based on the DP matching method. Their machine description languages consists of rewriting rules with code generation actions. Theoretically, this method gives the best results. However, the rules that are used to describe a machine do not correspond directly to any of the existing instructions of the machine owing to the characteristics and theory of the DP method. This mismatch often makes the work of describing a machine a lengthy and somewhat onerous process. On the other hand, the code generator that is used by GCC [15], one of the most widely used, retargetable compiler, employs a unique method that differs considerably from the DP method. Its description language consists of descriptive statements that correspond to each of the target's existing machine instructions. However, GCC code generator still has some limitations that prevent it from producing optimal code. Our code generator takes advantage of the good points of both the DP and GCC methods by translating the GCC descriptive statements into rewriting rules that are suitable for use by the DP matching method. Furthermore, DP matching is also implemented as a kind of transformation of an LIR program, and later transformations such as register allocation are applied to the resulting LIR program. The description language of our code generator is also equipped with a Scheme interpreter [16] and a simple macro feature to improve expressiveness.

The code generator described in this paper is currently not used as is in the COINS compiler. The code generator implemented in Scheme has been rewritten in Java to improve performance but the internal structure of the new code generator is essentially the same as that of the previous one and works with exactly the same interface.

The remainder of this paper is organized as follows. Our intermediate language, LIR, is introduced in Section 2. The overall structure of our code generator is described in Section 3. Section 4 describes the machine description language of our code generator. The instruction selection pass of our code generator is explained through the use of examples in Section 5. In Section 6, we compare our intermediate language and its code generator with other similar systems. Section 7 presents current performance of the COINS compiler. Finally, our concluding remarks and recommendations for future work are given in Section 8.

2. The LIR Intermediate Language

LIR and its formal semantics are introduced in the following subsections.

2.1 Examples of LIR

In this subsection, LIR is described by using examples to explain its overall structure. The examples are written in C, since it, like LIR, is a low-level language that is close to the actual hardware.

The program listings in **Fig. 1** are two C programs, `main.c` and `sub.c`, in which the function `prodv` returns the product of all the elements of the array `v` using the function `fold1`.

The first C program '`main.c`' is translated into the LIR code shown in **Fig. 2**, which is called an L-module. The translation assumes that an int, a pointer, and a float are all 32 bits long. It also assumes that their required alignments are in the same four-byte boundary. Machine instructions and data are stored in the segments `text` and `data`, respectively.

The syntax of LIR is built on top of S-expressions. A semicolon is used to indicate a comment. All of the characters following it in the same line are part of a comment. Also, the numbers that appear on the left side of the LIR listing are not part of LIR; they are used only as line references in the following explanation. The listing in Fig. 2 is an example of an L-module. An L-module consists of its module name, its L-association list, and the definitions of L-functions and L-data. An L-function definition consists of its name, its local function L-association list, and its L-sequence. An L-sequence is a list of L-expressions beginning with a `PROLOGUE` expression and ending with an `EPILOGUE` expression. An L-data definition consists of its name and a list of its contents.

An L-association list, beginning with the keyword `ALIST`, consists of several entries. The first element of each entry is a name that is defined by the rest of the elements of the entry. The second element of each entry is called a class, which is used to dictate the syntax for the rest of the entry. It also determines how the entry's remaining definitions are to be interpreted. Names declared in L-association lists are referred to by L-expressions in L-functions that follow the same static scope rules as those that are found in the C programming language. A name with the class `STATIC` represents a stat-

```
main.c :   extern float fold1(float f(float,float), float v[], int n);
           static float v[] = {1, 2.5, 3};
           static int n = sizeof v / sizeof v[0];
           static float fmul(float x, float y){float r=x*y; return r;};
           float prodv(){return fold1(fmul,v,n);};

sub.c :    float fold1(float f(float,float), float v[], int n)
           {
             int i; float r;
             for (r=v[0], i=1; i<n; i++) r = f(r,v[i]);
             return r;
           }
```

Fig. 1 A sample C program.

```
 1  (MODULE "main"
 2    (ALIST
 3      ;; name  class  L-type align segment linkage
 4      ("fold1" STATIC UNKNOWN  4   "text"  XREF)
 5      ("v"     STATIC 96       4   "data"  LDEF)
 6      ("n"     STATIC I32      4   "data"  LDEF)
 7      ("fmul"  STATIC UNKNOWN  4   "text"  LDEF)
 8      ("prodv" STATIC UNKNOWN  4   "text"  XDEF))
 9    ;; definition of the function fmul
10    (FUNCTION "fmul"
11      (ALIST
12        ;; name class  L-type align offset
13        ("x"    FRAME  F32     4       0)
14        ("y"    FRAME  F32     4       4)
15        ("r"    FRAME  F32     4       8))
16      ;; L-sequence
17      (PROLOGUE (12 0) (MEM F32 (FRAME I32 "x")) (MEM F32 (FRAME I32 "y")))
18      (SET F32 (MEM F32 (FRAME I32 "r")) ; r=x*y
19               (MUL F32 (MEM F32 (FRAME I32 "x"))
20                        (MEM F32 (FRAME I32 "y"))))
21      (EPILOGUE (12 0) (MEM F32 (FRAME I32 "r"))))
22    ;; definition of the data v and n
23    (DATA "v" (F32 1.0 2.5 3.0))
24    (DATA "n" (I32 3))
25    ;; definition of the function prodv
26    (FUNCTION "prodv"
27      (ALIST
28        ("t1" FRAME F32 4 0)) ; t1 is generated by the translator
29      (PROLOGUE (4 0))
30      (CALL (STATIC I32 "fold1") ; t1=fold1(fmul,v,n)
31            ((STATIC I32 "fmul") (STATIC I32 "v") (MEM I32 (STATIC I32 "n")))
32            ((MEM F32 (FRAME I32 "t1"))))
33      (EPILOGUE (4 0) (MEM F32 (FRAME I32 "t1")))))
```

Fig. 2 The LIR code for `main.c`.

ically allocated object. A name with the class `FRAME` represents an object that is allocated on the stack frame. For example, in line 6 of the L-module `main` above, the name `n` is declared to be a statically allocated object having the L-type `I32` (a 32-bit integer) along with its alignment, segment, and linkage information. An object's linkage information will always be one of three symbols, `LDEF`, `XDEF`, and `XREF`, which respectively mean that the name is locally defined, globally defined, or that the name is an external reference.

The declaration of `x` in line 13 is an example of a frame variable and another kind of L-type. `F32` is the L-type that is used to designate 32-bit floating point numbers. The 0 stands for its offset from the frame pointer. Although we assume the existence of a frame pointer, we treat it implicitly. In line 5 of the module, the L-type of the name `v` is just 96 but this is also another

L-type. It means the type of an object has 96-bits (12-bytes). We have introduced three kinds of L-types; namely, n-bit integers `I`n, n-bit floats `F`n, and just n. These comprise all of the kinds of L-types that we use. The symbol `UNKNOWN` is used to indicate a type whose size is unknown to the compiler. `UNKNOWN` is not an L-type.

Line 10 shows an example of a function definition. Two L-expressions, `PROLOGUE` and `EPILOGUE`, are used to specify the interface of the L-function. They are collectively called interface expressions. The second element of the `PROLOGUE` expression in line 17, `(12 0)`, has the syntax (w_f w_r). This syntax is used to designate the size of the frame and the register frame to be allocated (or deallocated in the case of `EPILOGUE`). We are not including any further explanation of register frames. The remaining elements of the `PROLOGUE` expression specify the arguments of the L-function. The remaining elements of the `EPILOGUE` expression specify the list of expressions to be returned as multiple return values.

Line 18 shows a typical example of an L-expression. Unlike the corresponding C code `r=x*y`, it explicitly represents memory accesses using the expression (`MEM` *type address*), which refers to the object with the specified type and address. The frame expression `(FRAME I32 "x")` represents the address of the variable `x` that was declared in line 13. `I32` is the type of type address.

Line 30 shows an example of a function call. Its syntax has the form:

`(CALL` *addr* `(`*args*`...)` `(`*results* `...))`,

where *addr* is the address of the function to be called, *args* are the L-expressions to be passed to the function, and *results* are the variables to which the multiple return values of the function will be assigned. As the CALL expression cannot be part of any other L-expression, the temporary variable '`t1`' is introduced by the translator.

Line 31 shows an example of accessing a global (as in the C language) variable, where the L-expression `(MEM I32 (STATIC I32 "n"))` represents the address of the variable `n` that was declared in line 7.

This L-module does not have any examples of registers. In LIR, a register is expressed as (`REG` *type name*) and the *name* is declared in an L-association list as (*name* `REG` *type offset*), where *type* is the type of the register and *offset* is the address of the register in register memory. There is no syntactic distinction between virtual and real registers.

As we have seen, our expressions for a calling convention are at a much higher level than the corresponding GCC expressions in RTL. To realize a calling convention with existing instructions is often called calling convention expansion.

GCC expands all of its calling conventions and designates some real registers at an early stage of its compilation process. The advantage of GCC's approach is that optimizers can achieve various machine specific optimizations. For example, as the stack pointer is a preallocated register, it explicitly exists throughout compiler passes; this enables optimizers to do some stack related optimizations, such as defer pop. The disadvantage is that the approach makes optimizers so complex since function interface codes are already expanded. Code optimizers can hardly recover them from a given code; this disables optimizers to do some higher-level optimizations, such as inline expansion, tail recursion elimination, and partial evaluation.

The constructs of LIR for a (multiple-valued) function consisting of `PROLOGUE`, `EPILOGUE`, and `CALL` expressions enable the code optimizer to handle registers by making the following assumptions:

(1) Arbitrariness of register names: Renaming registers local to a function does not change the meaning of the function.
(2) Independence of registers: Assigning a register local to a function does not alter any other registers.

As the parameters of a function are specified in an interface expression, renaming them does not change the meaning of the function. Designating real registers before code optimization clearly makes the above assumptions impossible, as the registers will now have unique names that sometimes partly overlap.

Our intention in introducing such higher-level constructs was to clarify and simplify code optimizers by separating and delaying the designation of real registers.

We can describe all of the passes of a compiler including code optimization, instruction pattern matching, register allocation, and peephole optimization in terms of program transformations using LIR. With this in mind, the task of instruction selection that is based on

```
 1  (MODULE "sub"
 2    (ALIST
 3      ("fold1" STATIC UNKNOWN 4 "text" XDEF))
 4    (FUNCTION "fold1"
 5      (ALIST
 6        ("f" FRAME I32 4 0)
 7        ("v" FRAME I32 4 4)
 8        ("n" FRAME I32 4 8)
 9        ("i" FRAME I32 4 12)
10        ("r" FRAME F32 4 16))
11      (PROLOGUE (20 0) (MEM I32 (FRAME I32 "f"))
12                       (MEM I32 (FRAME I32 "v"))
13                       (MEM I32 (FRAME I32 "n")))
14      (SET F32 (MEM F32 (FRAME I32 "r")) ; r=v[0]
15               (MEM F32 (MEM I32 (FRAME I32 "v"))))
16      (SET I32 (MEM I32 (FRAME I32 "i")) ; i=1
17               (INTCONST I32 1))
18      (DEFLABEL "L1")
19      (JUMPC (TSTLT I32 (MEM I32 (FRAME I32 "i")) ; if (i<n) goto L2; else goto L3
20                        (MEM I32 (FRAME I32 "n"))) (LABEL I32 "L2") (LABEL I32 "L3"))
21      (DEFLABEL "L2")
22      (CALL (MEM I32 (FRAME I32 "f")) ; r=f(r,v[i])
23            ((MEM F32 (FRAME I32 "r"))
24             (MEM F32 (ADD I32 (MEM I32 (FRAME I32 "v")) ; v[i]
25                               (MUL I32 (MEM I32 (FRAME I32 "i"))
26                                        (INTCONST I32 4)))))
27            ((MEM F32 (FRAME I32 "r"))))
28      (SET I32 (MEM I32 (FRAME I32 "i")) ; i++
29               (ADD I32 (MEM I32 (FRAME I32 "i"))
30                        (INTCONST I32 1)))
31      (JUMP (LABEL I32 "L1"))
32      (DEFLABEL "L3")
33      (EPILOGUE (20 0) (MEM F32 (FRAME I32 "r")))))
```

Fig. 3 The LIR code for `sub.c`.

a machine's description can be formalized as a program transformation that reforms each L-expression into one that expresses a real instruction of a real machine. The task of register allocation can also be formalized as a transformation to change local registers to global ones. After these transformations, the L-function `fmul` would take the following form:

```
(MODULE "main"
  (ALIST
    .......
    ("F0" REG F32 0) ; real reg F0
    ("F1" REG F32 4) ; real reg F1
    ......

  (FUNCTION "fmul"
    (ALIST )
    (PROLOGUE (0 0)
              (REG F32 "F0")
              (REG F32 "F1"))
    (SET F32 (REG F32 "F0")
             (MUL F32
                  (REG F32 "F0")
                  (REG F32 "F1")))
    (EPILOGUE (0 0) (REG F32 "F0")))
  .....
```

The modified registers now express the registers of a real target machine via their types and offsets.

The translated L-module for `sub.c` is shown in **Fig. 3**. Note that this example includes control structures. We never introduce 'structured' control constructs such as if and while. The expression (`DEFLABEL` *label*) defines a label *label* which is referenced by jump expressions such as an unconditional jump as shown in line 31, a conditional jump as shown in line 19, and a multiway jump. An example of an indirect call is shown in line 22. The address expression in the `CALL` expression must evaluate to the address of an L-function. In our model, program memory is not an arbitrary list of L-expressions. Rather, it is a list of L-functions that can be invoked only by `CALL` expressions. At the same time, the targets of any jump expressions are limited to those targets that are within the function to which the jump belongs.

2.2 The Formal Semantics of LIR

The benefits that have been realized by designing LIR as a self-contained program-

Table 1 Semantic functions.

component	meaning	syntax	semantics
L-program	a consistent set of modules	`Lprog`	$\mathcal{P}$
L-module	collection of data and functions	`Lmod`	$\mathcal{M}$
L-association list	mapping from string to its meaning	`Lalist`	$\mathcal{A}$
L-data	statically allocated data	`Ldata`	$\mathcal{D}$
L-function	L-sequence with function interface	`Lfunc`	$\mathcal{F}$
L-sequence	sequence of instructions	`Lseq`	$\mathcal{S}$
L-expression	instruction	`Lexp`	$\mathcal{E}$
L-type	type	`Ltype`	$\mathcal{T}$

ming language have already been described above. These benefits have been amply demonstrated throughout our development of compiler back-ends. In contrast to high-level programming languages, compiler intermediate languages have usually been private languages that are known only to the compiler writers who use them. Therefore, these compiler intermediate languages have not needed to have rigorous specifications. However, the COINS compiler is an open source program that can be freely modified and extended by any users. Thus, the intermediate language in COINS must be specified as rigorously as possible in a similar fashion to that of a general high-level programming language.

Many compiler bugs are caused by ambiguities in the specification of a compiler's intermediate language. In the case of a retargetable compiler, such a bug is often revealed when the compiler is used as a cross-compiler. Ambiguously specified intermediate code might well have two different interpretations; one for the machine on which the compiler is running, and another for the target machine.

2.2.1 Domains and Semantic Functions

The semantics of LIR is defined as denotational semantics [2], which is briefly discussed below. The semantics consists of the eight semantic functions shown in **Table 1**. Each function takes a corresponding syntactic object and returns a semantic value of a certain type.

For example, the semantic function $\mathcal{E}$ has the type shown below.

$$\mathcal{E} \ : \ \texttt{Lexp} \to \texttt{Env} \to \texttt{Mem} \to (\texttt{Mem} \times \texttt{Bits})_\perp$$

$$\begin{aligned}\texttt{Env} \triangleq \{\ & \texttt{reg}\colon \texttt{RegEnv},\\ & \texttt{sta}\colon \texttt{StaticEnv},\\ & \texttt{fra}\colon \texttt{FrameEnv},\\ & \texttt{lab}\colon \texttt{LabelEnv}\ \}\end{aligned}$$

$$\begin{aligned}\texttt{Mem} \triangleq \{\ & \texttt{pc}\colon \texttt{Location},\\ & \texttt{pm}\colon \texttt{PMem},\\ & \texttt{rm}\colon \texttt{RMem},\\ & \texttt{dm}\colon \texttt{DMem},\\ & \texttt{tr}\colon \texttt{Trace},\\ & \texttt{rs}\colon \boldsymbol{R}\ \}\end{aligned}$$

This means that the function $\mathcal{E}$ receives an L-expression of type `Lexp`, the current environment of type `Env`, the current memory (L-memory) of type `Mem`, and then returns a value of type $(\texttt{Mem} \times \texttt{Bits})_\perp$. Erroneous results are expressed as the bottom element. The record type `Env` consists of functions that bind four kinds of names appearing in L-expressions to their concrete location in an L-memory of type `Mem`. The record type `Mem` consists of a program counter `pc`, three kinds of memory, and special registers `tr`, and `rs`. We divide memory into program memory `PMem`, register memory `RMem`, and data memory `DMem`. The special registers are used to simulate volatile memory in denotational semantics, as explained in Section 2.2.3.

The domain of values is just a set of bits defined as follows.

$$\texttt{Bit}\, w \triangleq \{0,1\}^n$$

$$\texttt{Bits} \triangleq \bigcup_{n=0}^{\infty} \texttt{Bit}\, w$$

$$\texttt{Byte} \triangleq \texttt{Bit}\, 8$$

This decision is natural because the memory of ordinal machines is typeless and a value in the memory can be fetched as of a type different from the type when it was stored. Even a simple arithmetic operation such as addition is regarded as an operation on bits. To convert between a bit and an arithmetic value, conversion functions are used (**Fig. 4**).

2.2.2 Arithmetic Operations

Using the conversion functions, our specification of addition is given as follows.

$$\begin{aligned}&(\texttt{ADD}\ t\ x_1\ x_2) \quad t = t_1 = t_2\\ &\quad \mathcal{E}[\![(\texttt{ADD}\ t\ x_1\ x_2)]\!]\rho\,\sigma\\ &\quad\ \triangleq\ (\sigma_2, \texttt{bz}\, w\, (\texttt{zb}\, v_1 + \texttt{zb}\, v_2)) \ \textbf{if}\ t \in \texttt{Itype}\\ &\quad \mathcal{E}[\![(\texttt{ADD}\ t\ x_1\ x_2)]\!]\rho\,\sigma\end{aligned}$$

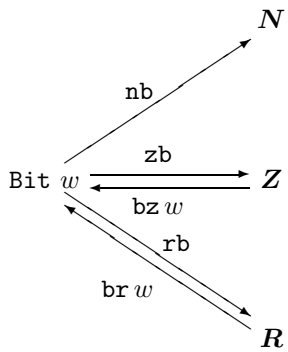


Fig. 4 Conversion functions.

$\triangleq$ $(\sigma_2, \texttt{br}\, w\, (\texttt{rb}\, v_1 + \texttt{rb}\, v_2))$ **if** $t \in \texttt{Ftype}$

It consists of the syntax, constraints, and semantic definitions of addition.

For brevity's sake, we assume the following rules for each such definition. If $x_i \in \texttt{TypedExp}$ and the symbols t_i, σ_i, v_i are not defined, then the following implicit definitions are assumed.

$$\begin{aligned} t_i &\triangleq x_i.\texttt{type} \\ w &\triangleq \text{number of bits of } t \\ (\sigma_1, v_1) &\triangleq \mathcal{E}[\![x_1]\!]\rho\sigma \\ (\sigma_2, v_2) &\triangleq \mathcal{E}[\![x_2]\!]\rho\sigma_1 \\ &\vdots \\ (\sigma_i, v_i) &\triangleq \mathcal{E}[\![x_i]\!]\rho\sigma_{i-1} \end{aligned}$$

Note that these definitions also specify the evaluation order of the arguments in L-expressions as left to right. In the above case, the constraints $t = t_1 = t_2$ say that both of the arguments x_1, x_2 must have the type equal to the result type t of the addition.

There are some operations that are hard to define. For example, the behavior of shift operations can be subtly different for each machine. To account for these potential differences, we allow an optional modifier $(s\ n\ d)$ at the end of an L-expression prefixed with `'&'`, where s indicates that the shift is signed ($s = \texttt{S}$) or unsigned ($s = \texttt{U}$), n gives the bit width of the shift count, and $d = \texttt{D}, \texttt{U}$ determines whether the result is defined ($d = \texttt{D}$) or undefined ($d = \texttt{U}$) if a given shift count cannot be expressed in n bits. We are not including a detailed explanation here. Rather, we just cite the definition of left shift as follows:

$\mathcal{E}[\![(\texttt{LSHS}\ t\ x_1\ x_2\ \&\ (s\, n\, d))]\!]\rho\sigma$
$\triangleq$ $(\sigma_2, \texttt{genericshift}\, w\, (\texttt{zb}\, v_1)$
$(\texttt{shiftcount}\, v_2\, [\![(s\, n\, d)]\!]))$
$\texttt{genericshift}\colon w\colon \boldsymbol{N} \to \boldsymbol{Z} \to \boldsymbol{Z} \to \texttt{Bit}\, w$
$\texttt{genericshift}\, w\, n\, c \triangleq \texttt{bz}\, w\, (\texttt{floor}(n * 2^c))$
$\texttt{shiftcount}\colon \texttt{Bits} \to \texttt{ShiftModifier} \to \boldsymbol{Z}_\perp$
$\texttt{shiftcount}\, b\, [\![(s\, n\, d)]\!]$
$\triangleq$ **if** $c_0 = c_1$ **then** c_0 **else**
case $[\![d]\!]$ **of** $\texttt{D} \Rightarrow c_1\ \texttt{U} \Rightarrow \perp$
where
$f \triangleq$ **case** $[\![s]\!]$ **of** $\texttt{S} \Rightarrow \texttt{zb}\ \texttt{U} \Rightarrow \texttt{nb}$
$c_0 \triangleq f\, b$
$c_1 \triangleq f(\texttt{bz}\, n\, c_0)$

The following table shows some examples of the correspondence between shift modifiers and actual machines.

S8D	VAX, V60/70
U5D	SPARC(32), MIPS(32), Intel x86
U6D	SPARC(64), MIPS(64), PowerPC(32)
U7D	PowerPC(64)
U64D	Intel MMX

2.2.3 Memory Access

The notion of volatile objects comes from the C language. Our definition of memory access operations takes this notion into account. To simulate a volatile object in the denotational semantics, we have introduced the trace register `tr` and the random state register `rs` as mentioned above.

The semantics of a memory read expression is defined as follows:

$\mathcal{E}[\![(\texttt{MEM}\ t\ x_1\ \&\ m)]\!]$
$\triangleq$
case $[\![m]\!]$ **of**
$\texttt{N} \Rightarrow (\sigma_1, \texttt{dmread}\, \sigma_1\, (\texttt{nb}\, v_1)\, [\![t]\!])$
$\texttt{V} \Rightarrow \texttt{random}[\![t]\!]\, (\texttt{addtotr}\, \sigma_1\, \texttt{READ}\, [\![t]\!]\, [v_1])$
where $(\sigma_1, v_1) \triangleq \mathcal{E}[\![x_1]\!]\rho\sigma$

This expression means the object of type t at the address x_1 in data memory. It represents a volatile object if a modifier $m \in \texttt{MemModifier}$ is given and $m = \texttt{V}$. In this case, the reading is recorded in the trace of the L-memory, and the result is an unpredictable value.

The semantics of a memory write expression is defined as follows:

$\mathcal{E}[\![(\texttt{SET}\ t\ (\texttt{MEM}\ t'\ x_1\ \&\ m)\ x_2)]\!] \triangleq (\sigma_4, v_2)$
where
$(\sigma_1, v_1) \triangleq \mathcal{E}[\![x_1]\!]\rho\sigma$

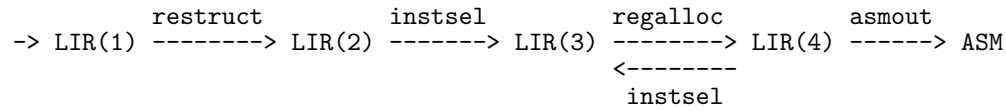


Fig. 5 Code generation passes.

$$
\begin{aligned}
(\sigma_2, v_2) &\triangleq \mathcal{E}[\![x_2]\!]\rho\sigma_1\\
\sigma_3 &\triangleq \texttt{dmwrite}\,\sigma_2(\texttt{nb}\,v_1)\,[\![t]\!]\,v_2\\
\sigma_4 &\triangleq \textbf{case}\ [\![m]\!]\ \textbf{of}\\
&\quad \texttt{N} \Rightarrow \sigma_3\\
&\quad \texttt{V} \Rightarrow \texttt{addtotr}\,\sigma_3\,\texttt{WRITE}[\![t]\!][v_1, v_2]
\end{aligned}
$$

This expression stores the value of x_2 into the object of type t at the address x_1. The `MEM` sub-expression represents a volatile object if a modifier $m \in$ `MemModifier` exists and $m = \texttt{V}$. In this case, the writing is recorded in the trace of the L-memory.

The trace register holds the history list recording the kind of memory accesses that have been done. The random state register is used to simulate the asynchronous modification of memory by an external device.

2.2.4 Function Call

The semantics of a function call is defined as follows:

$$
\begin{aligned}
&\mathcal{E}[\![(\texttt{CALL}\ x_1\ (x_2\ \cdots\ x_n)\ (y_1\ \cdots\ y_m))]\!]\rho\,\sigma\\
&\quad\triangleq\ (\sigma'_m, \emptyset)\\
&\quad\ \textbf{where}\\
&\quad\ (\sigma'_0, [b_1, \ldots, b_m])\\
&\qquad\triangleq\ \mathcal{F}[\![\sigma.\texttt{pm}\,v_1]\!]\,\rho\,[v_2, \ldots, v_n]\,\sigma_n\\
&\quad\ t_i \triangleq y_i.\texttt{type}\\
&\quad\ b'_i \triangleq \texttt{mkconst}[\![t_i]\!]\,b_i\\
&\quad\ (\sigma'_i, u_i) \triangleq \mathcal{E}[\![(\texttt{SET}\ t_i\ y_i\ b'_i)]\!]\,\rho\,\sigma'_{i-1}
\end{aligned}
$$

This expression takes the arguments $x_2, \ldots, x_n$, call the (multiple-valued) function stored at the memory location $\sigma.\texttt{pm}\,v_1$, and returns the values into $y_1, \ldots, y_m$.

$\mathcal{F}$ is the semantic function for function definitions of LIR as follows.

$$
\begin{aligned}
\mathcal{F}\ &:\ \texttt{Lfunc} \to \texttt{Env} \to \texttt{Args}\\
&\quad \to \texttt{Mem} \to (\texttt{Mem} \times \texttt{Rets})_\perp\\
\texttt{Args} &\triangleq [\texttt{Bits}]\\
\texttt{Rets} &\triangleq [\texttt{Bits}]
\end{aligned}
$$

The whole definition of $\mathcal{F}$ is lengthy. Rather, we just cite the toplevel definition of it:

$$
\begin{aligned}
&\mathcal{F}[\![(\texttt{FUNCTION}\ \mathit{name}\ \mathit{alist}\ \mathit{seq}@(\mathit{pro}\ \cdots\ \mathit{epi}))]\!]\\
&\quad\rho\,[a_1, \ldots, a_n]\,\sigma \triangleq (\sigma_6, [b_1, \ldots, b_m])\\
&\quad\textbf{where}\\
&\quad(\texttt{PROLOGUE}\ (w_f\ w_r)\ x_1\ \cdots\ x_n) \triangleq \mathit{pro}\\
&\quad(\texttt{EPILOGUE}\ (w_f\ w_r)\ y_1\ \cdots\ y_m) \triangleq \mathit{epi}\\
&\quad\rho' \triangleq \texttt{f\_newenv}\,\rho\,\sigma.\texttt{rm}\ \mathit{alist}\ \mathit{seq}\ w_r\\
&\quad(\sigma_1, v_1) \triangleq \mathcal{E}[\![\mathit{pro}]\!]\,\rho\,\sigma\\
&\quad\sigma_2 \triangleq \texttt{f\_args}\\
&\qquad\qquad [\![(x_1\ \cdots\ x_n)]\!]\,\rho'\,\sigma_1\,[a_1, \ldots, a_n]\\
&\quad\sigma_3 \triangleq \texttt{f\_exec}\,\mathit{seq}\,\rho'\,(\sigma_2 := \{.\texttt{pc} = 1\})\\
&\quad(\sigma_4, [b_1, \ldots, b_m])\\
&\qquad\triangleq \texttt{f\_rets}[\![(y_1\ \cdots\ y_m)]\!]\,\rho'\,\sigma_3\\
&\quad(\sigma_5, v_2) \triangleq \mathcal{E}[\![\mathit{epi}]\!]\,\rho\,\sigma_4\\
&\quad\sigma_6 \triangleq \texttt{f\_newmem}\,\sigma_5\,\sigma
\end{aligned}
$$

This function converts the syntactic object of an L-function definition into the mathematical multiple-valued function. As such mathematical functions are entirely independent from their 'implementation', we can define equivalence between L-functions in terms of the denotational semantics. And we can also define code optimizations as transformations of an L-function that keeps this equivalence.

3. The Overall Structure of Our Code Generator

Figure 5 depicts the overall flow of our code generator. LIR(1) to LIR(4) are all LIR codes and ASM is the assembler code of the target machine. The programs shown by the arrows do transformations that are dependent upon the target machine.

The parts of the programs that are target machine dependent are generated automatically by a tool from a file containing a description for the target machine. This description is called a target machine description or just a machine description. The tool is called a target machine description compiler (TMDC).

The program `restruct` does some program transformations that cannot be achieved by our instruction selector `instsel`. We call these transformations restructuring. A typical exam-

Table 2 Examples of machine parameters.

Symbol	Meaning
`*real-reg-symtab*`	List of all registers
`*cmplib-xref-symtab*`	List of external symbols of a compiler library
`*reg-I32*`, `*reg-I16*`, ...	List of registers
`*reg-call-clobbers*`	List of registers clobbered by procedure calls

ple of restructuring is expanding calling conventions. As a result of this restructuring, LIR(2) may include some real registers.

The program `instsel` is a tree that uses a DP matching based instruction selector. Like IBurg, our selector does DP matching at code selection time and the cost of an instruction specification can be arbitrary expression. It translates each L-expression in LIR(2) into a sequence of L-expressions that are existing instructions for the target machine. The resulting LIR code, LIR(3), still includes virtual registers.

The program `regalloc` is a graph-coloring-based register allocator that assigns real registers to all of the remaining virtual registers in LIR(3). This allocator may call `instsel` to fix partially modified L-expressions due to register spills. Then, `regalloc` is called again. This process is repeated until no virtual registers remain in LIR(4). LIR(4) is almost in the form of the object code for the target machine, except for its exact syntax. The program `asmout` generates the final assembler code, ASM, from LIR(4) for the target machine.

4. Machine Descriptions

A machine description for a target machine consists of the following parts:

(1) Machine parameters
(2) Instruction definitions
(3) Restructuring procedures
(4) Assembler language definitions

The various properties of a machine, such as the organization of its registers, are defined in variables called machine parameters. Machine instructions to be used in code generation are defined in the instruction definitions. The restructuring procedures are a collection of programs for restructuring LIR programs. The syntax of the target machine's assembler language is defined in the assembler language definitions.

As explained later, the syntax of all of these parts is in the form of S-expressions, like LIR. Thus LIR code fragments can naturally be embedded inside machine descriptions. We also employ a Scheme interpreter. Scheme is used mainly to write programs that handle LIR codes directly. For example, restructuring procedures and assembler language definitions are written in Scheme.

4.1 Machine Parameters

To set the machine parameter `<Symbol>` to a value `<Sexp>`, we use the following `<Def>` form.

```
<Def> ::= (def <Symbol> <Sexp>)
```

This is different from the `(define ...)` in Scheme. This form is used to inform the code generator about the various characteristics of a target machine via a machine parameter. Some examples of machine parameters are shown in **Table 2**.

4.2 Instruction Definitions

The instructions of a target machine are defined using two forms, `<DefRule>` and `<DefCode>`. The former defines the addressing modes while the latter defines the instructions. The differences between these two forms are explained in detail in Section 4.4.

4.3 Defrule Form

`defrule` forms correspond to rewriting rules in Burg-style generators. The syntax of a `defrule` form is as follows:

```
<DefRule> ::= (defrule <NonTerminal>
                  <Pattern>
                  {<InstInfo>})
<NonTerminal> ::= <Symbol>
       beginning with a lower letter
<Pattern> ::= An LIR expression
       with <NonTerminal> or _
```

A `defrule` form defines a non-terminal `<NonTerminal>` that can be rewritten to a pattern `<Pattern>`. A pattern is like an L-expression except that some sub-expression of it can be a non-terminal or an underscore '`_`'.

A `defrule` form can optionally have some `<InstInfo>`s consisting of the following:

```
<InstInfo>  ::=   <InstInfoAsm>
                | <InstInfoCost>
                | <InstInfoCond>
                | <InstInfoClobber>
```

where `<InstInfoAsm>` is an assembler output template. `<SchemeCode>` is evaluated and becomes a part of the assembler output.

```
<InstInfoAsm> ::= (asm <SchemeCode>)
```
<SchemeCode> is a Scheme code that can include some pattern variables of the syntax <PatRefVar>.
```
<SchemeCode> ::= Any Scheme code
                 with <PatRefVar>
<PatRefVar>  ::= $0 | $1 | ...
```
The pattern variable $n refers to the n-th non-terminal in the rule. The first one $0 corresponds to the non-terminal after the keyword defrule while the others correspond to each occurrence in <Pattern>.

Consider the following example.
```
(defrule reg (REG _ _)
  (asm `(reg ,(caddr $0))))
(defrule addr (ADD I32 reg reg)
  (asm `(add ,$1 ,$2)))
```
The first rule matches an L-expression denoting a register and the second rule matches a part of an addressing mode. When the second rule matches an L-expression, say (ADD I32 (REG I32 "r1") (REG I32 "r2")), the variables $1 and $2 correspond to the two occurrences of the non-terminal reg in the pattern. They are evaluated to (reg "r1") and (reg "r2"), respectively. These values are produced by (asm ...) in the first rule. The variable $0 in the first rule denotes the source of the rewriting rule, (REG I32 "r1") or (REG I32 "r2") in this case. Finally, the second rule returns (add (reg "r1") (reg "r2")). This S-expression denotes an addressing mode and is eventually converted into real assembler notation.

The other possibilites of <InstInfo> are as follows.
```
<InstInfoCost>
  ::= (cost <SchemeCode>)
<InstInfoCond>
  ::= (eq <PatRefVar> <PatRefVar>)
    | (regset {(<PatRefVar>
                <Symbol>)}
    | (cond <SchemeCode>)
<InstInfoClobber>
  ::= (clobber {<RegExp>})
```
<InstInfoCost> specifies the execution cost of the L-expression that matches the pattern. This value is used by DP matching to select the best instructions for a given L-expression in order to minimize the total cost of the L-expression. This specification can be omitted if the cost is 0.

<InstInfoCond> specifies the various conditions of instruction usage. The first specifies that two virtual registers must be allocated the same real registers. The second specifies a narrower set of virtual registers. The third specifies a special matching condition that is used when a pattern syntax either cannot be used or is very difficult to use as shown in the following. The following defrule is only applied for integer constants in the range 0 to 31.
```
(defrule con5 (INTCONST _ _)
  (cond (<= 0 (caddr $0) 31))
        (asm `(con ,(caddr $0))))
```
<InstInfoClobber> specifies the real registers that may be modified during instruction execution. An example of this specification is also found in a later section below.

4.4 Defcode Form

defcode forms are similar to defrule forms but they are used to describe entire instructions while defrule is used to describe addressing modes.
```
<DefCode> ::= (defcode <Name>
                       <Pattern>
                  {<InstInfo>})
<Name>    ::= <Symbol>
```
Unlike defrule forms, <Name> is not a non-terminal; it is just a name that is used to distinguish defcode forms. For example, in the following defcode form, the first non-terminal is reg and the pattern variables $0 and $1 correspond to reg and rc, respectively.
```
(defcode mov-I32 (SET I32 reg rc)
  (asm `(mov ,$1 ,$0))
       (cost 1))
```
Our system provides both the defrule and the defcode forms. Almost all retargetable code generators, including TMD and Burg-style generators such as Burg [7], Iburg [5], Lburg [6], and Jburg [8),9)], are based on the DP matching method. A defrule form in TMD directly corresponds to a rewriting rule in Burg-style generators. For example, consider the following defrule in TMD:
```
(defrule addr (ADD I32 reg reg)
  (asm `(add ,$1 ,$2)))
```
This is written in Burg-style generators as follows. The instruction costs are not essential to the following arguments and are omitted here.
```
addr : (ADD I32 reg reg) {...}
```
In Burg-style generators, both addressing modes and instructions are defined with this style of specification. It is also possible for TMD to specify both by using only defrule forms. However, we use defcode forms that are

Table 3 Examples of user defined functions.

Function name	Meaning
`tmd-restruct-prologue`	Restructures a prologue expression
`tmd-restruct-epilogue`	Restructures an epilogue expression
`tmd-restruct-call`	Restructures a call expression
`tmd-asmout-align`	Emits an alignment directive
`tmd-asmout-emit-data`	Emits data

specially designed to specify instructions. At the same time, we limit the usage of `defrule` forms to specifying addressing modes for reasons that are discussed below.

Consider the following rewriting rule:

```
reg : reg + reg {add $1, $2, $0}
```

This is a typical specification of an `add` instruction in Burg-style generators. An expression of the form `reg+reg` can be determined to be a `reg` type by executing the `add` instruction. This view seems natural, but the assignment operation that is done by the `add` instruction to store the result is implicitly handled. The pattern itself does not express exactly what the `add` instruction does. We think that this subtle mismatch in Burg-style generators often leads to lengthy and erroneous machine descriptions. The following is an example of a specification for the `add` instruction in a machine that has rich addressing modes, such as VAX.

```
mem : *reg
gen : reg
gen : mem
reg : gen + gen        {add $1,$2,$0}
stmt: gen = gen + gen {add $2,$3,$1}
```

The first three rules define the addressing mode `gen` consisting of registers or indirect registers. The last two rules define the same instruction `add` but from two different viewpoints. The first says that a sub-expression of the form `gen+gen` can be regarded as a register `reg` that holds an intermediate result. As Burg-style generators assume that any intermediate result is held in a register, this rule cannot cover an instruction's ability to store a result into memory. Hence, the second rule is needed. The second rule is applied only to the top level of an expression that stores the sum of two `gen`s into a memory location, `gen`. This problem may seem trivial, but machine descriptions tend to become larger by repeating similar definitions. Each instruction should be defined by just one specification.

There are two possible solutions to this problem:

(1) Generalize the notion of place to hold intermediate results.
(2) Generate two rules from one specification for each instruction.

We have chosen the second approach for its simplicity and efficiency. In TMD, `defrule` forms are used only to specify addressing modes and each instruction is defined using a `defcode` form as follows:

```
(defcode add (SET gen (ADD gen gen))
  (asm `(add $1, $2, $0)))
```

A `defcode` form by itself is not a rewriting rule, but it generates, at most, two rewriting rules. For each `defcode` form, the following conditions are first checked:

(1) The pattern is an assignment to a non-terminal.
(2) The non-terminal can be rewritten to `reg`.

Two `defrule` forms are then generated from a `defcode` that satisfy the conditions as follows:

```
(defrule reg (ADD gen gen)
  ;; rulename = add.reg
  (asm `(add ,$1 ,$2 ,$0)))

(defrule stmt (SET gen (ADD gen gen))
  ;; rulename = add.stmt
  (asm `(add ,$1 ,$2 ,$0)))
```

For debugging purposes, each `defrule` form is given a unique name that is included as a comment.

4.5 User Defined Functions

The remaining parts of machine descriptions are restructuring procedures and assembly language definitions. They are implemented in Scheme as user-defined functions. These parts cannot be formalized as simply as the process of instruction selection described above. Instead of seeking a formal realization for these parts, we have simply provided methods in the form of hook functions to customize these parts in the code generator. **Table 3** shows examples where functions prefixed with `tmd-restruct-` are used for restructuring and `tmd-asmout-` are used for describing a target machine's assembly code.

For example, the first three restructuring functions that the user must provide carry out calling convention transformations. **Fig. 6** is an example for a SPARC machine that takes

```
;; Before restructuring
(PROLOGUE (0 0) (REG I32 "x") (REG I32 "y"))
(CALL (STATIC I32 "foo") ((REG I32 "x") (REG I32 "y")) ((REG I32 "z")))
(EPILOGUE (0 0) (REG I32 "z"))

;; After restructuring of PROLOGUE, EPILOGUE, and CALL
(PROLOGUE (0 0) (REG I32 "%i0") (REG I32 "%i1"))
(SET I32 (REG I32 "x") (REG I32 "%i0"))
(SET I32 (REG I32 "y") (REG I32 "%i1"))
(SET I32 (REG I32 "%o0") (REG I32 "x"))
(SET I32 (REG I32 "%o1") (REG I32 "y"))
(CALL (STATIC I32 "foo") () ((REG I32 "%o0")))
(SET I32 (REG I32 "z") (REG I32 "%o0"))
(SET I32 (REG I32 "%i0") (REG I32 "z"))
(EPILOGUE (92 0) (REG I32 "%i0"))
```

Fig. 6 An Example of restructuring.

```
(define (tmd-asmout-emit-align sm n)
  ;; Emit alignment assembler directive.
  (fprintf sm "  .align  %d\n" n))

(define (tmd-asmout-emit-data sm type data)
  (case type
    ((I32) (fprintf sm "        .word   %s\n" data))
    ((I16) (fprintf sm "        .half   %s\n" data))
    ((I8) (fprintf sm " .byte   %s\n" data))
    ((F32)
     (fprintf sm "      .word   0x%08x ! %s\n" (float->bits data) data))
    ((F64)
     (let ((bits (double->bits data)))
       (fprintf sm "    .word   0x%08x ! %s\n"
                (logand (logshr bits 32) #xffffffff) data)
       (fprintf sm "    .word   0x%08x\n" (logand bits #xffffffff))))))
```

Fig. 7 Examples of assembler output functions in SPARC.

incoming arguments as real registers `%i0`, `%i1`, etc. Then, it returns the result as `%i0`. Finally, it sends outgoing arguments as `%o0`, `%o1`, etc. The calling convention of a SPARC machine becomes more complex when a call involves floating point arguments. Each machine has its own calling conventions that are hard to formalize in any general fashion. Therefore, our decision to use user-defined functions, a design decision that is common in other retargetable compilers, is a practical one.

Functions that are prefixed with `tmd-asmout-` are called from the code generator to produce the target machine's assembly code. Examples of these functions are shown in **Fig. 7**.

4.6 The Macro Feature

In a machine description, there is a tendency to repeat definitions that are very similar. These definitions are usually generated by using cut, paste, or modification operations in a text editor. To reduce this repetition, TMD provides the following simple macro feature. This example shows a macro form and its expanded form.

```
(foreach @x (a b)
  (foo @x))
=> (foo a) (foo b)

(foreach (@x @y) ((a 1) (b 2))
  (foo @x @y))
=> (foo a 1) (foo b 2)
```

5. Example of Instruction Selection

This section explains how `instsel` works by using a simple example. The machine description **Fig. 8** is part of the description for a SPARC machine. An older version of the SPARC machine does not have a multiply instruction. Instead, they are implemented as library function calls. The constraints specified in `cond` state that the result of an operation is stored in the first operand of the operation

```
(defrule reg (REG _ _) (asm $0))
(defrule addr (ADD I32 reg reg) (asm `(add ,$1 ,$2)))
(defrule con13 (INTCONST _ _)
  (cond (<= -4096 (caddr $0) 4095))
  (asm `(con ,(caddr $0))))
(defrule rc reg)  ; rc.1
(defrule rc con13); rc.2

(defcode mov-I32 (SET I32 reg rc)
  (asm `(mov ,$1 ,$0))
  (cost 1))

(defcode add (SET I32 reg (ADD I32 reg rc))
  (asm `(add ,$1 ,$2 ,$0))
  (cost 1))

(defcode lib-mul (SET I32 reg (MUL I32 reg reg))
  (cond (eq $0 $1)
        (regset ($0 *reg-o0-I32*) ; *reg-o0-I32* = (%o0)
                ($1 *reg-o0-I32*) ; *reg-o1-I32* = (%o1)
                ($2 *reg-o1-I32*)))
  (asm '(call (sta ".mul")) '(nop))
  (clobber (REG I32 "%o1")
           (REG I32 "%o2")
           (REG I32 "%o3")
           (REG I32 "%o4")
           (REG I32 "%o5"))
  (cost 10))
```

Fig. 8 Reduced description for SPARC.

```
SET I32
| * 13, stmt: (SET I32 reg (ADD I32 reg rc))  ; add.stmt
|
+--REG I32 "a"
|    * 0, reg: (REG _ _)  ; reg
|
+--ADD I32
   |
   +--MEM I32
   |  | * 1, reg: (MEM I32 addr)  ; load-I32.reg
   |  |
   |  +--ADD I32
   |     |  * 0, addr: (ADD I32 reg reg)  ; addr
   |     |
   |     +--REG I32 "b"
   |     |    * 0, reg: (REG _ _)  ; reg
   |     |
   |     +--REG I32 "c"
   |          * 0, reg: (REG _ _)  ; reg
   |
   +--MUL I32
      |   11, reg: (MUL I32 reg reg)  ; lib-mul.reg
      | * 11, rc: reg  ; rc.1
      |
      +--REG I32 "d"
      |    * 0, reg: (REG _ _)  ; reg
      |
      +--INTCONST I32 100
           * 1, reg: rc  ; mov-I32.reg
             0, rc: con13  ; rc.2
             0, con13: (INTCONST _ _)  ; con13
```

Fig. 9 DP matching result.

```
(SET I32 (REG I32 "V@1") (MEM I32 (ADD I32 (REG I32 "b") (REG I32 "c"))))
(SET I32 (REG I32 "V@2") (INTCONST I32 100))
(SET I32 (REG I32 "V@3") (MUL I32 (REG I32 "d") (REG I32 "V@2")))
(SET I32 (REG I32 "a") (ADD I32 (REG I32 "V@1") (REG I32 "V@3")))
```

Fig. 10 After tree expanding.

```
(SET I32 (REG I32 "V@4") (REG I32 "b"))
(SET I32 (REG I32 "V@5") (REG I32 "c"))
(SET I32 (REG I32 "V@1") (MEM I32 (ADD I32 (REG I32 "V@4") (REG I32 "V@5"))))
(SET I32 (REG I32 "V@2") (INTCONST I32 100))
(SET I32 (REG I32 "%o0") (REG I32 "d"))
(SET I32 (REG I32 "%o1") (REG I32 "V@2"))
(SET I32 (REG I32 "%o0") (MUL I32 (REG I32 "%o0") (REG I32 "%o1")))
(SET I32 (REG I32 "V@3") (REG I32 "%o0"))
(SET I32 (REG I32 "V@6") (ADD I32 (REG I32 "V@1") (REG I32 "V@3")))
(SET I32 (REG I32 "a") (REG I32 "V@6"))
```

Fig. 11 After relaxing.

and the operands of the operation are `%o0` and `%o1`. Each library function clobbers the listed registers in `(clobber ...)`.

The following is an input example for the instruction selector `instsel`:

```
;; register int a,b,c,d;
;; a = *(int*)(b+c) + d*100;
(SET I32 (REG I32 "a")
   (ADD I32
     (MEM I32
       (ADD I32
         (REG I32 "b")
         (REG I32 "c")))
     (MUL I32 (REG I32 "d")
              (INTCONST I32 100))))
```

Figure 9 shows the result of DP matching. Consult Iburg [5] for the details of Burg-style DP matching. Each node has the rules selected during the DP matching. For example, `MUL` (Line 21) has two rules (Lines 22, 23). Each rule consists of an optional star, the total costs, a rewriting rule, and a unique name (after a semicolon as it is a comment).

In general, each node does not correspond directly to an instruction. For example, the root node `SET` (Line 1) has the rule name `add.stmt` that is derived from `defrule` for `add`, and the node `ADD` (Line 7) is a part of the rule `add.stmt`. That is, the children of `SET` (Line 1) are `REG` (Line 4), `MEM` (Line 9), and `MUL` (Line 21). They correspond to the non-terminals `reg`, `reg`, and `rc` of the pattern shown in (Line 2), respectively.

`MUL` (Line 21) has two rules. The `lib-mul.reg` rule (Line 22) matches directly. The rule says that this node can be regarded as `reg`, but as explained before, this `MUL` node corresponds to the non-terminal `rc` from the pattern shown in (Line 2). This mismatch is solved by the second selected rule `rc.1` (Line 23). The rule to which a parent refers directly is marked by `'*'`. In this case, the rewriting rule `rc.1` can be applied without any additional cost.

On the other hand, the constant `INTCONST` (Line 28) requires additional cost to solve the mismatch of non-terminals. The `mov-I32.reg` rule is derived from `defcode` for `mov-I32`. Then it generates a `mov` instruction to regard the constant as a register.

After matching, this tree is first expanded into the sequence of instructions shown in **Fig. 10**. The order of the instructions is determined by the Sethi-Ullman numbering algorithm [12].

Secondly, the register constraints of the `defcode` forms are considered and the required transformations are applied. For example, register `b` and `c` are copied to fresh virtual registers so that they may be considered as candidates for coalescing. The `MUL` instruction (implemented as a library call) takes arguments from registers `%o0` and `%o1` and stores the result into `%o0`. To make it possible for the register allocator to solve these constraints, additional copies are needed to relax the constraints. We call this process relaxing.

The output of `instsel` is shown in **Fig. 11**. Each line of the sequence is an L-expression that corresponds to an existing instruction.

Even at this point, we never introduce any assembler notations that are dependent on the target machine. This allows the rest of the compiler passes, including register allocation, to be generic. The register allocator then tries to al-

locate real registers. `instsel` may be recalled when register spills arise.

Finally, the sequence of L-expressions is converted into real target machine assembly code. During this conversion, the hook functions in the target assembly code descriptions are used. Register allocation and coalescing are skipped in the following example.

```
mov     b,V@4
mov     c,V@5
ld      [V@4+V@5],V@1
mov     100,V@2
mov     d,%o0
mov     V@2,%o1
call    .mul
nop
mov     %o0,V@3
add     V@1,V@3,V@6
mov     V@6,a
```

6. Comparison with Other Systems

The idea to design an intermediate language of a compiler as a self-contained programming language is not new. The intermediate language that is based on the most similar concept to ours is `C--` [13], which is also a self-contained programming language designed as an intermediate language. However, `C--` was designed primarily to implement functional programming languages. As such, `C--` has some special features that are required for implementing functional programming languages, such as garbage collection [14] and exceptions [10]. These special features are not currently provided by LIR. `C--` provides several kinds of variables, including local, global, register, and aggregate, and their declarations are similar to those of the C programming language.

Compared to `C--`, LIR is closer to the conventional hardware and is more concrete. We have added only selected constructs to an assembler-level intermediate language. These constructs are essential to all back-end tasks including instruction selection and register allocation. Variable declarations in LIR include the details required for these tasks, such as frame offset, alignment value, and segment name. These details cannot be expressed in `C--`. LIR can be used as an intermediate language for instruction selection and register allocation as shown in our examples above.

We have previously pointed out a problem of Burg-style generators. Besides this problem, major Burg-style generators do not support the features that are necessary to specify various constraints, as the generators are designed to be independent of any intermediate languages. However, writing such constraints by hand separately from the corresponding instruction definitions is not easy and is often a very onerous task. The assumption that a fixed intermediate language exists can make a code generator more powerful. TMD is designed only for LIR. It provides the capability to specify various constraints as described above. TMD has also been incorporated with the Sethi-Ullman register minimizing algorithm [12].

Finally, Burg-style generators directly produce assembler codes by DP matching, while TMD transforms LIR programs by DP matching, whose results can be further transformed in later passes including register allocation.

The code generator that is used by GCC is designed for use with the RTL intermediate language. Machine descriptions in GCC do not have the problems inherent in Burg-style generators. However, features are provided that allow various constraints to be specified similar to TMD. Although the retargeting method used by GCC is a unique and flexible method, it cannot generate the best selection of instructions for each expression. As the method used by GCC is not based on the rewriting rules of Burg-style generators, the specification of addressing modes cannot be based on rewriting rules like the rules of Burg-style generators or the `defrule` of TMD. Instead, GCC provides a fixed but wide addressing mode. It can then be customized by using only a limited part of it. While at first glance this appears to be more awkward than necessary, this wide addressing mode includes a pre/post inc/decrement mode that is hard to implement otherwise. Therefore, GCC's approach is practical as far as the wide addressing mode covers the addressing modes of the target machine. Note that such special modes cannot be handled by Burg-style generators, including TMD.

The differences in design choices between LIR and RTL of GCC have already been discussed in detail in Section 2.1.

Table 4 summarizes the above comparison. The row 'One def for one inst' means whether one definition corresponds to one instruction in machine descriptions, and this property is not satisfied by Burg-style generators. The next row 'New addr modes' means whether new addressing modes can be easily defined. As we mentioned, the machine description language of

Table 4 Comparison.

	LBurg	GCC	TMD
DP matching	Yes	No	Yes
Sethi-Ullman	No	No	Yes
One def for one inst	No	Yes	Yes
New addr modes	Yes	No	Yes
Reg constraints	No	Yes	Yes
Complex addr modes	No	Yes	No

GCC does not provide an easy way to define new addressing modes, while in Burg-style generators, addressing modes can be easily defined by rewriting rules. The row 'Reg constraints' means whether constraints on registers can be specified in the machine description language. The final row 'Complex addr modes' denotes the feature of GCC that some intrinsic addressing modes are supported.

Machine descriptions for TMD were written for a SPARC machine and a (limited) X86 machine. The first version of our SPARC description was exactly based on that of LCC's retargetable code generator LBurg [6]. Thus it is worth comparing the total number of lines in their machine descriptions; the number of lines in LCC's description and that of ours are 1,163 and 877, respectively. Our machine description for the X86 machine excluding floating operations is 650 lines long, while the corresponding description in LCC is about 1,000 lines long. The machine description language of GCC is similar to ours except that the language does not support macro features because its underlying language is C. Thus we think that machine descriptions for TMD should be more concise than machine descriptions for GCC in general. However, the current machine descriptions for GCC cover a much wider range of SPARC and X86 machines, so it is impossible to directly compare the number of lines in those descriptions with ours or LCC's.

As we explained in Secion 2, another difference between the COINS compiler and GCC is that in COINS real registers do not appear in LIR programs before machine code generation. This convention has been widely preferred by those who implement code optimizations including SSA transformations [3], because they have much freedom in the use of registers, e.g., they can freely rename registers during transformations.

7. Performance of the COINS Compiler

The performance evaluation of object codes depends not only on the performance of TMD but also on other parts of COINS, such as the code optimizer and the register allocator. A portion of the results on the benchmark with Sun Microsystems Ultra5-10 workstation with a 300-MHz SPARC processor and 256-MB of memory is cited in **Table 5** to evaluate the current status of the COINS compiler where programs prefixed with digits are from the SPEC benchmark suite. It supports a relatively wide range of code optimizations based on SSA transformations [11], while some essential optimizations including pipeline scheduling and peephole optimizations are about to be implemented. The programs without a number prefix are not from the SPEC benchmark and the results are due to a version of COINS compiler with an experimental pipeline scheduler, which will soon be incorporated with the official version.

Table 5 Benchmark results.

PROGRAM	GCC-O2	coins-O2
164.gzip	2.74*	4.02
171.swim	2.67	2.50*
175.vpr	3.57*	5.82
181.mcf	0.741	0.738*
197.parser	5.12*	6.77
254.gap	2.07*	2.31
256.bzip2	9.99*	26.3
300.twolf	0.489*	0.622
isort	179.92	108.09*
ssort	184.55*	247.39
heap	68.81*	69.27
shell	63.57	63.47*
queen	69.46	67.91*
soukan	118.40	113.77*
komachi	27.90	27.52*
prime	160.37*	229.74

According to the above benchmark results, the object codes produced by TMD have an almost similar quality to those produced by GCC, though code optimizations in LIR are performed at a higher level compared with GCC. This means that the high-level features of LIR, such as L-functions and variable declarations, do not hinder the quality of the produced codes. Therefore, more sophisticated optimizations applied to LIR programs will outperform GCC in the future.

8. Conclusions

We have designed an intermediate language called LIR and have implemented the TMD retargetable code generator for this language. Machine descriptions for SPARC, x86, ARM, SH-4, PowerPC, and MIPS are currently avail-

able and the last three are done by students in a few month. Throughout the development of the back-end, LIR has been used without any major modification. The fact that LIR was designed as a programming language has provided a natural snapshot feature for the COINS compiler. An LIR program that is dumped at any point in the compilation process is very useful as a debug dump. Furthermore, it is also possible to restart the process at this dump point. Although the COINS compiler is implemented in JAVA, users can easily gain access to the compiler via any programming language. This is illustrated by the fact that TMD was originally implemented in Scheme.

TMD is a Burg-style generator with some additional features. One machine instruction can be naturally defined by a single `defcode` form, while addressing modes are defined by ordinary rewrite rules. This feature and the simple macro feature can reduce the size of machine descriptions. The embedded Scheme interpreter has proven to be helpful in generating concise descriptions of various conditions.

As a natural consequence that LIR is a programming language, the instruction selection by DP matching in TMD is also implemented as a kind of transformation of LIR programs.

In addition to TMD, various optimization techniques have already been implemented at the level of LIR programs. Those who implement such optimization techniques, including the third author, have reported that having a self-contained programming language makes implementation easier and clearer.

The elegance and quality of the DP matching method depends on the assumption that instruction selection is done for a single tree with at most one side effect. In fact, the method does not guarantee code generators based on the method to select optimal instructions for a whole program. It is difficult to select the best instructions for even a basic block; this problem is known to be NP-Complete. A few research-level retargetable code generators can select optimal instructions for expressions in a range beyond a single tree, e.g., Ertl[4]. But as far as we know, there are no such practical retargetable compilers. Any efforts towards such an ideal compiler are fruitful. A first step would be to cope with a pre/post inc/decrement mode.

Acknowledgments The authors would like to thank the members of the COINS project, in particular Dr.Nobuhisa Fujinami (SONY Corporation) for valuable discussions on the design of LIR and Kouichirou Mori (LSI Japan CO.,LTD.) for skillful re-implementation of compiler back-ends including the code generator described in this paper.

References

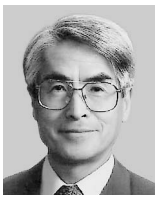
1) A compiler infrastructure project, http://www.coins-project.org/
2) Abe, S: A new model of pre-allocated registers and volatile variables, Master's Thesis, Department of Information Science, Graduate School of Science, the University of Tokyo (2002).
3) Cytron, R., et al.: Efficiently Computing Static Single Assignment From and the Control Dependence Graph, *TOPLAS*, Vol.13, No.4, pp.451–490 (1991).
4) Ertl, M.A.: Optimal Code Selection in DAGs, *26th POPL*, pp.242–249 (1999).
5) Fraser, C.W., Hanson, D.R. and Proebsting, T.A.: Engineering a Simple, Efficient Code Generator Generator, *ACM Letters on Programming Languages and Systems*, Vol.1, No.3, pp.213–226 (1992).
6) Fraser, C.W. and Hanson, D.R.: *A Retargetable C Compiler: Design and Implementation*, Benjamin/Cummings Publishing Co. (1995). (LBurg)
7) Fraser, C.W., Henry, R.R. and Proebsting, T.A.: BURG — Fast Optimal Instruction Selection and Tree Parsing, *SIGPLAN Notices*, Vol.27, No.4, pp.68–76 (1992).
8) http://jburg.sourceforge.net/
9) Kang, K.W.: A Study on Generating an Efficient Bottom-up Tree Rewrite Machine for JBurg, *LNCS*, Vol.3041 (2004).
10) Ramsey, N. and Jones, S.P.: A single Intermediate Language that Supports Multiple Implementations of Exceptions, *PLDI* (2000).
11) Sassa, M., Nakaya, T., Kohama, M., Fukuoka, T. and Takahashi, M.: Static Single Assignment Form in the COINS Compiler Infrastructure, SSGRR 2003w — International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, e-Medicine, and Mobile Technologies on the Internet, L'Aquila, Italy, No.54 (Jan. 2003).
12) Sethi, R. and Ullman, J.D.: The Generation of Optimal Code for Arithmetic Expressions, *J.ACM*, Vol.17, No.4 (1970).
13) Jones, S.P., et al.: `C--`: A Portable Assembly Language, *Implementing Functional Languages 1997*, LNCS (1998).
14) Jones, S.P., et al.: `C--`: A portable assembly language that supports garbage collection, *LNCS*, No.1702 (1999).
15) Stallman, R.M.: *Using and Porting GNU CC*,

Free Software Foundation (1999).
16) Yuasa, T.: A Lisp Driver to Be Embedded in Java Applications, *IPSJ Transactions on Programming*, Vol.44, No.SIG 4 (PRO 17), pp.1–16 (2003).

(Received February 21, 2005)
(Accepted June 27, 2005)
(Released October 26, 2005)
(Paper version of this article can be found in the IPSJ Transactions on Programming, Vol.46 No.SIG14(PRO27), pp.12–29.)

Seika Abe graduated in mathematics from Tokyo University of Science in 1984. After working for some companies, he received M.Sc. from the University of Tokyo in 2002. His current research interest is code generation with ILP. He is a member of COINS Compiler Infrastructure Project.

Masami Hagiya is a professor at Department of Computer Science, the University of Tokyo. After receiving M.Sc. from the University of Tokyo, he worked for Research Institute for Mathematical Sciences, Kyoto University, and received Dr.Sc. He has been working on modeling, formalization, simulation, and verification of computer systems, including various kinds of software and programming languages, mainly using deductive approaches. Recently, he is not only dealing with systems composed of electronic computers, but also biological and molecular systems with respect to their computational aspects, and is now working on DNA and molecular computing.



Ikuo Nakata received the M.S. degree in mathematics and the D.Sc. degree in information science from the University of Tokyo, in 1960, and 1977, respectively. He joined the Central Research Laboratory, Hitachi, Ltd. in 1960 and the Systems Development Laboratory, Hitachi, Ltd. in 1974. He had been a professor of University of Tsukuba from 1979 to 1997, a professor of University of Library and Information Science from 1997 to 2000, and a professor of Hosei University from 2000 to 2005. He is a director of the COINS Compiler Infrastructure Association. His current research interests include programming languages and language processors. He is an honorary member of the Information Processing Society of Japan.