The RTL System: A Framework for Code Optimization

Ralph E. Johnson
Carl McConnell
University of Illinois at Urbana-Champaign

Abstract

The construction of compiler front and back-ends is understood well enough for a great deal of the work to be automated. This paper describes the RTL System, which helps construct the rest of the compiler—the optimizer—by providing a flexible set of classes with a large number of predefined algorithms that the compiler writer can customize. The RTL System differs from systems to construct compiler front and back-ends because it does not specify the optimizations with a specialized language, but is instead an object-oriented framework. This paper describes the framework and how it can be used to build a code optimizer.

1 Introduction

Compiler front and back-ends are understood well enough for them to be generated automatically. A parser generator can create the parser from a grammar describing the language, and a code generator can be built automatically from a description of the machine. However, the construction of an optimizer is much less straightforward. This paper describes the RTL (Register Transfer Language) System, a framework for code optimization.

RTL is a register transfer language that compilers can use to represent programs during optimization. The RTL System is a set of classes for representing and manipulating programs, along with a large number of predefined algorithms that the compiler writer can customize. These algorithms include the common optimizations found in a typical compiler textbook [1][8], and it is easy to implement new optimizations. Thus, the RTL System is a toolkit for constructing code optimizers.

Such a toolkit is necessary because no code optimizer can be completely language- and machine-independent: when an optimizer is retargeted to a new language or machine, optimizations must nearly always be added, deleted, or modified to achieve the best results. For example, elimination of array bounds checks may be a worthwhile optimization for Pascal, but it is not for C; likewise, the criteria for when to perform constant propagation are different for a CISC architecture than for a RISC one. Such differences are best handled by providing a set of algorithms and components that can be "mixed and matched" to construct an optimizer for a given target. This is the philosophy behind the design of the RTL System.

The RTL System is not just a set of reusable software components, it is also a theory of how to represent and optimize programs. This theory is represented as a *framework* that is designed to be customized and extended [21]. An object-oriented framework is a reusable design for an application or a part of an application. It consists of a set of classes and a model of how instances of these classes interact.

The theory behind the RTL System is a combination of two different lines of work. The idea of using a register transfer language as an interme-

Authors' address: Department of Computer Science, 1304 W. Springfield, Urbana, IL 61801 USA. *Email*: `johnson@cs.uiuc.edu`, `mcconnel@cs.uiuc.edu`. *Phone*: (217) 244 0093

diate language in a compiler was borrowed from the work of Davidson and Fraser [5] [7] [6]. The RTL System also uses their algorithm for peephole optimization and code generation. The data dependence representation and the machine independent optimizations are based on SSA form [2][4][17]. Combining these two ideas was not trivial [15], but is not discussed here. This paper focusses on the object-oriented framework, not the details of the underlying theory.

The RTL System is part of TS, an optimizing compiler for Smalltalk. TS is one part of the Typed Smalltalk project [12]. The project has several components, such as developing a type system for Smalltalk [9] and redesigning some of the tools in the Smalltalk-80 programming environment to ensure that the compiler is compatible with the rapid-prototyping style of Smalltalk [22]. However, the largest component of the project is the TS compiler. The front-end of TS performs some optimizations, such as early-binding of message sends and in-line substitution, that are important because of peculiarities of Smalltalk. These optimizations convert Smalltalk programs into C-like programs, at which point conventional code optimization and generation is needed.

TS uses RTL not only as an intermediate form for programs, but also to define primitive Smalltalk methods (*i.e.*, procedures). After the TS front-end performs Smalltalk-specific optimizations on a method, it converts the entire method into RTL. Most optimizations are done by the RTL System, which also generates machine language. Thus, the RTL System also acts as a compiler's back-end, but this paper concentrates on its role as a code optimizer.

The next section describes RTL. Section 3 shows how RTL programs are represented by objects and how the different kinds of objects are described by several class hierarchies. Section 4 describes some of the algorithms in the RTL System. Section 5 shows how to customize the RTL System. Thus, these sections describe the framework for code optimization. Section 6 describes some lessons on framework design that we learned by developing the RTL System.

The GNU C compiler is a widely used compiler that also uses the Davidson and Fraser algorithms and represents programs with a register transfer language. It is easily ported to byte-addressable machines with 32 bit words. Although it was designed as part of the C compiler, the back-end has been converted to work with compiler front-ends for several languages. However, it was not designed to be reusable, and is a traditional C program that does not use object-oriented techniques.

Although they use different algorithms, the GNU back-end and the RTL System perform a similar set of optimizations. Thus, they represent two different approaches to implementing the same general design for the optimization and code generation phases of a compiler. This provides one of the few oportunities to compare similar programs written using object-oriented and traditional techniques. Section 7 compares the sizes of the comparable parts of the RTL System and the GNU back-end.

2 The Register Transfer Language RTL

The key lesson learned from the work of Davidson and Fraser was that a universal intermediate language for a compiler should be the *intersection* of machine languages, not the *union*. This implies that its operations should be so elementary that every machine has them, thus ensuring that every RTL program corresponds to some machine program. Unfortunately, this ideal is impossible to reach, since there are no operations that all machines use. An optimizer for a machine that lacks common operations like multiplication will have to transform missing operations into simpler ones, giving one more reason to customize the optimizer. In spite of this, we have found that a simple register transfer language makes an effective intermediate language for a compiler.

The sample program in Figure 1 illustrates the constructs available in RTL. RTL provides a typical set of numeric operations, as well as operations to read and write memory and to change the flow of control. RTL programs can use an

Compute the sum of 10 integers
starting at memory location 1000,
and put the result in memory location 500.

```
r1 ← 1000.       Initialize pointer to address 1000.
r2 ← 0.          Initialize sum to 0.
r3 ← 0.          Initialize counter to 0.

L1:
r3 >= 10 ↑ L2    If counter ≥ 10, jump to L2.
r4 ← *r1.        Fetch integer pointed to ...
r2 ← r2 + r4.    ... and add it to sum.
r1 ← r1 + 4.     Increment pointer.
r3 ← r3 + 1.     Increment counter.
↑ L1.            Jump to L1.

L2:
r5 ← 500.        Set pointer to 500 ...
*r5 ← r2.        ... and store sum there.
@.               Return.
```

Figure 1: An RTL program.

infinite number of logical registers. Logical registers can be preassigned to particular physical registers, enabling RTL to describe subroutine calling conventions.

The simplicity of RTL means that a relatively large quantity of RTL code is sometimes needed to represent high-level operations. However, this is exactly the idea: provide verbose but simple code to the RTL System, thus enabling it to uncover all possible optimizations.

The RTL System forms machine instructions by combining as many RTL statements together as it can; it never breaks them down into simpler ones. It works best when the RTL code it receives is as simple as possible. Computations that could be expressed using one RTL statement are instead broken up into several, with temporary registers holding intermediate results. A complex RTL statement may keep the RTL System from discovering an optimization. That is why the statements

```
r4 ← *r1.
r2 ← r2 + r4.
```

are used instead of

```
r2 ← r2 + *r1.
```

in Figure 1, even though the latter is equivalent and shorter.

One construct not shown in Figure 1 is the concurrency operator "`;`". For example,

```
r2 ← *(r1 - 4) ; r1 ← r1 - 4
```

expresses a pre-decrement memory access that might be written as

```
r2 = *--r1
```

in C. The concurrency operator is used to represent complex instructions and addressing modes that are discovered by the RTL System during optimization and instruction selection, and does not usually appear in the RTL System's input.

3 Program Representation

The RTL System represents RTL programs using objects of four kinds of classes. A **RTLProgram** contains the flow graph of a program. **RTLFlowNode** represents the nodes of the flow graph. **RTLTransfer** represents the register transfers in each flow node. **RTLExpression** represents the expressions in each register transfer.

Like most frameworks, the most important classes in the RTL System are abstract. **RTLFlowNode**, **RTLTransfer**, and **RTLExpression** are abstract, *i.e.* it is really their subclasses that are used to represent programs. **RTLProgram** is concrete, but it is just used to contain information global to the optimization process, and is never subclassed.

Figure 2 illustrates how a program is represented in terms of objects in these classes by showing the flow graph for the RTL program in Figure 1. It contains one instance of **RTLProgram** (representing the entire graph), four instances of subclasses of **RTLFlowNode**, nine instances of subclasses of **RTLTransfer**, and

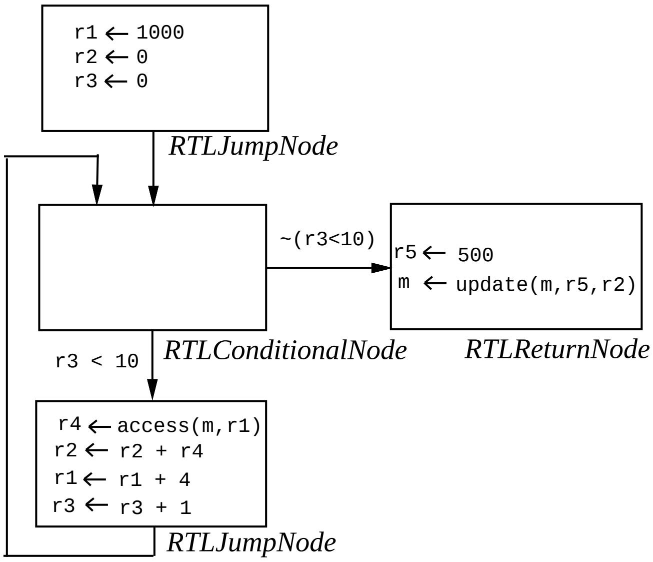

Figure 2: Flow graph for an RTL Program

over a dozen instances of subclasses of **RTLExpression**. Memory references are represented as functions taking an explicitly named memory object as an argument. This streamlines the treatment of memory operations, and is discussed in more detail later.

3.1 Static Single Assignment Form

Global optimizations require information about the global data flow properties of a program: Which transfers create the values used by a given transfer? This question is easy to answer if each register is assigned a value exactly once, and this is guaranteed by static single assignment (SSA) form, a program representation developed by researchers at IBM [4]. The RTL System uses SSA form to simplify algorithms for common subexpression elimination [2], code motion out of loops [17], and other optimizations.

It is simple to rename registers in a basic block so that there is only one assignment to (*i.e.*, definition of) each register. However, this isn't possible in the presence of conditionals or loops, since a given point in the program may be reached by conflicting definitions. SSA form avoids this problem by using an operator that merges (chooses) distinct registers defined on different paths. The first node reached by a set of conflicting definitions starts with a *ϕ-assignment* that merges them. As a consequence, each register is defined by no more than one register transfer.

A ϕ-assignment is an assignment whose right hand side is a ϕ function. Each operand of ϕ is a register, and there is one register for each predecessor of the flow node in which the ϕ-assignment appears. If control reaches the ϕ-assignment through the j-th predecessor, then the result of ϕ is the value of the j-th operand. Each execution of a ϕ-assignment uses only one of the operands, but which one depends on how control reached the flow node containing the ϕ-assignment.

SSA form makes computing data flow information easy. It also makes thinking about optimizations easy, since the case where different definitions reach the same point no longer occurs. Figure 3 shows how the flow graph in Figure 2 looks after SSA conversion. Note how conflicting definitions of loop variables are resolved by ϕ-assignments, which are shown as assignments with `phi(...)` on their right-hand sides in the figure. Also note the addition of *implicit assignments*, shown as assignments with `---` on their right-hand sides; these indicate that the destination was initialized outside the RTL program. Thus, every register is the destination of exactly one register transfer.

3.2 The Flow Graph Level

RTLProgram represents the control flow graph of a program. It acts as a container for the flow nodes, and for anything that is global with respect to the optimization process. For example, it contains an instance of **MachineDescription** that describes the target machine: all machine-dependent operations consult this object. **RTLProgram** also has methods for optimizing the RTL program, and for generating machine code, as well as for traversing the flow nodes in various ways.

RTLFlowNode and its subclasses represent vertices of flow graphs, and act as containers for register transfers. **RTLFlowNode**s also cap-

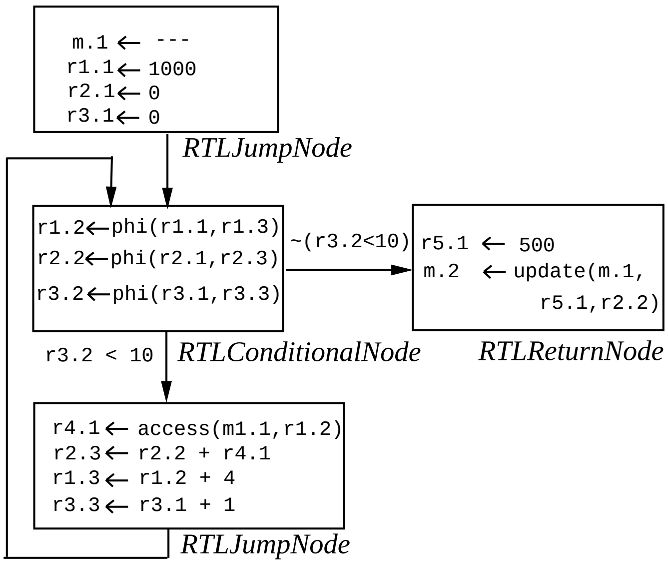

Figure 3: Flow graph for an RTL program after SSA conversion.

ture the structure of the flow graph by maintaining pointers to the nodes that precede and succeed them, as well as to their dominators and intervals. All flow-of-control optimizations, such as branch elimination, take place at the flow node level.

The **RTLFlowNode** protocol includes many of the same kind of optimization and code generation methods as **RTLProgram**; in fact, many **RTLProgram** methods do little more than delegate the message to the flow nodes. Since flow nodes are responsible for maintaining the structure of the flow graph, there are methods for changing a flow node's predecessors and successors.

Loops require special attention since a program spends most of its time executing them. In the RTL System, a loop is referred to as an *interval* [19]; a single-entry, strongly connected region of the flow graph. Sometimes it is convenient to treat an interval itself as a flow node. For this reason, an interval is represented by an instance of **RTLIntervalNode**, which is a kind of flow node that may act on behalf of the nodes inside it. Intervals can be nested, so **RTLIntervalNode** has a pointer to the surrounding interval.

Except for **RTLIntervalNode**, the subclasses of **RTLFlowNode** represent basic blocks. Thus, they are all subclasses of **RTLBasicBlock**, which is a subclass of **RTLFlowNode** that contains register transfers and is able to add, remove, and iterate over them.

The flow node hierarchy is

- **RTLFlowNode**
  - **RTLBasicBlock**
    - **RTLJumpNode**
      - **RTLCallNode**
      - **RTLConditionalNode**
    - **RTLReturnNode**
  - **RTLIntervalNode**

The reason for having a class hierarchy for basic blocks instead of a single class is that *the class of a flow node implicitly represents how control changes after the basic block in that flow node finishes executing.* In other words, rather than ending basic blocks with a register transfer indicating how control changes, the flow node itself indicates this. This principle has three corollaries:

- Basic blocks always end (by the flow node that contains them) with a change of control. They never just "fall through" to a successor.
- Basic blocks consist only of assignments. They never contain a register transfer representing a change of control.
- The targets of jumps are always flow nodes. Flow nodes play the role that labels do in conventional representations.

This scheme simplifies the implementation of the RTL System, since methods that depend on how control changes are implemented differently by each flow node class.

The subclasses play the following roles:

RTLJumpNode: basic blocks that end with a jump.

RTLCallNode: basic blocks that end with a call followed by a jump.

RTLConditionalNode: basic blocks that end with a conditional jump followed by a jump.

RTLReturnNode: basic blocks that end with a return.

3.3 The Register Transfer Level

RTLTransfer and its subclasses represent the register transfers that make up the program. They are components of basic blocks, and maintain the data flow information of the program. Each register transfer is able to find the register transfers that creates the values it uses and that use the values it creates.

A register transfer describes either a transfer of data—in other words, an assignment—or a transfer of control, such as a jump. However, as discussed in the previous section, control flow is represented at the flow node level. Control register transfers are not used during the optimization phases of the RTL System, but are only used during machine code generation.

The register transfer class hierarchy is

- **RTLTransfer**
  - **AbstractAssignment**
    - **ImplicitAssignment**
    - **Assignment**
    - **PhiAssignment**
  - **RegisterTransferSet**
  - **Call**
  - **Jump**
    - **CondJump**
  - **Return**

Some of these classes represent the transfer of data, and so are the main register transfer classes.

AbstractAssignment: the abstract superclass of all classes representing assignments to some storage.

ImplicitAssignment: represents assignments having an unknown source. These are used, for example, to define registers containing arguments.

Assignment: represents assignments having a destination and a source. These are the most common kind of **RTLTransfer**.

PhiAssignment: represents ϕ-assignments.

RegisterTransferSet: represents sets of concurrent **RTLTransfer**s.

The other register transfers represent the transfer of control. They are used primarily during machine code generation, where each register transfer produces machine code for itself.

Call: represents procedure calls.

Jump: represents unconditional jumps.

CondJump: represents conditional jumps.

Return: represents returns.

3.4 The Expression Level

Except for **RegisterTransferSet**, the components of a register transfer are expressions. Both the source and destination of a register transfer are instances of subclasses of **RTLExpression**. The expression hierarchy is

- **RTLExpression**
  - **Storage**
    - **Register**
      - **SpecialRegister**
    - **Memory**
  - **MemoryAccess**
  - **MemoryUpdate**
  - **BinaryExpression**
  - **Constant**
  - **UnaryExpression**

Storage is the abstract superclass of all classes representing expressions that can be the target of an assignment. **Register** is one subclass, with a subclass **SpecialRegister** that represents special registers such as the stack pointer. **Memory** is less naturally (at least at first glance) another subclass. However, as mentioned earlier, memory references are represented explicitly as functions in order to allow a unified treatment of operations on memory and operations on registers where dependency analysis and optimization are concerned. Instances of **Memory** serve as one argument to these functions. The functions themselves are

```
access(m, x)
```

which returns the item at address `x` in memory `m`, and is represented by **MemoryAccess**; and

```
update(m, x, v)
```

which returns a fresh **Memory** after storing value `v` at address `x` in memory `m`, and is represented by **MemoryUpdate**.

3.5 Creating an Expression

Since **RTLExpression**s are complicated objects, it is useful to have an object to build them. **RTLExpressionBuilder** plays this role. **RTLExpressionBuilder** makes testing the equality of **RTLExpression**s cheap by ensuring that only one version of each **RTLExpression** exists. Thus, two **RTLExpression**s are equal if and only if they are the same object. One consequence of this scheme is that an **RTLExpression** can never be altered, since changing one occurrence of an expression would change all of them. Therefore, the only way to update an expression is by creating a copy having the desired changes.

4 Optimizations

Some optimizations are performed by the objects that make up a RTL program, *i.e.* by the flow nodes and register transfers. Other optimizations are implemented by separate classes. Optimizations with their own classes are easier to customize and can encapsulate data local to the optimizations, but simple optimizations are better done by the objects that make up the RTL program. Also, some optimizations are naturally distributed, and it is easier to implement them as methods on flow nodes and register transfers.

Distributed optimizations help solve one of the main problems in in designing an optimizing compiler: deciding on the order in which to apply the optimizations. One optimization creates opportunities for another. The wrong order will miss optimizations, but there is often no ordering that is always best.

One way to keep from missing optimizations is to repeatedly retry optimizations until no further improvements can be made. However, this can be expensive and can even cause infinite loops if optimizations do not always shrink the program. Sometimes a single algorithm can carry out several optimizations, but no algorithm has been discovered that can perform all the optimizations that are needed.

The RTL System can sometimes solve the ordering problem by making each object responsible for optimizing itself. All the control flow optimizations are implemented this way, as well as dead assignment elimination. On the other hand, optimizations like register allocation and scheduling are better implemented as separate objects. It can be difficult to decide which implementation strategy is best for each optimization, and the strategy used for a particular optimizations often changes with experience.

4.1 Control Flow Optimizations

Optimizations that depend only on the control flow graph can be implemented entirely by the subclasses of **RTLFlowNode**. All of the control flow optimizations rely on the self-optimizing nature of the RTL program representation, and are performed whenever flow nodes lose predecessors or register transfers.

Each of these optimizations has two parts. The first part makes an invariant true, such as that there is no unreachable code. The second part maintains the invariant. It would be possible to require the front-end to take care of the first part by ensuring that these invariants are true from the start, but it is easier to check the invariants in the RTL System. Requiring as little as possible from the front-end also makes the RTL System more robust.

Eliminating Unreachable Code

Unreachable code is code to which there is no control path. An RTL program is effectively an acyclic graph, since the back edges of an interval are to a node inside an **IntervalNode**, not to the **IntervalNode** itself. Therefore, an unreachable flow node is one that is not the root node of the flow graph and that has no predecessors. Whenever a flow node is removed, it recursively removes any of its successors that consequently become unreachable.

Eliminating Jump Chains

A jump chain is a sequence of "jumps to jumps", and corresponds to an empty **RTLJumpNode**. Whenever a flow node removes its last register transfer, it eliminates itself.

Maximizing Basic Block Sizes

A pair of nodes is part of a larger basic block (and so should be merged) if the first has the second as its sole successor, and the second has the first as its sole predecessor. A node checks whenever its number of predecessors becomes 1 to see if it can merge with its predecessor.

4.2 Data Flow Optimizations

A data dependence exists between two register transfers if one writes to storage—either a register or memory—that the other uses. Traditional program representations have three kinds of data dependence [13]: flow dependence, anti dependence, and output dependence. By ensuring that a register is written by only one register transfer, SSA form has only one kind of data dependence, *flow dependence*. If transfer A defines storage that transfer B uses, then A is a *flow supporter* of B, and B is a *flow dependent* of A. This information is an integral part of SSA form; each register is responsible for knowing the register transfer that defines it and each register transfer is responsible for knowing the register transfers that use its result.

An RTL program is initially converted to SSA form using the algorithms in [4]. After SSA conversion, flow dependencies are easily computed in two steps. The first step associates each storage unit with the register transfer that defines it. The second step uses this information to link each transfer to its flow supporters by examining the storage used by the transfer. Once an RTL program is in SSA form and dependencies have been computed, the RTL System must maintain the dependencies. We do not have general incremental algorithms for updating a program in SSA form, but all the SSA-based optimizations maintain SSA form.

Not all storage can be modeled by registers in SSA form. Memory (*i.e.* arrays and records accessed by pointers) and registers that are preassigned to physical registers also have *anti dependencies*. Anti dependencies are not stored explicitly, but instead are computed whenever they are needed from the flow dependencies.

Propagating Copies

Copy propagation replaces the destination of a register-to-register assignment with the source everywhere it appears in the program. A transfer `r2` $\leftarrow$ `r1` finds all register transfers that use `r2` and tells them to replace `r2` with `r1`.

Eliminating Dead Assignments

Dead assignments are assignments whose destination is never used; in other words, assignments with no flow dependents. Thus, dependence information makes it easy to remove dead assignments. Register transfers with no dependents automatically remove themselves from the program.

Eliminating Common Subexpressions

Common subexpression elimination replaces expressions by equivalent but simpler expressions. The RTL System uses the algorithm described in [2], which detects when computations produce equivalent values. The algorithm is conservative in that any expressions found to be equivalent are in fact equivalent, but not all equivalences are found. The SSA representation makes the algorithm fast, simple, and powerful. The algorithm is complex enough that it is implemented in its own class.

Code Motion

Code motion moves invariant code out of intervals, where it will be executed less often; it also moves redundant code from different branches of a conditional to a common ancestor, thus reducing program size. The RTL System uses the algorithm described in [17], which globally recognizes invariant and redundant expressions, even those that are lexically different.

Eliminating Constant Conditionals

A constant conditional is a conditional jump in which the condition can be evaluated at compile time, thus making the jump unnecessary. The RTL System can eliminate conditional jumps even when the value of the condition is not constant in the flow node containing it. Sometimes the condition is constant in the preceding flow node, so duplicating the conditional node for each preceding flow node converts the conditional node into a set of jump nodes. These jump nodes usually consist only of dead assignments and so are eliminated. This particular optimization is language-dependent: it is useful for Smalltalk because of the way the TS front end works, but it would be less useful in a language like C.

4.3 Peephole and Scheduling Optimizations

An RTL program is a directed graph that describes control and data dependencies. The RTL System scheduler imposes a linear order on the register transfers in the graph. The goal is to find an order for the transfers that executes in as few time units as possible. The greedy "combining" algorithm described next is the one used for CISC architectures such as the 68020. It schedules sets of transfers to execute at a single time unit. The RTL System has a different scheduler for RISC architectures such as the SPARC, which orders register transfers to minimize pipeline delays.

The RTL code given to the RTL System is very simple, so machine instructions such as those with complex addressing modes are represented by sequences of register transfers. The scheduling phase for a CISC machine finds sets of register transfers corresponding to these complex machine operations. It does this by combining register transfers to form new transfers, and checking to see if these new transfers correspond to machine instructions. As a consequence, the scheduling phase performs peephole optimization.

The scheduling phase uses a variation of an algorithm for peephole optimization and code generation invented by Davidson and Fraser [5] [6] [7]. This algorithm was first used in a code generator called PO. PO has many phases; the *combiner* is the one that was borrowed for the RTL System. It was used in the RTL System because it seemed simple and easy to implement, easy to retarget to new machines, and the original papers showed it gave good results.

The RTL System's *scheduler* is similar in spirit to PO's combiner, but quite different in design: PO uses parsing algorithms to match combinations of register transfers, but the RTL System uses table look-up, which is much faster at the cost of space.

Both PO and the RTL System have an "instruction recognizer" that tells whether a register transfer represents a legal machine instruction. (The **MachineDescription** for a given target machine provides the instruction recognizer for it.) PO used strings to represent register transfers and a parser to recognize instructions. Register transfers in the RTL System are objects. There is a table mapping register transfers to corresponding instructions, and instructions are found by looking a given register transfer up in the table. Although the table is large (270K for the 68020), the combiner takes 50% of the execution time of PO while scheduling takes 9% of the execution time of the RTL System.

As in PO, the scheduling phase scans the program once, combining each transfer with members of its *window*—flow supporters, and possibly their flow dependents—and replacing the original transfers with the result if it corresponds to a machine instruction. For example, combining the third register transfer with the second (its flow supporter) in the RTL fragment

```
r2 ← r1 + 8.
r4 ← r3 + r2.
r5 ← *r4.
```

would yield `r5 ← *(r3 + r2)`, which does not correspond to a 68020 instruction. However, combining this new transfer with the first transfer in the fragment (one of the flow supporters of the new transfer) yields `r5 ← *(r1 + r2 + 8)`, which does correspond to an instruction. The RTL System would then replace the third transfer above with this new transfer, and delete the other two.

PO also forms instructions using a window. The difference is that SSA form automatically determines the window for the RTL System, while PO has to explicitly manage the window, which we found to be complex and error-prone. Thus, SSA form made RTL scheduling simpler and more reliable.

4.4 Register Assignment

Register assignment is based on the classical graph coloring algorithm [3]. Constructing the graph to be colored is complicated by SSA form.

SSA form breaks the lifetime of each physical register into many pieces, each represented by a logical register. Each assignment of a register becomes a new logical register and each place where logical registers have their values merged also becomes a logical register. The first task of register assignment is to merge these register lifetimes together again. However, optimizations performed since putting a program in SSA form might prevent register lifetimes from being merged.

Another way of looking at the problem is to note that the least expensive way to implement a ϕ-assignment `r3` $\leftarrow$ ϕ`(r1,r2)` is to assign `r1`, `r2`, and `r3` to the same physical register. However, this works only if the lifetimes of `r1` and `r2` don't extend beyond the ϕ-assignment. Otherwise, copy instructions must be introduced.

Moreover, the scheduling phase places constraints on register assignment. Many machine instructions require that their destination be the same register as one of their sources. Some machines have several kinds of registers, and each instruction must use the right kind. Thus, each instruction includes a *constraint* object that describes the constraint that it places on the registers it uses. The assigning phase checks each constraint to see whether it can be satisfied by merging register lifetimes. Otherwise, it must satisfy the constraints by introducing instructions to copy values from one register to another.

Register assignment can be customized in several ways. It is simple to replace the register assigner, and we have experimented with several versions[18]. The register assigner is an object, and there is a class hierarchy of the various register assignment algorithms that we have implemented. It is also easy to invent new kinds of constraints and to associate them with machine instructions so that scheduling will place constraints on register assignment correctly. Adding new kinds of constraints might require changing the register assigner.

Several parts of register assignment are machine-dependent. The number and type of registers is machine-dependent, and the way that registers are spilled is not only machine-dependent but run-time system dependent. Thus, parts of register assignment are delegated to the machine description, which can provide the list of available registers, and which is able to spill registers.

5 Customizing the RTL System

There are two ways to customize the RTL System. One way is to extend the RTL language that is used to represent programs. This is done by creating subclasses of the classes that represent RTL programs, namely **RTLFlowNode**, **RTLTransfer**, and **RTLExpression**. The other way is to modify or create algorithms that manipulate RTL programs. Algorithms implemented as single objects are usually easier to understand and modify than algorithms implemented as a collection of methods in various classes, but they are somewhat harder to create.

A good example of extending RTL by creating a new kind of register transfer or expression is the way that support for debugging was added to TS. The debugger required the compiler to produce functions that recovered the unoptimized values of variables at particular points in the program. Each point in the program at which the debugger might be invoked is an "inspection point" and is labeled with a new kind of register transfer, a **DebuggingUseMarker**. Each **DebuggingUseMarker** keeps track of the variables for which it computes a recovery function. As the various optimizations transform and manipulate the register transfers, the **DebuggingUseMarker** updates its recovery functions.

As might be expected, debugging support required more changes to the RTL System than simply adding a new class. Although debugging support required little change to most of the RTL System, some of the algorithms that manipulated register transfers were too specialized for **Assignment** and so had to be generalized to work with **DebuggingUseMarker**. However, these changes usually improved the clarity of the RTL System and made it easier to change in the future. Thus, the changes were essentially fixing "reusability bugs" in the RTL System.

The second way to customize the RTL System is to add new algorithms to it or to change the algorithms that it uses. Many of the optimizations are represented by objects, so a new algorithm is often a new class definition. The compiler can control the optimizations that will be performed by specifying the class of each optimization. The instance of **MachineDescription** keeps track of the classes used for each optimization. Thus, each machine can use a slightly different set of optimizations. This will probably be changed in the future, since the set of optimizations that should be used also depends on the source language, not just the target machine.

A good example of customizing the RTL System by modifying an algorithm expressed as an object is the scheduler. RISC machines may not need the peephole optimizations provided by the combining algorithm, but will instead need to fill branch delay slots or minimize pipeline delays. Representing the scheduler as an object means that it is easy to make slight variations using inheritance. Data structures that are not needed by other algorithms can be encapsulated in the new class, and the machine description can determine which scheduling algorithm is used by creating a scheduler of the appropriate class.

The machine description is involved in almost all customizations of the RTL System. It is responsible for mapping register transfers to their equivalent machine instructions, so it determines which table to use. It creates objects that perform register allocation, scheduling, and other optimizations. Thus, a new use of the RTL System often requires a new subclass of **Machine Description**. Fortunately, machine descriptions are not large and are always simple. For example, the largest subclass of **MachineDescription** that we have written, which is for the Motorola 68020, is 242 lines long.

6 Lessons Learned

The RTL System is not just a set of reusable software components, it is also a theory of how to represent and optimize programs. Probably any system for code reuse is also a theory of its application domain, so this should not have been surprising. Every time we discovered a weakness in our theory, we have had to revise the software. A brief history of the RTL System illustrates this.

The first version of the RTL System was written in 1986-1987 [20]. It was designed to be an object-oriented version of PO, so its design was as much like that of PO as possible. The only initial difference was that it recognized legal instructions by looking register transfers up in a hash table instead of by matching them with a parser, as in PO.

The first version of the RTL System differed greatly from the current one. RTL programs were represented by several classes in addition to the ones that the RTL System now uses. **RTLFlowNode** had no subclasses, in large part because the control flow graph was unimportant and only used by a few optimizations. An **RTLProgram** was a stream of register transfers, not a directed graph. Data dependencies were represented by def-use chains, and sets of register transfers were not considered a register transfer, so the **RTLTransfer** class hierarchy was different. None of the optimizations were implemented as separate objects, but as collections of methods in the classes representing RTL programs. Of the RTL classes that are now important, only **RTLExpression** has been relatively unchanged.

The first changes were primarily to correct errors and to make the RTL System easier to understand. The first major change was to make the flow graph the basis of the RTL program. This led to a class hierarchy for **RTLFlowNodes** that is similar to the current one. The RTL System worked well enough to compile some benchmarks for TS[12], but the optimizations were unreliable and hard to understand. It was clear that a less ad-hoc basis for code optimization was needed.

We read the papers on SSA form in mid-1988 and were convinced that they would provide a sound theoretical basis for code optimization. The RTL System was converted to use SSA form in 1989 [11][14]. A major change that occurred just prior to this (and partly in preparation for it) was to merge several classes used to represent RTL programs into the **RTLTransfer** hierarchy. A class that contained data-flow information was merged into **RTLTransfer**, and the list of register transfers that it contained was replaced by adding **RegisterTransferSet** to represent a group of simultaneous transfers. Also, the first instance of representing an optimization by an object occurred when the first version of the graph coloring based register allocator was built.

An important way of improving a framework is to use it in a way that was not originally foreseen. There were two such uses of the RTL System during this time. The first was by Javalina [10], a system for converting abstract specifications of digital filters to microcode for digital signal processors. This use showed a weakness in the way machine-dependent information was represented, and resulted in the design of **MachineDescription**. The second was in research on debugging optimized code [22]. This use verified the flexibility of the RTL System, and did not result in any major changes to it.

To test the representation of machine-dependent information, we wrote code generators for the National 32032 and the SPARC [16]. This led to making the scheduler a component, and to a class hierarchy for schedulers. This work showed a problem with memory references, which was subsequently fixed, but otherwise the basic representation of programs held up well.

The RTL System has constantly been revised to make it easier to understand and to change. However, the biggest changes have been caused by changes to the underlying theory. This includes the shift from viewing a program as a stream of instructions to viewing it as a directed graph, generalizing the notion of a register transfer to include a set of transfers, and SSA form. Changes such as converting optimization algorithms into separate classes are much easier to implement, and have little effect on the rest of the RTL System.

Uses of the RTL System usually point out ways to improve it, and we expect it to continue to improve as we use it in new applications. In addition to porting it to new architectures and adding new optimizations, we are also planning to use it with new source languages. In particular, we plan to use the RTL System as the back-end of a C compiler. These uses will help us refine and improve the RTL System.

7 Comparisons

The RTL System is one of the few Smalltalk systems that is directly comparable to a program written in C. The back-end of the GNU C compiler is quite similar to the RTL System. It does not use SSA form, but it does use a register transfer language and the Davidson/Fraser combining algorithm for code selection and peephole optimization. They implement nearly the same set of optimizations, though often with different al-

	classes	methods	lines
RTLProgram	1	73	799
Flow nodes	8	411	4451
Reg. transfers	12	447	3791
Expressions	23	432	3440
Reg. alloc.	15	151	2089
Scheduling	5	51	939
C.S.E.	9	45	462
Machine dep.	10	104	1245
Support	10	139	1286
Total	93	1753	18502

Figure 4: RTL System program statistics

gorithms. Thus, it is useful to compare them.

The RTL System is about 18,500 lines of commented Smalltalk code (counted using the Unix utility `wc`). Figure 4 gives the sizes of the pieces of the RTL System at the beginning of 1991. The first four categories are the classes described in Section 3, so over 65% of the total is for the basic representation of programs. In comparison, the back end of the GNU C compiler (version 1.37.1) is about 42,600 lines of C code, of which about 11,000 lines of code are for the basic representation, which is only 25% of the total.

One of the reasons that the classes representing RTL programs are so large is that many optimizations in the RTL System are implemented by methods in the flow node and register transfer classes. Another reason is that factoring out common operations like graph traversal makes the optimization classes smaller and the program representation classes larger. Thus, register allocation takes 3511 lines in the GNU compiler compared with 2089 lines for the RTL System (which contains a choice of 4 algorithms), common subexpression elimination takes 3920 lines instead of 462, and combining takes 2796 lines instead of 939 for scheduling of all kinds. Part of this size difference is because the classes that represent RTL programs provide much of the code for the optimizations. As we restructure the RTL System to turn optimizations into components, the classes that represent programs will get smaller.

It is obvious from the figures that the RTL System is much more compact than the GNU back-end. The GNU back-end is very well written and is compact for a C program. Both are heavily commented. Thus, the size differences are significant.

There are at least three possible explanations for the smaller size of the RTL System: compact optimization algorithms resulting from the SSA form, code reuse by inheritance and polymorphism, and non-object-oriented programming features of Smalltalk such as garbage collection and the ability to define control structures. In our experience, SSA form makes optimizations smaller, but not the program representation classes that make up the bulk of the RTL System. Most of the difference is caused by the implementation languages and not the features of the respective systems. One way to distinguish the effects of object-oriented features from the effects of the other features of Smalltalk is to make a version of the GNU back-end or the RTL System in a language like C++. The authors of the GNU compiler are reported to be doing this, and it will be interesting to compare the resulting system with ours.

There are other important metrics besides code size, but they are harder to measure. We believe that the quality of the generated code will be similar, though we will not be able to compare the relative speed and size of the output until the two code generators share a source language. This is the main motivation for building a C front-end for the RTL System. A C front-end will make other attributes easier to compare, such as speed of compilation. These other measurements might show ways that the GNU back-end is superior to the RTL System. For the moment, code size is the only metric by which we can compare the two systems, and RTL System is less than half the size of the GNU back-end.

8 Conclusion

The RTL System shows that object-oriented frameworks are valuable for application domains other than user interfaces. Code optimization is an example of an application domain that depends on complex algorithms. The RTL System provides ways to customize both the data structures that represent programs and the algorithms that manipulate these data structures. There are several families of classes, each family having a standard interface. Each family provides a different way to customize the optimizations, and the standard interfaces make it easy to combine new features.

The RTL System is relatively easy to understand and modify, and hence can significantly reduce the effort required to create new optimizations. This is not just the effect of implementing it in an object-oriented language; reusable software requires a theory upon which to base the system design. SSA form and the use of register transfers to describe both the program and the target architecture have been the foundations of the RTL System. The role of object-oriented programming is to express this theory in a natural and simple way, and the RTL System shows that this can be done much more concisely in a language like Smalltalk than in a language like C.

Acknowledgements

We thank Mike Lake and Gary Leavens for their comments on earlier versions of this paper.

This research was supported by the National Science Foundation under grant CCR-8715752 and by a gift from Tektronix.

References

[1] Alfred C. Aho, Ravi Seth, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, 1988.

[3] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, 1981.

[4] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, 1989.

[5] Jack W. Davidson. *Simplifying Code Generation Through Peephole Optimizations.* PhD thesis, University of Arizona, 1981.

[6] Jack W. Davidson and Christopher W. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505–526, 1984.

[7] Jack W. Davidson and Christopher W. Fraser. The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2):191–202, April 1980.

[8] Charles N. Fischer and Richard J. LeBlanc. *Crafting a Compiler.* Benjamin-Cummings, 1988.

[9] Justin Graver and Ralph Johnson. A type system for Smalltalk. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 136–150, 1990.

[10] Kurt J. Hebel. *An Environment for the Development of Digital Processing Software.* PhD thesis, University of Illinois at Urbana-Champaign, 1989.

[11] Richard Heintz. *Low Level Optimizations for an Object-Oriented Language.* Master's thesis, University of Illinois at Urbana-Champaign, 1990.

[12] Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. TS: An optimizing compiler for Smalltalk. In *Proceedings of*

OOPSLA '88, pages 18–26, November 1988. printed as SIGPLAN Notices, 23(11).

[13] D. J. Kuck. *The Structure of Computers and Computations.* John Wiley and Sons, 1978.

[14] Carl McConnell. *An Object-Oriented Code Optimizer Using Static Single Assignment Form.* Master's thesis, University of Illinois at Urbana-Champaign, 1989.

[15] Carl McConnell and Ralph E. Johnson. SSA form, dependencies, and peephole optimization. 1991. University of Illinois at Urbana-Champaign.

[16] J. David Roberts. *A Highly Portable Code Generator.* Master's thesis, University of Illinois at Urbana/Champaign, 1990.

[17] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 12–27, 1988.

[18] Charles B. Schoening. *A Family of Register Allocation Algorithms.* Master's thesis, University of Illinois at Urbana-Champaign, 1991. forthcoming.

[19] R. E. Tarjan. Testing flow graph reducibility. *J. Comp. Sys. Sci.*, 9:355–365, 1974.

[20] John David Wiegand. *An Object-oriented Code Optimizer and Generator.* Master's thesis, University of Illinois, Urbana-Champaign, 1987.

[21] Rebecca J. Wirfs-Brock and Ralph E. Johnson. Surveying current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, 1990.

[22] Lawrence W. Zurawski. *Source-Level Debugging of Globally Optimized Code with Expected Behavior.* PhD thesis, University of Illinois at Urbana-Champaign, 1990.