

1998

Specification-Driven Optimization

Sheena Day
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds



Part of the [Computer Sciences Commons](#)

Let us know how access to this document benefits you.

Recommended Citation

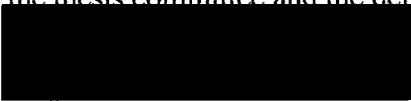
Day, Sheena, "Specification-Driven Optimization" (1998). *Dissertations and Theses*. Paper 6333.
<https://doi.org/10.15760/etd.8187>

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.


THESIS APPROVAL

The abstract and thesis of Sheena Day for the Master of Science in Computer Science were presented May 8, 1998, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS:


Andrew Tolmach, Chair


Jingke Li


Mike Driscoll
Representative of the Office of Graduate Studies

DEPARTMENT APPROVAL:


Richard Hamlet, Chair
Department of Computer Science

ABSTRACT

An abstract of the thesis of Sheena Day for the Master of Science in Computer Science presented May 8, 1998.

Title: Specification-Driven Optimization.

Traditionally, optimizing transformations have been built into compilers. The end-user has little or no control over guiding any optimizations that may be applied by the compiler. Moreover, the compiler-writer does not have a simple way to direct the optimizations. Thus, many potentially beneficial opportunities for code optimization may be lost. We have built a system that allows the user to participate in guiding source-to-source transformations via the specification of rewrite rules. A clean separation of the rules from the strategy of applying them makes the system easier to use and modify, compared to other integrated systems. This is especially relevant to the application-specific improvement of code, which is hard to achieve through the usual means. We anticipate our system to be a useful aid to both the end-user and the compiler-writer. A sophisticated end-user might use it very effectively by applying his knowledge of the problem domain to the rewrite rules. The compiler-writer, in addition, would benefit from the increased modularity, flexibility and simplicity of use that it provides. A desirable feature of the design is the ability to express the rewrite

rules in a language closely allied to the source language. We illustrate this with the help of several examples. Our system is intended to be an extension to the standard optimizing compiler. To our knowledge, existing compilers do not have this facility.

SPECIFICATION-DRIVEN OPTIMIZATION

by

SHEENA DAY

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Portland State University
1998

for
Mummy and Papa

Acknowledgments

I feel very privileged to have an advisor like Andrew Tolmach. It is hard to imagine how any student could have a more sincere or committed mentor. He has enriched not merely this thesis, but my entire graduate life with his knowledge and experience. I shall always remember his generosity and the sense of humor that have made the past few years a time to look back upon with warmth and pleasure.

I would also like to thank the committee members Jingke Li and Mike Driscoll for their advice and criticism of the thesis.

Eelco Visser has helped greatly by his thoughtful and detailed suggestions on how to improve the presentation of the material, as well as by referring me to some of the books I consulted during the time I was writing this thesis.

Finally, I cannot forget the support and encouragement of my husband Bikram, without whom so much might never have been.

TABLE OF CONTENTS

1. INTRODUCTION.....	1
2. OPTIMIZATION BY REWRITING	6
2.1 CONSTANT FOLDING.....	10
3. SYSTEM ARCHITECTURE.....	14
3.1 COMPILER ARCHITECTURE	14
3.1.1 <i>Generation of the optimizer</i>	14
3.1.2 <i>Compilation of the source program</i>	18
3.2 SYSTEM COMPONENTS.....	19
3.2.1 <i>RML Source Language</i>	19
3.2.2 <i>Templates</i>	20
3.2.3 <i>Rule Expression Language</i>	23
4. EXAMPLES.....	28
4.1 STRENGTH REDUCTION.....	28
4.2 MATRIX ADDRESSING.....	29
4.3 VECTOR LOOP FUSION	33
5. IMPLEMENTATION DETAILS.....	42
5.1 EXTENSIONS NEEDED TO THE RML COMPILER.....	42
5.2 GENERATION OF THE OPTIMIZER FROM USER SPECIFICATIONS	43
5.3 OPTIMIZATION VIA PATTERN-MATCHING	45
5.3.1 <i>Termination</i>	47
5.3.2 <i>Confluence</i>	48
5.4 OPTIMIZATION OF RML ABSTRACT CODE	48
5.5 TESTING.....	49
6. DISCUSSION	51
6.1 RELATED WORK	51
6.2 OUR APPROACH	53
6.3 FUTURE WORK	56
7. CONCLUSION.....	58
8. REFERENCES.....	59

LIST OF FIGURES

Figure 1. Template definition for optimizing operations on integers	12
Figure 2. Overall architecture of the RML compiler	15
Figure 3. Generation of the extended RML compiler	17
Figure 4. A template definition for the rule specification language.	25
Figure 5. An RML library for matrix manipulation.....	31
Figure 6. A template definition of primitives for vectors of integers.	32
Figure 7. An RML implementation of the conjugate gradient method.....	36
Figure 8. A library of primitive operations for matrix and vector manipulation.	38
Figure 9. An optimized version of the conjugate gradient method.....	41
Figure 10. Speedup obtained as a result of optimizing matrix address arithmetic.	50

1. Introduction

Optimizing compilers have become an integral part of modern-day computing. The fundamental role of a compiler is to translate source programs written in a higher-level language into machine language. An *optimizing* compiler, in addition, attempts to analyze the program and apply various transformations to it in order to create a more efficient target program, with a smaller size or running time. Thus, it enables the programmer to be less concerned with the underlying architectural details and to write higher-level, more intuitive and problem-oriented programming constructs.

A program can be optimized at several levels. These can range from improving the source code to optimizing it at the machine level. As compilation proceeds, newer opportunities for code optimization may be discovered as a result of previous transformation phases. It is then the responsibility of the compiler to detect and implement these optimizations. Typically, the programmer can only intervene at the source level. However, it is not always possible to perform certain types of transformations at the source-code level. For example, high-level languages often insulate the programmer from the details of memory management. A language like C may allow a programmer to manipulate pointers directly, but this is not the case with other languages. Moreover, even if it is possible for a programmer to improve the

source code, it may not be desirable, since it is an error-prone task and may result in an unclear source program.

Therefore a good optimizing compiler can make a significant difference to the overall program development time. Compilers often apply transformations to an intermediate language between the source and target language. Among the types of optimizations that may be applied here are “partial-evaluation” style optimizations like constant-folding or function inlining.

Partial evaluation is the general technique of performing part of a computation at compile time, producing a simplified program. In constant-folding, computations on constants are performed at compile time by replacing the value of an expression with the constant to which it evaluates. Inlining consists of replacing a function call by the body of the function, substituting each formal parameter with its value at the point of the call. Traditionally, constant-folding and other optimizing transformations have been hard-coded into compilers. This is unfortunate because the programmer has no control over the optimizations and the conditions under which they are enabled and thus cannot take advantage of specialized knowledge of any source language constructs. Moreover, the compiler writer has no foreknowledge of them since they may vary from one application to another. Therefore many opportunities for optimizing these kinds of application-specific constructs may be lost. Arguably, a

programmer should not have to be concerned with the internal implementation details of the compiler. However, having a knowledgeable programmer direct the compiler at a suitably high level can make a significant difference to the efficiency of the resulting code.

We use rewrite rules to specify optimizations to the program. We have provided a means to specify rewrite rules for the source language within a module that is separate from the main compiler. We distinguish between the specification for a problem and its implementation. A good specification abstracts away from the details of the underlying system architecture and enables one to define the problem in a language closer to the application domain. The rewrite rules are specified in a language similar to the application-development language. Moreover, they are specified in a manner independent of the mechanism by which they are integrated with other compiler transformations and applied to the source program.

The system we have developed enables both the end-user and the compiler-writer to participate in describing the kinds of transformations that may be applied to the source program. Our approach is conceived of as an extension to the traditional optimizing compiler. User-described transformations can be applied along with those already in place in the compiler itself. Designing a compiler in this modular fashion has several advantages. It is easier to write, debug, test, maintain and extend. The additional

transformations applied to the source code may lead to a significant improvement in runtime code efficiency without sacrificing the generality of the core compiler. The compiler can be built with optimizations in-place for operators that are common across all applications. The more general-purpose optimizations are thus hard-coded beforehand by the compiler writer. The more specialized ones can be linked in as and when necessary. An example is the application of constant-folding directives to primitives that are specific to an application and thus cannot be optimized by the core compiler. See §4.2 with reference to optimizing address arithmetic in matrix computations.

A facility for extending the core compiler gives us a way to specify and apply optimizing directives that are only marginally relevant in most situations and may therefore be impossible to predict. Via this system, the optimizations can be applied in an incremental fashion, without requiring the compiler writer to foresee all possible opportunities for optimization at once. Separate specification modules can be developed for each application domain and integrated with the core compiler as required. A module created for one application may be shared by another, reducing development time.

In addition, user-specified rewriting rules can supplement standard optimizations already applied by the compiler. For example, most optimizing compilers perform

some variety of function inlining. Inlining eliminates the expense of a procedure call and may uncover more opportunities for program simplification. However, the conditions under which this is attempted may be very restricted because aggressive inlining can lead to code blow-up. A conservative compiler may perform inlining solely in situations where the function is applied once. Therefore, in special cases where performance is critical, it might be a good idea to have a system that allows the user to guide the compiler to inline specific functions, ignoring its normal criteria for doing it. Our system is not suitable for achieving this directly in all situations. However, if the source code of a function is available, the user can replace each call by the function body via appropriate rewrite rules. He may also be able to rewrite the definitions of arbitrary functions, which conform to some known format, to include inlining directives. The latter are normally optional to a function definition.

We would like to perform optimizations early on in the compilation process without sacrificing retargetability. The results of many important optimizations like constant-folding may be platform-dependent. However, it is often desirable to apply the optimization in the early stages of the computation to take advantage of the other simplification opportunities that result. This indicates that it is preferable to factor out this transformation into a separate module, rather than to have it as an intrinsic part of the core compiler. Such a facility is simple to use and can be a very valuable aid to cross-platform compilation.

However, the modular approach is not without its drawbacks. The compile-time overhead due to the translation of the rules, their integration with the main compiler and the additional simplification passes that are needed for their effective application, can result in an increase in compilation time. This is the price we pay for being able to have increased flexibility within the framework of a general-purpose compiler. Hard-coded optimizations, though typically much more efficient, can only be incorporated to a limited degree within such a compiler.

The remainder of this thesis will discuss these and other issues involved in the development of the system, in greater detail, and explore the limitations of the implementation. §2 describes the fundamental basis of our optimization strategy and gives a motivating example. §3 outlines the layout of the compiler and the languages involved in specifying the optimizations. §4 presents a detailed discussion of how to optimize a domain-specific problem. The implementation details and results are discussed in §5. §6 presents related work, the unique features of our system and future enhancements to it. Finally §7 concludes our discussion.

2. Optimization by rewriting

We use term reduction systems for specifying optimizations to the program. Reduction systems are a class of formal computational structures [5]. Each such system consists

of a set of possible computation states and a relation that determines the transitions from one state to the next. A term rewriting or term reduction system is a reduction system where the states of computation are represented by terms. In our case, the terms describe program text.

The optimization of a source program by rewriting its abstract syntax tree is carried out by specifying directives to guide the rewrite process and then using them to implement the optimizations. The designer of the optimizer uses the rewrite system to specify one or more rules for optimizing source programs. These rules are generally specified just once for an entire class of applications to which they may be relevant. The actual process of rewriting the program involves several phases. The source program is parsed to generate an abstract syntax tree. A redex is any point in the tree that can be rewritten according to some rule. The rewrite system examines this abstract syntax tree via pattern-matching, to find possible redexes. Once a reduction point is discovered, the system uses some heuristics to select a matching rule, since it is possible that more than one such match exists. It then rewrites the redex as indicated by the rule and repeats this process as often as necessary until it can no longer be transformed according to any existing rules. After the rewriting phase is completed, the rewritten abstract syntax tree is compiled further to generate the target language.

Designing a rewrite system involves a variety of decision-making steps:

- (i) Choosing an efficient internal representation of the source program.
- (ii) Designing a language in which to express the rewrite rules.
- (iii) Constructing an algorithm to detect redexes by pattern-matching on the abstract syntax tree of the program.
- (iv) Selecting a strategy to traverse the program tree and apply the pattern-matching algorithm to it.
- (v) Inventing an algorithm for selecting the specific rule to apply at each step. This may be trivial in the ordinary case when only one possible rule matches the redex, but would need a predefined strategy when more than one possibility for rewriting exists.

Our design has been dictated by the underlying system and source language. We have implemented our rewrite system as an extension to the front-end of the RML (Restricted ML) compiler [12], which compiles concrete RML source code to C or Ada 83 target code. The compiler is written in SML/NJ (Standard ML of New Jersey) and runs under the SML/NJ system. RML is very similar to Revised SML (SML '97) [10]. We describe it in more detail in §3.2.1. We have solved the issues outlined above in various ways:

- (i) The internal representation of the program is important because reduction leads to repeated structural changes to the original expression tree. The source program is represented internally in RML abstract syntax, which is produced

from parsing the source concrete RML program. This abstract syntax is transformed through the application of the rewrite rules and other simplification processes.

- (ii) The choice of a language for specifying the rules has been motivated by the source-program language and the implementation language of the compiler. The rules are expressed in a form of RML concrete expression syntax interspersed with SML syntax.
- (iii) We have not written a specialized pattern-matching algorithm to detect redexes. Instead, we have borrowed the pattern-matching power of SML (Standard ML). The rules are compiled to produce an SML function that does the actual work of pattern-matching and transforming the expression.
- (iv) The implementation of the rewriting strategy is separate from that of the pattern-matching module. The SML rewrite function mentioned in step (iii) is applied to the nodes of the abstract syntax tree of the program in a bottom-up manner. For each expression node, the children are examined first for possible redexes and rewritten if such an opportunity exists, before any attempt is made to rewrite the parent node.
- (v) For selecting the next rule to apply at each reduction point, our approach is straightforward. If more than one rule exists for potentially rewriting a particular redex, we choose to apply the first one specified.

We use rewrite rules as the basis of our optimization strategy because they are simple to understand and specify. They make it easier to reason about the optimizations being carried out by making them more evident. However, we do not worry about standard issues related to rewrite systems. Some key issues for formal term reduction systems are to prove the termination and confluence properties. However, like many such systems, termination for our rewrite system is in general undecidable. Also, we do not have the notion of a unique normal form. These issues are discussed in more detail in §5.3.

2.1 Constant Folding

We begin by presenting a simple example to give a flavor of our rule specification language and how to express constant-folding through it. Constant folding can be classified as a “shrinking” optimization as it is guaranteed to make the program smaller. Moreover, folding of constants, followed by their propagation, can uncover more opportunities for program simplification. This can result in the evaluation of a significant portion of the program at compile time. Our rules define a natural and easily comprehensible way of specifying how to perform this optimization.

Figure 1 is a template definition of some common primitive operations on integers and the rules that may be used to perform constant-folding on them. A template defines the interface between the RML and 3GL (third-generation language) components of a

program. Templates are described in detail in §3.2.2. Primitive operations are defined in terms of the target language, i.e., the language to which the source (RML) code is compiled. The template library of primitives is a fixed set available for the user of our system and can be invoked from within the rules. The rules actually describe how to optimize the operations involving these primitives. We will ignore the details for now and try to explain the general nature of the specification.

The first part of the template defines primitive types and their target language representation. Notice that the type *integer* (line 2) is a primitive type, defined to be represented in C by the type *int*. For each operation, the template provides information about the name of the function to be used in the source RML program, the number of arguments, the types of the argument and return values and the target language definition of the function. Thus, for example, *+* is a primitive function that takes two arguments of type *integer* and returns a result of type *integer*. Similarly, *~* or unary minus, is another primitive function that takes a single argument of type *integer* and returns a result of type *integer*. In the interest of simplicity, Figure 1 omits the target language definitions (macros) of any of the primitive functions.

```

1  template IntegerCfoldTemplate
2  type integer "int"

3  (* some basic operations on integers *)

4  val + (x0:integer, x1:integer) : (res:integer)   "... "
5  val - (x0:integer, x1:integer) : (res:integer)   "... "
6  val * (x0:integer, x1:integer) : (res:integer)   "... "

7  (* unary minus *)
8  val ~ (x0:integer) : (res:integer)               "... "

9  rules

10 ~0          => 0
11 | ~(~(%a))  => %a
12 | ~(%a)    %% (isIntegerLit a) %% => %( let val i = SmlIntegerLit a
13                                     in mkIntegerLit (~i) end %)

14 | %a + 0    => %a
15 | 0 + %a    => %a
16 | %a + %b  %% (isIntegerLit a andalso isIntegerLit b) %%      =>
17                                     %( let val i = SmlIntegerLit a
18                                       val j = SmlIntegerLit b
19                                       in mkIntegerLit (i+j) end %)

20 | %a * 0    => 0
21 | 0 * %a    => 0
22 | %a * 1    => %a
23 | 1 * %a    => %a
24 | %a * 2    => %a + %a
25 | 2 * %a    => %a + %a
26 | %a * %b  %% (isIntegerLit a andalso isIntegerLit b) %%      =>
27                                     %( let val i = SmlIntegerLit a
28                                       val j = SmlIntegerLit b
29                                       in mkIntegerLit (i*j) end %)

30 | %a - 0    => %a
31 | 0 - %a    => ~(%a)
32 | %a - %b  %% (a = b) %% => 0
33 | %a - %b  %% (isIntegerLit a andalso isIntegerLit b) %%      =>
34                                     %( let val i = SmlIntegerLit a
35                                       val j = SmlIntegerLit b
36                                       in mkIntegerLit (i-j) end %)

```

Figure 1 Template definition for optimizing operations on integers

The second part of the template specifies rewrite rules involving the primitives. Essentially, each rule consists of a left-hand side, an \Rightarrow and a right-hand side. The left-hand side specifies a pattern that may occur in an un-optimized program and the right-hand side describes how to rewrite it. For example, the rule on line 10 simply rewrites ~ 0 to 0 .

Next, consider the rules that describe constant-folding operations on $+$. Obviously, this set can be extended to incorporate more of the standard constant-folding operations. The rules on lines 14 and 15 rewrite an expression consisting of a $+$ operation on two integers, to one of the integers, if the other integer happens to be 0 . The only thing here that is different from the first rule we mentioned, is the $\%$ which indicates a meta-pattern. The meta-pattern $\%a$ will match any argument in a call to the $+$ function.

We claim that in the typical case, our language is quite straightforward to use, especially for someone who is familiar with the source language RML. There may be cases, as the other rules in Figure 1 and in the later examples demonstrate, when the specifications become complicated by the embedding of arbitrary SML expressions. For example, the rule on lines 16 to 19 checks to see whether both the arguments to the $+$ function are literals. If they are, it proceeds to perform constant-folding on them. This evaluation involves expressing the computation process in SML. Finally, the result of the addition operation is converted back into RML. Thus the power of the

design lies in its expressiveness. However, we do pay a price for allowing such expressiveness. To explore the full functionality of the rule language, the author of the rules would need to be familiar with both RML and SML.

3. System Architecture

3.1 Compiler Architecture

We present a diagram of the various parts of the RML compiler in Figure 2 to demonstrate how our rewrite system interfaces with the rest of the compiler. The architecture of the compiler resembles a pipeline operating on a series of typed intermediate representations. Each step in the transformation preserves both semantics and types. A detailed description of the RML compiler can be found in [12].

There are two distinct phases in compiling an RML concrete source text - generation of the optimizer and compilation of the source with the optimizer.

3.1.1 Generation of the optimizer

The optimizer is generated once for each set of rewrite rules. It is then integrated with the main compiler. After that, it can be used as many times as required, to compile a source. The same optimizer can be used to compile many different source programs. A set of rules consists of all the rules that are specified in a template definition

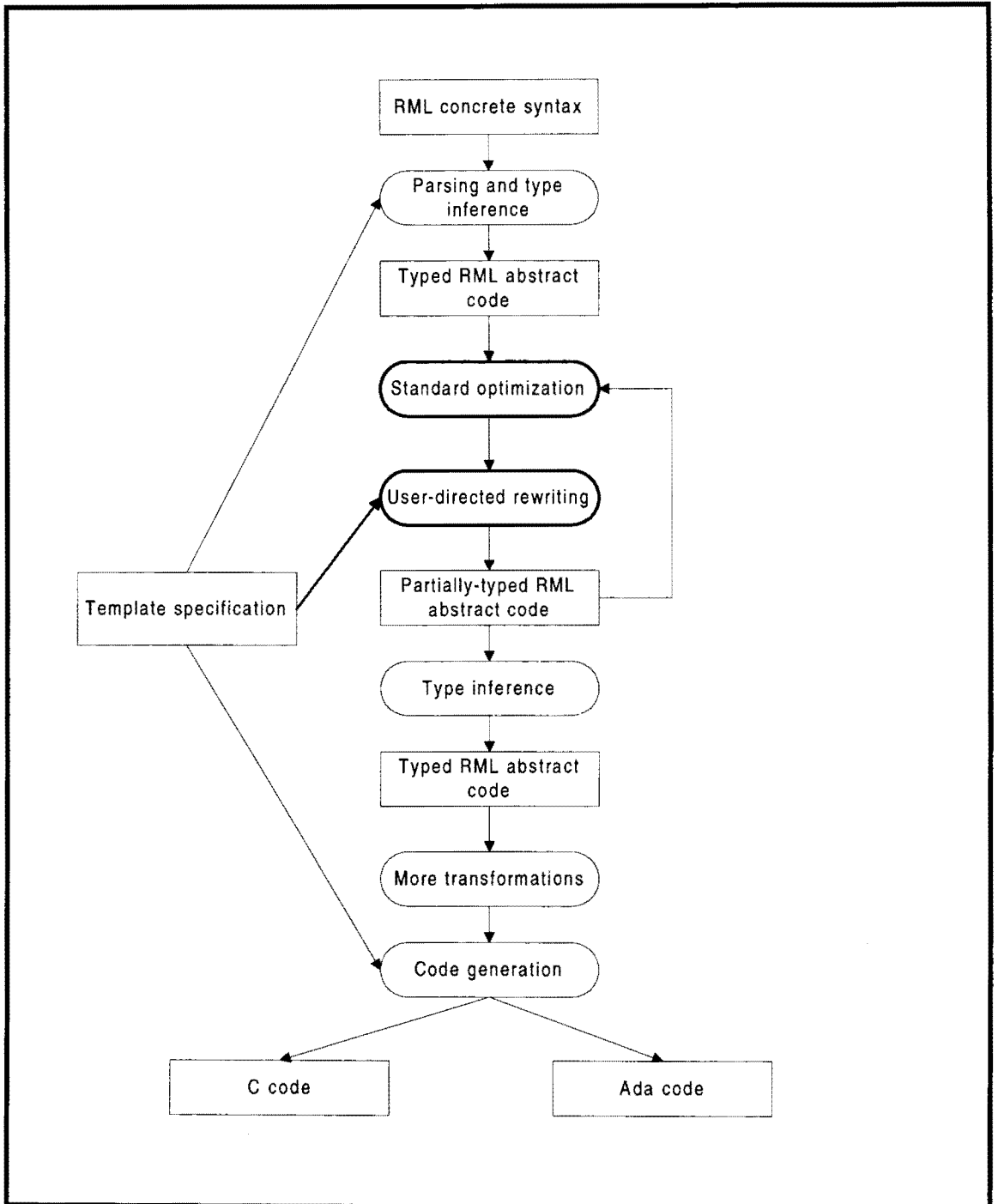


Figure 2 Overall architecture of the RML compiler

(§3.2.2). There are no semantic restrictions imposed on the transformations the rules may specify. The rules are typically specified just once for one or more related application areas that may use the optimizations defined by them. They are then used to generate the optimizer which is used hereafter exactly like a regular compiler. The optimizer needs to be recompiled only if the set of rewrite rules changes. The change could be due to the addition of new rules to the original set, or abandoning the old set in favor of an entirely new set. However, the generation of the optimizer is rarely repeated, as the role of this process is comparable in the traditional case to writing a new compiler. Of course, it is much easier and faster to use our system to generate an optimizer than to write an entire compiler by hand. We provide a diagram of the generation process in Figure 3. The optimizer must be produced by compiling the template (§3.2.2) before the execution of the pipeline described below in Figure 2. The template is the unit of the compiler where the rules are described. The generated optimizer is compiled and linked in with the rest of the compiler, and with a library of SML functions that is provided for use in the rewrite rules. We discuss this SML library later (§5.1) in detail. This compilation and linking process produces the specialized compiler which is used to compile the source in the usual manner as described later.

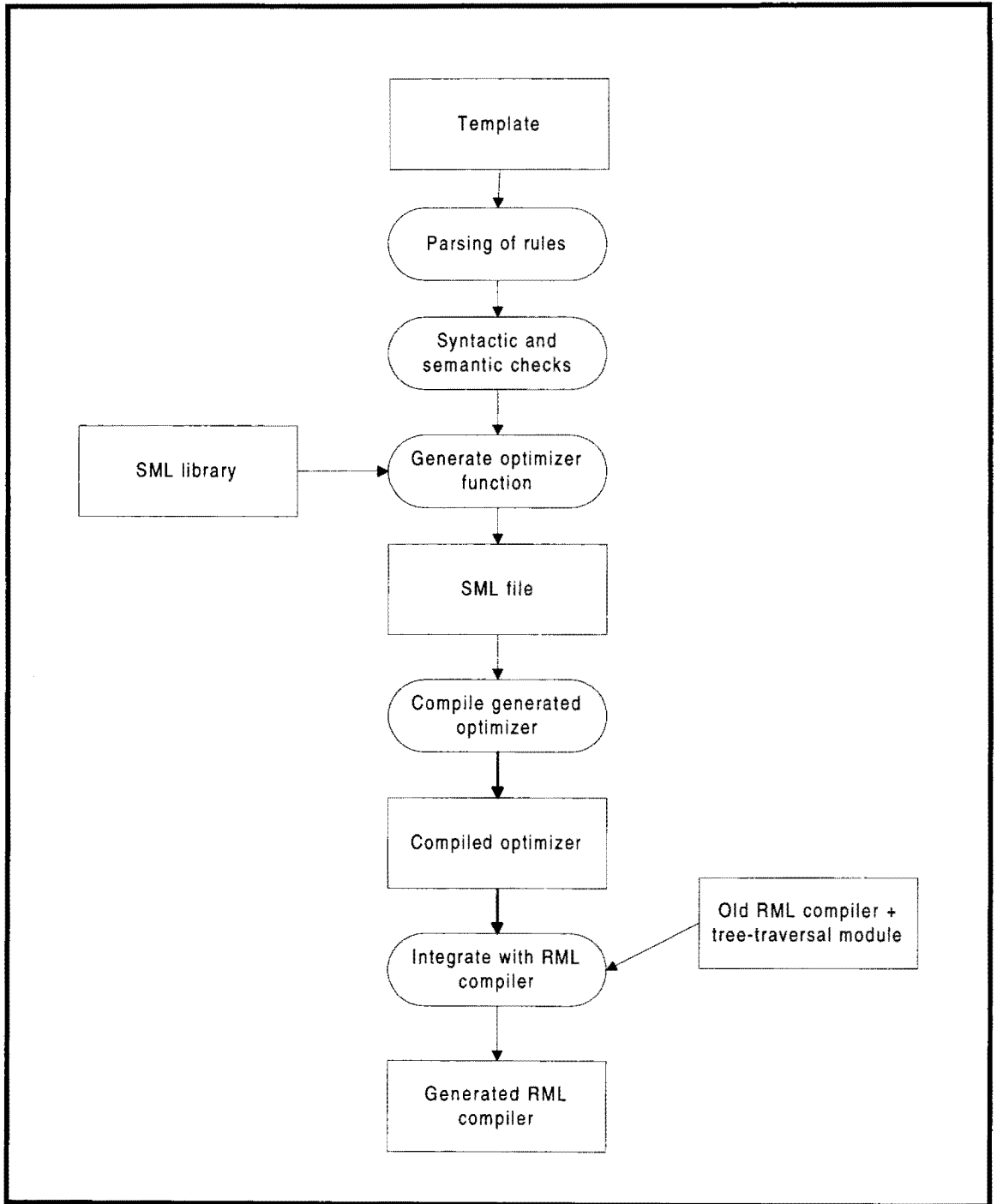


Figure 3 Generation of the extended RML compiler

3.1.2 Compilation of the source program

We have added steps (iii) and (iv), described below, to the original version of the RML compiler [12]. The main steps in the compilation process are:

- (i) RML code is parsed from a concrete text representation or loaded from a binary representation produced by a separate generator tool. The concrete representation is transformed to RML abstract syntax. Subsequent steps operate on this abstract form.
- (ii) The RML abstract code is annotated with type information using conventional Hindley-Milner type inference [9].
- (iii) The RML code is optimized by the repeated application of partial-evaluation style optimizations. These include inlining of functions used only once or marked for inlining (conservative function inlining), case-of-constructor simplification, propagation of variables and constants and elimination of dead bindings (dead code elimination).
- (iv) Each round of step (iii) is followed by a round of optimization driven by user specification. This is the place where our rewrite-specification system integrates with the core compiler. Each round of applying the simplifier in step (iii) alternates with step (iv), thereby possibly uncovering further opportunities for rewriting in either step.

- (v) The partial evaluation style optimizations are followed by another round of type-inferencing. This is essential because the rewrites may lead to a partially untyped program.
- (vi) The annotated code undergoes further transformations and optimizations before it is finally translated into C or Ada83 using the template macros.

3.2 System Components

3.2.1 RML Source Language

RML is very similar to the pure subset of core Revised SML (SML '97), if primitives are not considered. It is an eager, higher-order language with algebraic datatypes, true multi-argument functions and data constructors and parametric (Hindley-Milner) polymorphism. Unlike SML, it does not support nested patterns or many derived forms. Also, there are no records or tuples, but these can be built as datatypes with a single constructor. Datatypes can be marked as *flat* to indicate that they should not be heap-allocated. Primitives may have side-effects and can be used to implement I/O operations, arrays or mutable references. All functions are passed parameters by value.

The primary difference between concrete and abstract RML syntax is that the former is not annotated with types. The type-annotated 'abstract form is obtained from the concrete by standard Hindley-Milner type inference.

Each source program consists of a single RML component. It is translated *with respect* to a template to generate some 3GL code. The RML component may be formally defined as an environment mapping 3GL names to RML types and values. Each component has an export clause that describes the types and values to be exported for use by the 3GL components of the program and 3GL names for them. The main program or driver for an executable is always written in the target 3GL and invokes RML code via the exported functions defined in the RML component.

3.2.2 Templates

A template provides the interface between the RML and 3GL components of a program. Every RML component is translated with respect to a specialized template. Of course, several components may share the same template. Templates usually contain definitions relevant to a particular application. For example, one could imagine the template definition in Figure 1 as a subset of a more elaborate definition to specify a library of primitive operations on integers and some rewrite rules to guide constant-folding on integers. Some of the definitions were part of the pre-existing RML core system [12], whereas we have introduced the others to support the extended system. The latter are described below in (iv) and (v).

- (i) Abstract and primitive types, values and operators whose representation and implementation are specified in terms of the target 3GL. The template specifies which of these are to be visible to RML code. This information is essential for

the parsing and type checking of RML components. The operators are implemented via macro definitions in 3GL code fragments. Primitive types include both general-purpose (e.g., *integer*, *real*, *string*, etc.) and application-specific types (e.g., *vector*, *matrix*). They must be monomorphic. Their definition supplies information about the RML name for the type (e.g., *integer*) and the corresponding 3GL type name, built-in or user-defined, that provides its concrete realization (e.g., *int*). Primitive value declarations include the RML type of the value and the corresponding 3GL macro expansion string. Operation definitions specify formal names and types for the arguments (e.g., *x0* of type *integer*) and result (e.g., *res* of type *integer*) and a macro expansion string.

- (ii) Algebraic datatype declarations (e.g., *bool*), monomorphic instances of which may appear in the type signatures of primitive operators. They may also be used in the corresponding RML template. Monomorphic datatype declarations may include 3GL strings for type names (e.g., "*int*") and constructors (e.g., "*1*" and "*0*"). These, if present, will then be used in the generated 3GL code.
- (iii) Header declarations of 3GL library (Ada package or C file) names (e.g., *matrix_prims.h*) that are referenced by the 3GL code fragments mentioned before. This is done to bring the names defined in the library into the scope of the 3GL code that uses them.

(iv) Rules to specify transformations that may be applied to the intermediate representations to optimize and produce more efficient code. The rules described in the template are written in a concrete RML-like expression language with embedded SML meta-expressions. Therefore, the user merely needs to be familiar with the RML concrete expression language and basic SML, in order to be able to describe the optimizations. Thus the source language for rule specification and program coding is very similar. Some instances of rule patterns have already been presented in Figure 1 and we will discuss them in more detail later. These rule patterns are aggregated together to form the optimizer. The rules may involve the invocation of various SML auxiliary functions, some user-defined and some general-purpose library routines. The user-defined functions are specified in the template. The SML library routines are provided separately and linked in automatically during the compilation process. We have not presented their actual definition, though library routines (e.g., *SmlIntegerLit()*) have been mentioned in the right-hand sides of rules in examples like that described in Figure 1. We discuss the nature of this library later (§5.1). Now, the SML expressions embedded in the concrete RML must evaluate to RML abstract syntax. They can do so either by explicitly invoking library functions like *mkIntegerLit()* to construct the RML abstract expressions, or by having embedded RML concrete expressions inside them. The latter are translated to the abstract form during parsing of the rules.

- (v) User-defined SML functions that may be invoked from within the rules. There may be any number of these functions defined in the template. The reason that we have a separate section for the user-defined functions is that they may be invoked from more than one rule. The only place where these functions are visible is from within the rules. The examples presented in this thesis do not make any use of these functions.

We would like to emphasize the fact that in order to use our system, the only definitions that the user needs to specify are the rules and possibly the SML functions described above in (iv) and (v). Our system provides everything else necessary to aid in the specification as well as the implementation of the optimizations.

3.2.3 Rule Expression Language

The syntax of rule expressions is described below. A precise grammar follows in Figure 4. Essentially, a rule expression is a pair of concrete RML expressions and an optional conditional expression. The only difference is that a rule expression may contain meta-constructs prefixed by `%` or enclosed within `%(` and `%)`.

Rule ::= rml_pattern % % cond_exp % % => rml_action


```

1 Rule Specification Syntax :
2 (template)      templ ::= template name
3 (header declaration)  [header "string "]
4 (primitive types)    [type K ( size ) " string "]
5 (algebraic types)    [datatype [(t {, t)]] D
6                      [" string "][flat] = rator { | rator } }
7 (primitive values)   [primdef]
8 (user-specified rules) [rules rule { | rule}]
9 (user-specified functions) [functions %% string %%]

10 (user specified rule) rule ::= exp [cond_exp] => exp

11 (SML boolean expr) cond_exp ::= %% c_exp
12                      c_exp ::= string %%
13                      c_exp ::= string rml_exp c_exp

14 (embedded RML expr) rml_exp ::= `exp `

15 (RML concrete expr) exp ::= exp1
16                      exp ::= fn inlflag mpvnl => exp
17                      exp ::= case exp of calt { | calt }
18                      exp ::= exp1 :: exp

19                      exp1 ::= [exp1] exp2
20                      exp1 ::= exp1 ( [(exp ,) exp , exp] )
21                      exp1 ::= v ((exp ,) exp , exp) exp2

22                      exp2 ::= sml_pat OR sml_exp
23                      exp2 ::= const
24                      exp2 ::= v
25                      exp2 ::= let {decs ;} in exp {; exp} end
26                      exp2 ::= (exp {; exp})

27 (SML meta pattern) sml_pat ::= % meta_var

28 (SML meta expr)    sml_exp ::= % meta_var
29                      sml_exp ::= %( s_exp

30                      meta_var ::= v
31                      meta_var ::= _

32                      s_exp ::= string %)
33                      s_exp ::= string rml_exp s_exp

34                      calt ::= % meta_var [(mpvname {, mpvname)] => exp
35                      calt ::= v[(mpvname {, mpvname)] => exp
36                      calt ::= mpvname :: mpvname => exp
37                      calt ::= [ ] => exp

```

```

38      mpvname ::= % meta_var
39      mpvname ::= v

40      mpvnl ::= ([mpvname {, mpvname}])

41      decs ::= val v = exp
42      decs ::= fun fdecl

43      fdecl ::= fdec and fdecl
44      fdecl ::= fdec
45      fdecl ::= catamarker fdec

46      fdec ::= v {inlflag mpvnl} inlflag mpvnl = exp

47 (constructor)  rator ::= c [" string "]
48               rator ::= c [" string "] of ty
49               rator ::= c [" string "] of tuple_ty

50 (types)       ty ::= ty → ty
51               ty ::= tuple_ty → ty
52               ty ::= ty'
53               tuple_ty ::= ty' * ty' (* ty')

54               ty' ::= t
55               ty' ::= ((ty ,) ty , ty) D
56               ty' ::= ( ty )
57               ty' ::= [ty' ] D

58 (primitive value) primdef ::= val k : K " string "
59               primdef ::= val p ((k:K ,) ) : (k:K) [pure] "string"

60               inlflag ::= [(! inline !*)]

```

Figure 4 A template definition for the rule specification language.

Identifiers and symbols in **bold-face** refer to terminal symbols. All terminals usually stand for the actual word they represent. For example **pure** indicates the occurrence of the word itself as a keyword. The exceptions are listed below:

v	=> variable	K	=> primitive (abstract) type name
D	=> name of datatype	k	=> primitive constant name
t	=> type variable	p	=> primitive function name
c	=> type constructor	string	=> lexical string

OR => Indicates that its arguments are mutually exclusive. *sml_pat* applies to *exp2* for the left-hand side of the rules while *sml_exp* is for the right-hand side.

Rml_pattern (e.g., `%a * 0` on line 20, Figure 1) is the pattern against which source language constructs are compared to detect opportunities for rewriting. It is defined to be an RML concrete syntax expression, possibly with embedded meta-variables or identifiers (e.g., `%a` on line 20, Figure 1) and wildcards. The only valid wildcard is `%_`. The meta-variables act as placeholders for an actual construct and provide a means of referring to the variable or expression within *cond_exp* or *rml_action*. They are replaced by actual literals, variables or expressions during pattern matching. *Rml_action* describes the result of the expression to be substituted when the pattern matches.

Cond_exp is a conditional expression that is evaluated during pattern-matching (e.g., `(a = b)` on line 31, Figure 1). One of the prerequisites for a correct match is that this must evaluate to true. It may contain any boolean-valued SML expression which in turn may have RML concrete expressions embedded inside it.

Rml_action specifies the result of transforming an expression if a match succeeds. The syntax of *rml_action* is very similar to that of *rml_pattern*. However, it is more general in nature because the embedded constructs within RML concrete expressions are not restricted to be identifiers or wildcards. On the contrary, these may be arbitrary SML expressions. In the most general case, these in turn may have RML concrete expressions embedded in them to arbitrary levels. Thus the footnote in Figure 4

mentions the fact that the left hand side of the rule can merely contain a meta-variable whereas the right hand side can be a meta-expression. We have drawn our inspiration for the meta-expression syntax from the SML/NJ Quote/Antiquote mechanism. Quotes (`'%`) enclose SML expressions and Antiquotes (````) enclose RML concrete expressions. However, Antiquotes are implicit at the outermost level of specification, i.e., they are not written. In the simplest case, *rml_action* can be almost trivial (e.g., 0 on line 10, Figure 1). It is necessary that *rml_action* should evaluate to an RML abstract expression, this translation from concrete RML to the abstract form being taken care of by the optimizer-generator.

An invalid *rml_action*, i.e., one which does not result in an RML abstract expression, is rejected at the time of compilation of the generated SML optimizer function, before it is integrated with the main compiler. An invalid *cond_exp* is checked for in the same way.

The rule meta-expressions or patterns are ultimately compiled to SML patterns. However, unlike SML, our language allows duplicate meta-identifiers to appear within a single pattern. The semantics of duplicate meta-variables require that they match the identical actual construct. Thus, the rule on line 31 (Figure 1) can also be expressed as

$$\%a - \%a \Rightarrow 0.$$

The use of a specialized expression language (RML/SML), rather than pure SML, to describe rules confers several benefits. The syntax is very similar to the source language syntax and should be fairly simple for the programmer to master. Moreover, the ability to nest SML within RML expressions to potentially unlimited levels, makes the rewrite actions very powerful. In theory, this can make the right-hand sides of the rules arbitrarily complex, but, as our examples demonstrate, this need not be the case in practice. Some potential complexity in the system is tolerable in view of the flexibility and expressiveness it provides. Moreover, the primary target is the knowledgeable user, either the end-user or the compiler-writer, who has the requisite experience to deal with sophisticated systems.

4. Examples

We present a series of examples to demonstrate some of the different ways in which our system can be utilized.

4.1 Strength Reduction

Let us take a closer look at our previous example and see if it does anything other than the constant-folding we discussed. We mentioned before that constant-folding is a “shrinking” optimization i.e. it reduces the size of the transformed code. However, our rules are not restricted in any way to perform only shrinking optimizations.

Sometimes, it might be a good idea to replace an instruction sequence by a more optimal one. The rules in lines 24 and 25 (Figure 1) perform strength reduction. They replace a multiplication by 2 operation by an addition. This will speed up the object code if multiplication takes more time than addition, as is the case on many machines. Obviously, this is an improving transformation only if the target machine has the appropriate architecture. As such it requires knowledge of low-level details. Thus, it is one way in which the compiler-writer can exploit his knowledge of the underlying machine, to optimize the code in a modular and simple manner.

4.2 Matrix Addressing

Now, we shall discuss our system with respect to performing some important optimizations that are dependent on the memory layout of the application-specific data structures. An area that responds very favorably to optimizations is that of matrix addressing schemes. The elements of the matrix can be accessed very quickly if they are known to be stored in a block of consecutive locations.

For such a matrix M , stored in row-major form, the relative address of $M[i_1][i_2]$ may be calculated by the formula

$$base + ((i_1 - low_1) * n_2 + i_2 - low_2) * w$$

where $base$ is the relative address of the storage allocated for the matrix,

low_1 and low_2 are the lower bounds of the values i_1 and i_2 ,

w is the width of each matrix element and

$$n_2 = high_2 - low_2 + 1.$$

If we are compiling to a target language like C, where lower bounds are always 0, the above expression can be simplified even further to

$$base + ((i_1 * n_2) + i_2) * w$$

where $n_2 = high_2 + 1$.

We present an application that allocates memory for a two-dimensional array of integers and initializes each element to 0 (Figure 5). A number of RML functions are defined to manipulate matrices. Common operations on matrices include *subscript()* and *update()* that access a matrix with reference to a particular row and column number. We intend to use our knowledge of the fact that the matrix elements have been allocated contiguously, to optimize references to it. *get_addr()* calculates the address of a particular element from the base address and the row and column numbers.

In Figure 6 we describe an interface for operations on matrices of integers. This set of primitive functions is used to allocate a contiguous chunk of memory for a matrix and to access its elements. Calls to *get_val()* and *set_val()* can be used in the RML source, provided the address of the element is known.

```

1  (*! template "tmatrix_template.spec5" !*)
2  (*! exports
3     val doit: unit "doit" !*)

4  datatype (*! flat !*) param =
5     Param of integer * integer * integer * integer * integer
6  datatype (*! flat !*) matrix = Matrix of address * param

7  (* library of rml functions to manipulate matrices *)

8  fun get_addr (*! inline !*) m r c =
9     case m of
10      Matrix(b,p) => (case p of
11          Param(w,l1,l2,h1,h2) =>
12              let val n = h2 - l2 + 1
13                  val q = (r*n+c) - (l1*n+l2)
14                  val a = to_addr(b,q)
15              in a end)

16 fun subscript (*! inline !*) m r c = get_val(get_addr m r c)

17 fun update (*! inline !*) m r c i = set_val((get_addr m r c),i)

18 fun create_matrix (*! inline !*)p =
19     case p of
20     Param(w,l1,l2,h1,h2) => let val r = h1 - l1 + 1
21                             val c = h2 - l2 + 1
22                             val d = r * c * w
23     in newVector(d) end

24 (* program to create an initialized matrix *)
25 val doit = let val w = 4 and l1 = 0 and l2 = 0
26             val h1 = 1999 and h2 = 1999
27             val p = Param(w,l1,l2,h1,h2)
28             val b = create_matrix p
29             val m = Matrix(b,p)

30             fun init_all r =
31                 let fun do_cols c =
32                     if (c <= h2) then
33                         (update m r c 0; do_cols (c + 1))
34                     else ()
35                 in if (r <= h1) then
36                     (do_cols l2; init_all (r + 1)) else ()
37             end
38     in init_all l1 end

```

Figure 5 An RML library for matrix manipulation.


```

1  template MatrixTemplate
2  header "#include \"matrix_prims.h\""

3  (* vector of integers *)

4  type integer(4) "int"
5  type address(4) "int *"

6  datatype bool "int" = true "1" | false "0"
7  datatype unit "int" = Eunit "0"

8  (* basic integer operations *)
9  val - (x0:integer, x1:integer) : (res:integer)           "... "
10 val + (x0:integer, x1:integer) : (res:integer)           "... "
11 val * (x0:integer, x1:integer) : (res:integer)           "... "
12 val ~ (x0:integer) : (res:integer)                       "... "
13 val <= (x0:integer, x1:integer) : (res:bool)             "... "

14 (* add an integer to an address *)
15 val to_addr (x0:address, x1:integer) : (res:address)     "... "

16 (* vector creation *)
17 val newVector (x0:integer) : (res:address)               "... "

18 (* subscripting operations *)
19 val get_val (x0:address) : (res:integer)                 "... "

20 (* updating an existing vector*)
21 val set_val (x0:address, x1:integer) : (res:unit)        "... "

```

Figure 6 A template definition of primitives for vectors of integers.
(also includes the rules defined in Figure 1)

The program has been designed so that the primitive functions operate on low-level raw memory, while the RML functions impose the structure of a matrix on top of the allocated memory.

The functions listed in Figure 5 are intended for converting matrix accesses to references to an explicit address, relative to the base address. As a result of various simplification passes and inlining, *update()* and *subscript()* are replaced by calls to the corresponding routines that require explicit addresses as parameters. Following this, a call to *get_addr()* is replaced by the actual expression to calculate the address of an element, relative to the base address. This is where the constant-folding rules specified in Figure 1 come into play. They are used to evaluate the address expression as much as possible at compile time, as we discussed above.

4.3 Vector Loop fusion

Scientific and engineering problems often involve solving large systems of linear equations. Iterative methods for solving linear systems work by repeatedly improving an approximate solution until a sufficiently accurate solution is obtained. They are very suitable for solving large systems where direct methods may not succeed in finding an accurate solution. Efficiency becomes crucial as the problem size may be very large.

Sophisticated mathematical libraries exist for this problem as well as for performing standard operations on vectors and matrices. However, in areas which demand high-performance computing, general-purpose routines are simply not adequate. It is often necessary to write custom software, tailored for the system at hand. The reasons for this include the absence of the desired functionality in existing routines, data structures that are not natural for the problem and existing routines that are inefficient when applied to special instances.

The computational kernels of the iterative methods are a key criterion in determining performance [3]. They are typically coded to execute as fast as possible on the target architecture. Iterative schemes share most of their computational kernels. Some of the most time-consuming kernels are:

- vector updates
- inner products
- matrix-vector products.

The *conjugate gradient* method belongs to the class of iterative methods described before. It is very effective for solving linear systems where the coefficient matrix is symmetric positive definite. We present here (Figure 7) an implementation of the conjugate gradient method [11] that uses all the kernels we have mentioned and demonstrate how we can use our rewrite scheme to optimize it further. The

optimization rules that are applicable here are described in Figure 8. We would like to add that this is not intended to be a completely realistic or particularly efficient version of the algorithm. Rather, the aim is to demonstrate the applicability of our system to optimize situations similar to those arising here.

The method proceeds by generating successive approximations to the solution (vector sequences of iterates), corresponding residuals and search directions used in updating the iterates and residuals.

The heart of the computation is done by *do_calc()* which describes the work done in each iteration. It invokes several basic primitive routines that are defined in the template (Figure 8). Our template also contains some highly optimized routines which are specialized for this application area and are much more efficient than the general matrix/vector operations. Both these libraries are fixed and available to the user of our system for describing the rewrite rules. Moreover, they are separate and independent of each other. The intention is that in the usual case the source program would be written using the basic primitives. The latter could then be optimized by rewriting the original source program to use the more optimized versions. The automation of the optimization could be necessary for a variety of reasons, some of which are discussed in §6.2.

```

1  (*! template "cgrad_template.spec5" !*)
2  (*! exports
3     val cgrad : matrix->integer->integer->vector->unit "cgrad" !*)

4  (* a - coefficient matrix of linear system
5     b - right-hand side of equations
6     x - solution vector
7     d - direction vector
8     g - gradient vector
9     n - size of linear system
10    e - convergence criterion
11    s - step size
12    denom1,denom2,num1,num2 - temporary variables
13 *)

14 fun cgrad a n e b =

15     let val d = initv(n,0)
16         val x = initv(n,0)
17         val g = negv(b)

18         fun do_calc n1 =
19             if (n1 > 0) then (
20                 let val denom1 = inner_product(g,g)
21                     val g= addv(matrix_vector_product(a,x),negv(b))
22                     val num1 = inner_product(g,g)
23                 in if (e > num1) then ()
24                     else
25                         (let val s = div(num1,denom1)
26                             val d = addv(scalar_mulv(d,s),negv(g))
27                             val num2 = inner_product(d,g)
28                             val denom2 = inner_product(d,
29                                 matrix_vector_product(a,d))
30                             val s = div(num2,denom2)
31                             val x = addv(x,negv(scalar_mulv(d,s)))
32                         in (do_calc (n1 - 1)) end)
33                     end)
34             else ()

35     in do_calc n end

```

Figure 7 An RML implementation of the conjugate gradient method.

```

1 (* some operations on matrices and vectors of integers *)
2 template CgradTemplate

3 type integer(4) "int"
4 type vector(4) "vector"
5 type matrix(4) "matrix"

6 datatype bool "int" = true "1" | false "0"
7 datatype unit "int" = Eunit "0"

8 (* arithmetic operations on integers *)
9 val ~ (x0:integer) : (res:integer)           "... "
10 val - (x0:integer,x1:integer) : (res:integer) "... "
11 val > (x0:integer,x1:integer) : (res:bool)   "... "
12 val * (x0:integer,x1:integer) : (res:integer) "... "
13 val div (x0:integer,x1:integer) : (res:integer) "... "

14 (* some basic library operations on vectors *)

15 (* allocate a vector of integers of size x0,each element initialized to x1 *)
16 val allocv (x0:integer,x1:integer) : (res:vector) "... "
17 val negv (x0:vector) : (res:vector)         "... "
18 val addv (x0:vector,x1:vector) : (res:vector) "... "
19 val scalar_mulv (x0:vector,x1:integer) : (res:vector) "... "
20 val inner_product (x0:vector,x1:vector) : (res:integer) "... "
21 val matrix_vector_product (x0:matrix,x1:vector):(res:vector)"..."

22 (* an optimized library of some fused functions *)

23 (* x0 * x1 * x2 *)
24 val conj_mv_ip (x0:matrix,x1:vector,x2:vector) : (res:integer)
                                         "... "

25 (* add x0*x2 to x1*x3 *)
26 val scaled_addv (x0:vector,x1:vector,x2:integer,x3:integer) :
                                         (res:integer) "... "

27 (* add x0*x1*x3 to x2*x4 *)
28 val scaddv_mv_prod
29     (x0:matrix,x1:vector,x2:vector,x3:integer,x4:integer) :
30     (res:integer) "... "

31 (* ~(x0 * x1) *)
32 val neg_mv_product (x0:matrix,x1:vector) : (res:vector) "... "

33 (* negate x0 and then calculate x0 * x1 *)
34 val ip_negv (x0:vector,x1:vector) : (res:integer) "... "

```

```

(* rules to replace basic vector operations with the more optimized versions *)

35 rules

36 | inner_product (%v1, matrix_vector_product (%m,%v2)) =>
    conj_mvp_ip (%m,%v2,%v1)

37 | addv (%v1,%v2) => scaled_addv (%v1,%v2,1,1)

38 | scaled_addv (negv (%v1),%v2,%s1,%s2) =>
    scaled_addv (%v1,%v2,~(%s1),%s2)
39 | scaled_addv (%v1,negv (%v2),%s1,%s2) =>
    scaled_addv (%v1,%v2,%s1,~(%s2))

40 | scaled_addv (scalar_mulv(%v1,%i),%v2,%s1,%s2) =>
    scaled_addv (%v1,%v2, (%s1*%i), %s2)
41 | scaled_addv (%v1,scalar_mulv(%v2,%i),%s1,%s2) =>
    scaled_addv (%v1,%v2,%s1, (%s2*%i))

42 | scaled_addv (matrix_vector_product (%m,%v1),%v2,%s1,%s2) =>
    scaddv_mv_prod (%m,%v1,%v2,%s1,%s2)

43 | negv (initv (%n,%i)) => let val a = ~(%i)
44 |                               in initv (%n,a) end
45 | negv (negv (%v)) => %v
46 | negv (matrix_vector_product (%m,%v)) => neg_mv_product (%m,%v)

47 | inner_product (matrix_vector_product (%m,%v2),%v1) =>
    conj_mvp_ip (%m,%v2,%v1)
48 | inner_product (negv (%v1),%v2) => ip_negv (%v1,%v2)
49 | inner_product (%v1,negv (%v2)) => ip_negv (%v2,%v1)

50 | matrix_vector_product (%m,negv (%v)) => neg_mv_product (%m,%v)
51 | scaled_addv (%v2,matrix_vector_product (%m,%v1),%s2,%s1) =>
    scaddv_mv_prod (%m,%v1,%v2,%s1,%s2)

```

Figure 8 A library of primitive operations for matrix and vector manipulation.

In line 21 (Figure 7), the calls to *matrix_vector_product()* and *negv()* create entirely unnecessary temporaries, since their only purpose is to create vectors to add to each other. We can replace the calls with a call to a single routine that takes a , x and b , multiplies the former two, negates b , adds them together and returns the result. We achieve this optimization via a series of rewrite rules. The rule on line 37 (Figure 8) rewrites the call to *addv()* to that of the optimized routine *scaled_addv()*. This is then further rewritten by the rules on lines 39 and 42 (Figure 8) to finally use the optimized routine *scaddv_mv_prod()*. Note that we are saving not only time spent in allocation, but also an entire pass over a vector by using such an optimized routine. As these are common operations in vector manipulation and could make a significant difference to the computation time where the size of the vectors is large, providing fused routines makes a lot of sense.

Similarly, on line 26, the calls to *scalar_mulv()* and *negv()* create unnecessary intermediate vectors, only to add them together. We use the rules on lines 37, 39 and 40 (Figure 8) in order to completely specify the optimization possible in this case.

Once again, on line 28 (Figure 7), the call to *matrix_vector_product()* creates an entirely unnecessary temporary. We rewrite it according to the rule on line 36 (Figure 8), so that *conj_mvp_ip()* takes the matrix a and the vector d and does the actual computation. This routine can be written to perform this operation in the most

optimum way for the target architecture and to take advantage of the data layout of the matrix.

The final optimization opportunity exists on line 31 (Figure 7). This is executed by using the rules on lines 37, 39 and 41 (Figure 8). The optimized program is presented in Figure 9.

All these functions are called during each iteration and so optimizing them should make a measurable difference to the overall computation. Now, it might be possible for an optimizing compiler to perform some of the loop fusions described above, during the course of its standard optimization phase. To do this, it would need access to the library implementation code. However, as we noted before, these primitive routines are typically coded in a fairly low-level language for maximum efficiency. For example, they could be written to take advantage of the availability of vector processors. Thus the library code is not the same as that of the source language and often not available. This is a typical situation where our rules could prove useful.

Our library could have other optimized operations to support common functions which compose. In our listing, we have provided merely a subset of the possible combinations. The remaining rule patterns have been created to perform some other possible rewrites, though this particular algorithm does not use these rules. While all

```

1  (*! template "cgrad_template.spec5" !*)
2  (*! exports
3     val cgrad : matrix->integer->integer->vector->unit "cgrad" !*)

4  (* a - coefficient matrix of linear system
5     b - right-hand side of equations
6     x - solution vector
7     d - direction vector
8     g - gradient vector
9     n - size of linear system
10    e - convergence criterion
11    s - step size
12    denom1,denom2,num1,num2 - temporary variables
13 *)

14 fun cgrad a n e b =

15     let val d = initv(n,0)
16         val x = initv(n,0)
17         val g = negv(b)

18         fun do_calc n1 =
19             if (n1 > 0) then (
20                 let val denom1 = inner_product(g,g)
21                     val g = scaddv_mv_prod(a,x,b,1,~1)
22                     val num1 = inner_product(g,g)
23                 in if (e > num1) then ()
24                     else
25                         (let val s = div(num1,denom1)
26                             val d = scaled_addv(d,g,s,~1)
27                             val num2 = inner_product(d,g)
28                             val denom2 = conj_mvp_ip(a,d,d)
29                             val s = div(num2,denom2)
30                             val x = scaled_addv(x,d,1,~(1*s))
31                         in (do_calc (n1 - 1)) end)
32                     end)
33             else ()

34     in do_calc n end

```

Figure 9 An optimized version of the conjugate gradient method.

these rules do not introduce new fused operations, they do eliminate procedure-call overheads and redundant computations by rewriting. The idea behind the design of such a library is to have a set of operations and rules for each specialized application domain, in order to rewrite frequently occurring patterns in the code.

5. Implementation Details

5.1 Extensions needed to the RML compiler

We have built our system to extend the capabilities of the RML compiler [12]. In order to do this we had to design and implement

- (i) A language for the specification of the rewrite rules.
- (ii) A parser for the rule-specification language. It translates the rules to RML abstract syntax.
- (iii) An SML post-processing module that performs syntactic and semantic checks on the rules. It removes duplicate variables from the left-hand sides, replacing them with fresh names and inserting an appropriate conditional check to determine if the actual expressions bound to the variables are identical. It does error-checking on the left-hand sides of the rules to ensure that they contain valid meta-patterns.
- (iv) A library of general-purpose SML functions that aid in describing how to perform the rewrites.

- (v) An SML optimizer generator that takes as input the processed rule patterns, the template definitions including the user-defined functions and the library routines. It then proceeds to generate a SML routine to perform the pattern-matching, decide which rule to apply, and subsequent rewrite the redex.
- (vi) An SML module that implements our optimization strategy to traverse the abstract syntax tree of the program in a bottom-up manner. It compiles and invokes the routine described in (v), to perform each reduction.
- (vii) A simplifier (§5.4) for the abstract RML code, which performs standard optimizations like dead code elimination, inlining, value and variable propagation, etc. This was necessary, because we use our generated optimizer, together with these other simplifications, to improve the effectiveness of the optimization phases. The simplifier may uncover opportunities for applying the user-specified optimizations. These are then exploited by the optimizer to implement further reductions.

5.2 Generation of the Optimizer from User Specifications

The optimizer is produced from the rules and functions defined in the template. The result of the translation is an SML file. This contains the user-defined auxiliary functions and the actual optimizer function that matches each input RML abstract expression against the rule patterns. The SML optimizer function is compiled before being linked with the rest of the compiler. In addition, a library is available for the use

of the rule-writer. This library is linked in with the generated optimizer. It provides some general-purpose SML functions that may be invoked in the *cond_exp* and *rml_action* part of the rule. These routines are useful for constructing abstract RML expressions from within the embedded SML expressions and for converting abstract RML literal expressions to SML literals. This facility is important because we require the right-hand sides of all rules to evaluate to an abstract RML expression. At the same time, we do not wish to burden the rule-writer with having to know the details of the intermediate representation we are actually transforming. Having the rule-writer deal with these problems would complicate the specification and require exposing internal implementation details, which is not in the spirit of our specification language.

The library contains functions to construct abstract RML literal expressions (e.g., *mkIntegerLit()*, *mkCharLit()*, *mkStringLit()*, *mkRealLit()*) and deconstruct an RML literal to an SML literal (e.g., *SmlIntegerLit()*, *SmlCharLit()*, *SmlStringLit()*, *SmlRealLit()*). It has been developed for the purpose of providing implementations for tasks that are essential to the optimization process, but are not really a part of the high-level specification of the rules. The rules should describe “what to do” whereas the automatically generated optimizer and the library routines take care of “how to do it”. As more functionality is added, via our system, to the kinds of optimizations the core compiler can perform, the library too should be extended to support them. Thus, to use

our system to perform inlining “by hand”, for example, it would be necessary to add a routine to rename the bound variables in an expression.

The RML concrete source program is compiled into RML abstract syntax. Each rule expression is parsed to RML abstract syntax. The compiled rules are aggregated to form the optimizer function. This function is an SML function that takes an RML abstract expression as input and produces a possibly optimized RML abstract expression as output.

5.3 Optimization via Pattern-matching

This generated function is the heart of the optimizer. It is applied to the RML intermediate program to optimize it according to user directives. We have separated the specification of the rules from the strategy of how to apply them to build the resulting optimizer. The generated function encodes the actual transformations to be performed. It is compiled before being integrated with the core compiler. The tree-traversal strategy is implemented by an SML driver routine which invokes the optimizer function to do the actual rewriting. Our current implementation follows a fixed bottom-up strategy for traversing the program syntax tree. We rewrite each child node once, before rewriting the parent node. We may be able to improve the efficiency of the optimizer by modifying the driver so that each child node is rewritten repeatedly, before turning to the parent node.

The mechanics of pattern matching are implemented roughly as follows:

Each RML abstract expression in the compiled source program is compared to the compiled rule patterns (*rml_pattern*) according to the strategy discussed before in §2.

The rules are examined in the order they have been listed in the template. If the pattern matches, then the conditional expression, if any, is evaluated. The match succeeds only if the condition is true. Then the input expression is rewritten according to the action encoded in *rml_action*.

However, if a match fails, then the next rule is investigated and so on, until a match occurs or all the rules have been exhausted. A match may fail, either because the pattern does not match, or the conditional evaluates to false. If none of the rules match the input expression, the original expression remains un-transformed. If there is more than one potential match, the first rule specified in the template is actually chosen for the rewrite.

The rewriting strategy is type-oblivious, i.e., it does not consider the types of the expressions before reducing them. Therefore, this phase is followed by a round of type inference.

5.3.1 Termination

The rewrite rules are applied in several passes, each pass potentially uncovering further opportunities for rewriting, until no more changes are observed. This would normally guarantee termination. However, since the *rml_action* part of the rules can be arbitrarily complex, there exists the opportunity for writing code that does not terminate. This is a significant drawback of general rewrite systems like ours. Arguably, any problems that arise would be due to a misuse of the expressive powers of the rule language, rather than any inherent weakness in our design. We chose to implement the rewrite rules using SML to take advantage of its expressiveness and powerful pattern-matching facility and also because the underlying compiler itself is written in this language.

Non-terminating right-hand sides would result if the code itself contained ill-defined constructs like a call to a recursive function that does not have a well-formed base condition to check termination. We have no way of detecting such cases.

Another situation where non-termination occurs is when the right-hand side of a rule overlaps with some left-hand side, leading to cycles in the reduction graph. We make no attempt to prevent such situations.

5.3.2 Confluence

A desirable feature of any term reduction system is the ability to prove confluence. Since our optimizer has been implemented using term rewriting, we would like to compare it with other similar systems. The Church-Rosser or confluence property states that whenever an expression A may reduce to two different expressions B and C , then there exists another expression D , to which both B and C reduce. The existence of this property is required to guarantee uniqueness of normal forms. A normal form is a state from which no more transitions are possible.

Confluence depends on the properties of the rules in a rewrite system. We make no attempt to analyze the rules and therefore cannot guarantee confluence in our system. In fact, we do not expect confluence in an optimizer, although special sets of rules may produce confluence [2].

5.4 Optimization of RML Abstract Code

We have written an optimizer for the RML abstract code, to be used along with the user-directed optimizer. The motivation is to use the former to uncover further opportunities for rewriting, which can then be detected and implemented by the latter. One complements the other. Since both are separate modules, invoked in an appropriate order by the top-level driver, it is easy to apply each independently, and

observe the effects, should the need arise. We have used the output of one to feed the input of the other, in a controlled fashion.

The optimizer carries out a variety of standard partial-evaluation style improvements on the RML abstract syntax. They are similar to those described in [12]. These standard rewriting techniques, together with the user rules, are applied repeatedly until no changes are observed or a predetermined number of passes have been reached.

The optimizer guarantees not to reorder, duplicate, or eliminate any primitive applications or calls to potentially non-terminating functions. Only *pure* expressions can be eliminated. This is essential because RML has strict semantics and templates may include impure operators. For simplicity, all user function calls are treated as impure. Pure primitive operators are marked as such in the template function definition.

5.5 Testing

It is obviously not a trivial task to formally and exhaustively benchmark a system with the scope of applicability that our system has. The aim of this thesis has been to explore an attractive compiler design decision and to see if we could implement a useful tool in the limited time period available to us. To implement and test a realistic problem domain, together with its optimizations and realizations, is a separate project

in itself. Moreover, the potential gain in efficiency is dependent on many complicating factors like the nature of the problem domain, the scope of optimization opportunities within it, the manner in which the rules are specified, and the particular strategy being used.

However, we have tested our optimizer on the examples we have presented. We have examined the transformed intermediate code to satisfy ourselves that it is indeed performing the optimizations that we expect and claim it does. Moreover, we have implemented the matrix address optimizations used in the program of Figure 5. We tested this program, with the primitives written in C and used it to compare the running time of the optimized C code with the running time of the un-optimized version. As expected, the results show that the optimized program executes faster. The averages in Figure 10 were obtained by running the program five times, discarding the maximum and minimum run times and then calculating averages for the remaining three run times.

Average Run time (User + System) (seconds)	
Before optimization	1.74
After optimization	1.64
Percentage speedup	5.7

Figure 10 Speedup obtained as a result of optimizing matrix address arithmetic.
The program was tested on a 133MHz Pentium with 80MB of memory, running Linux version 2.0.27, for a matrix of size 2000 * 2000. The C code was compiled by gcc version 2.7.2.1 using the -O2 flag.

6. Discussion

6.1 Related Work

Many attempts have been made to write systems for program optimizations, some using rewriting strategies similar to ours. We indicate some of those more relevant to our discussion.

Appel and Jim [2] have studied a shrinking rewrite system. It implements only those optimizations that are guaranteed to make the program smaller, such as dead-variable elimination, constant folding and inlining of functions called just once. Their set of rules is fixed, unlike ours. They have been able to prove confluence for their system. They also show some efficient algorithms for implementing the optimizations. Our system is more general in nature, since the rewrite rules are not restricted to those that reduce the code size.

TXL [4], *Puma* [7] and *KHEPERA* [6] all provide languages for specifying tree transformations and mechanisms for matching sub-trees. *TXL* has a variety of “searching” primitives, that encapsulate both the rewriting rules and match-application strategies into a single unit. The system allows the application of a rule to be controlled, either replacing the first occurrence of a match in a sub-tree or all occurrences. In our system, we have simplified the design, by separating the tree-

traversal strategy from the facility for specifying the transformations. Thus, unlike some of the other systems, we do not require the programmer to explicitly program the traversal strategy.

Visser, Benaissa and Tolmach [13] describe a language for specifying program transformations via rewrite rules and explicit strategies and how to use it to build optimizers. Their strategy language is much more powerful than ours and uses operators like choice, sequential composition and recursion to develop transformations from some basic un-conditional rewrite rules. Moreover, their basic language can be extended with side-conditions and contextual rules, making their system more suitable for performing realistic compiler optimizations. However, they have not yet integrated their rewrite system into a compiler as we have successfully accomplished. The way we separate our rules from the strategy of applying them, is similar to their approach. They too have a flexible set of rules like us. Unlike them, we have not yet experimented with different strategies in the implementation of our optimizer. This and other related work is something we intend to do in the future.

Intentional Programming [1] is a system for program transformation that is similar in spirit to ours in that it allows domain-specific optimizations to be applied in an extensible manner. Since the system does not guarantee confluence, the authors emphasize controlling the order of transformation, so that it can be specified in a

compositional manner. Their rules may be dependent on other rules and the surrounding context. This requires the programmer to have some control over the transformation order by explicitly declaring dependencies between the rules he introduces and the existing ones. We have built our system so that the rules can be added incrementally, without making any assumptions about how they might interact with each other. This makes it easier to comprehend our design and frees the programmer from the burden of deciding how to implement the optimizer. Of course, complete absence of any guidance might lead to a less efficient optimization strategy in some cases.

Aspect-Oriented Programming (AOP) [8] suggests a generalized way of optimizing programs, using domain-specific *aspect* languages. Programs are sub-divided into *components* and *aspects*. Aspects are then merged with components, relying on a particular aspect language to specify the transformations and a *weaver* to integrate the optimizations into the original program. For each application area, new weavers and aspect languages need to be developed. Our rewrite rule specification might be viewed as one particular kind of aspect language.

6.2 Our approach

The goal of programming in a high-level language is to be able to write efficient, platform-independent programs in a simple and easily comprehensible language.

Unfortunately, highly optimized programs often tend to be obscure, hard to write and hence, difficult to develop and maintain. Therefore, we attempt to separate the tasks of program writing and optimizing as much as possible.

Our system has a dual purpose. It allows the end-user to use his knowledge of the problem domain to indicate possible optimizations to the source program. It also facilitates the task of the compiler-writer. We demonstrated some applications of how the end-user can use our system in the matrix examples. These optimizations, by their very nature, are specialized and so unlikely to be implemented within the core optimizing modules of a general-purpose compiler.

The complexity of the system may not permit a naïve end-user to use it very effectively or easily. Rather, we anticipate it to be more useful in allowing intervention by the sophisticated programmer who is likely to have the most significant knowledge of the problem area. Thus, naïve code can later be rewritten to be more efficient without sacrificing readability or ease of writing at the source-code level.

This approach also provides a great deal of flexibility to the compiler-writer. Optimization directives can be added or removed easily as required without modifying the rest of the compiler. This is a great benefit for developing a retargetable compiler.

Platform-specific optimizations like constant-folding can be performed in a modular fashion, very early in the compilation process.

We must emphasize the fact that having a system like ours to optimize the program is important because it may not always be possible for even a sophisticated programmer to edit the source code directly. A variety of reasons may be responsible:

- The patterns specified in the left-hand sides of the rules may not occur in the original source code. Rather, they could have been uncovered as a result of other simplification passes that perform transformations like inlining.
- Optimized libraries may not have been developed at the time the application was written. Our system enables these to be linked in as and when they become available.
- It is often simpler to write source code in an incremental fashion, using several smaller function calls rather than writing a highly optimized construct. Transforming the naively written program automatically, preserves readability but does not sacrifice runtime efficiency.
- The template mechanism promotes reuse of the rules and primitives.

A nice feature of the design is the clean separation of the specification rules from the strategy of applying them to the program tree. This should make it easier to reason about the properties of the rules in isolation from the implementation strategy. Perhaps

more relevant to us is the fact that development time for the optimizer can be reduced in this way. A set of rules can be reused by the generator, with different strategies for applying them, resulting in optimizers of varying efficiencies. In optimizers where the rewriting rules and application strategies are closely inter-woven, there really is no easy way to modify them to implement essentially a different set of optimizations. Also, debugging and testing the optimizer becomes much easier.

The specification language we have designed is very similar to the program development language. This should make it easier for the programmer to learn how to use the system. Moreover, since the user does not have to write the actual code for implementing the optimizations, he can use his time more effectively in making higher-level decisions to fine-tune the code, rather than in the more laborious, repetitive and error-prone task of writing the entire optimizer by hand.

6.3 Future work

A logical extension to this system would be to add parsing support for the application-specific constants. Our current system does not have an extensible parser and this limits its usefulness. A parser capable of recognizing all types of constants would certainly make it easier to express constant-folding rules and uncover optimization opportunities.

Our system separates the mechanism of rule specification from the strategy for applying the rules to the abstract syntax tree of the program. However, currently we have a fixed strategy for applying the rules. We would like to extend this system to enable the user to choose from one of several different strategies for rule application. This would lead to a faster optimizer.

An area worth exploring is to use this system for performing standard optimizations like hoisting, dead code elimination and inlining. However, to be truly useful for doing this, we would have to provide supporting routines in the SML library. These could be routines to determine if an expression is pure, for generating fresh variables, for determining the free variables in an expression and for renaming of variables in an expression.

Another area that needs further work is the development of good benchmarks to test the applicability and efficiency of our system in different areas, including that of standard compiler optimizations. This could involve testing the effects of separate rule application strategies on the efficiency of the resulting optimizer as well as measuring the execution speed of the optimizer-generated code for domain-specific applications.

We would like to test our rules to see whether they can be used to transform the true intermediate language of the RML compiler [12]. At the moment, we are transforming

the abstract syntax tree of the program. There are various complicating factors that make it hard to directly translate the rules to this intermediate language. It would be interesting to see if we can find a solution to this problem in the future.

7. Conclusion

We have attempted to demonstrate the advantage of designing a compiler with a facility for incorporating user-specified optimizations as a separate module. In doing so, we had to deal with various issues of language design and integration with the core compiler. We have certainly achieved our basic implementation goals, though in doing so we have come to realize its limitations and found new ways of improving the system. Since we have realized our main objective of designing a modular compiler, this facility will prove very useful in making these improvements. Thus, though we chose to implement the idea in the world of functional programming, we think that this feature would be a welcome addition to any general-purpose compiler. As such, the benefits might very well justify the development effort.

The simplicity and modular nature of our design is what makes it different from that of other systems of this kind. Moreover, unlike other existing monolithic systems, our optimizer can be extended fairly easily, to enhance its usefulness and efficiency in implementing a wider range of optimizations.

8. References

1. Aitken, W. and Dickens, B. and Kwiatkowski, P. and de Moor, O. and Richter, D. and Simonyi, C. Transformation in Intentional Programming. In *Proc. ICRS5*, June 1998 (to appear).
2. Appel, A. and Jim, T. Shrinking Lambda Expressions in Linear Time. *Journal of Functional Programming*, Volume 7, Number 5, pages 515-540, September 1997.
3. Barrett, R. and Berry, M. and Chan, T. and Demmel, J. and Donato, J. and Dongarra, J. and Eijkhout, V. and Pozo, R. and Romine, C. and Vorst, H. *Templates for the Solution of Linear Systems, 2nd Edition*. SIAM 1994. (http://netlib2.cs.utk.edu/linalg/html_templates/report.html).
4. Cordy, J. and Carmichael, I. and Halliday, R. *The TXL Programming Language, Version 8*. Legasys Corp., April 1995.
5. O'Donnell, M. *Equational Logic as a Programming Language* Chapter 18. MIT Press, 1985.
6. Faith, R. and Nyland, L. and Prins, J. KHEPERA: A System for Rapid Implementation of Domain-Specific Languages. In *Proc. USENIX Conference on Domain-Specific Language*, pages 243-255, October 1997.
7. Grosch, J. Puma - A Generator for the Transformation of Attributed Trees. Technical Report 26, Gesellschaft für Mathematik und Datenverarbeitung mbH, Forschungsstelle an der Universität Karlsruhe, November 1991.
8. Kiczales, G. and Lamping, J. and Mendhekar, A. and Maeda, C. and Lopes, C. and Loingtier, J. and Irwin, J. Aspect-Oriented Programming. Technical report, Xerox Palo Alto Research Center, 1997.
9. Milner, R. A Theory of Type Polymorphism in Programming. *Journal of Computer and Systems Sciences*, vol. 17, pp. 348-375, 1978.
10. Milner, R. and Tofte, M. and Harper, R. and MacQueen, D. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
11. Quinn, M. *Parallel Computing Theory and Practice Second Edition* Chapter 9. McGraw-Hill, Inc., 1994.
12. Tolmach, A. and Oliva, D. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 1998 (to appear).
13. Visser, E. and Benaissa, Z. and Tolmach, A. Building Program Optimizers with Rewriting Strategies. Submitted to *ICFP*, 1998.