Optimizing Compilation of CLP($\mathcal{R}$)

ANDREW D. KELLY and KIM MARRIOTT
Monash University
ANDREW MACDONALD and PETER J. STUCKEY
University of Melbourne
and
ROLAND YAP
National University of Singapore

Constraint Logic Programming (CLP) languages extend logic programming by allowing the use of constraints from different domains such as real numbers or Boolean functions. They have proved to be ideal for expressing problems that require interactive mathematical modeling and complex combinatorial optimization problems. However, CLP languages have mainly been considered as research systems, useful for rapid prototyping, but not really competitive with more conventional programming languages where efficiency is a more important consideration. One promising approach to improving the performance of CLP systems is the use of powerful program optimizations to reduce the cost of constraint solving. We extend work in this area by describing a new optimizing compiler for the CLP language CLP($\mathcal{R}$). The compiler implements six powerful optimizations: reordering of constraints, bypass of the constraint solver, splitting and dead-code elimination, removal of redundant constraints, removal of redundant variables, and specialization of constraints which cannot fail. Each program optimization is designed to remove the overhead of constraint solving when possible and keep the number of constraints in the store as small as possible. We systematically evaluate the effectiveness of each optimization in isolation and in combination. Our empirical evaluation of the compiler verifies that optimizing compilation can be made efficient enough to allow compilation of real-world programs and that it is worth performing such compilation because it gives significant time and space performance improvements.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*constraint and logic languages*; D.3.4 [**Programming Languages**]: Processors—*compilers*

General Terms: Languages, Performance

Additional Key Words and Phrases: Compilation, constraint logic programming, program analysis, program optimization, source-to-source program transformation

This work was partially supported by ARC grant A49531431. Author's addresses: A.D. Kelly and K. Marriott, Department of Computer Science and Software Engineering, Monash University, Clayton 2168, Australia; email: {`kelly,marriott`}`@cs.monash.edu.au`; A. Macdonald and P.J. Stuckey, Department of Computer Science, University of Melbourne, Parkville 3052, Australia; email: {`andrewdm,pjs`}`@cs.mu.oz.au`; R. Yap, Department of Information Systems and Computer Science, National University of Singapore, Lower Kent Ridge Rd, Singapore 119260; email: `ryap@iscs.nus.sg`.

1. INTRODUCTION

One of the most promising innovations in recent programming language design are constraint programming languages and, in particular, *constraint logic programming* (CLP) languages which combine constraints with logic programming (e.g., see Marriott and Stuckey [1998]). The key characteristic of these languages is a global constraint solver which is queried to direct execution and to which constraints are monotonically added.

CLP languages have proved to be ideal for expressing problems that require interactive mathematical modeling and for expressing complex combinatorial optimization problems. Major application areas have been electrical circuit analysis, synthesis and diagnosis, civil engineering, and mechanical engineering. This is because engineering applications tend to combine hierarchical composition of complex systems, mathematical or Boolean models, and—especially in the case of diagnosis and design—deep rule-based reasoning; hence they are particularly suited to constraint logic programming. Another major area has been options trading and financial planning where applications have taken the form of expert systems involving mathematical models. Other applications are in traditional operations research problems, such as resource allocation problems e.g. cutting stock and scheduling.

CLP languages have been considered mainly as research systems, useful for rapid prototyping, but not really competitive with more conventional programming languages when performance is crucial. This situation is not surprising, as current CLP systems are offshoots of first-generation research systems and since general-purpose constraint solving is expensive. The slowness of constraint solving is exacerbated by the addition without deletion of constraints, meaning that in current CLP systems the number of constraints in the global constraint solver grows rapidly as evaluation proceeds.[1]

One promising approach to improving the performance of CLP systems is the use of program optimizations to reduce the cost of constraint solving. We describe our experience with such an approach to the compilation of CLP($\mathcal{R}$). CLP($\mathcal{R}$) is the prototypical constraint logic programming language which extends Prolog by incorporating real arithmetic constraints.

Our optimizing compiler is a large piece of software, comprising around 54,000 lines of C++ code. It performs six different optimizations: reordering of constraints, bypass of the constraint solver, splitting and dead-code elimination, removal of redundant constraints, removal of redundant variables, and specialization of constraints which cannot fail. Optimization is performed in multiple phases and can be regarded as a source-to-source transformation of the intermediate language called CLIC. The associated analyzer is a generic tool supporting seven different analyses. In order to allow efficient optimization, it uses incremental updating of the results of the analysis whenever the CLP($\mathcal{R}$) program is modified by the optimizing compiler. Our main results are twofold:

—We systematically evaluate the effectiveness of each optimization and the cost of performing the optimization on a set of benchmarks.

[1] In principle, the constraint store grows monotonically during execution except that it can decrease upon backtracking.

—We consider each optimization in isolation and in combination.

Our empirical studies demonstrate that optimizing compilation is sufficiently fast for it to be employed in practical CLP compilation. In particular, three optimizations—bypass of the constraint solver, redundant-variable elimination and flagging that a constraint cannot fail (*nofail*)—can significantly improve the efficiency of the compiled CLP($\mathcal{R}$) code. Our results also demonstrate the usefulness of *multivariant specialization* [Winsborough 1992]. This is the process of duplicating (splitting or specializing) predicates for the different modes in which they are called and applying optimizations to each split copy, or variant, in a different way from any other split variant. Furthermore, our results show that the other more powerful optimizations of constraint reordering and constraint removal are less valuable, giving speedups only in the smaller, more mathematical programs.

This article continues our work on the implementation of CLP($\mathcal{R}$) [Jaffar et al. 1992a; 1992b; Macdonald et al. 1993] and on preliminary studies of each of the optimizations considered here [Jaffar et al. 1992a; Jørgensen et al. 1991; Macdonald et al. 1993; Marriott and Stuckey 1993; Marriott et al. 1994]. A preliminary version of these results appeared in Kelly et al. [1995,1996]. The incremental analyzer has been outlined in Kelly et al. [1997] and has appeared in an application for the detection of occur-checks in Prolog programs [Crnogorac et al. 1996].

Our compiler is related to experimental compilers for Prolog, which also make use of global program analysis [Muthukumar and Hermenegildo 1992; Taylor 1990; Van Roy and Despain 1990]. These too have given rise to good performance improvements. However these compilers are quite simple when compared to our compiler, as they apply only one or two noninteracting optimizations and only use simple nonincremental program analysis. Also the scope for performance improvement is greater in CLP languages because of the high cost of constraint solving.

Apart from our work, the only other work on optimizing compilation of CLP languages is that of Ramachandran and Van Hentenryck [1995] describing work in progress on a compiler for $CLP(RLin)$ and the CIAO system [Hermenegildo 1994]. $CLP(RLin)$ is essentially CLP($\mathcal{R}$) without Herbrand constraints or delayed nonlinear arithmetic constraints. Ramachandran and Van Hentenryck's compiler uses constraint reordering, constraint removal, and solver bypass (called refinement). They do not consider the nofail or redundant-variable removal optimizations and do not appear to perform multivariant specialization. Their compiler was developed independently and concurrently with the current version of our compiler. They give some experimental evaluation of the compiler but do not evaluate the effectiveness of each optimization in isolation, and the benchmarks used in their evaluation are rather small. Indeed, one of the findings of our empirical evaluation is that speedups encountered on small programs do not necessarily scale up to larger programs.

The CIAO system is a multidialect logic programming language which supports constraints. The optimizing compiler for CIAO is principally concerned with automatic parallelization of programs, and performs optimization of built-in predicates, so the optimizations performed are completely different from those described herein. The optimizer also supports multivariant specialization (see Puebla and Hermenegildo [1995,1997]) and makes use of incremental analysis [Hermenegildo et al. 1995]. The results from the CIAO system support our findings of the usefulness of mul-

```
%% Prin = principal, Time = no.  of months,
%% Rate = annual interest rate,
%% MP = monthly payment, B = balance.

mortgage(Prin, Time, Rate, MP, Bal) :-
   Time = 0,
   Prin = Bal.
mortgage(Prin, Time, Rate, MP, Bal) :-
   Time > 0,
   NTime = Time - 1,
   Int = Prin * Rate/1200,
   NPrin = Prin + Int - MP,
   mortgage(NPrin, NTime, Rate, MP, Bal).
```

Fig. 1. Mortgage program.

tivariant specialization and incremental analysis.

The remainder of the article is organized as follows. In Section 2 we describe CLP($\mathcal{R}$) using an example and briefly discuss the existing (nonoptimizing) compiler and abstract machine CLAM. Section 3 describes the suite of transformations by means of a simple example. In Section 4 we describe the optimizing compiler. Section 5 describes our experimental evaluation of the compiler, and Section 6 concludes.

2. CLP($\mathcal{R}$) AND EXISTING IMPLEMENTATION TECHNOLOGY

As in all CLP languages, execution of CLP($\mathcal{R}$) proceeds in Prolog style from an initial query or *goal* by repeatedly rewriting the goal using the rules in the program. Each rule H `:-` B may be used to replace the atom H in the goal by the body B of the rule. As constraints are encountered in the rules, they are added to the *constraint store*. The constraints in the store are always required to be consistent. If adding a constraint will lead to inconsistency, execution *backtracks* to the last point where there was a choice of rule for the rewriting step, and chooses a different rule.

Figure 1 shows a simple program for computing mortgage repayments. For example, one may ask "how much will I have to pay per month for an $80,000 mortgage of 30 years at 9.5%?" This is expressed by the goal `mortgage(80000, 360, 9.5, MP, 0)` which gives the answer `MP = 672.68`. Like pure-logic programs, a CLP($\mathcal{R}$) program can be used to answer many different types of goals. For example, instead of the goal above, we may ask a more complex question such as "what is the relationship between the principal, monthly payment, and the balance, given a 30-year mortgage at 15%?" The goal `mortgage(P, 360, 15, MP, B)` returns the appropriate relationship `P = 79.09*MP + 0.0114*B` as a constraint.

Constraint logic programs allow very simple and compact solutions to programming problems, as the same program can be used to answer many different types of goals. The above program really corresponds to a suite of conventional programs. However, there is a price to be paid for this flexibility—general-purpose constraint solving is expensive. This is especially true when the number of constraints in the store grows rapidly as evaluation proceeds. For example, in the above program, both goals generate more than 1,000 constraints. For this reason, efficient execution of constraint logic programs requires that the overhead of constraint solving and the number of constraints in the solver be kept as small as possible.

The first CLP($\mathcal{R}$) implementation was based on an interpreter which consisted of a PROLOG-like interpreter rewriting engine and a set of constraint solvers: a unification solver, simple arithmetic solver,[2] linear equation solver, linear inequality solver, and a nonlinear solver. The constraint solvers were combined with an interface which translated constraints into a canonical form suitable for the constraint solvers. They were also organized in a hierarchy: unification solver, simple arithmetic solver, linear equation solver, and linear inequality solver. The later solvers were more expensive to invoke than earlier ones. The interface sent constraints to the earliest solver in the hierarchy that could deal with the constraint. Thus, the more expensive solver was only invoked when the previous one was not applicable. For example, when solving a linear equation, the linear equation solver was often sufficient. Only when all the variables in the equation were involved in inequalities was it necessary to also use the inequality solver.

The current implementation of CLP($\mathcal{R}$) is a compiler-based system. It translates a program into code for an abstract machine, the CLAM. The abstract machine is an extension of the Prolog WAM architecture (e.g., see Aït-Kaci [1991]) to deal with arithmetic constraints. Since the constraint solvers deal principally with linear constraints, the main arithmetic instructions in the CLAM construct the data structures representing linear arithmetic forms. These data structures are in a form which can be used directly by the constraint solvers. The constraint-solving hierarchy that was used in the interpreter is retained, but is more effective, since some processing and other run-time decisions in the interpreter can now be shifted to compile-time.

To give a flavor of the CLAM (see Jaffar et al. [1992a] for details), let us describe the compilation of the constraint $5+X=Y$. Assume that X is a new variable and that the equation store contains $Y = Z + 3.14$. The following CLAM code would be generated:

`initpf`	5	*lf* :5
`addpf_var`	1,X	*lf* :$5+X$
`addpf_val`	-1,Y	*lf* :$1.86+X-Z$
`solve_eq0`		*solve* :$1.86+X-Z=0$

On the left are the CLAM instructions, while the right shows the effect on the constructed "linear form" (*lf*). The original constraint is rewritten into a linear canonical form, $5+X-Y=0$ to compile. The CLAM code executes as follows. First a new linear form is initialized to 5 (`initpf`), and X, being a syntactically new variable, is added directly (`addpf_var`). Then Y is added which entails adding its linear form (`addpf_val`), $Z + 3.14$. After the first three instructions have constructed a linear form, the last `solve_eq0` instruction is used to represent the equation *lf* $= 0$. In general, the `solve_eq0` may reduce to an assignment, a test, or a call to the equation solver. Equations are stored in the solver in a linear

[2]This could handle constraint checks (where all variables are fixed) and assignments (where one variable is new and the others fixed).

parametric solved form and inequalities in a variant of simplex solved form [Jaffar et al. 1992a; 1992b].

CLAM instructions operate at a granularity below that of a single constraint and take into account the operation of the constraint solver(s). This design allows for CLAM instructions to combine in various ways to optimize a single constraint. The highly optimizing compiler extends the earlier work on a *core CLAM* instruction set which is sufficient to execute a CLP($\mathcal{R}$) program together with some peephole optimizations and some rule-level optimizations. For example, when we know that a constraint is always satisfiable, it may be possible to decide at compile-time how the constraint is to be represented in the constraint store at run-time and simply add that data structure. The extension is obtained by adding new CLAM instructions to the core CLAM instruction set.

The released version of the CLP($\mathcal{R}$) compiler (v1.2) translates CLP($\mathcal{R}$) programs into a core set of CLAM instructions which can then be executed by an emulator. This is the baseline unoptimized reference which we use in the performance evaluation and empirical results. The system combines Prolog implementation technology with a specialized incremental constraint solver. Considerable effort has gone into the design and implementation of this compiler and the core CLAM instruction set, so that the compiler can perform many local optimizations. The constraint solver has been specialized to take advantage of simple cases, and, as mentioned earlier, employs a hierarchy of constraint solvers to reduce the cost of constraint solving.

The main factor limiting the quality of CLAM code generated by the released compiler (v1.2) is the lack of global analysis information and subsequent inability to perform multivariant code specialization. This consideration has led to the development of the new highly optimizing compiler described herein.

3. OPTIMIZATIONS

In this section we describe the operation of the optimizer and detail the suite of six optimizations it is based on by means of a worked example.

Consider the following CLP($\mathcal{R}$) program defining the relation `fib` where `fib(N,F)` holds if and only if `F` is the `N`th Fibonacci number. The program is for illustrative purposes only and is a direct translation of the usual (computationally inefficient) mathematical definition of Fibonacci numbers.

(FIB)

```
fib(N, F) :- N = 0, F = 1.
fib(N, F) :- N = 1, F = 1.
fib(N, F) :- N >= 2, F = F1 + F2,
             N1 = N - 2, N2 = N - 1,
             fib(N1, F1), fib(N2, F2).
```

Imagine that the compiler has been given this program in a file together with the declarations

```
:- export fib(N, F) where ground(N), free(F).
:- export fib(N, F) where true.
```

which tell the compiler to compile `fib(N,F)` for two modes of usage. In the first mode, `fib` is called with `N` a fixed value, (i.e., `N` is *ground*), and `F` is a variable which is unconstrained except that it may be aliased to other unconstrained variables (i.e., `F` is *free*). This mode of usage is intended to compute the `N`th Fibonacci number. In the second mode, there are no restrictions on how `fib` can be called. This second mode is useful for computing the first `N` Fibonacci numbers, or for checking if a number is in the Fibonacci sequence. We now detail in turn how the compiler will optimize `fib` for each of these modes.

3.1 Reordering of Constraints

The addition of a constraint to the store can be moved to a later point in execution if the constraint does not affect the control flow in the intervening computation. Control flow is only affected if the existence of the constraint in the store causes failure (by interaction with other constraints) and hence backtracking. The advantage of reordering is that it reduces the size of the constraint store and facilitates the other optimizations. As reordering delays constraint execution, it also has the advantage of saving work if the rule fails before the reordered constraint. Reordering requires determining possible interaction between constraints and hence possible unsatisfiability.

Consider the first mode of usage of `fib` and the third rule in the definition. Initially `F` is free. This means that the constraint `F = F1+F2` is satisfiable no matter what values `F1` and `F2` take. Thus the constraint `F = F1+F2` cannot cause failure in the rule body and so can be moved to the end of the rule. Because `N1` and `N2` are new, free variables, the constraints `N1 = N-2` and `N2 = N-1` can also be moved to just before the first point in which constraints are placed on `N1` and `N2`, respectively. The resulting program specialized for this usage is

(REO)
```
fib^gf(N, F) :- N = 0, F = 1.
fib^gf(N, F) :- N = 1, F = 1.
fib^gf(N, F) :- N >= 2, N1 = N - 2,
                fib^gf(N1, F1), N2 = N - 1,
                fib^gf(N2, F2), F = F1 + F2 .
```

Note that we use $\texttt{fib}^{gf}$ to indicate that `fib` has been called with the first argument ground and the second argument free. Recall that a *free* variable is a variable whose value is not constrained or restricted in any way. That is, the only constraints the variable may have appeared in are equality constraints with other free variables. The notation $\texttt{fib}^{ga}$ would indicate that the first argument was ground and that the second argument can be "anything," i.e., it can be either free or ground.

3.2 Bypass of the Constraint Solver

In many cases, by the time a constraint is encountered, it can be treated as a simple Boolean test or assignment. In this case a call to the solver can be replaced by the appropriate test or assignment. This can both decrease the size of the constraint store and remove calls to the solver. Our current implementation stores floating-point numbers in the solver; hence no space savings are obtained. Application of this optimization requires determining when variables are constrained to a unique value and when they are unconstrained.

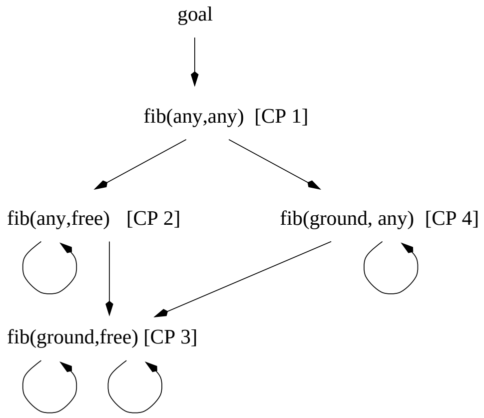

Fig. 2. Calling pattern graph for fib with second mode of usage.

Consider the execution of program **(REO)** in the first mode of usage. By replacing constraints with tests and assignments we can in fact remove all calls to the constraint solver. This results in the following program. Note that the instruction `>=`^{test} simply evaluates both sides and tests whether the `>=` relationship holds, while `=`^{assign} evaluates the right-hand side and sets the value of the left-hand side to this value. It is only because of reordering that it is possible to make `F = F1+F2` into an assignment.

(BYP)
```
fib^gf(N, F) :- N =^test 0, F =^assign 1.
fib^gf(N, F) :- N =^test 1, F =^assign 1.
fib^gf(N, F) :- N >=^test 2, N1 =^assign N - 2,
                fib^gf(N1, F1), N2 =^assign N - 1,
                fib^gf(N2, F2), F =^assign F1 + F2.
```

3.3 Splitting and Dead-code Elimination

In general the compiler will perform multivariant specialization by copying rule definitions for different patterns of usage if this enables more optimizations to be performed. This is called "splitting" and will continue until some upper limit of the number of versions of a rule is reached. Whenever a definition is split, bounds information can be used to eliminate those rules that cannot succeed. This is a form of dead-code elimination.

Now consider the second mode for `fib` where no information is known about the initial call. The compiler will first apply the reordering and solver bypass strategies. In this case as `F` is not free, `F = F1+F2` can only be moved before the second recursive call to `fib` (F2 is still free at this point). However, this movement means that `F1` in the first recursive call to `fib` is now free, which gives rise to a new pattern of usage for `fib` in which the first argument is unrestricted and the second is free. Also, after returning from the first call to `fib`, `N1` is ground, and hence `N` is ground. Thus `N2 = N-1` is an assignment, and on the second call to `fib` the first argument is ground, which is again a new pattern of usage. As both patterns of usage allow additional solver bypass optimizations, the compiler will "split" the definition of `fib` and perform multivariant specialization.

The calling pattern graph for this second mode of usage is shown in Figure 2. Note that the first calling pattern (CP 1) generates calls to the second and fourth calling patterns (CP 2 and CP 4). The second pattern leads to calls to itself and the third calling pattern. The third calling pattern generates two calls to itself, and the fourth pattern generates calls to calling pattern 3 and to itself.

(SPLIT)

```
fib(N, F) :- N = 0, F = 1.
fib(N, F) :- N = 1, F = 1.
fib(N, F) :- N >= 2, N1 = N - 2,
             fib^af(N1, F1), N2 =^assign N - 1,
             F = F1 + F2, fib^ga(N2, F2).
fib^af(N, F) :- N = 0, F =^assign 1.
fib^af(N, F) :- N = 1, F =^assign 1.
fib^af(N, F) :- N >= 2, N1 = N - 2,
                fib^af(N1, F1), N2 =^assign N - 1,
                fib^gf(N2, F2), F =^assign F1 + F2.
fib^ga(N, F) :- N =^test 0, F = 1.
fib^ga(N, F) :- N =^test 1, F = 1.
fib^ga(N, F) :- N >=^test 2, N1 =^assign N - 2,
                fib^gf(N1, F1), N2 =^assign N - 1,
                F = F1 + F2, fib^ga(N2, F2).
fib^gf(N, F) :- N =^test 0, F =^assign 1.
fib^gf(N, F) :- N =^test 1, F =^assign 1.
fib^gf(N, F) :- N >=^test 2, N1 =^assign N - 2,
                fib^gf(N1, F1), N2 =^assign N - 1,
                fib^gf(N2, F2), F =^assign F1 + F2.
```

This example did not demonstrate dead-code elimination. However, if the original call to `fib` contained the constraint `N >= 2`, `fib(N,F)` then splitting would occur as above, except that the first two rules for `fib` would be eliminated, as they cannot succeed for this particular calling pattern. However, variants of the first two rules for `fib` will exist for the subsequent recursive calls. It is often the case that the original call to a recursive predicate is specialized in a different way, that is, using a different split version, to the recursive calls. This is typically where dead-code elimination occurs.

3.4 Constraint Removal

A major source of inefficiency in the solver is caused by constraints which, after addition of other constraints to the solver, have become redundant, in the sense that their information is implied by these other constraints in the current constraint store. Execution can be optimized by adding instructions which remove these constraints from the store, as this decreases the size of the constraint store but cannot change the behavior of the program. In many cases this removal can occur even before the constraint becomes redundant as the constraint cannot cause failure between the time of removal and when it becomes redundant. An important case is when the constraint can be removed immediately after it is added to the solver. In this case the constraint is tested for satisfiability with respect to the current constraints, and then not added to the solver. This is the best case for removal, and is handled specially by the solver. This is called *future redundancy* [Jørgensen et al. 1991].

Consider the constraint `N >= 2` in the recursive rule of $\texttt{fib}^{af}$ in **(SPLIT)**. In the call to $\texttt{fib}^{af}$`(N1,F1)` the first constraint encountered is one of `N1 = 0`, `N1 = 1`, or `N1 >= 2`. But `N1 = N-2`, so in each case the constraint `N >= 2` is made redundant. Hence it can be removed before the call to $\texttt{fib}^{af}$`(N1,F1)` without affecting execution. Thus `N >= 2` is future redundant, and the compiler will transform $\texttt{fib}^{af}$ in **(SPLIT)** into **(REM)** below. Note that it is still necessary to check the future redundant constraint for satisfiability but it need not be kept in the store.

(REM)

$\texttt{fib}^{af}\texttt{(N, F) :- N = 0, F} =^{assign} \texttt{1.}$
$\texttt{fib}^{af}\texttt{(N, F) :- N = 1, F} =^{assign} \texttt{1.}$
$\texttt{fib}^{af}\texttt{(N, F) :- }$ *add_remove*`(N >= 2), N1 = N - 2,`
$\quad\quad\texttt{fib}^{af}\texttt{(N1, F1), N2} =^{assign} \texttt{N - 1,}$
$\quad\quad\texttt{fib}^{gf}\texttt{(N2, F2), F} =^{assign} \texttt{F1 + F2.}$

Application of future redundancy requires (1) knowledge about possible interaction between constraints and (2) tests for unsatisfiability of constraints. In the above example the call to $\texttt{fib}^{af}$ is unfolded, and the compiler checks that `N >= 2` is made redundant before subsequent atoms. More exactly the compiler checks the following:

$\texttt{N1 = N-2} \wedge \texttt{N1 = 0} \wedge \texttt{F1 = 1}$ implies $\texttt{N >= 2}$
$\texttt{N1 = N-2} \wedge \texttt{N1 = 1} \wedge \texttt{F1 = 1}$ implies $\texttt{N >= 2}$
$\texttt{N1 = N-2} \wedge \texttt{N1 >= 2} \wedge \texttt{N1}' \texttt{ = N1-2}$ implies $\texttt{N >= 2}$.

This is tested by checking that the constraints

$\texttt{N1 = N-2} \wedge \texttt{N1 = 0} \wedge \texttt{F1 = 1} \wedge \texttt{N < 2}$
$\texttt{N1 = N-2} \wedge \texttt{N1 = 1} \wedge \texttt{F1 = 1} \wedge \texttt{N < 2}$
$\texttt{N1 = N-2} \wedge \texttt{N1 >= 2} \wedge \texttt{N1}' \texttt{ = N1-2} \wedge \texttt{N < 2}$

are all unsatisfiable. This optimization is only applied to arithmetic inequalities, since the solved form of equations employed in the constraint solver is free from this form of redundancy.

Currently we only support constraint removal by future redundancy. Our experience has been that future redundancy captures most cases for removal, although we intend to investigate arbitrary constraint removal.

3.5 Dead Variables

Another common source of redundancy in the constraint solver is caused by variables which will never be referred to again, so-called *dead* variables [Macdonald et al. 1993]. Execution can be improved by adding instructions which eliminate these variables from the current constraints in the store as this helps keep down the size of the constraint store. Clearly this optimization requires determining which variables are still alive and is useful because it reduces the number of variables and constraints in the solver. This can be viewed as a form of compile-time semantic garbage collection of the constraint store.

Consider the calls to `fib`^{ga}`(N2,F2)` in **(SPLIT)**. After each call the variable `F2` is never again referred to and so is dead on return from the call. Thus, the variable `F2` in those calls, which is locally referenced as `F` in the definition of `fib`^{ga}`(N, F)`, can be removed from the constraint solver. This gives

(REV)

$\texttt{fib}^{ga}\texttt{(N, F) :- N} =^{test} \texttt{0, F}^{rem} \texttt{ = 1.}$
$\texttt{fib}^{ga}\texttt{(N, F) :- N} =^{test} \texttt{1, F}^{rem} \texttt{ = 1.}$
$\texttt{fib}^{ga}\texttt{(N, F) :- N} >=^{test} \texttt{2, N1} =^{assign} \texttt{N - 2,}$
$\qquad\qquad \texttt{fib}^{gf}\texttt{(N1, F1), N2} =^{assign} \texttt{N - 1,}$
$\qquad\qquad \texttt{F}^{rem} \texttt{ = F1 + F2, fib}^{ga}\texttt{(N2, F2).}$

In general determining when a variable is guaranteed never to be referred to again requires not just its last textual occurrence to be past but also that the variable not be accessed via shared data structures and that it not be in a delayed nonlinear constraint (for details see Macdonald et al. [1993]). Determining when these conditions hold requires global analysis for several different types of information, as is shown later in Figure 4.

Dead-variable elimination requires projecting a variable from the store, which could be a very expensive operation. We restrict ourselves to only eliminating variables appearing in equations. This can be managed by a Gaussian elimination step and is quite efficient.

3.6 Nofail Constraints.

Sometimes, when a constraint is encountered, it can be guaranteed not to fail because of the presence of free variables. The existing solver detects some such constraints at run-time due to the presence of new variables and uses this information to solve the constraint quickly. There is, however, still an overhead in detecting the possibility and manipulating the constraint into the required form. If the information about free variables is collected at compile-time, we can produce specialized instructions that reduce this overhead as well as allowing optimization in cases when the free variable guaranteeing that the constraint cannot fail is not a new variable. For an example (using nofail) we see that `fib(N,F)` can be transformed to:

(NOF)

$\texttt{fib(N, F) :- N = 0, F = 1.}$
$\texttt{fib(N, F) :- N = 1, F = 1.}$
$\texttt{fib(N, F) :- } \textit{add\_remove}\texttt{(N >= 2), } \textit{nofail}\texttt{(N1 = N - 2),}$
$\qquad\qquad \texttt{fib}^{af}\texttt{(N1, F1), N2} =^{assign} \texttt{N - 1,}$
$\qquad\qquad \textit{nofail}\texttt{(F2 = F - F1), fib}^{ga}\texttt{(N2, F2).}$

Thus we have seen that information about call patterns and multivariant specialization has allowed us to optimize the definition of `fib`, even in the case that we make no assumptions about the mode of usage.

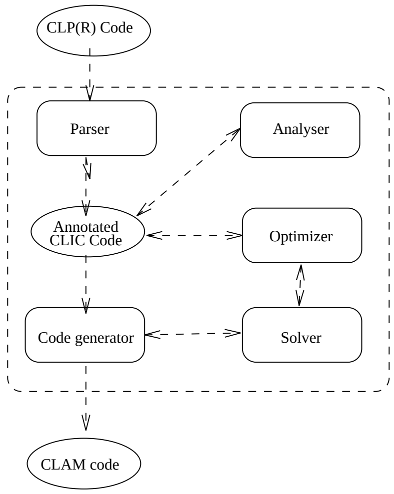

Fig. 3. The optimizing compiler.

4. THE COMPILER

In the previous section we described the transformations and the basic operations of the optimizing compiler. In this section we describe in more detail the design of the optimizing compiler.

The optimizing compiler has four main components (see Figure 3): the *optimizer* which performs the optimizations; a *global analyzer* and a *constraint solver* which provide information to guide the optimizer; and a *code generator* which produces CLAM abstract machine code.

The main complication in the design of the optimizer is the number of different optimizations. This is exacerbated by nontrivial interaction between the optimizations. Performing one transformation in one part of the program may preclude performing a different transformation in another part of the program. Another complication is the need for multivariant specialization. Our solution to these potential difficulties is to uniformly view all optimizations as source-to-source transformations on an intermediate language called CLIC, standing for Constraint Logic Intermediate Code. Optimizations consisting of transformations and multivariant specialization are applied to the CLIC code in multiple phases. This allows for a simple treatment of the optimizations and facilitates experimentation, as it is easy to apply optimizations in different orders and to add new optimizations. Both the optimizer and code generator make use of the CLP($\mathcal{R}$) run-time solver to reason about constraints in the CLIC program.

CLIC is an extension of CLP($\mathcal{R}$) which also provides the usual imperative arithmetic commands and various specialized solver instructions, a superset of the program annotations described in Section 3. Indeed CLIC can be thought of as a hybrid imperative-logic constraint programming language. CLIC is an intermediate language which is usable by the analyzer, optimizer, and code generator and not intended for direct use by the programmer as incorrect use of CLIC will destroy the declarative properties of a program. One important feature of CLIC is that it provides commands to remove constraints and variables from the global constraint solver. Thus execution of a CLIC program does not necessarily lead to more and more constraints in the constraint solver. Indeed, a major role of the transformations in the compiler is to transform a CLP($\mathcal{R}$) program with monotonic constraint addition into an equivalent CLIC program in which the number of constraints in the global solver is bounded.

In fact, the optimizer does not directly interact with the CLIC code; instead it works with an "annotated" CLIC program in which each program point in the program has an annotation which describes the constraints which will be encountered at this point during run-time. It is the role of the global analyzer to keep the annotation up to date whenever the optimizer changes the underlying CLIC program. For efficiency therefore, the analyzer is incremental and only reanalyzes those parts of the program which have been changed.

4.1 The Analyzer

The global analyzer consists of a generic abstract interpretation engine, similar to analysis engines such as PLAI [Muthukumar and Hermenegildo 1992] and GAIA [Le Charlier and Van Hentenryck 1994] originally developed for Prolog and now extended for constraints. The generic analysis engine is designed not only to perform many different analyses but also to allow for the easy addition of new analyses. The core of the analyzer is an algorithm for efficient fixpoint computation. Efficiency is obtained by keeping track of which parts of a program must be reexamined when a success pattern is updated. It is used together with seven different but interacting analysis domains.

The analyzer performs global analysis of programs (inter- and intra-procedural analysis); that is, program information (descriptions) is inferred about the calls "between" different rules of the program (as well as inferring the local information "within" each rule). The global analysis is based on abstract interpretation of constraint logic programs [Marriott and Søndergaard 1990; García de la Banda et al. 1996] in which operations in the execution of the goal are mimicked by abstract operations on the domain of descriptions (the analyses).

Conceptually, the analyzer takes a program and goal and annotates each point in the program with an approximate description of the constraints which will be encountered at that point when the goal is executed.

As an example consider the following Fibonacci program. If this program is analyzed for the class of calls in which the first argument is ground, then the following annotated program results. The description domain consists of Boolean functions which capture groundness information about variables and definite dependencies among variables [Armstrong et al. 1994]. For example, the function $N \wedge (F \leftrightarrow F2)$ indicates that the variable N is ground, and that if F is ever ground, then so is $F2$ and vice versa. The column to the right gives the dependency description of the rule variables after the corresponding statement in the program.

Program	**Annotated Program Point**
`fib(N, F) :-`	$\{N\}$
`    N = 0,`	$\{N\}$
`    F = 1.`	$\{N \wedge F\}$
`fib(N, F) :-`	$\{N\}$
`    N = 1,`	$\{N\}$
`    F = 1.`	$\{N \wedge F\}$
`fib(N, F) :-`	$\{N\}$
`    N >= 2,`	$\{N\}$
`    N1 = N - 2,`	$\{N \wedge N1\}$
`    N2 = N - 1,`	$\{N \wedge N1 \wedge N2\}$
`    fib(N1, F1),`	$\{N \wedge N1 \wedge N2 \wedge F1\}$
`    F = F1 + F2,`	$\{N \wedge N1 \wedge N2 \wedge F1 \wedge (F \leftrightarrow F2)\}$
`    fib(N2, F2),`	$\{N \wedge N1 \wedge N2 \wedge F1 \wedge F2 \wedge F\}$

For example, initially, when the third rule is entered, `N` is ground. The statement `N >= 2` does not change this. After the statement `N1 = N-1`, as `N` is ground, `N1` becomes ground. Similarly, after `N2 = N-1`, `N2` becomes ground. The effect of the call `fib(N1,F1)` is to ground `F1`. The statement `F = F1+F2` adds the information that `F` is ground if and only if `F2` is ground. The effect of the call `fib(N2,F2)` is to ground `F2` and hence to ground `F`.

Actually the analyzer does not exist as a separate entity in the compiler. Rather it is associated with the *AnnotatedProgram* class of which the current annotated CLIC program is an instance. The optimizer obtains analysis information by way of methods on *AnnotatedPrograms*. These methods are divided into two groups.

The first group of five methods provides information for a given program point in some rule, either for a single calling pattern for that rule, or for all such calling patterns. The methods respectively return the set of *ground arithmetic* variables, the set of *free* variables, the set of variables which are *nofail* for a particular constraint,[3] the set of variables which are *dead* in the sense that they will not be referenced directly or indirectly in the future after the next constraint, and for each arithmetic variable, the real interval it is constrained to lie in.

The second group of methods on the annotated CLIC program allows the optimizer to modify the CLIC program and associated goal. There are methods to split atom definitions for different call patterns and methods which reorder and remove constraints in the body of a rule. There is also a method which requests the analyzer to annotate a hypothetical goal. Such hypothetical reasoning is used to determine the applicability of reordering and allows the optimizer to obtain answers to questions of the form "what happens if this constraint is removed?"

The analyzer has two special features. First, the analyzer is incremental, and hence allows efficient reanalysis of modified programs and efficient analysis of hypothetical goals. Whenever the underlying program is modified by the optimizer it is not reanalyzed from scratch. Second, it handles "widening" (e.g., see Cousot and Cousot [1992]), which means that the analysis will terminate for description domains such as arithmetic intervals which have infinite ascending chains of descrip-

[3]A variable is called *nofail* at a program point before a constraint if it is free and is not aliased to any other variable appearing in the constraint. This is in fact the property we require to perform the nofail constraint optimization; simple freeness is not sufficient.

tions. Details about the algorithms used can be found in Hermenegildo et al. [1995] and Kelly et al. [1997]. The only other incremental generic logic program analyzer, that we are aware of, is PLAI [Muthukumar and Hermenegildo 1992] which also handles multivariant specialization incrementally, but does not handle hypothetical goals or widening.

There are three broad types of incremental events—*hypothetical goals, rule modifications*, and *splitting*—that are handled by the analyzer.

A *hypothetical goal* is a series of literals similar to a rule except that a head atom is not required. Therefore, the new rule is not reachable from any other point in the annotated program. It is added temporarily and can be analyzed for various calling patterns. Hypothetical goals, and all the new calling patterns they can create when analyzed, are removed after use to leave the annotated program in the original state. For efficiency reasons, this cleanup is normally performed after several hypothetical goals have been used.

The second kind of incremental event, *rule modifications*, occurs when the optimizer actually determines that some change to the program should take place. Two typical events are the reordering of a constraint and the removal of a constraint. Both these events require partial reanalysis to update the annotation information associated with the program. It is important to note that these reordering and removal events are only initiated when the answer patterns for the rule are not affected by the removal or reordering. In this way, we know that the analysis of the rest of the program remains correct.

The final incremental event occurs when *splitting* (duplicating) certain rules because multivariant specialization is being performed. At every program point the analyzer keeps a set of annotations corresponding to the different calling patterns in which the rule has been called. It does not combine these annotations to form the most general annotation that applies to all calling patterns. By maintaining a list, splitting is performed easily, as it involves simply copying the rule (with a new predicate name), and copying the corresponding program point annotations for the calling pattern which is being split. In this way, a split involves no new analysis, and a new modified program is created with minimal effort.

For *hypothetical goals* and *rule modifications*, the rerunning of the analysis heavily reuses all analysis information, and the fixpoint is typically reached much sooner than if the algorithm had been run from scratch. This is particularly true for a large program, where a minor change to a rule may require little reanalysis.

Another important part of the analyzer is the description domains. Details of the description domains are deliberately kept insulated from the optimizer so as to make it easier to change them. The most complex condition to analyze for is to determine if a variable is *dead*. This is because not being textually alive may not mean a variable is dead, for two reasons. The first reason is that "structure sharing" between terms allows variables to be accessed indirectly. The second reason is that "hard" constraints may be delayed by the constraint solver until they become simple enough to solve. This means that in the analyzer variables in hard constraints must be assumed to be alive unless, using information about groundness, we can guarantee the constraint is simple enough to be solved in the linear constraint solver. In particular the constraint solver in CLP($\mathcal{R}$) delays consideration of nonlinear arithmetic constraints until they become linear. This highly dynamic behavior must be modeled in the analysis. Currently the analyzer uses descriptions which are tuples of seven different domains:

Pos This domain captures groundness information about variables. We use reduced ordered binary decision diagrams (ROBDDs) [Armstrong et al. 1994] to represent the Boolean functions.

CallAlive This consists of lists of variables which may be directly referenced later in execution. For example, in the goal

```
X = 1, p(X,Y), q(Y)
```

both `X` and `Y` are initially textually alive; for the call `p(X,Y)` only `Y` is alive (that is, will be referenced after the call). For the call `q(Y)` no variables are textually alive.

Shar This captures information about possible structure sharing of variables between Prolog terms. It is based on the description domain introduced by Søndergaard [1986] for eliminating occur checks in Prolog. The description consists of a possible sharing relation for variables. Consider the goal

```
Y = f(X), p(X), Y = f(Z), q(Z).
```

After `Y = f(X)`, `X` and `Y` possibly share, but `Z` does not share with anything else. After `Y = f(Z)`, all variables possibly share. Note that arithmetic constraints do not cause sharing to take place. A special variable $\perp$ represents "hidden" alive variables. Thus if a variable shares with $\perp$ it cannot be dead.

Free The freeness domain consists of lists of variables which are free. It makes use of *Shar* to keep track of variable aliasing.

NonLin This consists of a list of variables which are possibly contained in a delayed nonlinear constraint. When a nonlinear constraint is first encountered, groundness information is used to check if it is definitely linear. If it is not, then the analyzer assumes it is nonlinear and adds its variables to the *NonLin* list. For instance, in `mortgage`, when the statement `I = P * R` is reached with `R` ground, the analyzer will determine that `I = P * R` is linear, so no variables will be added. If none of `I`, `P`, or `R` were ground then all three variables would be added to the *NonLin* list. A variable in the *NonLin* list is never dead.[4]

Type This indicates whether a variable is definitely arithmetic or possibly involved in term constraints. The implementation maintains two lists of variables. More accurate type information is obtained by keeping track of aliasing between variables for cases like `X = Y` when no type information is known about `X` or `Y`. This can be done efficiently by integrating type analysis with freeness analysis.

Bounds This gives an interval for each arithmetic variable in which the variable's value must lie. Widening is used with this domain to ensure termination.

[4]Note that this is a simplification of the approach of Macdonald et al. [1993].

There is frequent interaction between many of these domains. The actual implementation combines *Shar*, *Free*, and *Type* to form one domain. This leads to a more efficient implementation.

4.2 The Optimizer

The optimizer examines the rules of the annotated CLIC program and where possible performs optimizations upon them. The optimizer is designed to enable experimentation with different optimizations. In effect, a sequence of optimization phases can be defined, each of which determines the analyses required for that phase and the optimizations to be performed in that phase. Splitting may be performed or not, depending on whether splitting allows further optimization.

For example, the optimizer performs all the currently implemented optimizations in two phases. In the first phase the constraint reordering optimization is performed; for this optimization only a subset of the analysis domains is required. Reanalysis of the modified program sections is then performed, and the analysis information not required during the reordering phase is added. Then in the second phase the solver bypass, future redundancy, dead variable, and nofail optimizations are performed. Definition splitting (and dead-code elimination) are performed during both phases to maximize the number of optimization opportunities. In the second phase, if competing optimizations are available they are currently handled as follows: *solver bypass* is preferred to *nofail*, *add_remove*, and *dead*. Thus, for example, a *dead* optimization will not be applied in cases where the dead variable is always ground. This decision is based on the effects of the optimizations in the run-time system. Removing a fixed dead variable will not increase the speed of future constraint solving, but will cause extra overhead if backtracking occurs. Using assignment is preferable to *nofail*, and using tests is preferable to *add_remove*, since they do not involve the solver at all.

In each phase the optimizer examines and optimizes one strongly connected component (SCC) in the program call graph. The SCCs are examined in order from the bottom of the call graph upward, so that predicates at the bottom of the call graph are optimized first. This decision implements the heuristic that optimizations at a lower level are likely to be more important than at a higher level, since lower-level predicates are (in general) executed more frequently. In general, optimizations made for one predicate may prevent optimizations for other predicates. For example, constraint removal in one predicate can prevent constraint removal in another.

When optimizing the rules in a single SCC, the optimizer first performs optimizations which are valid for all calling patterns of each predicate. It scans each rule examining each constraint for possible optimizations. Also, if an atom (defined in a lower SCC) is present in the rule which is always called using a calling pattern for which the optimizer has created a specialized version, then the atom is replaced by a call to the specialized version.

Next the calling pattern graph is examined. If there are multiple calling patterns for any predicates in the SCC of the call graph under consideration, the optimizer examines each SCC of the calling pattern graph in turn, again bottom up. If, for a particular calling pattern, new optimizations are available, either because a new constraint optimization is possible or because an atom may be replaced with a more

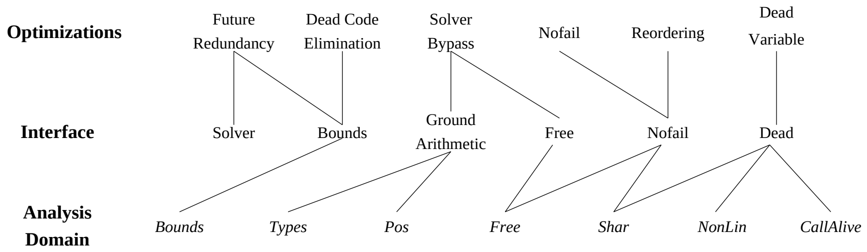

Fig. 4. Relationships between analysis domains and optimizations.

specific version, the optimizer creates a new version of the predicate for this specific calling pattern.

Once the optimizer has determined that a predicate must be split, it constructs a new copy of the code—containing the optimizations already made for all calling patterns—and optimizes this code further; that is to say, multivariant specialization is performed. Analysis information for the new code is extracted from the original. In general, some reanalysis may be required. Splitting an atom definition may allow further optimization of rules in the calling pattern SCC which have already been optimized. Thus for maximum improvement we should continue the process of splitting and optimizing until a fixpoint is reached. However, for simplicity, the optimizer presently examines each calling pattern only once in each phase. In practice, we have found this finds all possible optimizations.

The compiler determines the applicability of a particular optimization by querying the annotation information at program points to find variables which are free, ground arithmetic, dead, and nofail and to find the bounds on variables. Future redundancy also makes use of the constraint solver. The dependencies among the constraint solver, analysis domains, annotated program interface, and the optimizations are detailed in Figure 4.

Some of the analyses are quite complex. For example, to reorder a *nofail* constraint the optimizer creates as a hypothetical goal the remainder of the rule after that constraint. It then asks the analyzer for this goal to be annotated for the calling pattern occurring before the constraint. This mimics the behavior of the remainder of the rule if the constraint were removed. By examining the annotations of this hypothetical goal the optimizer can determine the last point where the *nofail* constraint is still guaranteed not to fail, and then move it to that position. It will not move it to the last position and break tail recursion unless this gains a solver bypass optimization. The old rule is replaced by the new rule with the constraint moved, and the analyzer updates the annotations. The future redundancy optimization is performed by examining an inequality to determine whether it is made redundant by the constraints occurring between it and the next atom A in the rule, together with the constraints occurring before any atoms in the rules for atom A and the inequalities determined by the bounds information. If the constraint is made redundant by each such combination of constraints (which is determined using the constraint solver) it is marked as *add_remove*.[5]

[5]Note that this strategy could sometimes delay failure, and hence is not guaranteed to always

Table I. Benchmark Descriptions

`fib`-*forw*	Fibonacci program for ground-free usage ($N = 18$).
`fib`-*back*	Fibonacci program for free-ground usage ($F = 2584$).
`mggnd`	Mortgage program to calculate balance given other variables are ground.
`sumlist`-*gnd*	Adds up 600 numbers from a list.
`sumlist`-*var*	Adds up variables from a list (600 elements), then grounds them.
`ackerman`	Computes Ackerman's function (3 and 3.)
`ode`	Solves ordinary differential equations.
`queens`	Solves the N Queens problem ($N = 8$).
`matmul`	Finding the inverse of a matrix.
`mg-extend`	Handles complex mortgages with provisions for a number of payments per month and special initial conditions and provisions.
`money`	SEND + MORE = MONEY cryptarithmetic puzzle.
`ladder`	Builds simple circuits.
`circuit`	Performs simple circuit analysis.
`pic`	A picture specification language.
`bridge`	Generates a finite-element model of a bridge.
`neural`	A neural net training program.
`amp`	Amplifier design and analysis.

4.3 Code Generator

The code generator maps the CLIC code into CLAM instructions which are executed by the CLAM emulator. The released version of the CLP($\mathcal{R}$) compiler (v1.2) also produced CLAM code, but used only a core set. The new compiler makes use of extended CLAM instructions for achieving the optimizations that are made possible by global analysis. The CLAM architecture is suited to this, as it operates below the level of a constraint, and the optimizations described require modifying the operation of constraint solving to be effective. Thus, they slot neatly with the rest of the CLAM architecture.

5. EMPIRICAL RESULTS

To illustrate the effects of the optimizations, we show their impact on a number of different programs and goals. The set of benchmark programs and goals are described in Table I. The benchmarks come from the CLP($\mathcal{R}$) library contributed from various CLP($\mathcal{R}$) users. They range from small, rather artificial programs, to larger typical "real-world" programs which make extensive run-time use of constraints. The sample has been selected as being representative of typical user-written CLP($\mathcal{R}$) program and applications.

Table II shows the size of the benchmarks after code which is unreachable from the goal is removed. It is measured by the number of rules in each program, the number of literals, and the maximum number of variables in a rule in the program. In addition it shows an approximate percentage of execution time that the original unoptimized program spends in the solver. As the optimizations discussed in this article principally apply to arithmetic constraints, the potential improvement is

improve performance; however, this will only occur in nonsensical programs and does not occur in any of the programs we examined.

Table II. Benchmark Statistics

	Rules	Literals	Max Vars in a Rule	Solver (% Used)
`fib`	4	16	6	73
`mortgage`	4	18	8	64
`sumlist`	9	24	5	100
`ackerman`	4	24	6	91
`ode`	7	25	9	59
`queens`	12	35	8	54
`matmul`	13	46	10	85
`mg-extend`	12	49	15	77
`money`	14	52	17	46
`ladder`	14	62	14	75
`circuit`	17	62	13	42
`pic`	11	68	24	57
`bridge`	19	97	20	67
`neural`	75	241	14	39
`amp`	62	326	30	78

limited by the time of execution spent outside the solver. All the benchmarks do make substantial use of arithmetic constraints with the amount of execution time being spent on unoptimized programs varying between 39% to practically 100% in the constraint solver.

The timings measured here are for running unoptimized and optimized CLAM code on a portable emulator written in C. Thus there is still the overhead of the CLAM emulator itself. To give an idea of the performance of the emulator, it is similar in speed to other Prolog WAM emulator-based systems.

To measure the effect of the optimizations we have performed a number of experiments. All the experiments were performed on a SUN SPARCstation 5 with 64MB of memory running Solaris 2.4. We optimized each program in the test suite using only one of the optimizations: future redundancy, solver bypass, nofail, reordering and dead-variable elimination. We tried these experiments with and without splitting enabled. In addition we tried the optimizations in combination. Beginning with solver bypass we added the optimizations in the following order: nofail, dead-variable removal, future redundancy, and finally reordering. Each combination was performed with splitting enabled. This corresponds to introducing the optimization methods in order of sophistication.

We do not give separate statistics for dead-code elimination, since it has had an insignificant effect on the execution statistics. For the benchmarks, dead-code elimination is only occasionally required. When applicable, it removed unreachable code built in the process of optimization, and in a few cases, removed rules which were not applicable for the first call to a specialized version of a recursive predicate.

In the following tables the execution time improvements are given with respect to the original unoptimized program. The peak space consumption of the constraint solver is measured in solver nodes (six words), and the figures are the percentage of the original space used by the optimized programs.

Table III gives speedups in execution time, (to the nearest 0.05), using the optimizations individually, with splitting enabled and disabled. A — entry indicates

Table III. Impact of Individual Optimizations (Speedups)

Program	Byp		Nofail		Dead		FR		Reo	
fib-*forw*	1.20		1.10	1.05	1.15	—	—		1.85	1.70
fib-*back*	1.35	≈	1.10		1.05	—	≈		1.25	1.15
mggnd	2.80		1.35		≈		≈		—	
sumlist-*gnd*	≈		1.10		1.30	—	≈	—	1.10	—
sumlist-*var*	≈		≈		290.4	—	≈		—	
ackerman	1.10		≈		—		1.40		—	
ode	≈		1.50		≈		—		1.05	
queens	3.20		1.20		—		—		—	
matmul	≈		1.05		1.35		—		—	
mg-extend	1.15		1.15		≈		≈		≈	
money	2.10	1.10	1.20	≈	—		—		—	
ladder	1.05		1.05		1.15		—		≈	
circuit	≈		1.05		1.10		—		—	
pic	1.25		1.20	1.10	≈	—	—		—	
bridge	2.95	2.00	1.20		≈		—		≈	
neural	1.10	1.05	1.10		1.05		≈		≈	
amp	≈		≈		1.25		—		≈	

that the optimization was not present. An ≈ entry indicates where the optimization did occur, but did not lead to any measurable improvement. Where splitting made a difference there are two entries in the column; the first is the result with splitting, the second without. Otherwise there is a single entry. It is clear from the table that each of the optimizations can give substantial improvements. Used individually, the maximum improvements were: solver bypass 3.20 (speedup), nofail 1.50, future redundancy 1.40, reordering 1.85, and dead-variable elimination 1.35 (ignoring `sumlist-`*var*).

Except for solver bypass, each of the optimizations has the capacity to change the parametric form of the solver. This can substantially modify the amount of work required by the solver. Hence they are not always guaranteed to lead to improved performance. Slowdown caused by this behavior is rare, but can be seen in the combined optimization of `ode` (see Table IV). Dead-variable elimination applied alone, and with splitting, to `sumlist-`*var* shows the best case of this behavior: the parametric form decreases from linear size to constant, giving a quadratic-to-linear improvement in the overall time taken.

The results show that the first three optimizations—solver bypass, nofail, and dead-variable elimination—are widely applicable and give moderate improvements in execution time in most cases. Solver bypass is the most significant, and for the benchmarks which merely use the solver as an arithmetic calculator, that is, `mggnd, queens`, and `money`, it provides all the improvement (although nofail can also gain part of this improvement). Reordering and future redundancy are far less applicable than the other optimizations. However, they have the potential to give significant performance gains when they arise. Most cases of future redundancy that were found to occur in our benchmarks happened to occur when the variables were ground. Hence, no improvement was gained.

The effect of splitting on the individual optimizations in terms of execution time was surprisingly limited, since it did not have an appreciable effect in many cases.

Table IV. Impact of Combined Optimizations (Speedups)

Program	Byp	+ Nofail	+ Dead	+ FR	+ Reo	All Split	All Nonsplit	Solver Speedup
fib-*forw*	1.20	1.25	1.50	—	5.20	5.20	2.50	20.0
fib-*back*	1.35	≈	1.50	1.55	3.60	3.60	1.45	98.6
mggnd	2.80	—	—	—	—	2.80	2.80	667
sumlist-*gnd*	≈	1.15	1.35	—	3.15	3.15	1.10	14.9
sumlist-*var*	≈	≈	23.2	—	—	23.2	1.00	23.2
ackerman	1.10	—	—	1.60	—	1.60	1.60	1.68
ode	≈	1.50	1.40	—	≈	1.40	1.40	1.90
queens	3.20	—	—	—	—	3.20	3.20	∞
matmul	≈	1.05	1.45	—	—	1.45	1.40	1.54
mg-extend	1.15	1.30	1.40	—	1.50	1.50	1.50	1.79
money	2.15	—	—	—	—	2.15	1.15	∞
ladder	1.05	1.10	1.25	—	≈	1.25	1.25	1.33
circuit	≈	1.05	1.15	—	—	1.15	1.15	1.32
pic	1.25	1.35	1.40	—	—	1.40	1.25	1.94
bridge	2.95	≈	—	—	≈	2.95	1.95	32.1
neural	1.10	1.15	1.20	—	≈	1.20	1.20	1.80
amp	≈	≈	1.25	—	≈	1.25	1.25	1.36

However, it was advantageous for others. It made significant differences using solver bypass for bridge and money. Using nofail optimization, pic was improved, and for reordering a difference was obtained with fib-*forw*, fib-*back*, and sumlist-*gnd*. With future redundancy, splitting made no appreciable difference. Dead-variable elimination was the most affected by splitting, since it was enabled in five cases. Interestingly, reordering requires splitting to gain the greatest advantage, because often the reordering optimization can move a constraint for the initial call to a recursive predicate, but not for later calls. Sometimes (as in fib-*forw*) the movement means that later calls can also be reordered.

Table IV gives the progressive improvement in execution time as each optimization is added. A — entry indicates that a program remains unchanged from the previous column. An ≈ entry indicates that the optimization was applied, but gave no appreciable change in execution time. This highlights which optimizations were applied at each stage. The second and third last columns give a summary of the results using all optimizations, with and without splitting. Note that, for sumlist-*var*, there is a change in parametric form with the combination of solver bypass with dead as compared to dead alone. Less speedup, though very substantial, is obtained.

Clearly, combining the optimizations leads to significant further gain. The best improvements for combined optimizations are on the small programs (e.g., fib). For the larger programs the benefit of the combinations tends to simply be the sum of the benefits from each of the optimizations individually. Again splitting is of considerable benefit.

The final column of Table IV gives the approximate speedup in constraint solving resulting from using all optimizations. It is calculated using the percentage solver use (from Table II) and factors out the time spent in the run-time engine. These speedups are very large for small programs and those which use the solver as an

Table V. Impact of Optimizations on Solver Space (Percent Space Used)

Program	Original Space	Dead		FR	Reo		All	
`fib`-*forw*	62608	64	—	—	33	41	33	41
`fib`-*back*	33665	86	—	≈	62	77	61	77
`mggnd`	189	≈		≈	—		≈	
`sumlist`-*gnd*	16003	89	—	≈	92	—	92	≈
`sumlist`-*var*	366004	1.6	—	≈	—		1.3	≈
`ackerman`	131418	—		63	—		63	
`ode`	10013	≈		—	≈		≈	
`queens`	200	—		—	—		—	
`matmul`	10659	68		—	—		76	
`mg-extend`	27150	82	86	≈	≈		56	
`money`	66	—		—	—		—	
`ladder`	7226	75		—	≈		77	75
`circuit`	472	65		—	—		66	65
`pic`	117	90	—	—	—		90	—
`bridge`	1345	≈		—	—		≈	
`neural`	4660	96	88	≈	≈		88	
`amp`	2117	75		—	—		77	75

arithmetic calculator. All programs show significant gain, never less than a speedup of 1.32.

While space savings are not our primary concern, we have also looked at space savings provided by these optimizations. Saving solver space at run-time allows larger problems to be run and can in some cases give rise to the most important speed change—"can run" versus "cannot run." Table V gives the reduction in peak solver space consumption measured in solver nodes (six words). We show the peak usage by optimized programs as a percentage of the peak usage by unoptimized programs. Again where splitting made a difference there are two numbers in the column, with splitting and without splitting. We show peak solver usage, as this is the largest amount of solver space used during computation and as such it is the limiting factor when executing a goal which requires significant amounts of memory. We show figures only for dead-variable elimination, future redundancy, reordering, and for the combination of all optimizations. This is because solver bypass and nofail optimizations cause no reduction in space used. Solver bypass could save solver space; but under the current implementation the solver is still used to store values of ground variables, so no savings take place.

Where a particular optimization did not occur for the benchmark its entry is marked with —, and where it had no effect, that is, 100% of space was used, it is marked ≈.

Of these optimizations dead-variable elimination is by far the most common optimization leading to a saving in solver space. Our expectation was that almost all cases of applying dead-variable elimination would require splitting. However, this was not the case. The space savings from dead-variable elimination are significant in many cases and overwhelming in `sumlist`-*var*. The space savings from future redundancy and reordering are also significant when present. Perhaps surprisingly, there is little evidence of space optimizations interacting. The space optimization achieved by all optimizations is usually just the sum of that achieved by individual optimizations. The exceptions are conflict where reordering can remove the possibility for dead-variable elimination (e.g., in `fib-`*forw*), and `mg-extend` where reordering costs us some space savings but substantially improves the other optimizations. The anomalous results where splitting slightly reduced the space savings (for `ladder`, `circuit`, and `amp`) result because of splitting enabling more solver bypass which prevents dead-variable removal.

Comparing the different combinations it is clear that for almost all the benchmarks the combination of the three commonly occurring optimizations: solver bypass, nofail, and dead-code elimination, provides all of the benefits of the optimizing compiler. Splitting is required in many cases to capture all the benefits of these optimizations. Only in the smaller programs (`fib, ackerman, sumlist-`*gnd*) were there significant advantages in the more complex optimizations, reordering and future redundancy. An explanation for this is as follows. Most programs are written with a single mode of use in mind for each predicate. Hence, the programmer carefully places the constraints where they shall be of most use, and does not introduce redundant constraints. Thus, there is little opportunity for reordering or future redundancy. In contrast, small programs written from mathematical definitions are intended to be used in many modes; so constraints are all placed as early as possible, and constraints which are not required for some modes are present. Hence, these kinds of programs can benefit from reordering and future redundancy.

We have also examined the cost of obtaining the optimized programs in terms of additional analysis and optimization time. Table VI shows the relative cost of the optimizations by comparing the combined analysis and optimization times for all optimizations in this article, with and without splitting, versus the subset of three commonly occurring optimizations, solver bypass, nofail, and dead-variable elimination (again with splitting). Even with all optimizations enabled the large benchmarks could be analyzed and optimized in under 10 seconds. The table shows that analysis and optimization for the potentially more powerful optimizations can take up to four times as long. Only one of the larger benchmarks (`mg-extend`) was improved by adding these extra optimizations. The table also shows the code expansion of the fully optimized splitting version of the program compared to the original. Note that the naive splitting strategy used does not lead to an unacceptable explosion in code size.

The examples here show that CLP($\mathcal{R}$) programs, particularly those with recursive definitions of constraints, are amenable to a comprehensive set of optimizations which can lead to improvements of an order of magnitude. A more unusual aspect of the optimizations is that we obtain not only time speedups but also substantial space savings, which is important, since executing long running goals may have unpredictable space requirements. Furthermore, it is possible to run out of space without the optimizations, so space optimizations may be desired even if some additional time cost is incurred.

Overall we see substantial across-the-board improvements in time and space for the benchmarks. Our results also show the advantage of multivariant specialization as this gives rise to considerably larger speedups and some space savings while it does not lead to a large increase in code size, even with the current naive splitting strategy. Speedups for some of the benchmarks, for example, `circuit` and `neural`, are not as great as for the smaller benchmarks. This is due to a number of reasons.

Table VI. Combined Analysis and Optimization Times (CPU sec)

	Analysis/Opt Times Split All	Analysis/Opt Times Nonsplit All	Analysis/Opt Times Split ByDeNo	Splitting Code Expansion
`fib`-*forw*	0.71	0.61	0.34	1.00
`fib`-*back*	1.84	1.05	0.48	2.75
`mortgage`	0.52	0.44	0.45	1.00
`sumlist`-*gnd*	0.55	0.47	0.46	1.00
`sumlist`-*var*	0.52	0.61	0.45	1.22
`ackerman`	1.05	0.91	0.38	2.50
`ode`	1.01	0.99	0.58	1.57
`queens`	0.65	0.79	0.51	1.00
`matmul`	1.52	1.26	0.79	1.46
`mg-extend`	3.47	2.60	0.98	1.42
`money`	1.18	0.96	0.67	1.64
`ladder`	1.25	1.23	0.70	1.14
`circuit`	1.46	1.43	0.95	1.12
`pic`	0.81	0.81	0.56	1.18
`bridge`	4.14	4.14	1.27	1.89
`neural`	4.86	4.21	2.05	1.33
`amp`	8.78	8.15	5.33	1.03
			Average	1.45

First, there is less opportunity for the more complex optimizations to be applicable. Second, information is lost during analysis due to arithmetic variables being passed around in data structures. The current analyzer, as we concentrate primarily on arithmetic constraints, does not keep track of complex nonarithmetic terms, unlike optimizing Prolog compilers [Taylor 1990; Van Roy and Despain 1990]. Finally, for some of these programs (e.g., `neural` and `circuit`), much of the execution time is not spent in the solver. Hence, the overall speedup available from the solver is limited.

The speedups obtained here are quite substantial taking into account the prototype optimizing compiler and the use of a CLAM emulator. Indeed the speedup due to global analysis, and the resultant simplification of constraint solving, is even larger than the speedup resulting from moving from the original CLP($\mathcal{R}$) interpreter to the compiler. In addition, some of the speedups and space savings have nonconstant factors associated with them and can have very large savings with different parameters in the goal. We have simply chosen some typical goals in such cases but much larger speedups can be obtained with larger parameters in the goals.

6. CONCLUSION

Although some details of the optimizations and compiler design are specific to CLP($\mathcal{R}$), we believe the general ideas behind the optimizations and compiler design are applicable to any CLP language and also to other constraint programming paradigms such as concurrent constraint programming languages, constraint databases, constraint imperative languages, and constraint functional languages. This is particularly true if the language provides arithmetic constraints—which most do.

Empirical evaluation of the prototype optimizing compiler has been extremely promising. Benchmark evaluation indicates that the optimizing compiler leads to a significant increase in execution speed and reduction in space requirements. The speed of the optimizing compiler is also impressive, taking only a few seconds to compile typical CLP($\mathcal{R}$) programs. The reason for the success of the optimizing compiler is, at least partly, due to the simple and declarative CLP semantics. This allows efficient and accurate global analysis as well as the use of many simple yet powerful optimizations. Indeed, in some sense the optimizing compiler can be viewed as an efficient program synthesizer as it takes an executable logical specification of the program and transforms this into specialized CLIC programs which handle different types of goals efficiently.

Our empirical evaluation has also taught us to be careful when extrapolating the usefulness of optimizations from small programs to larger programs. In contrast to preliminary findings about the usefulness of constraint reordering and constraint removal we have found that, at least on our benchmarks, they give little performance improvement for real world CLP($\mathcal{R}$) programs. Surprisingly, the most useful optimizations were found to be the simpler optimizations—solver bypass, no fail constraints, and redundant-variable removal—employed in conjunction with multivariant specialization. In part, this is probably because good CLP($\mathcal{R}$) programmers write code which is specialized for a particular mode of usage, implicitly performing constraint reordering and rarely use nonground variables in inequalities.

We noted in the previous section that much of the execution time of CLP($\mathcal{R}$) programs, particularly for larger programs, was spent not in the constraint solver but performing other tasks. These costs can be reduced by methods previously developed for use with Prolog programs [Taylor 1990; Van Roy and Despain 1990]. One next step is to take these optimizations and apply them to the CLP($\mathcal{R}$) system. We have not done this yet as our primary interest was the optimization of constraint solving.

The optimizations we currently perform could be applied more often if we had more precise analysis information. Currently we track only simple sharing, so a variable passed as an argument as part of a list or matrix has all of the information about it unavailable in that predicate. A deep sharing analysis to track such variables would allow us to retain that information and so perform more optimizations. However such an analysis may prove costly for large programs.

Other constraint optimizations are also available. A more general constraint removal optimization has been proposed as well as constraint refinement [Marriott and Stuckey 1993], a technique for adding constraints earlier to try to narrow the search space. Our experience with the constraint reordering and future redundant constraint removal optimizations raise some doubts about the benefits these other complex constraint optimizations would bring.

7. ACKNOWLEDGEMENTS

We would like to thank Harald Søndergaard for his involvement in initial discussions which led to the optimizing compiler.

REFERENCES

Aït-Kaci, H. 1991. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, Cambridge, Mass.

Armstrong, T., Marriott, K., Schachte, P., and Søndergaard, H. 1994. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In *Proceedings of the*

1st International Static Analysis Symposium, B. Le Charlier, Ed. Lecture Notes in Computer Science, vol. 864. Springer-Verlag, Berlin, 266–280.

Cousot, P. and Cousot, R. 1992. Abstract interpretation and application to logic programs. *J. Logic Program. 13,* 2/3 (July), 103–179.

Crnogorac, L., Kelly, A., and Søndergaard, H. 1996. A comparison of three occur-check analysers. In *Proceedings of the 3rd International Static Analysis Symposium*, R. Cousot and D. Schmidt, Eds. Lecture Notes in Computer Science, vol. 1145. Springer-Verlag, Berlin, 159–173.

García de la Banda, M., Hermenegildo, M., Bruynooghe, M., Dumortier, V., Janssens, G., and Simoens, W. 1996. Global analysis of constraint logic programs. *ACM Trans. Program. Lang. Syst. 18,* 5 (September), 564–615.

Hermenegildo, M. 1994. Some methodological issues in the design of CIAO, a generic, parallel concurrent constraint logic programming system. In *Proceedings of the 2nd International Workshop on Principles and Practice of Constraint Programming*, A. Borning, Ed. Lecture Notes in Computer Science, vol. 874. Springer-Verlag, Berlin, 123–133.

Hermenegildo, M., Puebla, G., Marriott, K., and Stuckey, P. 1995. Incremental analysis of logic programs. In *Logic Programming: Proceedings of the 12th International Conference*, L. Sterling, Ed. MIT Press, Cambridge, Mass., 797–814.

Jaffar, J., Michaylov, S., Stuckey, P., and Yap, R. 1992a. An abstract machine for CLP($\mathcal{R}$). In *Proceedings of the ACM Conference on Programming Language Design and Implementation.* ACM, New York, 128–139.

Jaffar, J., Michaylov, S., Stuckey, P., and Yap, R. 1992b. The CLP($\mathcal{R}$) language and system. *ACM Trans. Program. Lang. Syst. 14,* 3, 339–395.

Jørgensen, N., Marriott, K., and Michaylov, S. 1991. Some global compile-time optimizations for CLP($\mathcal{R}$). In *Logic Programming: Proceedings of the 1991 International Symposium*, V. Saraswat and K. Ueda, Eds. MIT Press, Cambridge, Mass., 420–434.

Kelly, A., Macdonald, A., Marriott, K., Sondergaard, H., Stuckey, P., and Yap, R. 1995. An optimizing compiler for CLP($\mathcal{R}$). In *Proceedings of the 1st International Conference on Principles and Practices of Constraint Programming.* Lecture Notes in Computer Science, vol. 976. Springer-Verlag, Berlin, 222–239.

Kelly, A., Macdonald, A., Marriott, K., Stuckey, P., and Yap, R. 1996. Effectiveness of optimizing compilation of CLP($\mathcal{R}$). In *Logic Programming: Proceedings of the 1992 Joint International Conference and Symposium*, M. Maher, Ed. MIT Press, Cambridge, Mass., 37–51.

Kelly, A., Marriott, K., Søndergaard, H., and Stuckey, P. 1997. A generic object oriented incremental analyser for constraint logic programs. In *Proceedings of the 20th Australasian Computer Science Conference.* 92–101.

Le Charlier, B. and Van Hentenryck, P. 1994. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM Trans. Program. Lang. Syst. 16,* 1, 35–101.

Macdonald, A., Stuckey, P., and Yap, R. 1993. Redundancy of variables in CLP($\mathcal{R}$). In *Logic Programming: Proceedings of the 1993 International Symposium.* MIT Press, Cambridge, Mass., 75–93.

Marriott, K. and Søndergaard, H. 1990. Analysis of constraint logic programs. In *Logic Programming: Proceedings of the North American Conference 1990*, S. Debray and M. Hermenegildo, Eds. MIT Press, Cambridge, Mass., 531–547.

Marriott, K., Søndergaard, H., Stuckey, P. J., and Yap, R. 1994. Optimizing compilation for CLP($\mathcal{R}$). In *Proceedings of the 17th Australian Computer Science Conference*, G. Gupta, Ed. 551–560.

Marriott, K. and Stuckey, P. 1993. The 3 R's of optimizing constraint logic programs: Refinement, removal and reordering. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages.* ACM, New York, 334–344.

Marriott, K. and Stuckey, P. 1998. *Programming with Constraints: an Introduction.* MIT Press, Cambridge, Mass.

Muthukumar, K. and Hermenegildo, M. 1992. Compile-time derivation of variable dependency using abstract interpretation. *J. Logic Program. 13,* 2/3 (July), 315–347.

PUEBLA, G. AND HERMENEGILDO, M. 1995. Implementation of multiple specialization in logic programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation.* ACM, New York, 77–87.

PUEBLA, G. AND HERMENEGILDO, M. 1997. Abstract specialization and its application to program parallelization. In *Sixth International Workshop on Logic Program Synthesis and Transformation*, J. Gallagher, Ed. Lecture Notes in Computer Science, vol. 1207. Springer-Verlag, Berlin, 169–186.

RAMACHANDRAN, V. AND VAN HENTENRYCK, P. 1995. Source to source optimizations of CLP(*RLin*). Brown University Technical Report.

SØNDERGAARD, H. 1986. An application of abstract interpretation of logic programs: Occur check reduction. In *Proceedings of the European Symposium on Programming.* Lecture Notes in Computer Science, vol. 213. Springer-Verlag, Berlin, 327–338.

TAYLOR, A. 1990. LIPS on a MIPS: Results from a Prolog compiler for a RISC. In *Logic Programming: Proceedings of the 7th International Conference*, D. Warren and P. Szeredi, Eds. MIT Press, Cambridge, Mass., 174–185.

VAN ROY, P. AND DESPAIN, A. 1990. The benefits of global dataflow analysis for an optimizing Prolog compiler. In *Logic Programming: Proceedings of the North American Conference 1990*, S. Debray and M. Hermenegildo, Eds. MIT Press, Cambridge, Mass., 501–515.

WINSBOROUGH, W. 1992. Multiple specialization using minimal-function graph semantics. *J. Logic Program. 13,* 2/3 (July), 259–290.

Received August; accepted December 1997