DETECTION AND OPTIMIZATION OF SUSPENSION-FREE LOGIC PROGRAMS *

SAUMYA DEBRAY, DAVID GUDEMAN, AND PETER BIGOT

▷ In recent years, language mechanisms to suspend, or delay, the execution of goals until certain variables become bound have become increasingly popular in logic programming languages. While convenient, such mechanisms can make control flow within a program difficult to predict at compile time and therefore render many traditional compiler optimizations inapplicable. Unfortunately, this performance cost is also incurred by programs that do not use any delay primitives. In this paper we describe a simple dataflow analysis for detecting computations where suspension effects can be ignored, and discuss several low-level optimizations that rely on this information. Our algorithm has been implemented in the `jc` system. Optimizations based on information it produces result in significant performance improvements, resulting in speed comparable to or exceeding that of optimized C programs. ◁

1. Introduction

In recent years, mechanisms to suspend the execution of a goal until certain variables in the goal become bound have become increasingly popular in logic programming languages. They are available in many modern implementations of Prolog, e.g., NU-Prolog, SICStus Prolog, Prolog-III, and Sepia. Such mechanisms also form the basis for synchronization in many concurrent logic programming languages such

*A preliminary version of this paper appeared in *Proc. 1994 International Symposium on Logic Programming*. This work was supported in part by the National Science Foundation under grant number CCR-9123520.

Address correspondence to Saumya K. Debray, Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA. E-mail: `debray@cs.arizona.edu`

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$3.50

as FGHC, Janus, and Strand. Delay mechanisms allow clear and concise expression of sophisticated control strategies, and can simplify programming significantly.

Despite the programming convenience provided by such features, they have the drawback that implementations are forced to contend with the possibility of suspension. This can greatly affect the performance of such systems. A significant problem is that the possibility of suspension, and the general unpredictability of when a suspended computation will be reactivated and eventually actually executed, complicates dataflow analysis and can render many traditional compile-time optimizations inapplicable; in particular, effective utilization of machine-level resources such as hardware registers becomes difficult. Also, additional testing may be necessary at runtime to determine whether or not suspension is necessary. This situation is especially undesirable because many compiler optimizations are precluded even for programs (or program fragments) that do not exhibit any suspension, effectively penalizing good programmers and carefully crafted programs. For example, it has been shown that significant improvements in performance can be obtained by returning output values of procedures in registers instead of in memory [1, 21], or with knowledge of lengths of dereference chains [17]. However, in a language with delay mechanisms, there is always the possibility that "normal" execution may be pre-empted by a newly awakened goal that may overwrite a register containing an output value or change the length of a dereference chain, thereby making these optimizations inapplicable.

In this paper we discuss simple analyses that can be used to detect situations where suspension effects can be ignored. The utility of this information is demonstrated by discussing a number of low-level compiler optimizations that rely on this information. We have implemented the analysis, and compiler optimizations based on it, in the `jc` system [13]: we present performance results to show that information about non-suspension is fundamental to a variety of low-level compiler optimizations that turn out to be very effective in producing significant performance improvements. As a result, the performance of our system very often approaches or beats that of C code written in a "natural" C style and optimized extensively. It should be emphasized that an important aspect of this work was to develop practical analysis methods to identify and optimize the common case of procedures and programs that do not make use of delay mechanisms. Simplicity, ease of implementation, and reasonable precision for commonly encountered programs were therefore our primary concerns.

2. Definitions and Notation

We assume that the reader is acquainted with the basic concepts and terminology of logic programming. The set of variables occurring in a syntactic object (i.e., term, atom, clause, etc.) t is denoted by $vars(t)$. A logic program consists of a finite set of predicate definitions; in addition, we suppose that we have (descriptions of) a set of queries of interest, from which analysis may be initiated. The set of all atoms of the language will be denoted by Atom. The set of (all alphabetic variants of) clauses defining a procedure p in the program under consideration will be denoted by $\mathsf{clauses}(p)$. The identity substitution over a set of variables V is denoted by id_V; when the set of variables under consideration is obvious from the context, we omit the subscript. The *most general unifier* of a pair of terms t_1 and t_2, which is unique up to variable renaming, is denoted by $mgu(t_1, t_2)$. The set of all (idempotent) substitutions is denoted by Subst. The empty sequence is denoted by ε. We denote the Kleene closure of a set S, i.e., the set of all finite sequences of its elements, by S^*; and the reflexive transitive closure of a relation R by $R^\star$.

Delay primitives in Prolog-like logic programming languages usually have the behavior that the execution of a goal suspends if certain variables, or arguments in a procedure call, are unbound. Different languages use different mechanisms to indicate which variables or argument positions are to be tested when determining whether a computation should suspend. Suspension of computations is also supported in concurrent logic programming languages, where a test in the guard of a clause suspends if the variables involved are not sufficiently instantiated. In order to abstract away from syntactic idiosyncracies of particular languages, we follow Marriott *et al.* [18] in assuming two (system-dependent) functions that specify the suspension/resumption behavior of goals: $delay(A, C, \theta)$ is true for a goal A, clause C and substitution θ if and only if the execution of the goal $\theta(A)$ using clause C should be delayed;[1] and, given a sequence of (suspended) goals G, the function $woken(G, \theta)$ yields a sequence of goals in G that are awakened by the substitution θ. Axioms specifying relationships between these functions are discussed in [18]. We assume that programs are *moded*, i.e., for each predicate p in a program its arguments are known to be "input" (if p uses the value of that argument) or "output" (if p defines that argument, i.e., binds it to a value).

The order in which awakened goals are executed, relative to each other and to the goals that are currently ready to execute, may differ in many ways depending on the language. For example, SICStus Prolog usually schedules goals as soon as they are awakened, ahead of other ready goals, though the relative order of goals that are awakened at the same time is unspecified [3]; the default policy in `jc` is to schedule awakened goals after all previously ready goals have finished executing (the more common "schedule awakened goals immediately" behavior can be obtained via a compiler option). KL1 provides a system of priorities that must be respected when awakened goals are executed [5]. To this end, we assume that there is a third system-dependent function

$$insert : \mathsf{Atom}^* \times \mathsf{Atom}^* \longrightarrow \mathsf{Atom}^*$$

such that, given a sequence of ready goals G and a sequence of awakened goals G', $insert(G, G')$ is the sequence of goals obtained by "inserting" the awakened goals G' into the appropriate positions within the ready goal sequence G.

3. Operational Semantics

To simplify the discussion that follows, we assume a language that uses nondeterministic clause selection. Goals within a clause are assumed to be executed in their left-to-right order. The operational behavior of a program can be characterized by the transition rules given below. A state consists of a sequence of "active" goals, a substitution, and a sequence of suspended goals:

$$State = \mathsf{Atom}^* \times \mathsf{Subst} \times \mathsf{Atom}^*.$$

[1] The *delay* function we consider is actually slightly different from that of Marriott *et al.* [18], in that theirs does not take a clause as a parameter.

For notational simplicity we say that a clause C is *variable-disjoint* from a state $S = (Ready, \theta, Susp)$ if $vars(C) \cap (vars(Ready) \cup vars(Susp) \cup vars(\theta)) = \emptyset$. The following transition rules specify the operational behavior of programs with suspension.

1. **Goal Reduction** : Given a state $S \equiv (p(\bar{u}) :: Ready, \theta, Susp)$, and a clause $C \equiv p(\bar{v})\ :-\ B \in \mathsf{clauses}(p)$ that is variable-disjoint from S, if it is the case that $delay(p(\bar{u}), C, \theta) = false$ and $\psi = mgu(\theta(\bar{u}), \bar{v}) \neq fail$, then the goal $p(\bar{u})$ is reduced to the clause body B:

   $$S \overset{r}{\leadsto} (B :: Ready, \psi \circ \theta, Susp)$$

   where '::' denotes concatenation of sequences. The annotation 'r' in the transition $\overset{r}{\leadsto}$ is intended as a mnemonic for "reduces."

2. **Variable Binding** : Given a state $S \equiv (`t_1 = t_2' :: Ready, \theta, Susp)$ such that $\phi = mgu(\theta(t_1), \theta(t_2)) \neq fail$, let $\psi = \phi \circ \theta$, and $A = woken(Susp, \psi)$, then

   $$S \overset{b}{\leadsto} (insert(Ready, A), \psi, delete(Susp, A)).$$

   The annotation 'b' in the transition $\overset{b}{\leadsto}$ is intended as a mnemonic for "binds."

3. **Suspension** : Given a state $S \equiv (p(\bar{t}) :: Ready, \theta, Susp)$, if there is no clause in $\mathsf{clauses}(p)$ that can reduce according to rule (1) above, but there is some clause $C \in \mathsf{clauses}(p)$ such that C is variable-disjoint from S and for which $delay(p(\bar{t}), C, \theta)$ is true, then the goal $p(\bar{t})$ suspends:[2]

   $$S \overset{s}{\leadsto} (Ready, \theta, p(\bar{t}) :: Susp).$$

   The annotation 's' in the transition $\overset{s}{\leadsto}$ is intended as a mnemonic for "suspends."

4. **Failure** : Given a state $S \equiv (p(\bar{t}) :: Ready, \theta, Susp)$, if there is no clause that can proceed via rules (1) or (2), or suspend according to rule (3) above, then execution fails. The action taken on failure depends on the language: in a Prolog-like language, it may trigger backtracking, while in a committed-choice language it may cause the entire execution to abort. For our purposes, it suffices to postulate a function

   $$fail\_action : State \longrightarrow State$$

   that specifies what happens on failure. The details of this function are not relevant for the purposes of this paper, and are not discussed further. We denote failure transitions by $\overset{f}{\leadsto}$.

[2] Primitive operations, e.g., involving arithmetic operations, in clause bodies may also suspend if their arguments are under-instantiated. This can be modelled either by rewriting the program to introduce auxiliary procedures with the appropriate delay conditions, or by additional rules similar to this one. The extensions involved are straightforward, and for simplicity of exposition we do not consider them explicitly.

For notational convenience in the discussion that follows, we will concatenate the mnemonic annotations on transitions to denote the union of the corresponding transition relations. Thus, $\overset{rbf}{\leadsto}$ denotes the relation $\overset{r}{\leadsto} \cup \overset{b}{\leadsto} \cup \overset{f}{\leadsto}$, i.e., the set of transitions not involving any suspension. Finally $\leadsto$ denotes all possible transitions, i.e., $\overset{rbsf}{\leadsto}$.

4. Dataflow Analysis for Non-suspension

4.1. Preliminaries

Given the "concrete" transition semantics described in the previous section, a collecting semantics associates a set of states with each program point. The obvious approach to defining a dataflow analysis would be to abstract sets of states to "abstract states" and define "abstract operations" over these objects to mimic the corresponding operations over the concrete domain. Correctness could then be inferred from the relationships between these objects and operations, using results of abstract interpretation [8]. However, the details can get quite complicated: for example, abstracting a sequence of goals into a sequence of abstract goals need not produce a Noetherian abstract domain, and an additional level of abstraction may be necessary to ensure termination. For example, the "star abstraction" of Codish *et al.* [6] allows at most one "abstract atom" with a particular predicate symbol, while Marriott *et al.* require bounds on the sizes of the multisets describing delayed goals [18]. Since our analysis algorithms are really quite straightforward, we felt that trying to formalize them as full-blown abstract interpretations would serve mainly to introduce a plethora of notation and to obscure, rather than illuminate, the essential underlying intuitions. For this reason, we have chosen to give direct proofs of correctness instead of formalizing them as abstract interpretations (though it will be obvious that we use ideas, such as "abstract substitutions" and "concretizations", that are derived from abstract interpretation).

Just as the concrete computation propagates substitutions, the analysis propagates descriptions of (sets of) substitutions that we refer to as "abstract substitutions". The set of abstract substitutions is denoted by ASub. We assume that there is a partial ordering $\sqsubseteq$ over ASub and that $(\mathsf{ASub}, \sqsubseteq)$ forms a complete lattice with meet and join operations $\sqcap$ and $\sqcup$ respectively. We also assume that there is a function

$$conc : \mathsf{ASub} \longrightarrow \wp(\mathsf{Subst})$$

that specifies the set of substitutions described by an abstract substitution. This function is assumed to be monotone, so that given $a_1, a_2 \in \mathsf{ASub}$, $a_1 \sqsubseteq a_2$ implies $conc(a_1) \subseteq conc(a_2)$. Thus, the higher an abstract substitution is in the lattice $(\mathsf{ASub}, \sqsubseteq)$ the larger a set of concrete substitutions it denotes, and therefore the lesser the information it conveys.

The basic idea behind our approach is very simple and quite general. Suppose we are given a (top-down) dataflow analysis for an "ordinary" logic programming language without any delay primitives, i.e., one whose operational semantics is defined by omitting the suspension rule from the transition rules defined in Section

3—in other words, by the transition relation $\overset{rbf}{\leadsto}$.[3] Such an analysis, which we refer to as the *underlying analysis* in the discussion that follows, can be extended in a simple way to deal with programs containing delay primitives, and the information gathered used to detect non-suspending procedures and calls. Broadly speaking, such an analysis typically iterates over the predicates in a program, analyzing each of the clauses for each predicate in turn, maintaining "abstract environments" at each program point that describe the possible bindings for variables at that point. Analysis usually starts with these abstract environments initialized to be empty, and information is propagated iteratively until there is no change to any of the information gathered: termination is ensured using additional mechanisms such as memo tables. Assume that the analysis provides two operations: $analyse\_call : \mathsf{Atom} \times \mathsf{ASub} \longrightarrow \mathsf{ASub}$, which describes the effects of a procedure call; and $extend\_abs\_env : \mathsf{ASub} \times \mathsf{ASub} \longrightarrow \mathsf{ASub}$, which describes how to extend an abstract environment to take into account the effects of a procedure call or unification. The analysis of a clause '$p(\bar{u})$:− $q_1(\bar{u}_1), \ldots, q_n(\bar{u}_n)$', given an abstract substitution α for a call, proceeds as follows:

1. Abstract the effects of head unification with the formal parameters $\bar{x}$, given the abstract substitution α describing the actual parameters, so as to obtain an initial abstract environment A_1.

2. Let the abstract environment at the program point immediately before the literal $q_i(\bar{u}_i)$ be denoted by A_i, $1 \leq i \leq n$.

   For $i := 1$ to n do:
   $A'_i := analyse\_call(q_i, A_i)$;
   $A_{i+1} := extend\_abs\_env(A_i, A'_i)$;

3. Compute and return the abstract environment A_{n+1} projected on the arguments in the head of the clause.

A call to a procedure is analyzed by processing each clause for that procedure, in turn, as described above. For each clause, the analysis yields an abstract environment, and these can be "summarized"—for example, using the join operator $\sqcup$ on ASub—to yield an abstract environment describing the substitutions that may be obtained when that call returns.

In the following sections, we describe how the underlying analysis described above can be extended to detect situations where suspension effects can safely be ignored. The underlying intuition is that, as long as we can guarantee that the "normal" left-to-right control flow of a clause will not be disrupted due to suspension effects, we can use (approximations to) the underlying analysis and be guaranteed soundness. There are two ways in which normal left-to-right execution can be disrupted: (*i*) a goal may suspend; and (*ii*) a suspended goal may be awakened and executed ahead of other ready goals. Section 4.2 discusses how to deal with the

[3] Strictly speaking, since there is no suspension, the operational semantics of such a language could be somewhat simpler than this: there is no need for list of suspended goals in a state, and the variable binding rule could be simplified. For the development of this paper, however, it is convenient to work with states whose structure is as in Section 3 but where the list of suspended goals is always empty, and have $delay(A, C, \theta)$ be *false* for all A, C, and θ.

first of these situations. We handle the second situation by analyzing the program to identify situations where no suspended goal will be awakened: this is discussed in Section 4.3.

4.2. Weak Non-Suspension

Define the parent relation on pairs of goals as follows: if a computation has a reduce-transition $(L :: \mathit{Ready},\ \theta,\ \mathit{Susp}) \overset{r}{\rightsquigarrow} (B_1 :: \cdots :: B_n :: \mathit{Ready}, \theta', \mathit{Susp})$ via a clause '$H\ :-\ B_1, \ldots, B_n$' then the goal L is said to be the *parent* of the goals $B_1, \ldots, B_n$. The *descendant* relation on pairs of goals is simply the reflexive transitive closure of the parent relation. A goal L is said to be weakly non-suspending if none of the descendants of L (which includes L itself) will suspend, i.e., take a $\overset{s}{\rightsquigarrow}$-transition.

A key notion in our approach to the analysis of weak non-suspension is that of the "demand" of a procedure. Intuitively, this is a description of the circumstances under which none of the clauses defining that procedure will suspend. More formally, we have the following definition:

Definition 4.1. Given a procedure p defined by clauses $C_1, \ldots, C_n$ in a program P, the *demand* of p, denoted by $\mathit{demand}(p)$, is given by

$\mathit{demand}(p) = \sqcap\{\alpha \in \mathsf{ASub} \mid \forall\theta \in \mathit{conc}(\alpha)\ \forall C \in \mathsf{clauses}(p) : \neg \mathit{delay}(p(\bar{x}), C, \theta)\}$.
where $\bar{x}$ is a tuple of distinct variables.

Example 4.1. Consider the following procedure to compute the factorial function: the language is our current incarnation of Janus, where guard tests suspend until the values of the operands are available:

```
fact(N, A, F) :- N = 0 | F = A.
fact(N, A, F) :- N > 0 | fact(N-1, N*A, F).
```

Suppose abstract substitutions map variables to the values `ground` and `any`, denoting, respectively, the set of ground terms and the set of all terms of the language. It is easy to see that neither guard will suspend if the first argument is bound to a (any) ground term. Thus, the demand of `fact`/3 is $\{\mathtt{N} \mapsto \mathtt{ground}, \mathtt{A} \mapsto \mathtt{any}, \mathtt{F} \mapsto \mathtt{any}\}$.

The analysis to identify weak non-suspension is a conceptually straightforward extension to the underlying analysis described in the previous section. With each literal L in a clause body (respectively, predicate p in the program) is associated a flag $L.\mathit{nosusp}$ (respectively, $p.\mathit{nosusp}$), indicating whether that literal (predicate) is weakly non-suspending. Initially, all these flags have the value `true`, indicating that our initial assumption is that no procedure or call will suspend. The program is first processed to compute $\mathit{demand}(p)$ for each procedure p in the program (note that this is a strictly local computation, and does not require any kind of global fixpoint computation). After this, a global fixpoint computation is carried out to identify weakly non-suspending procedures and calls. The processing of a clause '$p(\bar{u})\ :-\ L_1, \ldots, L_n$' is shown in Figure 1. When processing a body literal $L_i \equiv q_i(\ldots)$ under an abstract environment A_i, we first verify whether its activation can be guaranteed to perform at least one goal reduction step without suspending

1. Given an abstract substitution α describing the actual parameters in the call, abstract the effects of head unification with the formal parameters $\bar{u}$ to obtain an initial abstract environment A_1.

2. Let the abstract environment at the program point immediately before the literal $L_i \equiv q_i(\bar{u}_i)$ be denoted by A_i, $1 \leq i \leq n$.

   For $i := 1$ to n do:
   $L_i.nosusp := q_i.nosusp \wedge (A_i \sqsubseteq demand(q_i))$;
   $A'_i := analyse\_call(q_i, A_i)$;
   $A''_i := extend\_abs\_env(A_i, A'_i)$;
   $A_{i+1} :=$ **if** $\neg L_i.nosusp$ **then** $A_i \sqcup A''_i$ **else** A''_i;

3. $p.nosusp := p.nosusp \wedge (\wedge_{i=1}^{n} L_i.nosusp)$;

4. return the projection of the abstract environment $A_{n+1}(\bar{x})$ on the arguments in the head of the clause.

Figure 1. Weak Non-Suspension Analysis: Processing a clause '$p(\bar{u})$:− $L_1, \ldots, L_n$'

by checking whether $A_i \sqsubseteq demand(q_i)$.[4] Then, as in the underlying analysis, we determine the abstract environment A''_i that would be obtained if L_i were to execute without any suspension. The main change, compared to the underlying analysis, is that at this point, we compute the resulting abstract environment A_{i+1} as A''_i if L_i can be guaranteed to not suspend, and as $A_i \sqcup A''_i$ otherwise: here, $A_i \sqcup A''_i$ expresses what can happen whether L_i suspends or not. As before, a call to a procedure is processed by analyzing each clause defining that procedure, and summarizing the results. The analysis of a program involves iterating until there is no change to either the calling and success patterns computed for each procedure, or the suspension flags of any procedure or literal.

Define a *suspending transition sequence* to be any sequence in $\overset{rbf}{\leadsto}{}^{\star}\overset{s}{\leadsto}$. The following theorem states that, as long as we do not have to contend with goals being awakened, the algorithm of Figure 1 computes suspension bits correctly in the sense that any literal or predicate that may suspend has its *nosusp* bit set to `false`.

Theorem 4.1. Let $L \equiv p(\bar{u})$ *be a literal in a program such that for some initial query* Q, *it is the case that* $(Q, id_{vars(Q)}, \varepsilon) \leadsto^{\star} (p(\bar{u}), \theta, Susp)$; *and* $(p(\bar{u}), \theta, \varepsilon) \overset{rbf}{\leadsto}{}^{\star}\overset{s}{\leadsto} S$ *for some state* S *and suspended goals* $Susp$. *Then,* $L.nosusp$ *and* $p.nosusp$ *are set to* `false` *by the analysis algorithm.*

Proof. Suppose that $L \equiv p(\bar{u})$ is a literal in a program such that for some

[4] W. Winsborough has pointed out [22] that this check can be made more precise by checking also whether it may be possible to guarantee that some clause for the called predicate will reduce, in which case the possible suspension of other clauses is moot: this can be done, in the presence of type information, using the notion of *covering* discussed in [9].

initial query Q, $(Q, id_{vars(Q)}, \varepsilon) \rightsquigarrow^{\star} (p(\bar{u}), \theta, Susp)$. Consider transitions from the state $(p(\bar{u}), \theta, \varepsilon)$. We show, by induction on the length n of suspending transition sequences that if $(p(\bar{u}), \theta, \varepsilon) \overset{rbf}{\rightsquigarrow}^{\star} \overset{s}{\rightsquigarrow} S$, then the flags $L.nosusp$ and $p.nosusp$ are set of `false` by the analysis algotithm.

In the base case, $n = 1$, and it must be the case that $(p(\bar{u}), \theta, \varepsilon) \overset{s}{\rightsquigarrow} S$. From the rules defining the transition semantics, this can happen only if the goal $p(\theta(\bar{u}))$ is not able to reduce using any clause for p, and there is at least one clause C such that $delay(p(\bar{u}), C, \theta)$. Since the underlying dataflow analysis is assumed to be sound, it must be the case that $\theta \in conc(A)$ for the abstract environment A inferred immediately before L. Now suppose that $A \sqsubseteq demand(p)$. From the definition of $demand(p)$, this means that

$$conc(A) \subseteq \{\phi \in \mathsf{Subst} \mid \forall C \in \mathsf{clauses}(p) : \neg delay(p(\bar{u}), C, \phi)\},$$

i.e., the goal $p(\theta(\bar{u}))$ does not suspend. This is a contradiction, whence we conclude that it must be the case that $A \not\sqsubseteq demand(p)$. It is then straightforward to see, from the definition of the analysis, that $L.nosusp$ is set to `false` immediately. Because of this, in step (3) of the algorithm the flag $p.nosusp$ also becomes set to `false`.

For the inductive case, assume that the theorem holds for all suspending transition sequences with length less than k, and consider a suspending transition sequence from $(p(\bar{u}), \theta, \varepsilon)$ to S with length k. Then, there must be a clause $C \equiv$ '$p(\bar{x})$:- $L_1, \ldots, L_n$' in $\mathsf{clauses}(p)$ such that C is variable-disjoint from $p(\bar{u})$, $\neg delay(p(\bar{u}), C, \theta)$, and $\psi = mgu(\bar{x}, \bar{u}) \neq fail$, so that

$$(p(\bar{u}), \theta, \varepsilon) \overset{r}{\rightsquigarrow} (L_1 :: \cdots :: L_n, \psi \circ \theta, \varepsilon) \underbrace{\overset{rbf}{\rightsquigarrow} \cdots \overset{rbf}{\rightsquigarrow} \overset{s}{\rightsquigarrow}}_{k-1} S.$$

Now consider the transition sequence $(L_1 :: \cdots :: L_n, \psi \circ \theta, \varepsilon) \overset{rbf}{\rightsquigarrow} \cdots \overset{s}{\rightsquigarrow} S$ with length $k-1$: it must be the case that for some L_i, $1 \leq i \leq n$, there is a suspending transition sequence from (L_i, ϕ, ε) to S of length less than k, where ϕ is some substitution. Suppose that the predicate symbol for L_i is q, then from the induction hypothesis, the flags $L_i.nosusp$ and $q.nosusp$ will be set to `false` during analysis. Suppose that this happens during the j^{th} iteration of the algorithm, $j \geq 0$: since the values of some flags have changed during this iteration, the algorithm will iterate one more time. In the next iteration, $p.nosusp$ will be set to `false` because the $nosusp$ flag for the body literal L_i in the clause C defining p has become `false`. This change in the value of a flag will cause the algorithm to iterate once more, and this time the flag $L.nosusp$ will be set to false. The theorem follows.

Termination follows from the fact that there are only a finite number of flags manipulated by the algorithm, since each flag can change only from `true` to `false`.

The analysis described above is aimed at inferring, for any goal $p(\bar{u})$, whether any descendant of this goal can suspend. This information, while potentially interesting, is in general not sufficient for our purposes: we are interested in dataflow analysis for various low-level optimizations, such as register allocation and optimization of unification code, that require a more precise identification of what may happen at runtime. The problem that arises is the following: suppose that we have inferred that none of the descendants of a goal $p(\bar{u})$ will suspend. However, if $p(\bar{u})$ is executed in a state where there are already some suspended goals, it may happen that in the course of the resulting computation a variable x gets a binding that causes the awakening and execution of a suspended goal $q(\bar{v})$. Without a great

```
procedure analyse_clause(C)
begin
    let C ≡ p(x̄) :- L_1, ..., L_n;
    for each body literal L_j ≡ q(ȳ) do
        if q.s_nsusp = false then L_j.s_nsusp := false;
        else for each variable z in an output position in L_j do
            if (∃y ∈ aliases(z))(∃L_i to the left of L_j) : y occurs in an input position
                of L_i and L_i is not weakly non-suspending then
                    L_j.s_nsusp := false;
            fi
            if (∃u ∈ aliases(z)) : u is an output argument in the head of C and
                there is a call site M for p such that M.s_nsusp = false then
                    L_j.s_nsusp := false;
            fi
          od /* for */
        fi
    od
    p.s_nsusp := p.s_nsusp ∧ (∧_{i=1}^{n} L_i.s_nsusp);
end /* analyse_clause */
```

Figure 2. The function *analyse_clause* for strong non-suspension analysis

deal of knowledge about the possible suspended goals that may be awakened at any point, and about the scheduling policy for these awakened goals, it is very difficult to predict the behavior of the system under these circumstances (Marriott *et al.* show how a simple scheduling strategy, such as that used by SICStus Prolog, can be taken into account to some extent [18]; however, for more sophisticated strategies, e.g. involving multiple priority levels as in KL1, the task seems more difficult). It is for this reason that the statement of Theorem 4.1 assumed that the set of suspended goals is empty. In the next section, we describe an additional dataflow analysis that can be used to identify situations where it can be guaranteed that no suspended goal will be awakened in the course of a computation.

4.3. Strong Non-Suspension

A goal is said to be strongly non-suspending if it is weakly non-suspending and can additionally be guaranteed to not awaken any suspended goal. For strongly non-suspending goals, unpredictabilities arising from dynamic suspension and resumption of goals can be guaranteed to not arise, making their execution behavior much simpler to predict and enabling a variety of low-level optimizations to be carried out.

The intuition behind our analysis to detect strong non-suspension is very simple. Consider a clause '$p(\bar{x})$:- $L_1 \ldots, L_i, \ldots, L_n$.' Suppose that a variable z occurs in an output position in L_i, and assume that the analysis described in the previous section has identifed L_i as weakly non-suspending. Since the literals in the body of this clause are assumed to be executed from left to right, it is not difficult to see that if all occurrences of z in input positions in the body occur to the right of L_i, i.e., in L_j where $j > i$, then none of the literals L_j that need the value of z will be executed before L_i. Moreover, since L_i is weakly non-suspending, neither it nor any of its descendants will suspend, so they will produce a binding for z before any of the literals L_j that need the value of z are executed. We can generalize this idea slightly and require only that if z occurs in any L_k to the left of L_i, then L_k should be (weakly) non-suspending as well—in other words, that L_k does not actually use the variable z (the commonest example of this situation is where the L_k takes the variable z and simply puts it inside a structure without actually examining its value, e.g., in programs using difference lists).

Of course, this simple analysis is not quite adequate, for two reasons. First, all possible aliasing has to be taken into account. More importantly, it may happen that the variable z in question is also an output variable in the head of the clause, in which case it is necessary to consider all call sites for p to determine whether there may be any activation suspended on the corresponding actual parameter. At each such call site, the reasoning proceeds as before, and may involve looking at the call sites for the clause containing the call site, and so on. Overall, this gives rise to an iterative analysis algorithm that proceeds as follows:

1. initialize all strong non-suspension bits to `true`;
2. for every literal L (procedure p) that is not weakly non-suspending, set $L.s\_nsusp$ ($p.s\_nsusp$) to `false`;
3. iterate over the clauses in the program, and for each clause C execute the procedure $analyse\_clause(C)$ (shown in Figure 2), until there is no change to any strong non-suspension bit.

To reason about the soundness of this algorithm, we need the notion of the *rank* of a goal in a sequence of states obtained during a computation starting from an initial query. This notion is defined as follows:

1. Each atom in the initial query has rank 0.
2. If a state $(p(\bar{u}) :: Ready, \theta, Susp)$ reduces to a state $(L_1 :: \cdots :: L_n :: Ready, \theta', Susp)$ via a clause $C \equiv$ '$p(\bar{v})$:− $L_1, \ldots, L_n$' $\in \mathsf{clauses}(p)$, and the reduced goal $p(\bar{u})$ has rank k, then each of $L_1, \ldots, L_n$ has rank $k+1$.
3. The rank of a goal does not change if it suspends or is awakened.

The following establishes the correctness of this algorithm.

Theorem 4.2. If a literal L (respectively, procedure p) in a program is not strongly non-suspending, then L.s_nsusp (respectively, p.s_nsusp) is set to `false` *by the algorithm of Figure 2.*

PROOF. (sketch) : A literal or procedure may fail to be strongly non-suspending for either of two reasons: (*i*) it may not be weakly non-suspending; or (*ii*) its execution may result in a variable binding that causes an awakened goal to execute.

In the first case, Theorem 4.1 implies that it is inferred to be weakly non-suspending. In this case, its strong non-suspension bit is set to `false` at the beginning, before the analysis proper begins. Since the algorithm only changes strong nonsuspension bits from `true` to `false`, the value of this bit remains `false` at the end of the analysis.

In the second case, consider a goal L_1 that is weakly non-suspending, but that fails to be strongly non-suspending because there is some initial query Q during whose execution the goal L_1 is executed and causes a suspended goal L_2 to awaken. This implies that there is a clause C in the program that is a "common ancestor" of both L_1 and L_2, i.e., whose body B contains literals B_1 and B_2 such that L_1 is a descendant of B_1 and L_2 is a descendant of B_2;[5] moreover, since L_1 awakened L_2, it must be the case that L_2 suspended before the execution of L_1, so B_2 must be to the left of B_1. Let the rank of L_1 be m, and that of B_1 be m'. Since L_1 is a descendant of B_1, it must be the case that $m' \leq m$. We proceed by induction on $m - m'$.

In the base case, $m - m' = 0$. In this case, B_1 is the same as L_1. Since B_2 has a descendant that suspends, B_2 is not weakly non-suspending. Since B_2 is to the left of B_1 (i.e., L_1) in the clause C, it can be seen from Figure 2, that $L.s\_nsusp$ is set to `false` by the algorithm.

In the inductive case, assume that the theorem holds for $m - m' \leq k$, and consider a situation where $m - m' = k + 1$. In this case, suppose that L_1 occurs in the body of a clause for a procedure p, then there must be a goal L'' with rank m'' that reduced using this clause. By the definition of rank, $m = m'' + 1$, which means $m'' - m' = k$. From the induction hypothesis, $L''.s\_nsusp$ is set to `false` by the algorithm. This means that at the next iteration of the algorithm, the fact that $L''.s\_nsusp =$ `false` for the call site L'' will be detected, and $L.s\_nsusp$ will be set to `false`.

As with the algorithm for weak non-suspension, termination follows from the fact that there are only a finite number of flags manipulated by the algorithm, and each flag can change only from `true` to `false`.

5. Implementation

5.1. Background

Janus is a committed-choice logic programming language that, in its present incarnation, closely resembles Strand [10]. The `jc` system is a sequential implementation where clause bodies are executed from left to right as in Prolog. The `jc` compiler translates Janus programs to C and then uses a C compiler (the performance numbers in this paper correspond to `gcc` 2.6.3 invoked with `-O2 -fomit-frame-pointer`) to compile the resulting program to executable code. An early version of the system is described in [13], and a prototype of the system, including the dataflow analyses and the optimizations based on this analysis that are discussed in Section 6, is available by anonymous FTP from `ftp.cs.arizona.edu`. As with many other committed-choice languages, Janus uses dataflow synchronization. Thus, a procedure call will suspend if the input arguments are sufficiently underinstantiated that none of the clause guards for that clause can commit and

[5] It may help the reader's intuition to think of (the body of) C as a common ancestor of L_1 and L_2 in a proof tree for the query Q, although the presence of delays complicates the connections between the proof trees for queries and their operational behavior.

Program	Procedures		Call Sites	
	total	non-susp	total	non-susp
`aquad`	8	7	26	26
`bessel`	11	10	35	35
`binomial`	7	7	24	24
`chebyshev`	3	2	9	9
`e`	2	2	6	6
`fib`	1	1	6	6
`log`	6	5	27	27
`mandelbrot`	10	2	27	19
`muldiv`	1	1	7	7
`nrev`	3	3	10	10
`pascal`	15	14	49	49
`pi`	3	3	9	9
`sum`	2	2	4	4
`tak`	1	1	8	8
Total:	73	60	247	239

Table 1. Weak Non-suspension Analysis : Precision

they do not all fail; and even if a procedure call commits to a clause, primitive operations in the body of the clause will suspend if their operands are not sufficiently instantiated. Because of this, non-suspension analysis is crucial for any kind of low-level compiler optimization.

We have implemented weak non-suspension analysis in our system, based on a very simple groundness analysis. The underlying abstract domain for our analysis consists of only two points: *ground* and *any*, denoting, respectively, the set of all ground terms and the set of all terms of the first order language defined by the program under consideration. The ordering on this domain is $ground \sqsubseteq any$. We use reexecution of primitive operations (see [16]) to improve the precision of the analysis. We have not yet implemented strong non-suspension analysis. The reason weak non-suspension has been sufficient so far is that by default, goals awakened after suspension are scheduled in "batch mode" in `jc`, i.e., executed at the end after all non-suspended goals have been executed. In this case, there is no possibility that binding a variable will cause an awakened goal to be executed ahead of a ready goal. The more familiar scheme for scheduling awakened goals, where a goal is executed as soon as possible after it is awakened, is available via a compiler option, but optimizations based on non-suspension analysis, such as returning output values in registers instead of in memory, are currently turned off in this case: we expect to relax this restriction in the future when strong non-suspension analysis is implemented and incorporated into our compiler. The algorithm for strong non-suspension analysis is sufficiently similar structurally to that for weak non-suspension analysis that we do not anticipate any significant loss in precision of analysis when strong non-suspension is taken into account.

We have not separately measured the time taken to analyse programs, because dataflow analysis and optimization accounts for a very small part of the overall compilation time. Because Janus programs are compiled to C code which is then processed by a C compiler, most of the overall time for translation to the object code is spent in I/O operations and in the C compiler (other systems that compile to C, e.g., KLIC [5], report similar experiences). As a result, there is no perceptible decrease in the overall compile time when dataflow analysis and optimizations are switched off.

5.2. Precision

Since our analysis is not tied to any particular abstract domain, the question of precision is, in some sense, moot: the overall precision of a non-suspension analysis could, in principle, be improved where necessary simply by using a more precise underlying analysis with a more elaborate abstract domain. Our experience has been that in practice, the simple groundness analysis that we use, with a limited amount of reexecution, turns out to produce results that are quite reasonable. This is illustrated in Table 1. The benchmarks used include the following programs: *aquad* performs a trapezoidal numerical integration $\int_0^1 e^x dx$ using adaptive quadrature and an epsilon of 10^{-8}; *bessel* computes the Bessel function $J_{75}(3)$, and involves both integer (for factorial) and floating point (for exponentiation) computations; *binomial* computes the binomial expansion $\sum_{i=0}^{30} x^i y^{30-i}$ at $x = 2.0,\ y = 1.0$; *chebyshev* computes the Chebyshev polynomial of degree 10000 at 1.0; *e* computes the value of $e = 2.71828\ldots$ by iteratively summing the first 2000 terms of the series $1 + 1/1! + 1/2! + 1/3! + \cdots$; *fib* computes the Fibonacci value $F(16)$; *log* computes $\log_e(1.999)$ using the expansion $\log_e(1+x) = \sum_{i\geq 0}(-1)^{i+1}x^i/i$, to an accuracy of 10^{-6}; *mandelbrot* computes the Mandelbrot set on a 17×17 grid on an area of the complex plane from $(-1.5, -1.5)$ to $(1.5, 1.5)$; *muldiv* exercises integer multiplication and division, doing 5000 of each; *nrev* is the usual naive reverse program on an input list of length 100; *pascal* is a benchmark, by E. Tick, to compute Pascal's triangle; *pi* computes the value of π to a precision of 10^{-3} using the expansion $\pi = 4\sum_{i\geq 0}\frac{(-1)^i}{2i+1}$; *sum* adds the integers from 1 to 10,000—it is essentially similar to a tail-recursive factorial computation, except that it can perform a much greater number of iterations before incurring an arithmetic overflow; and *tak*, from the Gabriel benchmarks, is a heavily recursive program involving integer addition and subtraction. Of these programs, *aquad*, *bessel*, *binomial*, *chebyshev*, *e*, *log*, *mandelbrot*, and *pi* are floating-point intensive computations. These were chosen in part to focus on numerical computations, which—according to folklore—are not considered to be especially efficiently executable in logic programming languages.

The `jc` compiler works by first duplicating the code (i.e., the abstract syntax tree) for each procedure in the module being compiled: the intent is that one copy is for invocations that could potentially suspend, the other for invocations that can be guaranteed to be non-suspending. This is followed by weak non-suspension analysis, after which the program is transformed to take non-suspension information into account. Two things happen during this transformation phase: non-suspending procedure calls are modified to call the non-suspending version of the called procedure; and possibly suspending primitive operations in clause bodies, e.g., arithmetic, are transformed into out-of-line procedure calls to ensure correct behavior on suspension and resumption. Finally, the call graph of the resulting program is examined starting at the roots, which correspond to the possibly-suspending versions of exported procedures, and any version of a procedure that is found to be unreachable is deleted.

Because of the initial duplication of code and the subsequent deletion of unreachable versions, the number of procedures and call sites in the program that is actually compiled can be quite different from that in the original source program. To reduce confusion, therefore, the numbers in Table 1 are given relative to the original source program. Since the language semantics specifies that each operation in a clause body, regardless of whether it is a primitive operation or a call to a user-defined procedure, can potentially suspend, each body literal is counted as a "call site" that can, in principle, suspend. A procedure is then counted as non-suspending if a non-suspending version of that procedure is retained in the final program, while a call site i.e., body literal, is counted as non-suspending if the final program contains a non-suspending instance of that literal. Columns 2 and 3 of Table 1 give, respectively, the total number of procedures in the program and the number inferred to be non-suspending, while columns 4 and 5 give the same information for individual call sites. Note that some programs, such as `aquad` and `bessel`, have suspending versions of procedures but no suspending call sites: this is because it is possible for the user to invoke the top level goal in a manner that causes it to suspend; our analysis infers, however, that once the top level exported goal commits, there will be no further suspension. For these programs, the program contains a suspending version of the exported procedure, but there are no suspending call sites in the program. Other programs, such as `binomial` and `pi`, have top level goals that do not take any user input (this can happen, for example, if the only exported procedure is `main`/0), and therefore cannot suspend: for these programs, the generated code contains neither suspending versions of any procedures nor any suspending call sites.

It can be seen that for most of the programs tested, most of the procedures and call sites were inferred to be non-suspending. The `mandelbrot` benchmark is an exception to this, primarily because the underlying analysis is not sophisticated enough to carry out the kind of inductive reasoning necessary to infer groundness of arrays that are being defined an element at a time. However, as Tables 2 and 3 show, enough information is available to allow the inner loops of this program to be optimized, resulting in significant performance improvements despite the apparently poor precision of analysis.

In general, our experience has been that the precision of our non-suspension analysis is intimately tied to the precision of the underlying analysis. The simple abstract domain used in our implementation is adequate for programs that do not have much in the way of dependencies between variables, e.g., that do not use techniques such as difference lists and where there is little aliasing. The precision of our analysis degrades if the underlying analysis is not precise enough to allow us to determine whether or not the procedure demands are being satisfied. The situation can sometimes become fairly subtle. As an example, consider the following procedure:

```
p([]).
p([msg(In,Out)|Msgs]) :- Out is In+100, p(Msgs).
```

In order to allow us infer that `p`/1 is weakly non-suspending, the underlying analysis must be able to provide information about the structure of terms with considerable precision. While such (underlying) analyses have been studied in the literature (see,

Benchmark	Execution Time (μsecs)			RT/M	RU/M
	M	RT	RU		
`aquad`	45600	33242	20569	0.729	0.451
`bessel`	12516	12467	12364	0.996	0.988
`binomial`	4362	3924	5720	0.900	1.311
`chebyshev`	23689	23689	8500	1.000	0.359
`e`	12495	12372	9832	0.990	0.787
`fib`	12330	4774	4711	0.387	0.382
`log`	35025	35432	17198	1.011	0.491
`mandelbrot`	64888	71464	23942	1.101	0.369
`muldiv`	13285	13303	12705	1.001	0.956
`nrev`	7842	8006	8018	1.021	1.022
`pi`	25031	27860	12144	1.113	0.485
`sum`	1691	1691	1694	1.000	1.002
`tak`	13505	4732	5340	0.350	0.395
Geometric Mean				0.844	0.623

Key:
M : memory returns only
RT: register and memory returns (tagged registers only)
RU: register and memory returns (tagged and untagged registers)

Table 2. Output Value Placement in Registers: Performance (`jc` on a Sparcstation-IPC)

for example, [14, 20]), whether or not it is reasonable to expect this degree of precision from the underlying analysis depends on the kinds of programs one expects to encounter and perhaps also on the speed with which the compiler is expected to work (though our experience has been that the time taken for I/O operations can in many cases dominate the overall compilation time, suggesting that within reasonable limits, efficiency of analysis is not as critical as one might imagine). Nevertheless, it is important to note that imprecision in non-suspension analysis is due fundamentally to shortcomings in the underlying analysis: that is, the lost precision in suspension analysis can be recovered simply by improving the underlying analysis, with no changes necessary to the suspension analysis algorithms.

6. Applications

6.1. Returning Output Values in Registers

Most implementations of logic programming languages treat input and output arguments to procedures in a fundamentally asymmetric way: input values are passed in registers, but output values are returned in memory. For programs where predicate modes are known, output arguments can be returned in registers instead. This avoids unnecessary work arising from memory reads and writes due to initialization and dereferencing, and can cause significant performance improvements. For languages that support suspension of activations, however, the problem is complicated by the fact that if a call to a procedure can suspend, the output registers will contain garbage when control returns to the caller (since the suspended computation will not have computed values into them) unless additional work is done—both at suspension time and when the suspended activation is resumed—to ensure that values are propagated correctly. This can become fairly complicated and incur performance penalties. A much simpler solution that works well in practice (see [1]) is to consider returning output values in registers only for procedures that can be guaranteed to be non-suspending. It turns out, however, that weak non-suspension is inadequate for this optimization. The reason is that if an output value is returned in a register, that register must not be overwritten until that value has been used or stored into memory. However, if a procedure p that returns some outputs in registers can only be guaranteed to be weakly non-suspending, it may happen that some awakened goal is executed as soon as p has finished executing, but before the goal that would have used the value returned in a register by p. This would either overwrite the register containing p's output value and thereby produce incorrect results, or would require complex and expensive runtime schemes to save and restore output value registers where necessary. This problem can be avoided if output values are returned in registers only for procedures that can be inferred to be strongly non-suspending.

This optimization (returning output values in registers) has been implemented in `jc`: the interested reader is referred to [1]. Performance results for a number of benchmark programs on a Sparcstation-IPC (with garbage collection turned off) are shown in Table 2.

6.2. Maintaining Unboxed Values

In languages with delay operations, the low level representation of a data object at a particular program point cannot always be predicted in a precise way at compile time, since this depends on whether the value of an expression has been computed or not, which in turn depends on the suspension behavior of the program. The code generated for programs in such languages must, therefore, be able to deal with different kinds of representations that may arise at runtime. There are two different but related issues that arise here. First, it is necessary to be able to determine how a bit pattern, encountered at runtime, is to be interpreted—e.g., as an unbound variable or as a value of a particular type. Second, different data objects may have different sizes: for example, the size of an integer value may not be the same as that of a double precision floating point value. The usual way to address the first problem is to attach a descriptor to each value, to specify how its bit pattern is to be interpreted: such descriptors are usually referred to as *tags* [12, 19]. The second problem is usually handled by making values of different sizes "look the same" by manipulating pointers to them rather than the values themselves: such an indirect representation is often referred to as a *boxed* representation. In general, therefore, operations have to contend with the manipulation of tags and/or a level of indirection, and as a result incur a performance penalty.

This performance overhead is especially serious in numerical computations, because implementations of logic programming languages very often represent floating point numbers as boxed values (see, for example, [4]). This incurs a significant performance penalty, for a number of reasons. First of all, since floating point values are heap-allocated, numerical computations involving boxed floating point values fail to exploit hardware registers effectively, and generate a lot more memory traffic. The allocation of fresh heap cells may also result in additional checks for heap overflow. Finally, the high rate of memory usage also results in increased garbage collection and adversely affects cache and paging behavior. However, if enough information is available, at compile time, about a value at a particular program point, it is possible to (generate code to) maintain the value in its native machine representation, i.e., without any tagging or boxing, and thereby avoid these overheads. For example, in general it is not enough to know that a value will be a number—we need to know whether it will be an integer or a floating point value. Such information can be obtained in various ways, e.g., via type analyses or from programmer annotations: the details are orthogonal to the topic of this paper, and are not discussed further.

The problem of optimizing the low-level representations of objects by maintaining them in untagged and unboxed form becomes more complicated in languages with delay operations because in this case it is no longer enough to have precise type information about an object: it is necessary to guarantee also that for all executions of the program (for the inputs of interest), the value of that object will have been computed by the time control reaches the program point of interest. The reason for this is not difficult to see: since a value in native machine format does not have a descriptor that can be used to identify its type, it may not be possible, in general, to distinguish an unbound variable from an untagged integer or floating point value. We therefore need (weak) non-suspension analysis to identify variables whose values can be guaranteed to have been computed at a particular program point.

This optimization has been implemented in `jc` [2]. At this time, only numeric values, i.e., integers and floating point values, are considered for untagged and unboxed representation. The use of untagged values is not restricted to intraprocedural computations: untagged values may be stored on the stack, passed to other procedures as arguments, and returned from procedures as outputs. Since untagged values may be stored on the stack, the garbage collector must be modified so that it can correctly identify objects in stack frames: this is done by adding a word to each stack frame that can be used by the garbage collector to index into a symbol table that identifies the procedure that frame belongs to and specifies the structure of its stack frames. Currently, untagged values on the heap are not supported because the structure of the heap is a lot less predictable than that of the stack, making the identification of untagged objects during garbage collection more difficult. This has to do primarily with the tagging scheme used by an implementation: if the tagging scheme used by an implementation is rich enough to support descriptors that encode the structure of (some types of) heap-allocated objects, in particular information about elements that are untagged, then the problem with identification of untagged values on the heap goes away. In this case, our approach can be readily extended to handle untagged values on the heap. We are currently considering extensions to our tagging scheme to allow untagged objects on the heap. However, these details are largely orthogonal to the topic of this paper.

Table 3 shows the improvements in speed and memory usage resulting from the use of untagged and unboxed values (garbage collection was turned off for these

Program	Memory Returns			Reg+Mem. Returns		
	T	U	U/T	T	U	U/T
`aquad`	45600	27300	0.599	33242	20569	0.619
`bessel`	12516	11013	0.880	12467	12364	0.992
`binomial`	4362	5919	1.357	3924	5720	1.458
`chebyshev`	23689	8500	0.359	23689	8500	0.359
`e`	12495	9641	0.772	12372	9832	0.795
`fib`	12330	12260	0.994	4774	4711	0.987
`log`	35025	16174	0.462	35432	17198	0.485
`mandelbrot`	64888	24875	0.383	71464	23942	0.335
`muldiv`	13285	12708	0.957	13303	12705	0.955
`nrev`	7842	7851	1.001	8006	8018	1.001
`pi`	25031	11944	0.477	27860	12144	0.436
`sum`	1691	1694	1.002	1691	1694	1.002
`tak`	13505	13462	0.997	4732	5340	1.128
Geometric Mean :			0.727			0.738

(a) Execution Time (μsecs)

Program	Memory Returns			Reg+Mem. Returns		
	T	U	U/T	T	U	U/T
`aquad`	30884	10255	0.3320	23332	544	0.0233
`bessel`	689	418	0.6067	689	452	0.6560
`binomial`	1208	249	0.2061	1026	6	0.0058
`chebyshev`	30002	6	0.0002	30002	6	0.0002
`e`	6005	6	0.0010	6005	6	0.0010
`fib`	6389	6389	1.0000	5	5	1.0000
`log`	28870	12	0.0004	28866	6	0.0002
`mandelbrot`	69533	654	0.0094	69533	654	0.0094
`muldiv`	5	5	1.0000	5	5	1.0000
`nrev`	7842	7851	1.001	8006	8018	1.001
`pi`	20007	9	0.0004	20007	6	0.0003
`sum`	5	5	1.0000	5	5	1.0000
`tak`	7121	7121	1.0000	5	5	1.0000

(b) Heap Usage (words)

Key : T : tagged values; U : untagged values

Table 3. Performance Improvements due to Untagged and Unboxed Objects

timings, so the speed improvements do not take into account reductions in garbage collection time due to reduced heap usage). Programs that involve mostly integer arithmetic may not benefit much from this optimization, since integers do not need to be boxed, and operations on tagged integers are not much more expensive than on untagged ones: this is illustrated by `fib`. However, for programs that involve a lot of floating point computation, the use of untagged values generally leads to significant improvements in speed and memory usage (The `binomial` program is an exception: its slowdown using untagged values is due to the use of C as the back-end compiler for `jc`, and the concomitant lack of control over hardware register allocation). Overall, this optimization produces a speed improvement of about 30% for the programs tested.

6.3. Inlining

Inlining refers to the replacement of a procedure call by (the appropriate instance of) the body of the called procedure. One reason inlining is potentially important for logic-based languages is that such languages lack nestable iterative constructs, but instead implement iteration using tail recursive procedures. This can incur significant performance penalties, relative to traditional imperative languages, due to additional procedure calls. As an example, the multiplication of two $n \times n$ matrices requires three different tail-recursive procedures in a logic-based language, one of which is called n times and the other n^2 times. Thus, the multiplication of two 100×100 matrices—which requires no procedure calls in a nested-loops Fortran implementation—can incur the cost of $10,100$ procedure calls in a straightforward implementation of a logic-based language.

Inlining is conceptually straightforward in languages that do not support delay primitives. The situation is more complicated for languages that allow suspension because of the need to save state information when an activation suspends and restore it when it is resumed. An implementation that allows suspension at arbitrary program points has to deal with saving and restoring arbitrary amounts of local state, leading to implementation complications and runtime performance overheads. An alternative approach—taken, for example, by SICStus Prolog [3] and `jc` [13]—is to allow suspension to occur only at specific predetermined program points. Such schemes require the manipulation of only a limited amount of state during suspension and resumption and are much simpler to implement than the previous scheme, but they essentially rule out inlining since inlining a procedure that may suspend can cause suspension to occur at arbitrary program points. Thus, inlining presents implementation problems in languages that support delay primitives no matter how we deal with suspension and resumption. However, these problems disappear if we restrict inlining to procedures that can be guaranteed to not suspend. In particular, note that in traditional algorithms designed for imperative languages—for example, the matrix multiplication routine mentioned above, or any other scientific program—computations necessarily do not suspend. This implies that in logic programs implementing such algorithms, procedures can be inlined in order to avoid the overheads of additional procedure calls associated with the lack of nestable iterative constructs in logic programming languages.

With regards to the implementation of inlining, it is not difficult to see that in order to inline a goal L, it suffices to check whether L is weakly non-suspending. However, depending on the language semantics for the scheduling of awakened goals, it may also be necessary to determine whether L is strongly non-suspending. This is because otherwise, inlining L may change the behavior of the program if there are awakened goals that are required to be executed as soon as they are awakened: if L is not inlined, such awakened goals would be executed before L, but if L is inlined this is not possible (otherwise we are faced with the earlier problem of saving and restoring an arbitrary amount of state so that L can be correctly executed later).

Currently, the `jc` system implements a special case of this optimization: in general, numerical operations such as '`X = Y+Z`' occurring in clause bodies are compiled as out-of-line procedure calls where the called procedure checks whether the operands are available, and suspends if they are not. However, numerical operations that can be guaranteed to not suspend are compiled into in-line code. This is significantly faster and more compact than the general case. We have not implemented general procedure inlining at this time, but intend to do so soon.

6.4. Reducing Suspension Tests

Obviously, a procedure p that has been inferred to be non-suspending will not suspend, and therefore need not test its arguments to check whether it should suspend. For this, it suffices to verify that at each call site L for p we have $A_L \sqsubseteq \mathit{demand}(p)$, where A_L is the abstract environment obtained at the end of weak non-suspension analysis at the program point immediately before L. The pragmatic benefits of this optimization depend greatly on the details of how suspension is implemented. For example, in `jc` we have optimized the system for non-suspending code, and suspension testing is done at the end after all other tests, so the main benefit of deleting suspension tests would be a reduction in code size. However, Marriott *et al.*, using SICStus Prolog 2.1, report significant performance improvements from the removal of suspension tests [18].

6.5. General Prolog Optimizations

In recent years there has been a great deal of work on optimization of (non-suspending) Prolog. For example, Märien *et al.* show that significant performance improvements are possible for Prolog programs if the lengths of dereference chains can be statically predicted [17], while Van Roy shows that execution speed can be improved significantly if the initialization of variables can be avoided [21]. All of these optimizations become applicable for (strongly) non-suspending programs in logic programming languages with delay mechanisms.

7. Discussion

While delay mechanisms can be very convenient for programming purposes, they make control flow difficult to predict and thereby render many low level compiler optimizations difficult or impossible. We have described simple compiler analyses to identify program fragments whose control flow can be guaranteed to not be affected by suspension and resumption of activations, and several low level optimizations that rely on this information. We have implemented weak non-suspension analysis in the `jc` system: this turns out to be of fundamental importance to the compiler optimizations we perform. In this section we discuss some of the performance

Program	Execution Time (μsecs)				J/gcc:2	J/cc:2	J/cc:4
	J	gcc:2	cc:2	cc:4			
`aquad`	20569	16604	28883	26433	1.238	0.712	1.119
`bessel`	12364	12644	20635	20123	0.978	0.599	0.614
`binomial`	5720	5075	8894	6098	1.127	0.643	0.938
`chebyshev`	8500	7207	18067	18065	1.179	0.470	0.470
`e`	9832	9392	10148	10154	1.047	0.969	0.968
`fib`	4711	4727	4598	4584	0.997	1.025	1.028
`log`	17198	17487	35029	35029	0.984	0.491	0.491
`mandelbrot`	23942	19403	78423	46195	1.234	0.305	0.518
`muldiv`	12705	10605	11688	11669	1.193	1.087	1.089
`nrev`	8018	4904	4900	4272	1.635	1.636	1.877
`pi`	12144	11998	22528	22520	1.012	0.529	0.529
`sum`	1694	1606	1606	406	1.055	1.055	4.172
`tak`	5340	4384	4085	4070	1.218	1.298	1.303
Geometric Mean :					1.134	0.752	0.940

Key:

J :	`jc -O`
gcc:2 :	`gcc -O2`
cc:2 :	`cc -O2`
cc:4 :	`cc -O4`

Table 4. The speed of `jc` compared to optimized C (Sparcstation IPC)

improvements accruing from these optimizations. The numbers shown are for a Sun Sparcstation IPC with 36 MB of main memory, running Solaris 2.3, with garbage collection turned off.

The optimization of returning output values in registers is discussed in Section 6.1. Performance numbers are given in Table 2. The average speed improvement is about 15% even without the use of unboxed values, which is quite significant for this kind of low-level optimization. For many programs the improvements are much greater: for example, the speed of the `tak` benchmark almost triples. When unboxed values are maintained, and the passing of arguments and return values via unboxed registers allowed, the gains are even greater, averaging about 37%. Furthermore, while for most programs the improvements are primarily in speed and, in some cases, in the amount of stack space used (which can decrease because fewer variables may have to be stored on the stack when outputs are returned in registers), a few programs, such as `aquad` and `fib`, exhibit significant reductions in the amount of heap space used as well because fewer "unsafe" variables are necessary.

The optimization of maintaining unboxed values is discussed in Section 6.2, with performance numbers given in Table 3. Again, the speed improvements of about 26% on the average are quite significant. Heap usage also improves dramatically, in many cases to the point where giving an "improvement ratio" seems meaningless.

Interestingly, it can be seen that on a few programs there is actually a small loss in performance when the two optimizations discussed so far are combined, and output values are allowed to be returned in unboxed registers. This is due partly to suboptimal placements of format conversion operations in some cases, leading to additional conversions from tagged to untagged form and back and partly to the use of C as the target language, and the concomitant lack of control over the register allocation decisions made by the underlying C compiler. However, it can be argued that these numbers provide a conservative lower bound on the performance level achievable using such low level optimizations.

Table 4 shows the absolute performance of `jc` compared to heavily optimized C code written in a style one would expect of a competent C programmer, i.e., using iteration rather than recursion wherever possible, using macros and avoiding function calls where this is reasonable, and relying heavily on destructive assignment. For the simple programs we tested, `jc` is only about 13% slower than C code compiled under `gcc` and optimized at the highest level possible. For the Sun C compiler `cc`, the results are even better: `jc` is almost 25% faster than `cc -O2` and 6% faster than `cc -O4`.[6] Moreover, `jc` outperforms `cc` on precisely those programs—namely, floating-point intensive computations—where one would expect a dynamically typed declarative language to do considerably worse than a statically typed imperative language. The superior performance of `jc` compared to `cc` is due partly to the fact that `cc` does not generate especially good code for floating point computations; however, as Tables 2 and 3 illustrate, this would not have been possible without extensive low-level optimization. One program where `cc` performs significantly better than `jc` is `sum`: this is due greatly to the fact that at optimization level `-O4`, `cc` inlines a user-defined function, while `jc` has not yet implemented this kind of inlining. On small recursion-intensive benchmarks, the presence of register windows on the SPARC architecture removes the need to save and restore registers at recursive calls; because of this, and parameter passing in hardware registers, procedure calls at the C level are not as expensive as in older architectures, and so the performance of the C code on recursion-intensive programs such as `aquad`, `fib` and `tak` are not as bad as one might expect them to be. Overall, our numbers illustrates the fact that it is possible for logic programs to outperform imperative programs that are written in a natural imperative style. This illustrates the fundamental importance of the sorts of low level optimizations we have described in attaining good performance. Since all of our optimizations depend fundamentally on information about non-suspension, the analyses described here are crucial for attaining this level of performance.

[6] Since `jc` uses `gcc` as its back end translator, one might wonder whether this comparison with `cc -O4` is "fair" or question what it proves. We claim that `jc`'s use of `gcc` is purely a matter of convenience: we could, in principle, have achieved the same results by writing our own back ends and using all of `gcc`'s technology in it. The point of this comparison, therefore, is merely to show that simple dataflow analyses and careful attention to low level concerns can allow implementations of declarative languages to attain performance that is competitive with the performance of imperative programs written in an imperative style. We acknowledge, of course, that performance comparisons between different languages are fundamentally dubious and very often have a strongly religious flavor, and caution the reader against reading too much into these results.

8. Related Work

The work most closely related to this is that of Marriott *et al.* [18], who also consider the analysis of sequential logic programs with delay primitives, and of Hanus, who considers the analysis of functional logic programs using residuation [15]. The main difference between their work and that reported here is that of focus. While our work is aimed at identifying program fragments that will not suspend and dataflow behavior for such fragments, the work of both Marriott *et al.* and Hanus is aimed at accurately approximating the suspension behavior of literals and predicates, including when a particular atom is delayed, when it is awakened, and which atoms are delayed at some program point. Because of this, both Marriott *et al.* and Hanus make more assumptions about the scheduling policy for reawakened goals than we do—specifically, they assume that goals are executed as soon as they are awakened—and use this to obtain a more precise description of the behavior of suspending programs. This additional precision comes at a price, however: experiences with a prototype implementation of the analysis of Marriott *et al.* indicate that large amounts of time and space may be needed to analyze programs of even modest size if there are many goals that can suspend [11]. Moreover, the details of such an approach become somewhat complicated under more elaborate scheduling policies, e.g., the priority-based system of KL1 [5]. Our approach, by contrast, makes no assumptions about how suspended goals might be scheduled after they are awakened. This results in a less precise analysis for computations that may suspend; on the other hand, the fact that our approach does not try to keep track of the set of suspended goals and predict which goals might be awakened at various program points simplifies the implementation significantly and improves its efficiency considerably.

Also related is work on analysis of concurrent logic languages, e.g., the deadlock analyses described in [6, 7]. The primary difference between the work of these authors and that described here is that they make no assumptions regarding the scheduler (we assume that goals in a clause body are executed from left to right), and as a result are faced with the formidable problem of accounting for all possible interleavings of primitive actions during the execution of a program. Moreover, it seems difficult to reason about the kind of suspension behavior that we are interested in without making any assumptions at all about the order in which the body goals of a clause are executed, so in general the properties considered by these authors are very different from those we consider.

9. Conclusions

While language mechanisms that allow the execution of a goal to suspend until certain variables have become bound have become increasingly popular in logic programming languages, they can make the execution behavior of programs difficult to predict, and thereby make many traditional compiler optimizations inapplicable. This paper discusses two different notions of non-suspension in sequential logic programs with delay mechanisms, describes simple dataflow analyses to identify non-suspending programs, and discusses various low-level optimizations based on this information. Experimental results from the `jc` system are presented to show that such analyses can improve the performance of programs significantly.

Acknowledgements

Thanks are due to Will Winsborough for many interesting discussions, and to the anonymous referees for their thoughful comments, which helped improve both the contents and the presentation of the paper. This work was supported in part by the National Science Foundation under grant number CCR-9123520.

REFERENCES

1. P. Bigot, D. Gudeman, and S. K. Debray, "Output Value Placement in Moded Logic Programs", *Proc. Eleventh Int. Conf. on Logic Programming*, June 1994, pp. 175–189. MIT Press.

2. P. A. Bigot and S. K. Debray, "A Simple Approach to Supporting Untagged Objects in Dynamically Typed Languages", *Proc. 1995 Int. Symp. on Logic Programming*, Dec. 1995. MIT Press.

3. M. Carlsson and J. Widen, *SICStus Prolog User's Manual*, Swedish Institute of Computer Science, Oct. 1988.

4. M. Carlsson, "The SICStus Prolog Emulator", Technical Report T91:15, Swedish Institute of Computer Science, Sept. 1991.

5. T. Chikayama, T. Fujise, and D. Sekita, "A Portable and Efficient Implementation of KL1", *Proc. Int. Symp. on Programming Language Implementation and Logic Programming*, Sept. 1994, pp. 25–39

6. M. Codish, M. Falaschi, and K. Marriott, "Suspension Analysis for Concurrent Logic Programs", *Proc. Eighth Int. Conf. on Logic Programming*, June 1991, pp. 331–345. MIT Press.

7. C. Codognet, P. Codognet, and M. Corsini, "Abstract Interpretation of Concurrent Logic Languages", *Proc. 1990 North American Conf. on Logic Programming*, Nov. 1990, pp. 215–232. MIT Press.

8. P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Apporoximation of Fixpoints", *Proc. Fourth ACM Symp. on Principles of Programming Languages*, 1977, pp. 238-252.

9. S. K. Debray, P. López García, M. Hermenegildo, and N.-W. Lin, "Lower Bound Cost Estimation for Logic Programs", manuscript, April 1994.

10. I. Foster and S. Taylor, "Strand: A Practical Parallel Programming Tool", *Proc. 1989 North American Conference on Logic Programming*, Cleveland, Ohio, Oct. 1989, pp. 497-512. MIT Press.

11. M. Garcia de la Banda, personal communication, Oct. 1994.

12. D. Gudeman, "Representing Type Information in Dynamically Typed Languages", Technical Report TR 93-27, Dept. of Computer Science The University of Arizona, Oct. 1993.

13. D. Gudeman, K. De Bosschere, and S.K. Debray, "`jc`: An Efficient and Portable Sequential Implementation of Janus", *Proc. Joint Int. Conf. and Symp. on Logic Programming*, Nov. 1992, pp. 399–413. MIT Press.

14. G. Janssens, *Deriving run-time properties of logic programs by means of abstract interpretation*, Ph. D. thesis, Katholieke Universiteit Leuven, Belgium, March 1990.

15. M. Hanus, "On the Completeness of Residuation", *Proc. Joint Int. Conf. and Symp. on Logic Programming*, Nov. 1992, pp. 192–206. MIT Press.

16. B. Le Charlier and P. Van Hentenryck, "Reexecution in Abstract Interpretation of Prolog", *Proc. Joint Int. Conf. and Symp. on Logic Programming*, Nov. 1992, pp. 750–764. MIT Press.

17. A. Mariën, G. Janssens, A. Mulkers, and M. Bruynooghe, "The Impact of Abstract Interpretation: An Experiment in Code Generation", *Proc. Sixth Int. Conf. on Logic Programming*, June 1989, pp. 33-47.

18. K. Marriott, M. Garcia de la Banda, and M. Hermenegildo, "Analyzing Logic Programs with Dynamic Scheduling", *Proc. 21st. ACM Symp. on Principles of Programming Languages*, Jan. 1994, pp. 240–252.

19. P. A. Steenkiste, "The Implementation of Tags and Run-Time Type Checking", in *Topics in Advanced Language Implementation*, ed. P. Lee, pp. 3–24. MIT Press, 1991.

20. P. Van Hentenryck, A. Cortesi, and B. Le Charlier, "Type Analysis of Prolog using Type Graphs", *J. Logic Programming* vol. 22 no. 3, March 1995, pp. 179–209.

21. P. Van Roy, *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, 1990.

22. W. Winsborough, personal communication, May 1994.