Compiling Dataflow Analysis of Logic Programs

Jichang Tan I-Peng Lin

Department of Computer Science and Information Engineering
National Taiwan University
Taipei, 10764
Taiwan
jctan@csman.csie.ntu.edu.tw

Abstract

Abstract interpretation is a technique extensively used for global dataflow analyses of logic programs. Existing implementations of abstract interpretation are slow due to interpretive or transforming overhead and the inefficiency in manipulation of global information. Since abstract interpretation mimics standard interpretation, it is a promising alternative to compile abstract interpretation into the framework of the WAM (Warren Abstract Machine) for better performance. In this paper, we show how this approach can be effectively implemented in a low-cost manner. To evaluate the possible benefits of this approach, a prototype dataflow analyzer has been implemented for inference of mode, type and variable aliasing information of logic programs. For a subset of benchmark programs in [15], it significantly improves the performance by a factor of over 150 on the average.

1 Introduction

Like many conventional languages, the performance of the logic programming language Prolog has been significantly improved through compilation [21]. In particular, the WAM (Warren Abstract Machine) [22] is a virtual machine that has emerged as the de facto standard for the compilation and implementation of Prolog. The benefits of the WAM basically rest on local optimizations through a simple intra-procedural (clause-level) program analysis. However, substantial optimizations [12, 13, 15, 18, 23] all depend on inter-procedural information such as *mode* (the input and output direction of procedure arguments), *type* (the possible bindings of data-structures for program variables), and *variable aliasing* (the co-references of logical variables). The dataflow information is not only important to enable more performance improvement of Prolog, it also paves the way for efficient implementation of different classes of logic programs which support Independent And-Parallelism [23, 13], concurrent processes [2], or constraint satisfaction [10].

Instead of inferring various information about a program through different analysis procedures, a technique called *abstract interpretation* can be used for global dataflow analyses of a programming language based on a unified model [4]. The idea is to execute a program to be analyzed over a finite *abstract domain* rather than the possibly infinite *concrete domain* over which the language is defined. This includes the definition of a mapping from elements and operators of the standard (i.e., concrete) domain to those of the abstract domain. Abstract interpretation is then carried out by a *fixpoint* execution of the abstracted program. The result of the execution gives information about the original program over the specified abstract domain. For abstract interpretation of logic programs, there are two major issues. First, the unification procedure and variable substitutions have to be redefined over the abstract domain. For soundness of the dataflow analysis, there are certain criteria to be met in their definitions. Second, termination and completeness of the control scheme must be ensured. This means a different interpretation strategy is required instead of the top-down depth-first method used by ordinary Prolog systems. Due to general interests and applications, both issues have been extensively studied [1, 2, 3], [5]-[16], [20].

To the best of our knowledge, all global dataflow analyzers for logic programs have been implemented on top of Prolog. The implementation can be based on a meta-circular interpreter [6, 17] or a program transformer [5, 23]. The *meta-interpreting approach* directly interprets programs by using the redefined control scheme and unification procedure over an abstract domain. Instead of direct interpretation, the *transforming ap-*

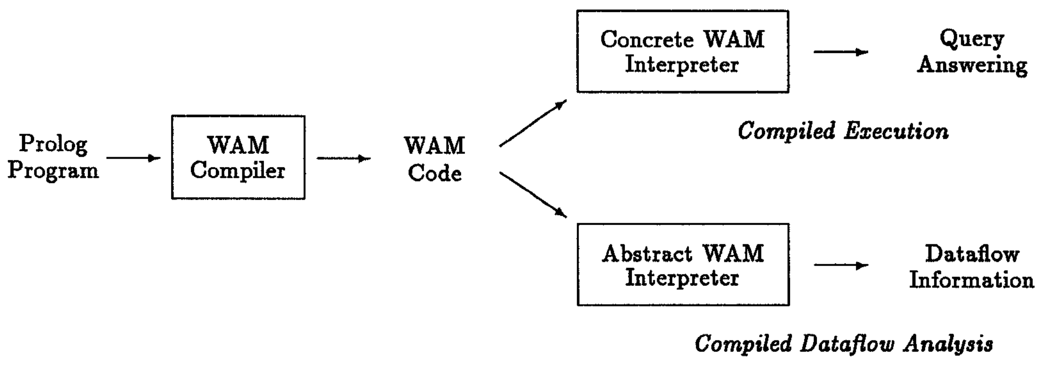

Figure 1: Compiled Execution and Compiled Dataflow Analysis

proach first partially evaluates the programs over the abstract domain, and then runs transformed programs to do the abstract interpretation. For efficiency, a different control scheme can be incorporated [11, 12, 15]. However, both approaches are slow due to the interpretive or transforming overhead and the inefficiency in manipulation of global dataflow information. Since abstract interpretation mimics standard interpretation, it should be a promising alternative to compile abstract interpretation into a similar framework like the WAM for better performance.

Rather than reinventing a new virtual machine and a dedicated compiler for it, it is much cheaper to *reuse* the framework of the WAM and current compilers to the maximal extent. This idea turns out to be quite successful because there is almost no design in the WAM which is restricted to execution over the concrete domain. For a typical abstract domain such as the one considered in this paper, the instruction set of the WAM can be easily reinterpreted according to the redefined control scheme and unification procedure. The outline of this approach is illustrated in Figure 1. We will use the term *abstract WAM* for the rest of this paper to refer to this reinterpretation of the WAM for dataflow analysis, and the term *concrete* or *standard WAM* for the original framework of Prolog compilation. To evaluate the possible benefits of this approach, we have implemented a prototype dataflow analyzer for the inference of mode, type and variable aliasing information of logic programs. For a subset of benchmark programs in [15], it significantly improves the performance by a factor of over 150 on the average.

The most obvious source of performance improvements is the removal of the overhead of interpretation and transformation incurred in other approaches. As a comparison, the technique of compilation improved the performance of Prolog by a factor of about 30 in Warren's original work [21]. Of course, the WAM code does not come for free. However, it appears that a translation to unification primitives like those of the WAM before dataflow analyses is indispensable for other approaches as well [15, 17]. Another important source of performance improvements is the efficiency in manipulation of global dataflow information. The handling of global information is required in abstract interpretation while it is discouraged by existing implementations of Prolog. Therefore, concerns for its efficiency of implementation have been explicitly addressed in the work of [23, 15, 17]. For our approach, it does not pose any problem because we use a conventional language like C for the implementation.

2 Preliminaries

2.1 Warren Abstract Machine

The basic idea of optimization is to generate specialized code for different cases of source programs. This works for Prolog as well as other conventional languages. Consider executing the head of a clause with two arguments in a Prolog program:

$$p(a, [f(V)|L]) \leftarrow \ldots$$

If it is directly carried out by an interpreter, a *general* unification procedure will be used for both arguments 'a' and '$[f(V)|L]$' when this predicate is invoked. Since the interpreter cannot take advantage of knowledge of the source program, the information has to be generated at run-time. Due to the abstract semantics of logic programs, this overhead can be considerably expensive. If the head is compiled into the framework of the WAM, a sequence of unification primitives will be generated as the one given in Figure 2. We can see that there are specialized unification subroutines, i.e., the WAM instructions, defined for each type of term. For example, the instructions `get_const`, `get_list`, and `get_struct` are defined for constants, lists, and structures that appear in the head of a clause. For a complex

```
get_const  a,A1    % The first coming-in argument must unify with 'a'.
get_list   A2      % The second argument has to be a list.
unify_var  X3      % Keep the first subterm (car) of the list using a temporary register.
unify_var  L       % Unify its next subterm (cdr) with L.
get_struct f/1,X3  % The car of the list must be an f/1 term.
unify_var  V       % And its subterm unifies with V.
```

Figure 2: The WAM code instructions for the head of the clause

term such as $[f(V)|L]$, the generated instructions will proceed in a *breadth-first* order. For example, the two `unify_var` instructions after the `get_list` instruction correspond to the unification of the first level subterms, and the third `unify_var` instruction which is after the `get_struct` instruction corresponds to the unification of the second level subterm. Through this sequence of primitive unification subroutines, much of the overhead in the general unification procedure is eliminated.

Based on similar ideas, the instruction set of the WAM can be defined. According to Warren [22], they are classified into *get, put, unify, procedural,* and *indexing* instructions. The *put* instructions are specialized *get* instructions which are dedicated to term construction in the body of a clause. The *procedural* instructions are responsible for the control transfer and environment allocation associated with procedure calling. The *indexing* instructions, the last group of instructions, control the access to different clauses of a predicate procedure. For other relevant details of the WAM, appropriate descriptions will be given in the following sections.

2.2 The Control Scheme for Abstract Interpretation

Due to the requirement of *compile-time* completeness and termination, researchers have proposed several control schemes for abstract interpretation of logic programs [1, 3, 8, 9, 16, 20]. In this paper, we only consider the *extension table-based* method [8] (a variant of *OLDT Resolution* [16]) because it is simple and can be easily incorporated into the compilation approach presented.

The basic idea of the extension table-based method is to employ a *dynamic programming technique* to program interpretation in which intermediate results are saved and later reused to avoid redundant work and to prevent the execution from diving into an infinite loop. The table is simply a memo structure containing a set of pairs of "calling pattern" (the description of the call) and its "success patterns" (the description of the successful return) which have been found so far. Instead of repeated computation, the extension table is consulted first whenever we try to interpret a calling pattern. If any such solutions are found, they are returned directly; if the call has been made earlier but no solution has been recorded, this call is considered failed. To ensure *completeness*, the interpretation has to execute in multiple iterations such that another iteration will be initiated whenever a new success pattern is found in one iteration. When the interpretation terminates, the table will provide approximate run-time information for each predicate in the program to be analyzed.

The outline of an extension table-based abstract interpretation can be described as follows. The interpretation starts from a given top-level calling pattern. Globally, it executes in the standard top-down depth-first order. When a clause is successfully executed, the extension table is updated with the success pattern. However, the control is then transferred to the next clause of the predicate procedure by an *artificial failure* such that all clauses will be explored. If there are multiple success patterns found for a calling pattern, they may be summarized into a single pattern by taking the *least upper bound* based on the abstract domain used. By keeping at most one success pattern for a calling pattern, it also simplifies the control flow of the interpretation since any invocation of a predicate always results in a deterministic return. Through iterative deepening, the interpretation will eventually terminate for a finite abstract domain when *least fixpoint* of the dataflow analysis is reached.

There are a few analyzers [5, 17, 23] which are based on this control scheme. However, the extension table is expensive to implement in Prolog because it is an inherently global data structure.

3 An Abstract Domain for Global Dataflow Analysis

To infer run-time information about a logic program, we classify all run-time terms into interesting subsets that form a complete lattice ordered by set inclusion. Each subset is represented by an *abstract type* in the abstract domain. An instance of an abstract type is referred to as an *abstract term* of that type. The abstract domain is briefly defined as follows.

- any represents the set of all terms, the top element ($\top$) of the abstract domain.
- nv represents all non-variable terms.
- ground (or g, for short) represents the set of ground terms.
- const represents the set of constant terms. It is the union of two subsets which are represented by atom and integer.
- struct($f/n, \alpha_1, .., \alpha_n$) represents the set of structure terms which principal functor is f/n and each argument is of type α_i ($1 \leq i \leq n$) in the abstract domain. For sake of clarity, the set of terms will be written as $f(\alpha_1, .., \alpha_n)$ when there is no ambiguity.
- α-list represents the set of lists with a type parameter α. It is a precise type for the union of constant '[]' (the empty list) and [α|α-list] (the structure terms struct('.'/2,α, α-list)) while [$\cdot$|$\cdot$] in general remains to be an ordinary structure term. For example, glist stands for the set of terms [], [g], [g, g], and so forth. Due to the extensive application of lists in logic programs, list-awareness is usually very useful.
- var represents the set of all variables.
- empty represents the set of non-existing term, the bottom element ($\perp$) of the abstract domain.

To make the abstract domain finite, we impose a *term-depth restriction* for complex abstract terms in the analysis. Given an artificial limit k, the subterms with depth k or greater in a complex term will be simplified to other simple elements in the abstract domain. Consequently the precision of the analysis for general recursive datatypes is traded for a guarantee of analysis termination.

The abstract domain presented here is actually a slightly simplified version of the one used by Taylor's analyzer [17]. The dataflow analysis based on this abstract domain implements an inference of mode, type, and variable aliasing information in logic programs. This domain is considerably more complex than the one used by the Aquarius analyzer [15] whose run-time performance will be compared to that of our analyzer in Section 6.

4 Compiling Abstract Unifications

4.1 Representation and Manipulation of Abstract Terms

Since each abstract term represents a set of terms in the concrete domain, *abstract unifications* can be characterized by *set unification* as follows. Let T_1 and T_2 be two sets of terms. Set unification of T_1 and T_2 can be defined as:

$$\text{s\_unify}(T_1, T_2) \;=\; \{t \mid t = \mathit{unify}(t_1, t_2) : t_1 \in T_1 \wedge t_2 \in T_2\}$$

Abstract substitution can also be formalized as a set of concrete substitutions:

$$\{X/T\} \;=\; \{X/t \mid t \in T\}$$

where X is a variable occurred in T_1 or T_2, and T is a set of terms. More theoretical details can be found in [1, 6]. Here are some examples for the abstract domain considered:

- s_unify(any,ground) = ground.
- s_unify(any, $f(X, Y)$) = f(any, any) with an abstract substitution $\{X/\text{any}, Y/\text{any}\}$.
- s_unify(glist,[Head|Tail]) = [g|glist] with an abstract substitution $\{\mathit{Head}/\text{g}, \mathit{Tail}/\text{glist}\}$.

By these examples, we can see that the instances of most abstract types, such as any or list, are similar to logical variables in that they become *more specific* through unifications. It is therefore natural to represent these abstract terms like variables such that each of them is encoded in a single word before unification, and can be instantiated to another type of term or a complex structure later. They will also be subscripted hereafter to stand for different instances. Finally, the definition of abstract substitution should be extended to include substitutions for these abstract terms in addition to that for variables. Instances of atom, integer and specific structure terms such as f(atom), however, remain unchanged through unifications because there is no smaller (i.e., more specific) element other than '$\perp$' in the abstract domain considered.

Based on the definitions, consider the abstract interpretation of the head of the example clause in Section 2:

$$p(a, [f(V)|L]) \leftarrow ...$$

```
                     % call p(atom, glist_1);
get_const  a, A1     %                         { (1) Succeeds for a ≈ atom.          }
get_list   A2        % glist_1 ← [g_1|glist_2];  { (2.1) Get a [·|·] instance of glist. }
unify_var  X3        % X3 ← g_1;                { The car of glist_1                   }
unify_var  L         % L ← glist_2;             { The cdr of glist_1                   }
get_struct f/1, X3   % X3↑ ← f(g_2);            { (2.2) Get an f(·) instance of g.     }
unify_var  V         % V ← g_2;                 { The argument of f/1                  }
```

Figure 3: The WAM Code Reinterpreted

over the abstract domain given a calling pattern $p(\text{atom}, \text{glist}_1)$. In breadth-first order, the abstract unification for this particular head can be decomposed into a sequence of three simple s_unify's:

(1) s_unify(atom,atom) = atom with an empty substitution { }.

(2.1) $\text{s\_unify}(\text{glist}_1, [\cdot|L]) = [\text{g}_1|\text{glist}_2]$ with a substitution $\{\text{glist}_1/[\text{g}_1|\text{glist}_2],\ L/\text{glist}_2\}$.

(2.2) $\text{s\_unify}(\text{g}_1, f(V)) = f(\text{g}_2)$ with a substitution $\{\text{g}_1/f(\text{g}_2),\ V/\text{g}_2\}$.

Composing the substitutions, the head of the clause succeeds with an abstract substitution:

$$\{\text{glist}_1/[f(\text{g}_2)|\text{glist}_2], L/\text{glist}_2, V/\text{g}_2\}.$$

4.2 Reinterpretation of the Instruction Set

To continue the refinement, the three s_unify's can be mapped into the sequence of WAM code instructions described in Section 2 as given in Figure 3. By this example, it is not difficult to define an appropriate reinterpretation of these WAM instructions over the abstract domain. For example, the algorithm to reinterpret `get_list` is outlined in Figure 4 where the following primitive operations about abstract terms are required.

- *Primary Approximation.* Function $\mathit{AbsType}(T)$ is defined to approximate a term T, either concrete or abstract, to an abstract type regardless of its subterms. For example, $\mathit{AbsType}(a) = \text{atom}$, $\mathit{AbsType}(\text{glist}) = \text{glist}$, $\mathit{AbsType}(f(V)) = f(\top)$ or simply denoted by struct$(f/1)$.
- *Approximate Unifiability.* Ignoring the subterms, two terms T_1 and T_2 are unifiable, denoted by $T_1 \approx T_2$, if $\text{s\_unify}(\mathit{AbsType}(T_1), \mathit{AbsType}(T_2)) \neq \emptyset$. For example, $a \approx$ atom, glist $\approx [\cdot|\cdot]$, and g $\approx f(\cdot)$ have been used for the instructions `get_const`, `get_list`, and `get_struct` $f/1$, respectively, in abstract interpretation of the call $p(\text{atom}, \text{glist}_1)$.
- *Complex-Term Instantiation* (ComplexTermInst). If an abstract term is approximately unifiable with a complex term, the generation of the unification result will be necessary for subsequent subterm unifications. For example, the term $[\text{g}_1|\text{glist}_2]$ and $f(\text{g}_2)$ are complex-term instantiations of glist_1 and g_2, generated by the instructions `get_list` and `get_struct` $f/1$, respectively.

If we represent each run-time object by a tag and a value encoded in a computer word as in the standard WAM [22], the primary approximation function *AbsType* simply returns the tag of a simple object. For predefined abstract types (e.g. any or g), tabulation is sufficient to implement approximate unifiability and complex-term instantiation. For inferred data-types, such as glist, a type-description database will be necessary.

The instructions `get_struct` and `get_const` shown in the example can be reinterpreted in a similar way, while the interpretation of instruction `unify_var` over the concrete domain can be completely reused over this abstract domain.

5 Compiling the Control Scheme

Consider a simple predicate procedure p and its WAM code as follows.

$p(X) \leftarrow q,\ r(X).$	% clause $p.1$
$p(a).$	% clause $p.2$

```
p.1:  get args of p(X)
      call q
      put args of r(X)
      execute r(X)
```

```
get_list $A_i$ ≡
begin
    deref($A_i$);                              { Same as standard WAM interpretation }
    if concrete($A_i$) then                    { True for atom, integer, struct, list, and var }
        concrete WAM code definition
    elsif $A_i \approx$ struct('.'/2) then     { True for ground, nv, any and list }
        $A_i\uparrow \leftarrow$ tag(list,$H$); Trail($A_i$);       { Generate a $[\cdot|\cdot]$ instance of $A_i$ on Heap }
        $S \leftarrow$ ComplexTermInst($A_i$, $H$);   { Set $S$ pointer to the con-cell for subsequent }
        $H \leftarrow H + 2$; $Mode \leftarrow$ Read;    { subterm unifications and proceed in read mode }
    else fail
    fi
end
```

Figure 4: The Outline of the Reinterpreted get_list Instruction

```
p.2:  get args of p(a)
      proceed
```

The instruction proceed corresponds to a successful *return* of a clause, instruction call corresponds to a predicate *procedure invocation*, and instruction execute is equivalent to a call immediately followed by a proceed (the last-call optimization). If we apply the extension table-based interpretation scheme described in Section 2 to the source program, the result is a set of transformed programs:

```
$p'(X) \leftarrow abstract(X, X_\alpha)$,
        if $explored(p(X_\alpha))$ then $lookupET(p(X_\alpha))$
        else $assert(explored(p(X_\alpha)))$, $p(X_\alpha)$
        fi .

$p(X) \leftarrow q', r'(X), updateET(p(X)), fail.$
$p(a) \leftarrow updateET(p(\text{atom})), fail.$
$p(Lub) \leftarrow lookupET(p(Lub)).$
```

where the predicate *explored*(P) is true if a calling pattern P has been explored in current iteration. The predicate *lookupET*(P) and *updateET*(P) are used to look up and update the extension table entry for P respectively. Every invocation of predicate p in the source program is replaced by an invocation of an artificially-introduced predicate p' such that the calculation of the calling pattern (through *abstract*) and the consulting of the extension table will be performed before exploration of p's clauses. The transformed predicate p is a deterministic procedure of the original code such that it updates the extension table for p and then fails to the next clause if a clause of p has been completely executed. When all clauses have been explored, the summarized success pattern, if any, will be returned.

This control scheme can be incorporated into the abstract WAM as follows. Instruction call should be reinterpreted to execute like the artificially-introduced predicate by calculating calling patterns, consulting the extension table, or exploring the clauses. Instruction proceed should be reinterpreted as the part of code "*updateET, fail*" in the end of each transformed clause and "*lookupET*" when the original clauses are exhausted. Instruction execute can simply be reverted to a call followed by a proceed. The reinterpretation technique can be better explained by the annotated WAM code sequence in Figure 5.

In this discussion, we have intentionally omitted some relevant WAM instructions, including some *procedural* instructions and all *indexing* instructions. Their reinterpretation is almost identical to that of a standard WAM with few exceptions. For instance, creation and reclamation of backtracking points would better be incorporated into instructions call and proceed rather than instructions try and trust. By such reinterpretation, the WAM code compiler and the code it generates can be reused without any modification.

6 Implementation and Performance Evaluation

To evaluate the benefits of the proposed approach, we have implemented a prototype analyzer. It is written in C and ported to several platforms. The input WAM code programs are generated by the PLM Prolog compiler[1] for a subset of benchmark programs [16]. The analyzer is about 2500 lines of code, including lots of comments, debugging and conditional compilation statements. In comparison to the size of other analyzers reported (from 1200 lines [16] to over 3000 lines [18]

[1]The original code was from Peter Van Roy [15] and modified by Herve' Touati for the Aquarius Project at UC Berkeley, 1989.

```
p.1:  get args of p(X)
      call q                % If explored(q) then ... else ... call q fi .
      put args of r(X)
      call r(X)             % ... if explored(r(X_α)) then ... else ... call r(X_α) fi .
      proceed               % ⇒ updateET(p(X)), fail.
p.2:  get args of p(a)
      proceed               % ⇒ updateET(p(atom)), fail.
                            % last clause of p ⇒ lookupET(p(Lub)).
```

Figure 5: The Reinterpretation of the Control Instructions

of Prolog code), the implementation cost of this analyzer is relatively low considering the precision of the analysis.

Important implementation features of the analyzer are briefly described as follows. For term abstraction before a predicate invocation, the constant for term-depth restriction is set to 4, which is also used by Taylor's analyzer [17]. Like the analyzer reported in [23], this analyzer tries to keep complete aliasing information of variables (and also other abstract terms). The extension table is implemented as a linear list of (calling-pattern, success-pattern) pairs. Multiple calling patterns are maintained for a predicate while the success patterns for each of the calling patterns are "lubbed" (least-upper-bound is taken) together. This implementation of the extension table is equivalent to the *assert* database technique described in [23, 17]. However, it is obviously more straightforward and efficient to be implemented in C. It is worth mentioning that the three-stack scheme used by the standard WAM (i.e., the *Heap*, the *Stack*, and the *Trail*) and most other design considerations remain effective for reinterpretation over the abstract domain considered. Nevertheless, it is possible to ignore some of the optimization techniques designed for the concrete domain. In particular, the *environment trimming* technique which strives for a quick stack reclamation appears to be overkill in this abstract WAM.

The analysis time is measured on a Sun 3/60 to compare with the performance of the Aquarius analyzer [15] which is the only publicly available source for both benchmark programs and analyzer efficiency. The Aquarius analyzer's times are running under Quintus Prolog version 2.0. The run-time performance of the analyzer is typical among the analyzers reported (including [23] and [17]). Table 1 gives the results of the measurements of analysis time of our analyzer (*Ours*), the speed-up factors (*Speed-Up Factor*), and the average of speed-up factors. A rough profile of the benchmarks is given by the number of argument places (*Args*) and predicates (*Preds*) in each source program. Additional information includes the static code size (*Size*) and the number of (abstract) WAM code executed (*Exec*) at analysis-time. Neither the times of our analyzer nor those of the Aquarius analyzer (*Aquarius*) include any preprocessing time. Since we use WAM codes generated from the PLM compiler, the compilation times are also included in the table (*PLM*). However, the PLM compiler is not especially fast by doing many optimizations which do not appear justified as far as our analyzer is concerned.

The result of the improvements in analysis time performance is significant and encouraging. The speed-up factors range from 14 (zebra) to 575 (tak), and have an (arithmetic) average of 152. The fluctuation of the speed-up factors is mainly due to the different abstract domains and interpretation algorithms used. If the same abstract domain is used, our conservative estimation is that the speedup would be over 100 for all benchmarks. As a prototype, the implementation was meant to be simple. We believe that there is plenty of room left for more improvements in performance based on better algorithms for abstract interpretation such as those described in [3, 20]. For a reference, the measurements of analysis time on several other platforms are given in Appendix A. This information should be useful for those who do not have access to a Sun 3/60.

7 Conclusions

Theoretically, the time complexities of most interesting dataflow analyses have been shown to be exponential in the worst case [7]. In particular, the precise dataflow analysis considered in this paper is an example. In the average case, however, it appears that the analysis time is proportional to the number of arguments in the program and the precision of the abstract domain used. This is shown in the measurement results presented in this paper and also the results given in [15, 23, 17]. For a practical system, it becomes a design tradeoff between time and precision of the analysis. More precise

Benchmarks	*Args*	*Preds*	*Aquarius* (in sec)	*PLM* (in sec)	*WAM code*		*Ours* (in msec)	*Speed-Up Factor*
					Size	*Exec*		
log10	3	2	2.9	4.5	179	749	38.6	75
ops8	3	2	3.0	4.5	180	400	23.3	129
times10	3	2	3.0	4.5	186	971	48.4	62
divide10	3	2	2.9	4.6	186	1043	50.7	57
tak	4	2	2.3	1.2	53	110	4.0	575
nreverse	5	3	2.2	1.6	99	479	26.7	82
qsort	7	3	3.4	2.5	164	763	44.0	77
query	7	5	4.2	4.3	264	626	25.8	163
zebra	9	5	3.5	7.5	271	1262	257.9	14
serialise	16	7	4.2	3.6	205	912	53.4	79
queens_8	16	7	6.0	3.1	117	324	16.5	364
average								152

Table 1: The Efficiency of Dataflow Analyzers

dataflow analysis can be used if the analyzer is more efficient.

The compilation approach of dataflow analysis presented in this paper integrates two important ideas: the efficient compilation framework of the WAM, and the similarity between abstract and concrete interpretations. It is more efficient than other implementation approaches for abstract interpretation such as meta-interpretation or program transformation because the interpretive or transforming overhead incurred is removed. Moreover, the manipulation of global dataflow information is straightforward and efficient with this approach. Most of all, the approach can be incorporated into an existing compilation-based system in a low-cost manner. Our experience in implementing an abstract WAM and its excellent improvement in analysis-time performance have demonstrated the significance of the approach. It is probable that this approach can also be effectively applied to different classes of dataflow analyses of logic programs or even to different classes of logic programming languages other than Prolog.

Acknowledgements

Martha Kosa, Jin-Kun Lin, and Ting-Wei Hou helped to proof-read this paper. Jieh Hsiang and Jin-Kun Lin always help to provid necessary information promptly. The theoretical work by Saumya K. Debray, David S. Warren, and Suzanne W. Dietrich helps us to do this work in a very simple way. We thank Peter Van Roy for his helpful comments and the results from the Aquarius project. The first author was supported in part by the NSC Taiwan.

References

[1] M. Bruynooghe and G. Jenssens, "An Instance of Abstract Interpretation Integrating Type and Mode Inferencing," In *Proc. of 5th Int'l Logic Programming Conf.*, 1988.

[2] C. Codognet, P. Codognet, and M.-M. Corsini, "Abstract Interpretation for Constraint Logic Languages," In *Proc. of North American Conf. on Logic Programming,* 1990.

[3] B. Le Charlier, K. Musumbu and P. Van Hentenryck, "A Generic Abstract Interpretation Algorithm and its Complexity Analysis," In *Proc. of 8th Int'l Conf. on Logic Programming*, Paris, 1991.

[4] P. Cousot and R. Cousot, "Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Constructions of Fixed Points," In *Conf. Rec. of 4th POPL,* pp. 78-88, 1977.

[5] S. K. Debray and D. S. Warren, "Automatic Mode Inference for Prolog Programs," In *Proc. of 3rd Symp. on Logic Programming*, Salt Lake, 1986.

[6] S. K. Debray, "Efficient Data Flow Analysis of Logic Programs," In *Proc. of the 14th Symp. of Principles of Programming Languages*, San Diego, 1988.

[7] S. K. Debray, "The Mythical Free Lunch (Notes on the Complexity/Precision Tradeoff in Dataflow Analysis of Logic Programs)," In *Proc. of Int'l Logic Programming Symp.* 1991.

[8] S. W. Dietrich, "Extension tables: memo relations in logic programming," In *Proc. of 4th IEEE*

Benchmarks	*Aquarius 3/60*	*Ours 3/60*	*Mac IIx TC 4.0*	*μVax 3100*	*Vax 8530*	*DecS 3100*	*SS1+*	*DecS 5000*	*SS2*
log10	1	75	37	49	86	284	363	500	630
ops8	1	129	63	59	139	469	612	833	1034
times10	1	62	30	37	71	231	294	400	500
divide10	1	57	28	34	65	215	266	372	453
tak	1	575	288	383	639	2091	3286	3833	5750
nreverse	1	82	41	56	108	297	333	595	579
qsort	1	77	38	45	95	281	318	548	540
query	1	163	84	60	183	618	894	1167	1556
zebra	1	14	5.7	9.4	16	55	63	95	107
serialise	1	79	39	47	94	296	375	538	656
queens_8	1	364	182	200	448	1364	1935	2500	3333
average	1	152	76	89	177	564	794	1035	1376
Index	.007	1	.50	.58	1.2	3.7	5.21	6.8	9.0

Table 2: The Speed Ratios on Various Platforms

Int'l Symp. on Logic Programming, San Francisco, 1987.

[9] K. Marriott and H. Søndergaard, "Bottom-Up Abstract Interpretation of Logic Programs," In *Proc. of 5th Int'l Conf. on Logic Programming,* 1988.

[10] K. Marriott and H. Søndergaard, "Analysis of Constraint Logic Programs," In *Proc. of North American Conf. on Logic Programming,* 1990.

[11] C. S. Mellish, The Automatic Generation of Mode Declarations for Prolog Programs, DAI Research Paper No. 163, 1981.

[12] C. S. Mellish, "Some Global Optimizations for a Prolog Compiler," In *Jour. of Logic Programming,* 1985:1:43-66.

[13] K. Muthukumar and M. Hermenegildo, "Determination of Variable Dependence Information Through Abstract Interpretation," In *Proc. of North America Conf. on Logic Programming,* 1989.

[14] R. A. O'Keefe, "Finite Fixed-Point Problems," In *Proc. of 4th Int'l Conf. on Logic Programming,* 1987.

[15] P. Van Roy, "A Prolog Compiler for the PLM", Report No. UCB/CSD 84/203, UC Berkeley, November 1984.

[16] P. Van Roy and A. M. Despain, "The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler," In *Proc. of North American Conf. on Logic Programming,* 1990.

[17] H. Tamaki and T. Sato, "OLD Resolution with Resolution," In *Proc. 3rd Int'l Logic Programming Conf.,* London, 1986. LNCS 255.

[18] A. Taylor, "Removal of Dereferencing and Trailing in Prolog Compilation," In *Proc. of 6th Int'l Conf. on Logic Programming,* Lisbon, June 1989.

[19] A. Taylor, "LIPS on a MIPS: Results form a Prolog Compiler for a RISC," *Proc. of 7th Int'l Conf. on Logic Programming,* Jerusalem, 1990.

[20] A. Wærn, "An Implementation Technique for the Abstract Interpretation of Prolog," In *Proc. of 5th Int'l Conf. on Logic Programming,* 1988.

[21] D. H. D. Warren, Applied Logic - its Use and Implementation as a Programming Tool, Ph. D. Thesis, Univ. Edinburgh, Scotland, 1977.

[22] D. H. D. Warren, An Abstract Prolog Instruction Set, Tech. Note 309, SRI, October, 1983.

[23] R. Warren, M. Hermenegildo and S. K. Debray, "On the Practicality of Global Flow Analysis of Logic Programs," In *Proc. of 5th Int'l Conf. on Logic Programming,* 1988.

A Performance Evaluation on Various Platforms

Table 2 is supplied to compare the speed ratios for the Aquarius analyzer [16] running on a Sun 3/60 workstation (*Aquarius*) and our implementation on eight different platforms:

- Sun 3/60/SunOS 4.1.1 (*Ours 3/60*).
- Apple Macintosh IIx with THINK C 4.0 (*Mac IIx, TC 4.0*).
- DEC microVax 3100/VMS 5.4 (*μVax 3100*).
- DEC Vax 8530/Ultrix-32 v3.0 (*Vax 8530*).
- DECstation 3100/Ultrix 4.2 (*DecS 3100*).
- Sun Sparc station 1+/SunOS 4.1.1 (*SS1+*).
- DECstation 5000/200/Ultrix 4.2 (*DecS 5000*).
- Sun Sparc Station 2/SunOS 4.1.1 (*SS2*).

For simplicity, the figures of the analysis time on each platform have been omitted. They can be recalculated based on the figures given in Table 1. All timings of our analyzer have been measured with a resolution of 0.1 mSec, and represent the average execution times over 100 to 1000 iterations. For Unix-based versions, the "`cc -O`" option on the C compiler is used. The average speed-indexes of the analyzer/platform configurations are given in the last row (*Index*) of the table.