Using The Abstract Interpretation Technique for Static Pointer Analysis

H. Cassé, L. Féraud, C. Rochange, P. Sainrat
email:{casse, feraud, rochange, sainrat}@irit.fr

Institut de Recherche en Informatique de Toulouse,
Université Paul Sabatier, 118 route de Narbonne
31062 Toulouse cedex, France

Abstract

Static pointer analysis by aliasing becomes more and more significant for low-level optimisation while superscalar processors exploit more and more instruction-level parallelism. This paper describes an original method to compute static pointer dependencies. Static pointer approximations are extracted from the source language program by the so-called Abstract Interpretation technique and used for improving the efficiency of memory access reordering by allowing some ready loads to be launched in presence of former blocking stores. Our approach is implemented using the SUIF compiler and performance results are obtained thanks to the SimpleScalar simulator.

1 Introduction

Since the beginning of compilation theory, there has been a break between the earliest analysis phases and the latest code generation and optimisation phases due to distinct abstractions needed to render easier the design and development of compilers . This break was not a matter while optimisations stayed obvious.

Now, with the use of superscalar processors, the need for more and more efficient and powerful optimisation methods is crucial. More accurate program analysis is needed especially for low-level optimisations which are limited because the only thing they know about the program is the machine instruction sequence. Thus, a lot of time is spent to rebuild information about control flow and data organization which are explicitly contained in the source language program and lost during the compilation process. This information should be transmitted to the latter phases of compilation.

Some modern and experimental compilers like *SUIF* [16] and *McCat* [8] have shown the interest and the feasibility of intermediate languages which are able to embed any kind of secondary information which could be used by optional optimisation passes. Our contribution is to demonstrate that better low-level optimisations can be achieved by embedding high-level information inside the intermediate language. To hit the mark, we will use an original method based on *Abstract Interpretation* [3] which allows us (i) to obtain the so-called "high-level information" by static analysis applied on program structures that are very close to the source language and (ii) to use this information to improve low-level optimisations.

In this paper, we focus on the problem of memory disambiguation and memory operations reordering. Memory disambiguation is usually hard to solve at machine language level but it could profit from the data type and structure information contained in most high-level languages. The efficiency of our method is measured by providing the information to the load/store queue of an "agressive" superscalar processor in order to allow loads to bypass stores if permitted.

The remainder of this paper is organized as follows. Section 2 introduces *Abstract Interpretation* and its connection to denotational semantics. In section 3, an abstract interpretation devoted to static analysis of C pointers is proposed. Section 4 describes an implementation of this analysis within the *SUIF* compiler. Section 5 displays some performance measures obtained using the superscalar processor simulator *SimpleScalar*. The conclusion discusses existing static pointer analysis algorithms compared to our method and presents future extensions of our work.

2 Abstract Interpretation

Clearly, when some properties cannot be computed using raw data of a problem, *Abstract Interpretation* is the projection and approximation of these data, and the computation of these properties into an abstract domain where they are computable.

Whereas *Abstract Interpretation* is often used without knowing its name, P. and R. Cousot [5] have formalized this empirical approach. They propose a validated framework and theoretical approach to use *Abstract Interpretation*. After an example for clarifying the notions handled in this paper, we give a generic definition of Abstract Interpretation and its use in denotational semantics.

The key of *Abstract Interpretation* is the fact that very little information is used to get the result of the static analysis of a program. By abstraction, the wanted properties become easily computable by bounding the program to the range of interesting data.

2.1 An example: signing expressions

Let the set of real expressions defined by the abstract grammar below where k stands for a constant and v for a variable:

$$\begin{aligned} expr &\rightarrow expr + expr \mid expr - expr \mid expr \times expr \\ &\rightarrow k \mid v \end{aligned}$$

We are looking for a way to find the sign of an expression like "$(x+1) \times 0 + (-3) \times (-5)$" where x is unknown. First, the abstract set { +, -, 0, ±} is defined. Then, real values and abstract set elements are associated by the following application f:

$f(x) = +$ if $x > 0$,
$f(x) = 0$ if $x = 0$,
$f(x) = -$ if $x < 0$,
$f(x) = \pm$if x is a variable.

$+^\#$	**+**	**-**	**0**	**±**	$\times^\#$	**+**	**-**	**0**	**±**
+	+	±	+	±	**+**	+	-	0	±
-	±	-	-	±	**-**	-	+	0	±
0	+	-	0	±	**0**	0	0	0	0
±	±	±	±	±	**±**	±	±	0	±

Figure 1: abstract forms of + and ×

Now, $+^\#$and $\times^\#$, which match respectively + and × in the abstract domain, are defined in figure 1, where the columns contain the first operands and rows the second ones. The element ± is a trick to handle cases where the sign of an expression cannot be computed so that it is the value of variables the sign of which cannot be known. Notice the relation between ± and other values of this abstract set: ± contains all of them but is less accurate than them. This fact creates a complete partial order on the abstract domain which could be used to prove some interesting properties of *Abstract Interpretation*.

Using the above definition and figure 1, the expression becomes computable in this abstract domain. The expression "$(x+1) \times 0 + (-3) \times (-5)$" is abstracted and computed as follows.

$$(x \quad + \quad 1) \quad \times \quad 0 \quad + \quad (-3) \quad \times \quad (-5)$$

$$\underbrace{\underbrace{\underbrace{\pm \quad +^\# \quad +}_{\pm} \quad \times^\# \quad 0}_{0} \quad +^\# \quad \underbrace{- \quad \times^\# \quad -}_{+}}_{+}$$

The result of this abstract interpretation shows that this expression is always positive without computing the expression for any value. This example shows the power of *Abstract Interpretation* which is able, in this case, to give an approximation of any expression. Proof about signs is here obvious but we need more theoretical bases in order to deal with programmation language semantics.

2.2 Main Principles of Abstract Interpretation

Let Γ be a programming language. A semantic $S : \Gamma \mapsto D$ associates a value in a domain D - respectively called concrete semantic and concrete domain - to each program p of Γ. An abstract interpretation is the choice of an abstract domain $D^\#$, an abstract semantic $S^\# : \Gamma \longmapsto D^\#$ and an abstraction application $\alpha : D \longmapsto D^\#$. Symmetrically, we could define γ the concretization function from $D^\#$ to $\wp(D)$, construction of set of subsets is due to the fact that α is an application.

An abstract domain and an abstract semantic are chosen such as a problem without a computable solution in the concrete domain could be solved in the abstract one. The soundness of an interpretation for a given application $f : D \longmapsto D$ and its matching abstract application $f^\# : D^\# \mapsto D^\#$ may be expressed by the relationship below:

$$\forall x \in D,\ f(x) \subseteq \gamma(f^\#(\alpha(x)))$$

The inclusion-or-equal operator shows the approximation nature of *Abstract Interpretation*. The matter here is to extract the more accurate abstraction for getting the best results from interpretation. P. and R. Cousot have presented a view of this approach as a hierarchy of abstract interpretations according to their accuracy [4] and worked on automatic methods to get minimal abstractions [5][2].

One step forward, the construction of complete partial orders on concrete and abstract domains allows the continuity properties for applications inside these domains and for abstraction relation to be shown, so that the Tarsky fix point computation could be used to approximate loops and recursive function calls. The "more-defined" operator is usually used because it allows to build a complete partial order by adding $\perp$ as the least element for creating a flat order (figure 2).

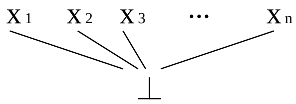

Figure 2: graph of flat complete partial order.

Moreover, when denotational semantics is used in *Abstract Interpretation*, it could be projected in Σ-algebras. Complete partial orders and continuity are easily proved. As programming languages are usually non-contextual and non-ambiguous, denotational semantics Σ-algebras are free. Then, structural induction could be used for computing abstract interpretations.

Now, we have enough tools to build and to prove the soundness of our abstract interpretation about C pointer static analysis.

3 Abstract Interpretation for Pointer Analysis

This section presents our approach applied to C pointer static analysis. Our choices and our design process for the abstract interpretation are explained here.

3.1 Pointer analysis

Even though compilers optimize code to minimizing number of memory access, LISP-like languages do an heavy use of pointers. Moreover, actual superscalar processors with aggressive parallelism leads formerly-avoided memory access hazards. In the past, memory accesses were considered as always dependent to preserve program safety. Current optimisation algorithms need a more accurate knowledge about dependences between memory accesses.

A dependence between two memory accesses exists in three cases ("/" stands for "followed by"): write/write, write/read and read/write. No hazard could arise in the last combination read/read because memory is not modified. Historically, the dependences between pointers were called *aliasing* and were divided into two problems: the *may-alias* problem answers the question "may two pointer alias in some cases ?", the *must-alias* problem "must two pointers alias in all cases ?". The *may-alias* problem is studied here because it is the most used for instructions and loops reordering.

Our abstract interpretation approach described below is substantially lighter than traditional *may-alias* algorithms. Most of them need a great deal of memory and are often not computable for real applications. Moreover, our interpretation does not solve the problem but produces an approximation of pointer values which could be computed for the *may-alias* problem. Safety is ensured by over-estimating constraints.

3.2 Standard and Extended Semantics for the C language

C has a very powerful expression capacity which is a drawback for optimisation but an advantage for writing programs like operating systems. A complete semantic description of C would have to proceed at assembly level but it doesn't bring any gain for static analysis.

As only few programs use low-level memory access facilities of C and as most of the specific C constructions, like `goto` or `switch` instructions, assignment in expressions and so on, could be transformed into algorithmic ones [6], we choose to place our concrete semantics of C to an higher level more adapted to static pointer analysis.

First, the domains on which constructors will be defined must be described:

Val is the set of values composed of the Int integer set, the $Real$ real set and the Ref reference set. Values are the atoms of memory. Ref is the pointer type values which allow values to be indirectly retrieved in memory. Ref is more deeply discussed in 3.5.

Mem is the application which associates a value to each reference. If $m \in Mem$ and $r \in Ref$, $m\,r$ is the value associated with reference r while $m' = m[r \longmapsto v] \in Mem$, with $v \in Val$, is a new memory such as $\forall r' \neq r \in Ref$, $m'\,r' = m\,r'$ and $m'\,r = v$. Note the special dereference property of pointer languages like C, that is, the result of a memory application could be used for composition with itself.

Standard semantics are the semantics defined on concrete domains whereas extended semantics is the standard one improved by addition of a mechanism for getting the wanted property and the flat complete partial order least element to Val. The behaviour of operations confronted with $\perp_{Val}$ in their arguments is very easy to define: all of them return the undefined $\perp_{Val}$. Notice that this property preserves continuity for each application on Val.

For pointer analysis, we look for the value of some pointers at some points in the program. S is defined as a set of unique identifiers of each program statement. The trace set gives for each program point the different states of the memory. It is defined as an application from S to the set of all subsets of Mem, $S \mapsto \wp(Mem)$. For the sake of abstraction, S is extended with a least element $\perp_S$ which leads to a complete partial order and a continuity property for functions defined on it.

The composition of Val and S by the cartesian product, application or set of subsets, $Mem = Ref \rightarrow Val$, $S \rightarrow \wp(Mem)$ and so on, inherits obviously the properties of a least element, a complete partial order and continuity. Now a definition of two main components of extended semantics could be given:

$$Expr = Mem \times S \times (S \rightarrow \wp(Mem)) \rightarrow Val \times (S \rightarrow \wp(Mem))$$
$$Instr = Mem \times S \times (S \rightarrow \wp(Mem)) \rightarrow Mem \times (S \rightarrow \wp(Mem))$$

Domains for instruction $Instr$ and expressions $Expr$ definitions show the fundamental difference between them, expressions do not modify the context memory and therefore the expression tree is free from context dependencies. Expressions include unary operators, binary operators, constants while instructions are composed of assignment, sequence, condition and repetition.

3.3 Abstraction of Domains

We display here an abstract domain for statically computing pointer values and the abstraction relation. The key idea is to abstract single values by a set of probable ones. Lets see figure 3, from a static

```
int x, y;
scanf( ''%d'', &y );
if( y < 100 ) x = 0;
else x = 1;
```

Figure 3: a sample of value set meeting.

point of view, no one can state about the `y` value. This introduces a special value $\top_{\overline{Val}}$ which embeds all constraints and represents all values. Therefore, the condition expression of `if` cannot be computed then `x` may be assigned any of the two values 0 or 1. For preserving safety, no choice can be made but the constraints of both possibilities must be used in continuation of computation. So, `x` may be represented by a set of two values `x =` $\{$ `0, 1` $\}$. Now, another question arises: what are the values of `x` and `y` just after their declaration? The answer is straightforward: they contain the undefined value. Thus, we need a new special value: $\perp_{\overline{Val}}$, the invalid value.

One step forward, an abstract value may be one of a values set of $\wp(Int) \cup \wp(Real) \cup \wp(Ref)$, $\top_{\overline{Val}}$ or $\perp_{\overline{Val}}$, the two last values must be expressed in the set of subsets for finding out a useful complete partial order and some computation properties. $\top_{\overline{Val}}$ could be seen as the set of all values, that is, Val such as it contains all constraints of all values. On the contrary, $\perp_{\overline{Val}}$ does not have any valid value or any constraint and is well represented by the empty set, Φ. A complete partial order and, even, a complete lattice could be build on the set of abstract values thanks to the inclusion operator with a least element $\perp_{\overline{Val}}$ and a greater one $\top_{\overline{Val}}$.

The abstract domain can be expressed as:

$$\overline{Val} \;=\; \wp(Int) \cup \wp(Real) \cup \wp(Ref) \;\text{ with }\; \top_{\overline{Val}} = \overline{Val} \text{ and } \perp_{\overline{Val}} = \Phi$$
$$\overline{Mem} = \wp(Ref) \to \overline{Val}$$
$$\overline{S} = S$$

Now, we are able to display the definition of the main components of the non-standard semantics, that is, of the semantics built on abstract domains:

$$\overline{Expr} \;=\; \overline{Mem} \times \overline{S} \times (\overline{S} \;\to\; \wp(\overline{Mem})) \;\to\; \overline{Val} \times (\overline{S} \to \wp(\overline{Mem}))$$
$$\overline{Instr} \;=\; \overline{Mem} \times \overline{S} \times (\overline{S} \;\to\; \wp(\overline{Mem})) \;\to\; \overline{Mem} \times (\overline{S} \to \wp(\overline{Mem}))$$

Abstraction of a single value v becomes quite easy: it is the singleton $\{\, v\,\}$. For the sake of complete partial order preservation, $\perp_{Val}$ must be abstracted by the least element of $\overline{Val}$, $\perp_{\overline{Val}}$. There is no concrete element the abstraction of which matches $\top_{\overline{Val}}$ but it will be useful for qualifying any program inputs.

3.4 Abstraction of Constructors

In order to interpret non-standard semantics, we discuss here the way constructors (expressions and instructions in denotational semantics) are derivated from standard semantics. Abstract semantics must be designed keeping in mind three points: computability, safety and termination.

Computability is achieved by the choice of the domain abstraction. The problem is to find the abstraction which provides the best approximation. Safety is relative to the wanted property. Structural induction is useful for showing preservation of a property through each constructor but safety often is the result of the abstract semantics design which is oriented by the property which is searched for. The abstract interpretation termination influences the design because of the static nature of the interpretation which induces potential endless loops due to undefined values.

Termination is achieved in this abstract interpretation by binding program values to a finite set and by building the memory state in an incremental fashion. First, new constant creations are avoided by the operator definition: the result may be either $\perp_{\overline{Val}}$, or $\top_{\overline{Val}}$ for integers and reals computations. Pointers are processed in 3.5. Then, fixed point iteration memory contexts are computed by the union of the previous context and the result context. This union ensures incrementation of memory context until each memory data converges to the finite set Val. Practically, this way of operator computation makes it converging very quickly, e.g. two or three iterations.

The use of denotational semantics provides a non-contextual and non-ambiguous grammar for programs description. Therefore, derivated Σ-algebra are free and abstract interpretation is performed by structural induction. A short description of constructor behaviour follows: (i) operators always return a $\top_{\overline{\mathrm{Val}}}$ in sake of termination (ii) conditional and repetition instruction make union, respectively, either of each alternative memories, or of memories after each iteration.

3.5 Symbolic Pointers for the C Language

We present here a system for concrete symbolic pointer valuation and their abstraction and a solution for the abstraction of the dynamic memory. At last, the choice of symbolic pointers is discussed.

The C language memory structure can be seen as a tree. Leafs are memory atoms, that is, integers, reals or references[1]. Other nodes are either structures, or arrays. The root looks like the global memory of the program. A reference - the value contained in a pointer - is a path in the memory tree. An important property of such a memory representation is the independence of two data - either leafs, or nodes - while either their references are different, or the reference to one of them is the subpath of the second one. Therefore, aliasing between two pointers will be easily established if an approximation of their value can be computed.

If a reference is a path in the memory tree, it is composed of a sequence of nodes named using identifiers (for variable or field names) and integers (for array indexes).

As there are several ways to access a same memory locations in C, we need a canonical form to represent correspoding tree references in C programs as a list of pairs composed of an identifier and an index. From figure 4, the single variable `p` could be addressed by [(`p`, 0)], reference of `s.t[3]` becomes [(`s`, 0), (`t`, 3)] then i_3 and i_4 access the same memory unit referenced by [(`st`, 0), (`t`, 0)]. In the case of a sequence of fields, the reference looks like [(variable, 0), (field_1, 0), (field_2, 0), ...] with 0 as indexes. Cases with several indexes do not exist because indexes of multiple-dimension arrays are accessed with a single integer computed from indexes. For example, `t[3][4]` has for reference [(t, $3\times$t'*size*+4)]. Now, as memory accesses only work on memory atoms at semantics level, two pointers are aliased only if their references are equivalent.

```
     struct {
            int f;
            int t[5];
     } s, st[ 3 ], *sp;
     int *p;
i1   sp = st;
i2   p = sp->t;
i3   *p = 0;
i4   st[0].t[ 0 ] = 0;
```

Figure 4: a C description of memory structure.

Abstraction of pointers obviously produce sets of references. Yet, as an index becomes a set of index, an array access operator must apply each index element to each reference composing the pointer set. Moreover, the index $\top_{\overline{Val}}$ could be viewed either as all indexes of an array, or as a special index value. For sake of simplicity and independence of array access operators and memory representation, we choose the second solution even though the dependence test is a bit more difficult in the presence of $\top_{\overline{Val}}$ in any pair of the list. Shortly, dependences between all pairs of references of each set must be searched with the exceptional rule that $\top_{\overline{Val}}$ index is dependent with all others.

During the abstract interpretation, dynamic memory allocations pose a problem. As loop computations are fixed-point computations, there's no way to know the number of memory allocations performed

[1] Either random memory accesses, or "wild" type casts are here forbidden.

within them in the concrete interpretation. As abstract interpretation is a static approximation of programs, another one must be used for dynamic allocations.

The key is to gather multiple memory allocations into one memory place which stores all constraints of this memory block. Thus, there are two kinds of memory: a static one which is assigned by the replacement of old values, a dynamic one which is assigned by the union of the old values and of the new values. This union ensures no loss of constraints for the different blocks sharing the abstract memory place.

Now, the matter is to find how dynamic memory allocation must share the abstract memory place. A first idea is to gather all allocations in the same place but it is too simple to lead to a good approximation. Our method assigns a memory place at each allocation statement. All allocations at a given statement may share the same data structure and allow to design this place as a static memory tree. This is a good tradeoff between utopian concrete-like execution and one-place dynamic memory because list elements are often created in few constructor functions.

Though symbolic pointers are very easy to process, they well fit abstract interpretation because they contain all semantics information of the pointer building. Therefore, symbolic pointers avoid some incoherent memory accesses and our symbolic pointer system has a good behavior in presence of $\top_{\overline{Val}}$ indexes. In a processor-like integer valuation, this index applied to a pointer makes it equal to all references while symbolic pointers bound the impact of $\top_{\overline{Val}}$ to referenced arrays. One step forward, as a memory tree is finite in the abstraction, pointer building is made in a finite way for structure fields and array indexes are bounded by the integer system. In case of pointer arithmetic operators[2], the computation is done on the index part of the reference pair list, bounding assignments to data area locations. So, there is a finite number of references, constructors work in an incremental way, and abstract interpretation always completes.

[2] That is, `+`, `-`, `++` and `--` operators in C.

4 Implementation of the Compilation Process

Our implementation was made using the well-known *SUIF* compiler [16] from Stanford University. *SUIF* was first designed for program parallelization and low-level optimisation. It features a sequence of different passes, working on a program file and sharing the same format. Compilation is the transformation from the semantic-powerful elements of the source language to simpler ones in the machine language. Either original passes can be reorganized by user, or new ones can be added.

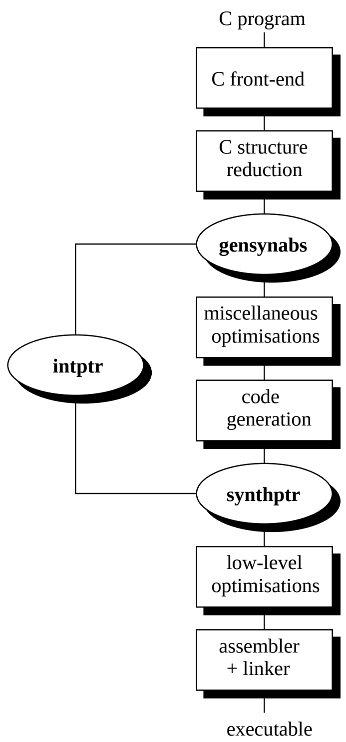

Figure 5: Pointer analysis compilation process.

This shared program format is well-suited for our approach. It provides an easy way to hook auxiliary information at each instruction thanks to the annotation system. Annotations are data structures saved in the program file which may be retrieved at any point of the compilation process. Therefore, the result of a static analysis on the high-level program representation can be used in all passes, even the low-level optimisation and code generation ones.

Figure 5 shows the organization of passes in our approach for static pointer analysis. Added passes are displayed with bold text in ovals. After the classical front-end passes, some transformations are applied to reduce the instructions of the program (in *SUIF*, representation of abstract trees and instructions are melted) to the subset of the C language defined in 3.2. Then several passes of high-level optimisations, code generation, low-level optimisation, assembly and link edition follows.

The three passes we add are used together for the same goal, pointer static analysis though the work is really only done by *intptr*. *gensynabs* allows translation from *SUIF* abstract trees to our semantic language well suited for *Abstract Interpretation* and installs a trace system in order for *synthptr* to reinsert analysis results into the machine language instructions. These two passes could be reused for several static analysis either by *Abstract Interpretation*, or working on pure semantics. The following optimisation passes can then retrieve the analysis results in order to be more efficient. Notice the inherent parallelism coming from the abstract trees trace system and from the fact that several abstract interpretations are entirely independent.

5 Case Study: Pointer Analysis

5.1 Introduction

In this part, we show how our static pointer analysis can be used to improve load/store dynamical reordering for superscalar processors. Out-of-order execution of loads and stores is critical for the performance of modern ILP processors because of the long memory latencies and limited memory bandwidth. In order to measure the effects of our analysis, we use the SimpleScalar [1] package to simulate an aggressive superscalar processor (16-way, 2-level branch prediction, 18 integer and 6 floating-point functional units, 256-entry reorder buffer, ...).

5.2 Related Work

Several hardware schemes have been proposed to schedule loads and stores out-of-order.

Program	IPC	Mean Number of blocked loads	Ready but blocked loads
anagram	4.23	12.95	39%
compress	3.01	6.30	34.8%
go	2.04	15.74	32.3%
li	3.43	8.59	32.9%
m88k	4.48	3.50	41.5%
perl	3.07	16.36	44.7%

Table 1: Load/Store Queue measures for a 16-way superscalar processor

The first one, implemented in Simplescalar, allows a load to be issued before a preceding store if (i) the store address is known, (ii) the store data is known (when the store address matches the load address; the data is forwarded to the load without accessing the cache). However, table 1 shows that many non-ready stores penalize the performance of the processor by blocking following loads even if their operands are ready.

Another scheme [7] suggests that the processor can execute loads even if the address of previous stores is not known. This optimistic speculative execution of loads increases the parallelism but it needs to quickly recover a coherent state in the case of a misprediction. Finally, [13] suggests to predict memory dependences to improve the accuracy of the previous scheme.

All these schemes are quite complex. They need a memory order buffer (or load/store queue) and an efficient recovery scheme for the speculative ones.

A software scheme, run-time memory disambiguation [14][9], consists in keeping two versions of code fragments, a conservative one which assumes that if static analysis fails to demonstrate unaliasing, the references alias and an aggressive one that assumes the contrary and thus can be better reordered. These two versions are guarded by conditions evaluated at run-time in order to choose the version to execute. The main drawback of this solution is, of course, the increase of code size.

5.3 Efficiency of abstract interpretation for pointer analysis

In order to evaluate the maximum efficiency of our method, we exploit the results of the analysis at run-time. Thus, Simplescalar was modified to straightforward supply information coming from our static pointer analysis to the load/store reordering module of the simulator. Whenever a store address is unknown in the queue, a dependence test is performed for every following load instruction. Keeping in mind the conservative nature of our static analysis, all cases cannot be resolved but the processor performance is likely to be improved according to the accuracy of the abstract interpretation. Encouraging results for small meaningful programs are summarized in table 2. We are currently measuring performances for the SPEC benchmarks.

6 Conclusion

We have presented an original approach for melting high-level static analysis into low-level optimisation thanks to *Abstract Interpretation*. This approach has been used in a concrete example for static pointer analysis and its efficiency has been measured by using the results to dynamically reorder loads and stores. The reported performance gain showed the interest of our approach.

Our method is radically different from usual static pointer analysis which compute *may-alias* problem matrix on control flow graphs [15] [12] [11] [10]. Even though neither function call, nor recursive call are implemented yet, our method is computable in a finite and realistic time thanks to the *Abstract Interpretation* approach.

Yet, it should still be improved in order to obtain more useful static pointer analysis results for instruction scheduling or loop optimisations. For example, integers used as indexes are very rawly approximated while more accurate evaluation would allow to get a better analysis of scientific programs which make an heavy use of arrays. Likewise, recursive structures like lists and trees could benefit from a mechanism which could detect circular pointers. All these extensions may be achieved in one or more parallel abstract interpretation passes.

Moreover, our approach may be generalized for moving more information from high-level program representation to all levels of optimisation where information about the program structure is needed.

Finally, we believe that this information might also be useful at run-time: it could improve the accuracy of a hardware dependence predictor. Thus, this approach could lead to the design of a new instruction set architecture embedding either dependencies informations, or program structure encoding which could make better the work of processors. However, an efficient way for transmitting the static analysis information to the hardware has to be found.

References

[1] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. Computer Science Department Technical Report 1342, University of Wisconsin-Madison, June 1997.

[2] P. Cousot and R. Cousot. Comparison of the Galois Connexion and Widening/Narrowing Approaches to Abstract Interpretation (extended abstract). *Bigre 74*, pages 107–110, octobre 1991.

[3] P. Cousot and R. Cousot. Relational Abstract Interpretation of Higher Order Functional Programs (extended abstract). *Bigre 74*, octobre 1991.

[4] Patrick Cousot. Program Analysis: The Abstract Interpretation Perspective. *ACM Computing Surveys*, 28A(4), december 1996.

Program type	Number of blocked loads	Ready but blocked loads	Speedup
String storage list construction	27.91	68.3%	1.15
Matrix multiplication	4.49	40.5%	1.11
Binary tree management	17.45	17.45%	1.03
Lexical analyser	8.12	8.12%	1.10
Hashing table management	28.22	94.3%	1.47

Table 2: optimisation results for some small programs.

[5] Patrick Couzot. Abstract Interpretation. *ACM Computing Surveys*, Vol.28, No.2:324–328, June 1996.

[6] A.M. Erosa and L.J. Hendren. Taming Control Flow: A Structured Approach to Eliminating Goto Statements. ACAPS Technical Memo 76, McGill University, 3480 Uinersity St, Monreal, Canada, september 1993.

[7] M. Franklin and G. S. Sohi. ARB: a Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computer Science*, 45(5):552–571, May 1996.

[8] L.J. Hendren, C. Donawa, M. Emami, G.R. Gao, Justiani, and B. Drifharan. Designing the McCat Compiler Based on a Family of Structured Intermediate Representations. ACAPS Technical Memo 46, McGill University, 3480 University St, Monreal, Canada, june 1992.

[9] A. S. Huang, G. Slavenburg, and J. P. Shen. Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation. In *21st International Symposium on Computer Architecture*, pages 200–210. IEEE/ACM, 1994.

[10] Alexandru Nicolau Joseph Hummel, Laurie J. Hendren. A General Dependence Test for Dynamic, Pointer-Based Data Structures. In *ACM SIGPLAN'94 Conference on Programming Language and Design Implementation*, pages 218–229. ACM Press, 1994.

[11] James R. Larus and Paul N. Hilfinger. Detecting Conflicts Between Structure Accesses. *SIGPLAN'88 Conference on Programming Language Design and Implementation*, 23(7):21–34, july 1988. ISBN 0-89791-269-1.

[12] T. J. Marlowe, W. G. Landi, B. G. Ryder, J.D. Choi, M. G. Burke, and P. Carini. Pointer-Induced Aliasing: A Clarification. *ACM SIGPLAN Notices*, 28(9):67–70, september 1993.

[13] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic Speculation and Synchronization of Data Dependencies. In *24th Annual International Symposium on Computer Architecture*, pages 181–193. IEEE/ACM, May 1997.

[14] A. Nicolau. Run-Time Disambiguation: Coping with Statically Unpredictable Dependencies. *IEEE Transactions on Computers*, 38(5):663–678, May 1989.

[15] R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, June 1995.

[16] R.P. Wilson, R.S. French, C.S. Wilson, S.P. Amarasinghe, J.M. Anderson, S.W.K. Tijang, S.-W. Liao, C.-W. Tseng, M.W. Hall, M.S. Lam, and J.L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.