# Data Dependence Analysis of Assembly Code

Wolfram Amme, Peter Braun, Eberhard Zehendner, François Thomasset

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Data Dependence Analysis of Assembly Code*

Wolfram Amme, Peter Braun, Eberhard Zehendner
Computer Science Department
Friedrich Schiller University Jena
D-07740 Jena, Germany
François Thomasset
INRIA, Rocquencourt

78153 Le Chesnay Cedex, France

**No 3764**

Septembre 1999

THÈME 1

*apport de recherche*

ISSN 0249-6399

# Data Dependence Analysis of Assembly Code

Wolfram Amme, Peter Braun, Eberhard Zehendner *

Computer Science Department

Friedrich Schiller University Jena

D-07740 Jena, Germany

François Thomasset †

INRIA, Rocquencourt

78153 Le Chesnay Cedex, France

Thème 1 — Réseaux et systèmes

Projet A3

Rapport de recherche n˚3764 — Septembre 1999 — 33 pages

**Abstract:** Determination of data dependences is a task typically performed with high-level language source code in today's optimizing and parallelizing compilers. Very little work has been done in the field of data dependence analysis on assembly language code, but this area will be of growing importance, e.g. for increasing Instruction Level Parallelism. A central element of a data dependence analysis in this case is a method for memory reference disambiguation which decides whether two memory operations may/must access the same memory location. In this paper we describe a new approach for determination of data dependences in assembly code. Our method is based on a sophisticated algorithm for symbolic value propagation, and it can derive value-based dependences between memory operations instead of just address-based dependences. We have integrated our method into the SALTO system for assembly language optimization. Experimental results show that our approach greatly improves the accuracy of the dependence analysis in many cases.

**Key-words:** compiler, static analysis, assembler, dependence analysis

*(Résumé : tsvp)*

* {amme,braunpet}@informatik.uni-jena.de, zehendner@acm.org
† Francois.Thomasset@inria.fr

# Analyse de dépendances de codes assembleur

**Résumé :** Si de nos jours le calcul des dépendances de données est une tâche effectuée couramment par les compilateurs optimisants ou parallélisants pour des langages de haut niveau, il existe encore très peu de travaux sur l'analyse de code assembleur ; ce domaine devrait pourtant acquérir une importance croissante, par exemple pour aider la portabilité des codes en exploitant le parallélisme à grain fin des processeurs. Le point capital de telles analyses est une méthode de résolution des références mémoire, qui permette de décider si deux opérations d'accès à la mémoire *doivent* (ou *peuvent*) accéder à la même cellule mémoire. Dans cet article, nous décrivons une approche nouvelle de détermination des dépendances de données dans le codes assembleur ; notre méthode est basée sur un algorithme de propagation de valeurs symboliques ; elle est capable de calculer les dépendances de flot (*"value-based"*) au lieu des dépendances classiques basées sur la simple comparaison des adresses. Nous avons intégré notre méthode dans l'environnement SALTO, système développé à l'IRISA pour aider l'optimisation de codes assembleur. Nous donnons quelques résultats expérimentaux, qui indiquent que notre approche améliore sensiblement la précision de l'analyse dans un grand nombre de cas.

**Mots-clé :** compilation, analyse statique, assembleur, analyse de dépendances

# 1 Introduction

The determination of data dependences is nowadays most often done by parallelizing and optimizing compiler systems on the level of source code, e.g. C or FORTRAN 90, or some intermediate code, e.g. RTL [25]. Data dependence analysis on the level of assembly code aims at increasing instruction level parallelism. Using various scheduling techniques like list scheduling [6], trace scheduling [9], or percolation scheduling [18], a new sequence of instructions is constructed with regard to data and control dependences, and properties of the target processor. Most of today's instruction schedulers only determine data dependences between register accesses and consider memory to be one cell, so that every two memory accesses must be assumed as data dependent. Thus, analyzing memory accesses becomes more important while doing global instruction scheduling [3].

Performing optimizations at the level of assembly code has many benefits. The developer of machine-dependent optimization techniques needs an easily programmable tool, which exposes (almost) all processor properties. Implementing new techniques in an existing compiler (even if the compiler is retargetable, such as the GNU C Compiler `gcc` [24]) fails, because new processor features may not be expressible within the machine description. In addition, working on assembly code also gives the opportunity to optimize during link-time. Wall [27] gives an overview of systems which perform the so–called *late-code-modification*. A full description of such techniques can be found in Srivastava [23]. Another aspect concerns optimization of delivered code. Fisher [10] makes the suggestion to translate an executable program during loading, e.g. replace multimedia extensions by 'normal' instructions or vice versa. When processors of the same family differ in the number of functional units or number of registers, *rescheduling* [21] can be used to optimize in the new processor. A program can be made executable on a completely different processor by the use of binary translation [22]. In all these areas a better analysis of assembly code and more accurate data dependence information would offer significant benefits.

In this paper, we describe an intraprocedural value-based data dependence analysis [15]. When analyzing data dependences in assembly code we must distinguish between accesses to registers and those to memory. In both cases we derive data dependences from reaching definitions and reaching uses information that we obtain by a monotone data flow analysis. Register analysis does not involve any complication: the set of used

and defined registers in one instruction can be established easily, because registers do not have aliases. Therefore, determination of data dependences between register accesses is not in the scope of this paper.

For memory references we have to solve the *aliasing problem* [26, 12]: decide whether two memory references access the same location. We have to prove that two references always point to the same location (must-alias) or must show that they never refer to the same location. If we cannot prove the latter, we would like to have a conservative approximation of all alias pairs (may-alias), i.e., memory references that might refer to the same location. To derive all possible addresses that might be accessed by one memory instruction, we use a symbolic value propagation algorithm. To compare memory addresses we use a modification of the GCD test [28].

We implemented our technique in the context of the SALTO tool [20]. SALTO is a framework to develop optimization and transformation techniques for various processors. The user describes the target processor using a mixture of RTL and C language. A program written in assembly code can then be analyzed and modified using an interface in C++. SALTO has already implemented some kind of *conflict analysis* [13], but only determines address-based dependences between register accesses and assumes memory to be one cell. The technique we present in this paper goes beyond the one we have already implemented in the SALTO tool. But our first experimental results indicate that even this simplified form of our analysis can be more accurate in the determination of data dependences than other previous methods.

The rest of this paper is structured as follows: In section 2 we introduce our programming model. Section 3 gives a brief introduction in the field of data dependence analysis and alias analysis in assembly code. Section 4 describes the concept of monotone data flow systems, which is the theoretical basis of our approach. We present our method in detail in section 5. Section 6 gives experimental results, in section 7 we discuss related work, and in section 8 we conclude with an outlook to further developments.

# 2 Programming Model and Assumptions

In the following we assume a RISC instruction set, which is strongly influenced by the SPARC architecture[1]. Memory is only accessed through load (`ld`) and store (`st`) instruc-

[1] Although our analysis is not limited to the SPARC.

tions. Memory references can only have the following format: a) `mem` = `%rx` + `%ry` or b) `mem` = `%rx` + `offset`. Use of a scaling factor is not provided in this model, but an addition would not be difficult. Memory accesses normally read or write a word of four bytes. By use of special instructions it is possible to read and write only one or two bytes, too. For global memory access, the address (which is a label) first has to be moved to a register. Then it can be read or written using a memory instruction. Initialization of registers or copying the contents of one register to another can be done using the `mv` instruction. All logic and arithmetic operators have the following format: `op` $src_1, src_2, dest$. The operation `op` is executed on operand $src_1$ and operand $src_2$; the result is written to register $dest$. An operand can be a register or an integer constant.

**Arithmetics on adresses** We want to take care of the fact that computations of adresses may wrap around beyond $2^L$, where $L$ is the number of bits used to represent adresses. Arithmetic operations are based on modulo arithmetic, i.e., we assume that all integer registers have the same width $L$ and calculations are performed modulo $2^L$ with wrap around where applicable. Calculation of all coefficients in section 5 is modulo $2^L$ and with nonnegative results.

**Control Flow** Control flow is modeled using unconditional (`b`) or conditional (`b`*cc*) branch instructions. Further, we assume that the control flow graph is reducible; every cycle of the control flow graph then constitutes a loop in the usual sense[2]. Runtime-memory can be divided into three classes [1]: static or global memory, stack, and heap memory. When an address unequivocally references one of these classes, some simple memory reference disambiguation is feasible (see section 3). Unfortunately it is not easy to prove that an address always references the stack, when no interprocedural analysis is done from which one could obtain information about the frame pointer. In our approach we do not make such assumptions.

[2]In the appendix (section 10), we have reproduced a formal defi nition, together with an algorithm to identify all loops of a control fbw graph [1].

| | | |
|---|---|---|
| `1: ld [%fp-4],%o1`<br>`2: st %o2,[%fp-8]` | `1: ld [%fp-4],%o1`<br>`2: sethi %hi(.LLC0),%o2`<br>`3: st %o3,[%o2+%lo(.LLC0)]` | `1: add %fp,-20,%o1`<br>`2: st %o2,[%o1-4]`<br>`3: ld [%fp-20],%o3` |
| (a) | (b) | (c) |

Figure 1: Sample code for different techniques of alias detection: (a) and (b) can be solved by instruction inspection, whereas (c) needs a sophisticated analysis.

# 3 Data Dependences in Assembly Code

In assembly programs we have two classes of locations in which a program can store data: registers and memory. For a definition of data dependences we can group both classes and treat them both as memory locations:

A statement $S_2$ has a value-based data dependence on a statement $S_1$ if $S_2$ can be reached after the execution of $S_1$, and the following conditions hold:

(i) Both statements access a common memory location $l$.

(ii) At least one of them writes to $l$.

(iii) Between the execution of $S_1$ and $S_2$, there is no statement $S_3$ that also writes to $l$.

A data dependence is called a *flow dependence* if $S_1$ writes to $l$ whereas $S_2$ reads from $l$. It is called an *anti dependence* if $S_1$ reads from $l$ and $S_2$ writes to $l$; it is an *output dependence* if both statements write to $l$. $S_1$ and $S_2$ are in *conflict*, if conditions (i) and (ii) hold, whereas (iii) may or may not be fulfilled. Another name for a conflict is *address-based data dependence*. Conflict analysis is a frequently used approximation of data dependences.

The determination of data dependences can be achieved by different means. The most commonly used is the calculation of reaching definitions (resp. reaching uses) for all statements. This can be described as the problem of determining all statements where the value of a specific memory location has been written last resp. has been used last. Once the reaching definitions and uses have been determined, we are able to infer def-use, def-def, and use-def associations; a def-use pair of statements indicates a flow dependence between them, a def-def pair an output dependence, and a use-def pair an anti dependence.

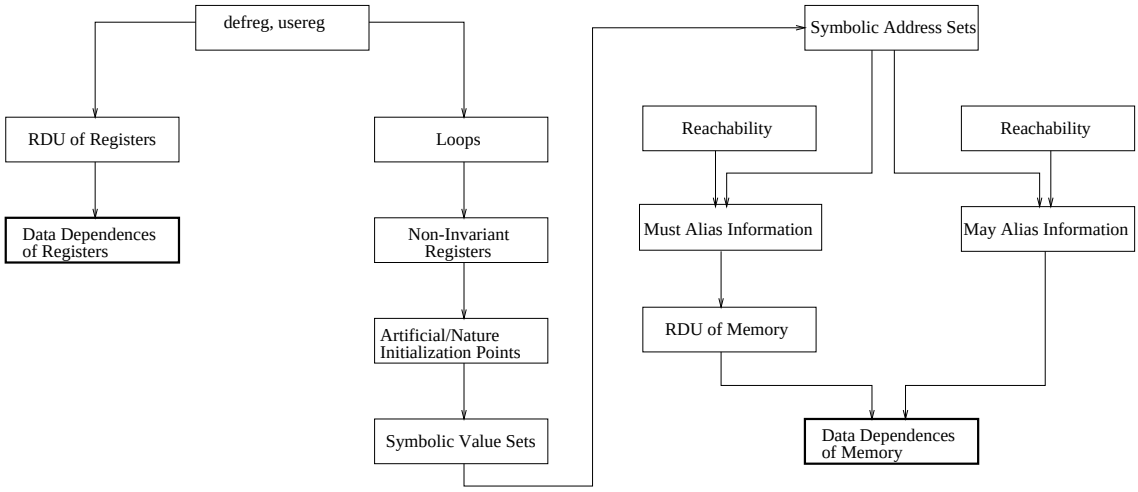


Figure 2: Overview of the determination of data dependences.

Data dependence analysis of registers causes no problems. The set of used and defined registers can be established for each instruction by its semantics. For memory references we have to solve the aliasing problem, i.e., we have to determine, whether two memory references access the same location. In the following we briefly review techniques for alias analysis of memory references.

Doing *no alias analysis* leads to the assumption that a memory load instruction is always dependent on a store instruction, whereas a store instruction is always dependent on any memory instruction. A common technique in compile-time instruction schedulers is *alias analysis by instruction inspection*, where the scheduler looks at two instructions to see if it is obvious that different memory addresses are referenced. With this technique, independence of the memory references in Fig. 1 (a) and (b) can be proved, because the same base register but different offsets are used (a), or different memory classes are referenced (b). Fig. 1 (c) shows an example where this technique fails. By looking only at register `%o1` it must be assumed that register `%o1` can point to any memory location, and therefore we cannot exclude that $S3$ is data dependent on $S2$ — in fact, these instructions are independent. This local analysis disables notice of the definition of register `%o1` in the first statement. This example makes it clear that a two-fold improvement is needed. First, we need to save information about address arithmetic, and second, we need some kind of copy-propagation. Provided that we have such an algorithm, it would be easy to show that in statement $S2$ register `%o1` has the value $\%\texttt{fp} - 20$ and therefore there is no overlap between the 4 bytes memory blocks starting at $\%\texttt{o1} - 4$ resp. $\%\texttt{fp} - 20$.

# 4 Monotone Data Flow Analysis

The most important passes of our analysis are performed by *data flow analysis* [11]. Therefore, several parts of the analysis have to be described as a *monotone data flow framework*. Formally, a monotone data flow framework is a triple *MDFS* $= (DF, \sqcap_{DF}, F)$. *DF* is called the *data flow information set*, which is a formal description of the information that will be propagated through the control flow graph. The *meet operator* $\sqcap_{DF}$ of a data flow framework describes the effect of joining paths in the flow graph. It answers the question, how we can compute data flow information for a node that has more than one predecessor. $F \subseteq \{f : DF \to DF\}$ is the set of our semantic functions. To each node of the control flow graph we assign one of these semantic functions. A semantic function specifies the effect of an application of this node to the data flow information.

If the semantic functions are monotone and $(DF, \sqcap_{DF})$ forms a bounded semi-lattice with a one element and a zero element [28], we can use a general iterative algorithm [11] that determines for each statement of the control flow graph an element $d \in DF$ that is a safe approximation of the data flow information reaching the statement. The solution of a monotone data flow framework thus can be expressed as a function mdfs : $STMT \to DF$. The initial data flow information at each statement is the one element of the corresponding semi-lattice.

We can define a relation $\sqsubseteq$ based on the semi-lattice $(DF, \sqcap_{DF})$ of a monotone data flow framework, that is defined as:

$$\forall a, b \in DF : \; a \sqsubseteq b \iff a \sqcap_{DF} b = a$$

Monotonicity of the semantic functions can be expressed using the relation $\sqsubseteq$. Semantic function $f \in F$ is monotone iff $\forall a, b \in DF : a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$. Note that for a statement *s* and for all $e \in DF$ that may reach statement *s*, the solution obtained when applying the general algorithm satisfies: mdfs$(s) \sqsubseteq e$.

**Description of semantic functions** A unifying description of semantic functions can simplify the proving of monotonicity. If there is a suitable definition of a difference operator $\backslash$, the general form of a semantic function is given as[3]:

$$f_s(D) = (D \backslash K_s(D)) \sqcap_{DF} G_s(D)$$

[3] $K_s(D)$ is a "kill function", and $G_s(D)$ is a "generate function".

In this formula, *D* stands for data flow information that is reaching statement *s*. An application of $K_s(D)$ returns the part of the incoming data flow information that will be destroyed by the execution of statement *s*. $G_s(D)$ describes the data flow information that is introduced by the execution of *s*.

# 5 Determination of Data Dependences

Fig. 2 shows an overview of our data dependence analysis for machine code. Our technique determines data dependences for register accesses (resp. memory accesses) by the calculation of reaching definitions and reaching uses (RDU). For registers the determination of reaching definitions/uses can be performed by a well-known standard algorithm described in [1]. To use this algorithm for data dependence analysis of memory accesses we have to derive the *may-alias* information, i.e., we have to check whether two storage accesses could refer to the same storage object. To improve the accuracy of the data dependence analysis the *must-alias* information is needed, i.e., we have to check whether two storage accesses always refer to the same storage object.

## 5.1 Informal Description of the Method

The main task of our analysis concerns the determination of memory addresses. Therefore, we have to determine all possible values of registers in memory expressions. In some rare cases solving this problem is trivial, e.g., moving a constant to a register or adding two registers, for which the values are known. In contrast, there are situations, in which it is in general impossible to derive the value of a register, e.g. when the register is defined by a load instruction. As we want to perform an accurate analysis, we have to look for a way to work with these as yet unknown values.

**Symbolic Value Sets** In our approach we use the concept of symbolic values. A statement *j* is called an *initialization point* if *j* defines a register with an unknown value. The finite set of all natural initialization points of *P* is given by the image of function $ip$, defined further below. In our programming model, natural initialization points are load instructions, call nodes, or entry nodes of a procedure. If an initialization point *j* defines the contents of register $r_i$ we refer to this value through the symbol $R_{i,j}$.

| | | | | | |
|---|---|---|---|---|---|
| 0: | entry | | | | |
| 1: | ld [%r31+68],%r1 | | | | |
| 2: | cmp %r1,189 | $\%\mathtt{r1} = \{R_{1,1}\}$ | | | |
| 3: | bne .LL8 | $\%\mathtt{r1} = \{R_{1,1}\}$ | | | |
| 4: | mov 0,%r2 | $\%\mathtt{r1} = \{R_{1,1}\}$ | | | |
| 5: | b .LL9 | $\%\mathtt{r1} = \{R_{1,1}\}$ | $\%\mathtt{r2} = \{0\}$ | | |
| 6: | .LL8: | | | | |
| 7: | mov 4,%r2 | $\%\mathtt{r1} = \{R_{1,1}\}$ | | | |
| 8: | .LL9: | | | | |
| 9: | add %r2,4,%r3 | $\%\mathtt{r1} = \{R_{1,1}\}$ | $\%\mathtt{r2} = \{0,4\}$ | | |
| 10: | add %r31,68,%r1 | $\%\mathtt{r1} = \{R_{1,1}\}$ | $\%\mathtt{r2} = \{0,4\}$ | $\%\mathtt{r3} = \{4,8\}$ | |
| 11: | st %r2,[%r1+%r3] | $\%\mathtt{r1} = \{R_{31,0}+68\}$ | $\%\mathtt{r2} = \{0,4\}$ | $\%\mathtt{r3} = \{4,8\}$ | |

Table 1: Results of a symbolic value set propagation for a simple program. Registers $r_i$ that are not mentioned have the value $\{R_{i,0}\}$.

With our method we calculate possible *symbolic value sets* (SVS) for each register and each program statement. A symbolic value set is a set of polynomials, whereby each polynomial stands for a possible content of the associated register. Variables of such polynomials are represented by the symbols $R_{i,j}$, i.e., definition values of initialization points. Table 1 shows an example of the calculation of symbolic value sets for a simple assembly program. For each statement we determined symbolic value sets that describe the register contents immediately before the execution of the statement.

Without limiting the cardinality of symbolic value sets our propagation algorithm might lead to infinite sets. Registers whose contents could change at each loop iteration are responsible for this phenomenon. Performing an *l-bounded* analysis, the calculated symbolic value set for these registers would comprise only the special value $\perp$, i.e., we cannot determine the value of the register with our method. Such an inaccuracy in the analysis should not be accepted in practice. We improve the symbolic value set propagation algorithm by introducing the concept of non-invariant registers in a loop.

A *non-invariant register* of a loop $G'$ is a register $r_i$ used in $G'$, whose value is not proven to be constant in each loop iteration. The set of non-invariant registers contains, for instance, registers containing induction variables and registers which will be defined by a load instruction in $G'$. For each non-invariant register $r_i$ of a loop we insert an artificial initialization point $n$ *: init* $r_i$ into the control flow graph at the beginning of the

loop. The concept of artificial initialization points has two advantages: the number of iterations of the general iterative algorithm, which we use for data flow analysis, will be reduced. Additionally, we can compare memory addresses even though they depend on non-invariant registers.

Fig. 3 shows the results of a symbolic value set propagation using artificial initialization points for a simple program. The non-invariant registers of the loop are `%r1`, `%r2`, `%r3`, and `%r4`. For each non-invariant register an artificial initialization point is inserted into the program. As a consequence, the data flow algorithm terminates after the third iteration. The concept of artificial initialization points allows a more accurate analysis of memory references inside the loop. Without special treatment of non-invariant registers the value of register `%r1` would have been set to $\perp$ eventually.

**Symbolic Address Sets** Calculation of symbolic value sets of registers is necessary for the determination of symbolic address sets of a statement (SAS). In a subsequent step of our analysis we use the SAS information and information of control flow for the determination of may and must alias information as well as reaching definitions and reaching uses. In the last step we derive data dependence information of memory accesses.

In order to obtain all this information we need a mechanism which checks whether the index expressions of two storage accesses *X* and *Y* could (resp. must) represent the same value. To solve this problem, we replace the appearances of registers in *X* and *Y* with elements of their corresponding symbolic value sets, and check for all possible combinations whether the equation $X - Y = 0$ has a solution.

As an example, we refer to Fig. 3. Obviously, instruction 5 is a reaching use of memory in instruction 8. The derived memory addresses are $R_{1,12} - 40$ and $R_{1,12} - 80$, respectively. With the assumption that both instructions are executed in the same loop iteration, we can prove that different memory addresses will be accessed.

## 5.2 Basic definitions

In this section we introduce basic definitions that we need for the formal description of our method. Let $REGS$ be the set of all registers and $STMT$ the set of all statements. The set of all symbols is denoted as $SYM$. Note that all symbols that we introduced for initialization points are elements of $SYM$. Furthermore, let $defreg$ and $usereg$ be defined as following:

| | | 1. iteration | 2. iteration |
|---|---|---|---|
| 0 | `entry` | | |
| 1 | `mov 1,%r2` | | |
| 2 | `.LL11:` | | |
| 12 | `init %r1` | $\%\texttt{r2} = \{1\}$ | $\%\texttt{r1} = \{R_{1,0}, R_{1,12}\}$,$\%\texttt{r2} = \{1, R_{2,13}+1\}$<br>$\%\texttt{r3} = \{R_{3,0}, (R_{3,3}+R_{4,5})\}$,$\%\texttt{r4} = \{R_{4,0}, R_{4,5}\}$ |
| 13 | `init %r2` | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r2} = \{1\}$ | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r2} = \{1, R_{2,13}+1\}$<br>$\%\texttt{r3} = \{R_{3,0}, (R_{3,3}+R_{4,5})\}$,$\%\texttt{r4} = \{R_{4,0}, R_{4,5}\}$ |
| 14 | `init %r3` | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r2} = \{R_{2,13}\}$ | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r2} = \{R_{2,13}\}$<br>$\%\texttt{r3} = \{R_{3,0}, (R_{3,3}+R_{4,5})\}$,$\%\texttt{r4} = \{R_{4,0}, R_{4,5}\}$ |
| 15 | `init %r4` | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r2} = \{R_{2,13}\}$<br>$\%\texttt{r3} = \{R_{3,14}\}$ | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r2} = \{R_{2,13}\}$<br>$\%\texttt{r3} = \{R_{3,14}\}$,$\%\texttt{r4} = \{R_{4,0}, R_{4,5}\}$ |
| 3 | `ld [%r1-40],%r3` | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r2} = \{R_{2,13}\}$<br>$\%\texttt{r3} = \{R_{3,14}\}$,$\%\texttt{r4} = \{R_{4,15}\}$ | $\%\texttt{r1} = \{R_{1,12}\}$, $\%\texttt{r2} = \{R_{2,13}\}$<br>$\%\texttt{r3} = \{R_{3,14}\}$,$\%\texttt{r4} = \{R_{4,15}\}$ |
| 4 | `add %r2,1,%r2` | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r2} = \{R_{2,13}\}$<br>$\%\texttt{r3} = \{R_{3,3}\}$, $\%\texttt{r4} = \{R_{4,15}\}$ | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r2} = \{R_{2,13}\}$<br>$\%\texttt{r3} = \{R_{3,3}\}$,$\%\texttt{r4} = \{R_{4,15}\}$ |
| 5 | `ld [%r1-80],%r4` | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r3} = \{R_{3,3}\}$<br>$\%\texttt{r2} = \{(R_{2,13}+1)\}$,$\%\texttt{r4} = \{R_{4,15}\}$ | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r3} = \{R_{3,3}\}$<br>$\%\texttt{r2} = \{(R_{2,13}+1)\}$, $\%\texttt{r4} = \{R_{4,15}\}$ |
| 6 | `cmp %r2,9` | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r3} = \{R_{3,3}\}$<br>$\%\texttt{r2} = \{(R_{2,13}+1)\}$,$\%\texttt{r4} = \{R_{4,5}\}$ | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r3} = \{R_{3,3}\}$<br>$\%\texttt{r2} = \{(R_{2,13}+1)\}$,$\%\texttt{r4} = \{R_{4,5}\}$ |
| 7 | `add %r3,%r4,%r3` | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r3} = \{R_{3,3}\}$<br>$\%\texttt{r2} = \{(R_{2,13}+1)\}$,$\%\texttt{r4} = \{R_{4,5}\}$ | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r3} = \{R_{3,3}\}$<br>$\%\texttt{r2} = \{(R_{2,13}+1)\}$,$\%\texttt{r4} = \{R_{4,5}\}$ |
| 8 | `st %r3,[%r1-40]` | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r4} = \{R_{4,5}\}$<br>$\%\texttt{r2} = \{(R_{2,13}+1)\}$<br>$\%\texttt{r3} = \{(R_{3,3}+R_{4,5})\}$ | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r4} = \{R_{4,5}\}$<br>$\%\texttt{r2} = \{(R_{2,13}+1)\}$<br>$\%\texttt{r3} = \{(R_{3,3}+R_{4,5})\}$ |
| 9 | `add %r1,4,%r1` | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r4} = \{R_{4,5}\}$<br>$\%\texttt{r2} = \{(R_{2,13}+1)\}$<br>$\%\texttt{r3} = \{(R_{3,3}+R_{4,5})\}$ | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r4} = \{R_{4,5}\}$<br>$\%\texttt{r2} = \{(R_{2,13}+1)\}$<br>$\%\texttt{r3} = \{(R_{3,3}+R_{4,5})\}$ |
| 10 | `ble .LL11` | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r4} = \{R_{4,5}\}$<br>$\%\texttt{r2} = \{(R_{2,13}+1)\}$<br>$\%\texttt{r3} = \{(R_{3,3}+R_{4,5})\}$ | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r4} = \{R_{4,5}\}$<br>$\%\texttt{r2} = \{(R_{2,13}+1)\}$<br>$\%\texttt{r3} = \{(R_{3,3}+R_{4,5})\}$ |
| 11 | `retl` | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r4} = \{R_{4,5}\}$<br>$\%\texttt{r2} = \{(R_{2,13}+1)\}$<br>$\%\texttt{r3} = \{(R_{3,3}+R_{4,5})\}$ | $\%\texttt{r1} = \{R_{1,12}\}$,$\%\texttt{r4} = \{R_{4,5}\}$<br>$\%\texttt{r2} = \{(R_{2,13}+1)\}$<br>$\%\texttt{r3} = \{(R_{3,3}+R_{4,5})\}$ |

Figure 3: Symbolic value set propagation involving a loop. Registers $r_i$ that are not mentioned have the value $\{R_{i,0}\}$.

| | |
|---|---|
| $defreg\colon STMT \rightarrow \mathcal{P}(REGS)$, | statement $s$ may write to all registers in $defreg(s)$ |
| $usereg\colon STMT \rightarrow \mathcal{P}(REGS)$, | statement $s$ may read from any register in $usereg(s)$ |
| $ip\colon SYM \rightarrow STMT$, | symbol $x$ is defined by statement $ip(x)$ |

As mentioned before, $defreg$ and $usereg$ can be derived directly from the semantics of the instruction set of the supposed target processor.

## 5.3 Detecting non-invariant registers

A non-invariant register of a loop $G'$ is a register $r_i$ used in $G'$, the contents of which can change. There are several well-known algorithms for the determination of non-invariant registers of a loop. We use a technique that is based on a simple iterative algorithm [1].

Let $G'$ be a loop and $S$ a statement inside $G'$. A statement $S$ is called *loop invariant* if the destination register $r_i$ is defined with the same value in each loop iteration. The determination of loop invariant statements of $G'$ can be performed in two steps:

1. Mark all statements as loop invariant, which only use constants as operands or operands defined outside of $G'$.

2. Iteratively, mark all untagged statements of $G'$ as loop invariant which only use operands that are defined only by a loop invariant statement. The algorithm terminates if no further statement can be marked.

By using the concept of loop invariants we can determine the non-invariant registers of a loop $G'$ in a simple way. For this, a register $r_i$ is a non-invariant register in $G'$ iff $r_i$ is defined by a statement in $G'$ that is not a loop invariant statement in $G'$.

## 5.4 Symbolic value propagation

To represent functional relations between register contents and certain initial values represented by symbols from $SYM$, we use polynomials that are linear in the symbols. Each such polynomial can be described formally as a mapping from the symbols to the integers. For representing a constant additive term we introduce $\perp$ as an artificial symbol:

$\overline{SYM} = SYM \cup \{\perp\}$

Then, the space $SV$ of all formal linear polynomials in the symbols with coefficients from the integers can be described as follows:

$SV = [\overline{SYM} \rightarrow \mathbb{Z}]$

A more familiar representation of such a polynomial is as a formal sum:

$$v = a + \sum_h a_h \cdot x_h \qquad \text{with} \qquad a, a_h \in \mathbb{Z}, x_h \in SYM$$

**Free symbols** We describe the set of symbols with nonzero coefficients in a polynomial—the so-called *free symbols*—by a function $free$:

$free\colon SV \to \mathcal{P}(SYM), v \mapsto \{x \in SYM : v(x) \neq 0\}$

We write $v = a$ as a shortcut when $free(v) = \emptyset$ and $v(\bot) = a$.

**Operations** Since we want to stay within the space of linear polynomials, only some of the known operations on polynomials are feasible. These are:

$$\forall v, w \in SV, x \in \overline{SYM}\colon (v \oplus w)(x) = v(x) + w(x)$$

$$\forall v, w \in SV, x \in \overline{SYM}\colon (v \ominus w)(x) = v(x) - w(x)$$

$$\forall v \in SV, x \in \overline{SYM}\colon (\ominus v)(x) = -v(x)$$

$$\forall v \in SV, x \in \overline{SYM}, c \in \mathbb{Z}\colon (c \odot v)(x) = (v \odot c)(x) = c \cdot v(x)$$

For representing unknown or approximated values, we use $\overline{SV} = SV \cup \{\bot\}$.

**A bounded semi-lattice** Since we abstract from the predicates of conditionals, we must work with sets of values from $SV$. For doing an effective analysis each such set should contain only a small number of elements. An appropriate "bounding concept" is introduced via the space $SVS$:

$SVS = \{S \subseteq SV : |S| \leq l\}$

where $l$ is some predetermined natural. The size of $l$ influences the precision of the analysis and should be chosen carefully to fit the demands. Furthermore, since $\bot$ stands for any value, we can restrict in this respect to a single set containing $\bot$ alone:

$\overline{SVS} = SVS \cup \{\{\bot\}\}$

We use $\{\bot\}$ instead of $\bot$ to have the opportunity to enumerate sets from $\overline{SVS}$ or to build cartesian products from them.

On $\overline{SVS}$ we can define a meet operator $\sqcap_{\overline{SVS}}\colon \overline{SVS} \times \overline{SVS} \to \overline{SVS}$:

$$S \sqcap_{\overline{SVS}} T = \begin{cases} S \cup T & \text{if } S, T \in SVS \text{ and } |S \cup T| \leq l, \\ \{\bot\} & \text{else} \end{cases}$$

Remark: $(\overline{SVS}, \sqcap_{\overline{SVS}})$ is a bounded semi-lattice with zero element $\{\bot\}$ and "one" element $\emptyset$.

**Approximation of the values in registers** What we are now looking for is an approximation of the values that the registers may contain when reaching a certain statement. This approximation must be safe, in the sense that not every value found during the analysis necessarily appears during runtime but all values that do in fact show up have to be included in the approximating set.

$regvals\colon STMT \to [REGS \times \overline{SVS}]$,

$(r, v) \in regvals(s) \Leftrightarrow$ register $r$ may contain value $v$ when read by statement $s$

We calculate $regvals$ by the following MDFS:

Data flow information set[4]: $RSV = [REGS \times \overline{SVS}]$

One element: $\emptyset$

Meet operator $\sqcap_{RSV}\colon RSV \times RSV \to RSV$:

$$\forall r \in REGS\colon (f \sqcap_{RSV} g)(r) = f(r) \sqcap_{\overline{SVS}} g(r)$$

Semantic functions:

$$K_s(D) = defreg(s) \times \overline{SVS}$$

$$G_s(D) = \{r_i \times \mathcal{F}(D, g_i, r_{i_1}, \ldots, r_{i_k})\colon r_i \in defreg(s)\} \qquad \text{(abbreviation: } [g_i; r_{i_1}, \ldots, r_{i_k}]_i\text{)}$$

where the functions $g_i$ depend on the instruction performed, and

$$\mathcal{F}(D, g, r_{i_1}, \ldots, r_{i_k}) = \sqcap_{\overline{SVS}}\{\hat{g}(v_1, \ldots, v_k)\colon (r_{i_j}, v_j) \in D\}$$

with

$$\hat{g}(v_1, \ldots, v_k) = \{g(v_1, \ldots, v_k)\}$$

in the sequel (although you may think of more general situations).

Semantic functions have to be defined for any instruction. The most important cases then are:

| | |
|---|---|
| `mov a,ri` | $[a;]_i$ with $a() = a$ |
| `mov rj,ri` | $[id; r_j]_i$ with $id(v) = v$ |
| `init ri` | $[unknown;]_i$ with $unknown() = R_{i,s} \in SYM$ |
| `ld [mem],ri` | $[unknown;]_i$ |
| `entry` | $[unknown;]_i$ for all registers $r_i$ |

[4]Actually, the information at each program point is a *function* from the registers, but we found it convenient to define it as a relation, so as to simplify the definition of the semantic functions.

`call` $[unknown;]_i$ for a suitable subset of the registers

`add rj,rk,ri` $[add; r_j, r_k]_i$ with $add(v,w) = \begin{cases} \bot & \text{if } v = \bot \text{ or } w = \bot \\ v \oplus w & \text{else} \end{cases}$

`sub rj,rk,ri` $[sub; r_j, r_k]_i$ with $sub(v,w) = \begin{cases} \bot & \text{if } v = \bot \text{ or } w = \bot \\ v \ominus w & \text{else} \end{cases}$

`mul rj,rk,ri` $[mul; r_j, r_k]_i$ with $mul(v,w) = \begin{cases} 0 & \text{if } v = 0 \text{ or } w = 0 \\ c \odot w & \text{if } v = c \in \mathbb{Z} \text{ and } w \neq \bot \\ v \odot c & \text{if } v \neq \bot \text{ and } w = c \in \mathbb{Z} \\ \bot & \text{else} \end{cases}$

`div rj,rk,ri` $[div; r_j, r_k]_i$ with $div(v,w) = \begin{cases} v & \text{if } w = 1 \\ \ominus v & \text{if } v \neq \bot \text{ and } w = -1 \\ \bot & \text{else} \end{cases}$

**Remark:** For division, we can only handle trivial cases since we are reasoning in $\mathbb{Z}$ (we can reduce coefficients modulo $2^L$, but can not for symbols) whereas the machine is calculating in $\mathbb{Z}_{2^L}$, the ring of residues modulo $2^L$. For addition, subtraction, and multiplication, there appear no essential problems because these operations provide ring homomorphisms between $\mathbb{Z}$ and $\mathbb{Z}_{2^L}$. Division is more critical in this sense, as shows the following example:

Let $r_1 = 2^{L-1}$ and $r_2 \equiv 0 \pmod 2$. After the statement sequence

```
mul r1,r2,r3
div r3,r2,r1
```

$r_1 = 2^{L-1}$ in $\mathbb{Z}$ but $r_1 = 0$ in $\mathbb{Z}_{2^L}$.

There are variants of these instructions with integer constants used instead of operand registers; these are treated analogously, one of the polynomials degenerating to the constant additive term.

We have $D_s^{out} = D_s^{in}$ if $defreg(s) = \emptyset$.

For all other instructions not treated so far we could set $[bottom;]_i$ for all registers $r_i$ with $r_i \in defreg(s)$, where $bottom() = \bot$.

Of course, most of these functions could be modelled a bit more precisely—e.g. shiftleft like mul, shiftright like div, exact handling of boolean operators—but we are not sure that it would be worth the effort.

**Remark:** The assumption that all calculations be modulo $2^L$ is essential for addition, subtraction, and other instructions. Should the processor trap with overflow, functions like $add$ must be defined in a much more restrictive way than above to get a safe approximation. For instance, we could set

$$add(v,w) = \begin{cases} v \oplus w & \text{if } v,w \neq \bot \text{ and } v \oplus w \in Int, \\ w & \text{if } v = 0, \\ v & \text{if } w = 0, \\ \bot & \text{else} \end{cases}$$

and

$$sub(v,w) = \begin{cases} v \ominus w & \text{if } v,w \neq \bot \text{ and } v \ominus w \in Int, \\ \bot & \text{else} \end{cases}$$

where $Int$ is the set of integers representable in a register.

$\bot$ here also models possible overflow situations that could terminate the execution of the instruction with an exception.

Remark: The reader might doubt that a symbol propagated in the different registers to an add instruction that uses both registers as operands has the same meaning in the polynomials and thus can be added. The feasibility of this operation follows from the fact that initialization points introduce new incarnations of symbols in each iteration for all such registers.

## 5.5 Evaluating address expressions

The results of our symbolic value propagation have to be substituted into the address expressions of all memory instructions to get a safe approximation of the addresses accessed by these instructions. To perform this substitution we have to touch each instruction only once. We assume that the only instructions involving memory are loads, stores, and calls. The set of addresses possibly accessed by an instruction is calculated as follows:

$addr\colon STMT \rightarrow \overline{SVS}$,

$v \in addr(s) \Leftrightarrow v$ may be the address of a memory cell accessed in statement $s$

$$addr(s) = \begin{cases} \mathcal{F}(regvals(s), add(a,.), r_j) & \text{if } \texttt{[rj+a]} \text{ is the address expression} \\ \mathcal{F}(regvals(s), add, r_j, r_k) & \text{if } \texttt{[rj+rk]} \text{ is the address expression} \\ \{\bot\} & \text{if } s \text{ is a call instruction} \\ \emptyset & \text{else} \end{cases}$$

Now, we have to distinguish whether an instruction reads from memory or writes to it. Thus, we derive functions $defmem$ and $usemem$ from $addr$.

$defmem\colon STMT \to \overline{SVS}$,

$v \in defmem(s) \Leftrightarrow$ statement $s$ possibly writes to address $v$

$$defmem(s) = \begin{cases} addr(s) & \text{if } s \text{ is a store or call instruction} \\ \emptyset & \text{else} \end{cases}$$

$usemem\colon STMT \to \overline{SVS}$,

$v \in usemem(s) \Leftrightarrow$ statement $s$ possibly reads from address $v$

$$usemem(s) = \begin{cases} addr(s) & \text{if } s \text{ is a load or call instruction} \\ \emptyset & \text{else} \end{cases}$$

The handling of call instructions here is as approximative as could be when staying conservative in the analysis, and could possibly be improved by distinguishing certain storage areas with specific bottom symbols.

## 5.6 Reaching definitions and uses of memory

We are now ready for setting up reaching definitions and uses of memory. We formulate this pass as a MDFS that propagates a set of possible reaching definitions (resp. uses) to the successors of a statement. In order to keep the reaching sets as small as possible—and thus the analysis as precise as could be—we use a special form of must alias analysis where we check if the memory effect of a previous statement $t$ is completely invisible to the successors of a statement $s$. To assure this, $defmem(s)$ must be an alias of $addr(t)$; we formulate this by a predicate $covers$.

Function for reaching definitions: $rdmem\colon STMT \to \mathcal{P}(STMT)$,

$t \in rdmem(s) \Leftrightarrow$ statement $s$ can possibly observe a storage contents written by statement $t$

Function for reaching uses: $rumem\colon STMT \to \mathcal{P}(STMT)$,

$t \in rumem(s) \Leftrightarrow$ statement $s$ can possibly observe a storage contents read by statement $t$

Data flow information set for reaching definitions and uses: $STMT$

One element: $\emptyset$

Meet operator $\sqcap_{STMT}$: $STMT \times STMT \rightarrow STMT$: $S \sqcap_{STMT} T = S \cup T$

Semantic functions for reaching definitions:

$$G_s(D) = \begin{cases} \{s\} & \text{if } s \text{ is a store or call instruction} \\ \emptyset & \text{else} \end{cases}$$

$$K_s(D) = \{t \in D \colon covers(s, defmem(s), t, defmem(t))\}$$

Semantic functions for reaching uses:

$$G_s(D) = \begin{cases} \{s\} & \text{if } s \text{ is a load or call instruction} \\ \emptyset & \text{else} \end{cases}$$

$$K_s(D) = \{t \in D \colon covers(s, defmem(s), t, usemem(t))\}$$

For the must alias relation, we find the following situation:

An address set $A$ can only cover another address set $B$ if each element in $B$ is covered by each element in $A$.

$$covers(s, A, t, B) = \bigwedge_{(a,b) \in A \times B} covers2(s, a, t, b)$$

The unknown address $\perp$ can never cover any address, and can never be covered by any address.

$$covers2(s, a, t, b) = \begin{cases} false & \text{if } a = \perp \text{ or } b = \perp \\ covers3(s, a, t, b) & \text{else} \end{cases}$$

The predicate $sep$ (formally introduced later) describes the situation where symbols to be compared can never be generated both in one run of the analyzed code. Cases where such symbols appear together in comparisons are spurious; they are due to the abstraction of control flow where the predicates of conditionals are not used in deciding on paths to be taken.

$$covers3(s, a, t, b) = \begin{cases} true & \text{if } sep(x, y) \\ & \text{for some } x \in free(a), y \in free(b) \\ covers4(s, a, t, b \ominus a) & \text{else} \end{cases}$$

When comparing two addresses for coverage we have to form the difference between them. This difference is itself a polynomial, and when it contains a free symbol then we might find a valuation of the symbols where this difference is large in magnitude and thus coverage is not assured. Thus, $covers4$ never has the value $true$ when there are any free symbols in the difference polynomial.

$$covers4\,(s,a,t,p) = \begin{cases} false & \text{if } free(p) \neq \emptyset \\ covers5(s,a,t,p) & \text{else} \end{cases}$$

Moreover, eliminating free symbols from both argument polynomials by assuming equality is only feasible when such a symbol has the same meaning in both contexts. To delete a statement $t$ from the reaches (resp. uses) set arriving at statement $s$, we have to be sure that (i) the memory contents addressed in statement $t$ is overwritten every time the control flow passes statement $s$, and that (ii) there is no way to bypass statement $s$ between the use of a symbol $x$ in statement $t$ and its redefinition. The latter is checked with a function $notpassing$, explained below[5].

Finally, we check whether the memory block of $size(s)$ bytes starting at address $a$ completely covers the memory block of $size(t)$ bytes starting at address $b$[6].

$$covers5(s,a,t,p) = \begin{cases} false & \text{if } (t,s) \in notpassing(ip(x)) \\ & \text{for some } x \in free(a) \\ (p \leq size(s) - size(t)) & \text{else} \end{cases}$$

The function $size\colon STMT \rightarrow \mathbb{N}$ reflects the number of bytes accessed in memory.

**Symbols defined on a common path: predicate** $sep$

$sep \subseteq SYM \times SYM$

$(x,y) \in sep \Leftrightarrow$ the symbols $x$ and $y$ are not defined on a common path

$sep$ can be derived from a function $reaches$, describing which statements reach a statement.

[5]Function $ip(.)$ has been introduced on page 13.

[6]If the number of bytes read or written by a memory instruction is always the same, the formula $p \leq size(s) - size(t)$ in $covers5$ simplifies to $p = 0$

$sep(x, y) \Leftrightarrow ip(x) \neq ip(y) \wedge ip(x) \notin reaches(ip(y)) \wedge ip(y) \notin reaches(ip(x))$

$reaches\colon STMT \rightarrow \mathcal{P}(STMT)$,

$t \in reaches(s) \Leftrightarrow$ there is a path from statement $t$ to statement $s$

We calculate $reaches$ by the following MDFS:

Data flow information set: $STMT$

One element: $\emptyset$

Meet operator $\sqcap$: $STMT \times STMT \rightarrow STMT$: $S \sqcap T = S \cup T$

Semantic functions:

$$K_s(D) = \emptyset$$

$$G_s(D) = \{s\}$$

**Predicate** $notpassing$

$notpassing\colon STMT \rightarrow \mathcal{P}(STMT \times STMT)$

$(t, u) \in notpassing(s) \Longleftrightarrow$ there is a path from statement $t$ to statement $s$ not passing statement $u$

To calculate $notpassing$ with a MDFS:

Data flow information set: $\mathcal{P}(STMT \times STMT)$

One element: $\emptyset$

Meet operator: $S \sqcap T = S \cup T$

Semantic functions:

$$K_s(D) = \{(t, s) \in D\}$$

$$G_s(D) = \{(s, t)\colon s \in reaches(t)\}$$

or equivalently in our case: $G_s(D) = \{(s, t)\colon t \in reaches(s)\}$.

## 5.7 Determination of memory based data dependences

To derive memory based data dependences from reaching definitions (resp. uses) for memory, we have to check whether the memory cells accessed in one statement may intersect with the memory cells accessed in another statement. The intersection is checked using the $cut$ predicate. Given this, the data dependences are as follows:

$t$ is flow dependent on $s \Leftrightarrow s \in rdmem(t)$

$\wedge\ cut(size(s), defmem(s), size(t), usemem(t))$

$t$ is anti dependent on $s \Leftrightarrow s \in rumem(t)$
$\wedge\ cut(size(s), usemem(s), size(t), defmem(t))$
$t$ is output dependent on $s \Leftrightarrow s \in rdmem(t)$
$\wedge\ cut(size(s), defmem(s), size(t), defmem(t))$

An intersection between sets of addresses can only be excluded when there is definitely no intersection for any pair of addresses.

$$cut(ss, A, st, B) = \bigvee_{(a,b)\in A\times B} cut2(ss, a, st, b)$$

An unknown address $\perp$ may intersect with any address.

$$cut2(ss, a, st, b) = \begin{cases} true & \text{if } a = \perp \text{ or } b = \perp \\ cut3(ss, a, st, b) & \text{else} \end{cases}$$

If addresses contain symbols that are incompatible they can never appear together.

$$cut3(ss, a, st, b) = \begin{cases} false & \text{if } sep(x, y) \text{ for some } x \in free(a), y \in free(b) \\ cut4(ss, a, st, b) & \text{else} \end{cases}$$

Now, we have to form a difference polynomial for the final intersection test. A common free symbol from the polynomials $a$ and $b$ might take different values in both polynomials. For correctly setting up the difference polynomial we have to substitute a new symbol for the common one into one of the polynomials if its value is not fixed. The latter property is checked by the predicate $variant$.

$$cut4(ss, a, st, b) = cut5(ss, a', st, b)$$
$$\text{with } a' = a[dup(x)/x, \text{ for all } x \in free(a) \cap free(b) \text{ such that } variant(x)]$$

In this formula, $a[dup(x)/x \ldots]$ means that we substitute a new symbol for each conflicting old one, all at the same time. The function $dup$ has to provide unique duplicates for the symbols appearing in the semantic functions without interfering with the latter. An easy way to achieve this would be the following definition (assuming that statements are counted up from 0):

$dup\colon SYM \rightarrow SYM, R_{i,s} \mapsto R_{i,-s-1}$

We define $variant(x) \Leftrightarrow ip(x) \in reaches(ip(x))$. However, in special contexts like loop body analysis, we might want to be more precise.

$$cut5(ss, a', st, b) = ( \lceil \frac{1 - ss - c}{\delta} \rceil \leq \lfloor \frac{st - 1 - c}{\delta} \rfloor )$$

$$\begin{aligned} \text{where} \quad & c = p(\perp), \\ & p = a' - b, \\ & \text{and } \delta = \gcd(2^L, \gcd_{x \in free(p)}\{p(x)\}) \end{aligned}$$

We refer to the appendix (section 9) for the derivation of this formula. This implements what can be considered as an extension of the GCD test—the best we can do in the current situation[7].

# 6 Implementation and Results

The method for determining data dependences in assembly code presented in the last sections was implemented as a user function in SALTO on a Sun SPARC 10 workstation running Solaris 2.5, with the following simplifications:

- $covers3(ss, a, st, b) = covers4(ss, a, st, b \ominus a)$
- $covers5(ss, a, st, p) = \begin{cases} false & \text{if variant(x)} \\ & \text{for some } x \in \text{free(a)} \\ (p \leq size(s) - size(t)) & \text{else} \end{cases}$
- $cut3(ss, a, st, b) = cut4(ss, a, st, b)$

Presently, only the assembly code for the SPARC V7 processor can be analyzed, but an extension to other processors will require minimal technical effort. Results of our analysis can be used by other tools in SALTO. For evaluation of our method we have taken a closer look at two aspects:

[7] As shown in the appendix, the test amounts to asking that an integer multiple of $\delta$ be comprised between the numbers $(1 - s - c)$ and $(t - 1 - c)$; so it amounts to the classical GCD test, applied to all values in this interval.

1. Comparison of the number of data dependences using our method against the method implemented in SALTO; this shows the difference between address-based and value-based dependence analysis concerning register accesses.

2. Comparison between the number of data dependences using address-based and value-based dependence analysis for memory accesses.

As a sample we chose 160 procedures out of the sixth public release of the Independent JPEG Group's free JPEG software, a package for compression and decompression of JPEG images. We distinguish between the following four levels of accuracy: in level 1 we determine address-based dependences between register accesses, memory is modeled as one cell, so that every pair of memory accesses is assumed to introduce a data dependence. Level 2 models the memory the same way as in level 1, and does value-based dependence analysis for register accesses. From level 3 on, register accesses are determined the same way as in level 2, and we analyze memory accesses with our symbolic value set propagation, but in level 3 the derivation of dependence is address-based. In level 4 we perform value-based dependence analysis. Level 1 analysis is performed by SALTO [20], but SALTO does not even consider control flow. Two instructions are assumed to be data dependent, even if they cannot be executed one after another. Level 2 is a common technique used by today's instruction schedulers, e.g. the one in `gcc` [25] or the one used by Larus et. al. [21]. Systems that do some kind of value propagation, but only determine address-based dependences, are classified as level 3. In section 7 we will have a closer look at other techniques for value propagation. Our method is classified as level 4. As yet, we know of no other method which also determines value-based dependences. The table 2 only contains those 39 procedures in which an improvement, i.e., less dependences, was noticeable from level 3 to level 4. It shows the number of dependences (sum of true, anti-, and output dependences), where we distinguish different levels of accuracy, as well as register and memory accesses. Table 2 also shows in the two rightmost columns the effect of a value-based analysis against an address-based analysis. For every procedure it is clear to see the proportion of data dependences that our method disproves.

| Procedure Name | LOC | Level 1 | | Level 2 | | Level 3 | | Level 4 | | Improvement | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Reg. | Mem. | Reg. | Mem. | Reg. | Mem. | Reg. | Mem. | Reg. | Mem. |
| keymatch | 59 | 643 | 81 | 149 | 81 | 149 | 58 | 149 | 47 | 77% | 19% |
| test3function | 15 | 24 | 29 | 15 | 29 | 15 | 16 | 15 | 13 | 38% | 19% |
| is_shifting_signed | 33 | 178 | 38 | 87 | 38 | 87 | 31 | 87 | 22 | 51% | 29% |
| jpeg_CreateCompress | 126 | 4273 | 1945 | 423 | 1945 | 423 | 1664 | 423 | 1619 | 90% | 3% |
| jpeg_suppress_tables | 74 | 1127 | 396 | 143 | 396 | 143 | 229 | 143 | 184 | 87% | 20% |
| jpeg_finish_compress | 144 | 10432 | 2333 | 1121 | 2333 | 1121 | 2210 | 1121 | 2197 | 89% | 1% |
| emit_byte | 42 | 433 | 214 | 119 | 214 | 119 | 189 | 119 | 184 | 73% | 3% |
| emit_dqt | 125 | 4794 | 1097 | 575 | 1097 | 575 | 771 | 575 | 726 | 88% | 6% |
| emit_dht | 134 | 5219 | 1461 | 589 | 1461 | 589 | 980 | 589 | 870 | 89% | 11% |
| emit_sof | 100 | 6389 | 1282 | 661 | 1282 | 661 | 1087 | 661 | 1077 | 90% | 1% |
| emit_sos | 100 | 5252 | 1285 | 574 | 1285 | 574 | 873 | 574 | 840 | 89% | 4% |
| write_any_marker | 41 | 561 | 175 | 184 | 175 | 184 | 110 | 184 | 106 | 67% | 4% |
| write_frame_header | 142 | 4309 | 1368 | 679 | 1368 | 679 | 870 | 679 | 744 | 84% | 14% |
| write_scan_header | 86 | 3656 | 626 | 934 | 626 | 934 | 486 | 934 | 459 | 74% | 6% |
| write_tables_only | 83 | 2495 | 390 | 716 | 390 | 716 | 324 | 716 | 267 | 71% | 18% |
| jpeg_abort | 38 | 268 | 84 | 93 | 84 | 93 | 67 | 93 | 63 | 65% | 6% |
| jpeg_CreateDecompress | 124 | 4878 | 1972 | 507 | 1972 | 507 | 1716 | 507 | 1659 | 90% | 3 % |
| jpeg_start_decompress | 135 | 4097 | 902 | 674 | 902 | 674 | 860 | 674 | 856 | 84% | 1% |
| post_process_2pass | 111 | 2583 | 1385 | 278 | 1385 | 278 | 907 | 278 | 878 | 89% | 3% |
| jpeg_read_coefficients | 113 | 3783 | 897 | 538 | 897 | 538 | 853 | 538 | 851 | 86% | 1% |
| select_file_name | 104 | 5631 | 1146 | 473 | 1146 | 473 | 714 | 473 | 644 | 92% | 10% |
| jround_up | 20 | 80 | 29 | 30 | 29 | 30 | 20 | 30 | 15 | 62% | 25% |
| jcopy_sample_rows | 46 | 354 | 197 | 115 | 197 | 115 | 89 | 115 | 64 | 68% | 28% |
| read_1_byte | 48 | 653 | 84 | 186 | 84 | 186 | 70 | 186 | 67 | 72% | 4% |
| read_2_bytes | 93 | 3115 | 360 | 555 | 360 | 555 | 297 | 555 | 285 | 82% | 4% |
| next_marker | 42 | 567 | 137 | 305 | 137 | 305 | 112 | 305 | 98 | 46% | 12% |
| first_marker | 84 | 1989 | 259 | 360 | 259 | 360 | 197 | 360 | 187 | 82% | 5% |
| skip_variable | 33 | 533 | 83 | 258 | 83 | 258 | 83 | 258 | 73 | 52% | 12% |
| process_COM | 107 | 6901 | 979 | 1147 | 979 | 1147 | 697 | 1147 | 592 | 83% | 15% |
| process_SOFn | 75 | 4545 | 729 | 670 | 729 | 670 | 601 | 670 | 598 | 85% | 1% |
| scan_JPEG_header | 34 | 804 | 82 | 306 | 82 | 306 | 78 | 306 | 77 | 62% | 1% |
| keymatch | 59 | 643 | 81 | 149 | 81 | 149 | 58 | 149 | 47 | 77% | 19% |
| read_byte | 43 | 415 | 105 | 129 | 105 | 129 | 102 | 129 | 97 | 69% | 5% |
| read_colormap | 67 | 2221 | 668 | 305 | 668 | 305 | 583 | 305 | 568 | 86% | 3% |
| read_non_rle_pixel | 40 | 368 | 93 | 125 | 93 | 125 | 84 | 125 | 83 | 66% | 1% |
| read_rle_pixel | 80 | 976 | 289 | 268 | 289 | 268 | 280 | 268 | 279 | 73% | 1% |
| jcopy_sample_rows | 46 | 354 | 197 | 115 | 197 | 115 | 89 | 115 | 64 | 68% | 28% |
| flush_packet | 44 | 468 | 187 | 131 | 187 | 131 | 187 | 131 | 182 | 72% | 3% |
| start_output_tga | 215 | 12870 | 3272 | 974 | 3272 | 974 | 2937 | 974 | 2876 | 92% | 2% |

Table 2: Number of dependences (sum of true, anti-, and output dependences) found in four levels of accuracy. The results are divided into register-based and memory-based dependences. The two rightmost columns show the improvement of a value-based dependence analysis on an address-based dependence analysis.

# 7 Related Work

So far, only limited amount of work has been done in the field of memory reference disambiguation. Ellis [8] presented a method to derive symbolic expressions for memory addresses by chasing back all reaching definitions of a symbolic register, the expression is simplified using rules of algebra, and two expressions are compared using the GCD test. The method is implemented in the Bulldog compiler, but it works on an intermediate level close to high-level language. Other authors were inspired by Ellis, e.g. Lowney et. al. [14], Böckle [4], and Ebcioğlu et. al. [16]. The latter approach is implemented in the Chameleon compiler [17] and works on assembly code. First, a procedure is transformed into SSA form [5], and loops are normalized. For gathering possible register values the same technique as in the Bulldog compiler is used. If a register has multiple definitions, the algorithm described in [16] can chase all reaching definitions, whereas the concrete implementation in the Chameleon compiler seems to not support this. Comparing memory addresses makes use of the GCD test and the Banerjee inequalities [2, 28]. The results of their method are alias information. Debray et. al. [7] present an approach close to ours. They use address descriptors to represent abstract addresses, i.e., addresses containing symbolic registers. An address descriptor is a pair $< I, M >$ where $I$ is an instruction and $M$ is a set of $mod - k$ residues. $M$ denotes a set of offsets relative to the register defined in instruction $I$. Note that an address descriptor only depends on *one* symbolic register. A data flow system is used to propagate values through the control flow graph. $mod - k$ sets are used as a bounded semi-lattice is needed (in the tests it is $k = 64$). However this leads to an approximation of address representation that makes it impossible to derive must-alias information. The second drawback is that definitions of the same register in different control flow paths are not joined in a set, but mapped to $\perp$. Comparing address descriptors can be reduced to a comparison of $mod - k$ sets, using some dominator information to handle loops correctly. They do not derive data dependence information.

# 8 Conclusions

In this paper we presented a new method to detect data dependences in assembly code. It works in two steps: First we perform a symbolic value set propagation using a monotone data flow system. Then we compute reaching definitions and reaching uses for memory access, and derive value-based data dependences. For comparing memory references we use a modification of the GCD test. All known approaches for memory reference disambiguation do not propagate values through memory cells. Remember that loading from memory causes the destination register to have a symbolic value. When we compare two memory references we must have in mind that registers defined in different instructions may have different values, even if they were loaded from the same memory address. To handle this situation we plan to extend our method to propagate values through memory cells.

Software pipelining will be one major application of the present work in the near future; this family of techniques overlaps the execution of different iterations from an original loop, and therefore requires a very precise dependence analysis with additional information about the distance of the dependence. Development of this work entails in particular discovering induction variables, which is possible as a post-pass, as soon as loop invariants are known. Then coupling with known dependence tests, such as Banerjee test or Omega test [19] can be considered.

Finally, extending our method to interprocedural analysis would lead to a more accurate dependence analysis. Presently we have to assume that the contents of almost all registers and all memory cells may have changed after the evaluation of a procedure call.

# 9 Appendix : derivation of formula $cut5$

See section 5.7 for the use of this predicate and notations. Note that $\delta$ is always positive, as it is the gcd of several quantities including $2^L$.

$$
\begin{aligned}
cut5(ss, a', st, b) \Leftrightarrow & \\
& \exists \alpha, \beta \in \mathbb{Z}, h, k \in \mathbb{N}_0 : h < ss, k < st : \\
& \qquad a' + h + \alpha \cdot 2^L = b + k + \beta \cdot 2^L \\
& \Leftrightarrow \\
& \exists \alpha, \beta \in \mathbb{Z}, h, k \in \mathbb{N}_0 : h < ss, k < st : \\
& \qquad p + (\alpha - \beta) \cdot 2^L = k - h \\
& \qquad \text{with } p = a' - b \\
& \Leftrightarrow \\
& \qquad \exists \alpha, \beta \in \mathbb{Z}, h, k \in \mathbb{N}_0 : h < ss, k < st : \\
& \qquad (p - c) + (\alpha - \beta) \cdot 2^L = k - h - c \\
& \qquad \text{with } c = p(\bot) \\
& \Leftrightarrow \\
& \exists \gamma \in \mathbb{Z}, h, k \in \mathbb{N}_0 : h < ss, k < st : \\
& \qquad \gamma \cdot \delta = k - h - c \\
& \qquad \text{with } \delta = \gcd(2^L, \gcd_{x \in free(p)}\{p(x)\}) \\
& \Leftrightarrow \\
& \exists \gamma \in \mathbb{Z} : \\
& \qquad -ss + 1 \leq \gamma \cdot \delta + c \leq st - 1
\end{aligned}
$$

Hence the conclusion:

$$cut5(ss, a', st, b) = \lceil \frac{1 - ss - c}{\delta} \rceil \leq \lfloor \frac{st - 1 - c}{\delta} \rfloor$$

# 10 Appendix: What is a loop

Let $G = (N, E, s)$ be a control flow graph. Let $G' = (N', E', s')$ denote a subflowgraph of $G$, i.e., $N' \subseteq N$ and $E' = E \cap (N' \times N')$. $G'$ is a loop with entry point $s' :\Leftrightarrow \forall (n, n') \in E : n' \in N' \Rightarrow n' = s' \vee n \in N'$ and for every pair of nodes $n, n' \in N'$ there are non-trivial paths from n to n' and vice versa.

In Fig. 4 we show the well-known algorithm for the determination of all nodes associated with loop $G'$ with entry point $s'$ and the backward edge $(n', s')$, which can be found by using the dominance relation [28, 1]. In the algorithm, $pred(a)$ stands for the set of all predecessors of node a. At the end, set $L$ contains all nodes of the loop.

```
procedure insert(x)
    if x ∉ L then
        L := L ∪ {x};
        stack.push(x);
    end if
end procedure

procedure main
    L := { s' };
    insert( n' );
    while stack.notempty() do
        a := stack.pop();
        foreach b ∈ pred(a) do insert(b) end
    end
end procedure
```

Figure 4: Algorithm for the determination of all nodes of a loop.

# References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1988.

[2] U. Banerjee. *Dependence analysis for supercomputing*. Kluwer Academic, Boston, MA, USA, 1988.

[3] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 241–255, Toronto, Canada, June 1991.

[4] G. Böckle. *Exploitation of Fine-Grain Parallelism*, volume 942 of *LNCS*. Springer-Verlag, Berlin, Germany, 1995.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadek. An efficient method of computing static single assignment form. In *Proceedings of the Sixteenth Annual ACM Symposium on the Principles of Programming Languages*, pages 25–35, Austin, Texas, Jan. 1989.

[6] S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallet. Some experiments in local microcode compaction for horizontal machines. *IEEE Trans. on Computers*, 30(7):460–477, July 1981.

[7] S. Debray, R. Muth, and M. Weippert. Alias analysis in executable code. In *Proceedings of the Twenty-fifth Annual ACM Symposium on the Principles of Programming Languages*, pages 12–24, San Diego, California, 1998.

[8] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, 1985.

[9] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. on Computers*, 30(7):478–490, July 1981.

[10] J. A. Fisher. Walk-time techniques catalyst for architectural change. *IEEE Computer*, 30(9):40–42, September 1997.

[11] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[12] W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *Proceedings of the Eighteenth Annual ACM Symposium on the Principles of Programming Languages*, pages 93–103, Orlando, Florida, Jan. 1991.

[13] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. *ACM SIGPLAN Notices*, 23(7):21–34, July 1988.

[14] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7:51–142, 1993.

[15] V. Maslov. Lazy array data-flow dependence analysis. In *Proceedings of the Twenty-first Annual ACM Symposium on the Principles of Programming Languages*, pages 311–325, 1994.

[16] S.-M. Moon and K. Ebcioğlu. A study on the number of memory ports in multiple instruction issue machines. In *Proc. of the 26th Annual International Symposium on Microarchitecture MICRO-26*, pages 49–58, 1993.

[17] M. Moudgill, J. H. Moreno, K. Ebcioğlu, E. Altman, S. K. Chen, and A. Polyak. Compiler/architecture interaction in a tree-based VLIW processor. In *Workshop on Interaction between Compilers and Computer Architectures '97 in conjunction with HPCA-3*, San Antonio, TX, Feb. 1997.

[18] A. Nicolau. Percolation scheduling: A parallel compilation technique. Technical report, Dep. of Computer Science, Cornell University, 1985.

[19] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.

[20] E. Rohou, F. Bodin, and A. Seznec. SALTO: System for assembly-language transformation and optimization. In *Proc. 6th Workshop on Compilers for Parallel Computers*, pages 261–272, Aachen, Dec. 1996.

[21] E. Schnarr and J. R. Larus. Instruction scheduling and executable editing. In *Proc. of the 29th Annual International Symposium on Microarchitecture MICRO-29*, pages 288–297, Paris, France, Dec. 1996.

[22] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary translation. *Digital Technical Jornal*, 4(4):137–152, 1992.

[23] A. Srivastava and D. W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, December 1992.

[24] R. M. Stallman. Using and porting the GNU CC. Technical report, Free Software Foundation, Cambridge, MA, 1989.

[25] M. D. Tiemann. The GNU instruction scheduler. Technical report, Free Software Foundation, June 1989.

[26] D. Wall. *Systems for Late Code Modification*. In R. Giegerich and S. L. Graham, editors, *Code Generation — Concepts, Tools, Techniques*, Workshops in Computing, pages 275–293. Springer-Verlag, 1992.

[27] D. Wall. Limits to instruction level parallelism. *4th Architectural Support for Programming Languages and Operating Systems*, pages 176–188, April 8-11, 1991.

[28] H. Zima and B. Chapman. *Supercompilers for parallel and vector computers*. Addison-Wesley, Reading, MA, 1991.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifi que,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399