Data Dependence Analysis of Assembly Code

Wolfram Amme, Peter Braun, Eberhard Zehendner
Computer Science Department
Friedrich Schiller University Jena
D-07740 Jena, Germany
P.Braun@computer.org

François Thomasset
INRIA, Rocquencourt
78153 Le Chesnay Cedex, France
Francois.Thomasset@inria.fr

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 ILP. 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 address-based dependences, only. 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.*

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 [21]. 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 [17], 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]. In this paper, we describe an intraprocedural value-based data dependence analysis, (see Maslov [14] for details about address-based and value-based data dependences), implemented in the context of the SALTO tool [19]. 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* [12], but their approach only determines address-based dependences between register accesses and assumes memory to be one cell.

When analyzing data dependences in assembly code we must distinguish between accesses to registers and those to memory. In both cases we derive data dependence from reaching definitions and reaching uses information that we obtain by a monotone data flow analysis. Register analysis makes no complications: 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* [22]: whether two memory references access the same location. See Landi and Ryder [11] for more details on aliasing.

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 this, 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 [23].

Experimental results indicate that in many cases our method can be more accurate in the determination of data dependences than other previous methods.

2. Programming Model and Assumptions

In the following we assume a RISC instruction set. Memory is only accessed through load (`ld`) and store (`st`) instructions. 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. 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. Control flow is modeled using unconditional (`b`) or conditional (`b`*cc*) branch instructions. 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 can obtain information about the frame pointer. In our approach we do not make such assumptions.

3. Alias Analysis of Assembly Code

In this section we briefly review techniques for alias analysis of memory references. Doing *no alias analysis* leads to the assumption that a store instruction is always dependent on a load or store 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 have to determine that $S3$ is data dependent on $S2$. 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 secondly 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 byte memory blocks starting at $\%\texttt{o1} - 4$ resp. $\%\texttt{fp} - 20$.

4. Symbolic Value Set Propagation

In this section we present an extension of the well-known constant propagation algorithm [23]. Our target is the determination of possible *symbolic value sets* (contents) for each register and each program statement. In a subsequent step of the analysis this information will be used for the determination of data dependences between storage memory accesses—meaning store and load instructions. The calculation of symbolic value sets is performed by a *data flow analysis* [10]. Therefore, we have to model our problem as a *data flow framework* $(L, \vee, F)$, where L is called the *data flow information set*, $\vee$ is the *union operator*, and F is the set of *semantic functions*. If the semantic functions are monotone and $(L, \vee)$ forms a bounded semi-lattice with a one element and a zero element, we can use a general iterative algorithm [10] that always terminates and yields the least fix-point of the data flow system.

4.1. Data Flow Information Set

Our method describes the content of a register in the form of symbolic values. Therefore, we have to define the initialization points of program *P*. A statement *j* is called an *initialization point* $R_{i,j}$ of *P* if *j* is a load instruction that defines the content of r_i, a call node, or an entry node of a procedure. The finite set of all initialization points of *P* is given by *init*(P). The finite set SV of all symbolic values consists of the symbol $\perp$ and all proper symbolic values that are polynomials:

$$SV = \{\perp\} \cup \left\{ \sum_{i,j} a_{i,j} \cdot R_{i,j} + c : R_{i,j} \in \mathit{init(P)}, a_{i,j}, c \in \mathbb{Z} \right\}.$$

A variable $R_{i,j}$ of a symbolic value represents the value that is stored in register r_i at initialization point $R_{i,j}$. We use the value $\perp$ when we cannot make any assumptions on the content of a register.

As we are performing a static analysis, we are not able to infer the direction of branches taken during program execution. Therefore, it could happen that for a register r_i, more than one symbolic value is valid at a specific program point. As a consequence, we must then describe possible register contents by the so–called k-bounded symbolic value sets. The limitation of the sets is to ensure the termination of the analysis. Let $k \in \mathbb{N}$ be arbitrary, but fixed. Then a *k-bounded symbolic value set* is a set

$$A \in \mathbf{SV_k} = \{X : X \subseteq (SV \setminus \{\perp\}) \wedge |X| \leq k\} \cup \{\perp\}.$$

```
1: ld [%fp-4],%o1
2: st %o2,[%fp-8]
```
(a)

```
1: ld [%fp-4],%o1
2: sethi %hi(.LLC0),%o2
3: st %o3,[%o2+%lo(.LLC0)]
```
(b)

```
1: add %fp,-20,%o1
2: st %o2,[%o1-4]
3: ld [%fp-20],%o3
```
(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.

In the following let *REGS* stand for the set of all registers. We call a total map $\alpha : REGS \rightarrow \mathbf{SV_k}$ a *state*. By this means the data flow information set we use for the calculation of symbolic value sets is given by the set of possible states *SVS*.

4.2. Union Operator

If a node in a control flow graph has more than one predecessor, we must integrate all information stemming from these predecessors. In data flow frameworks, joining paths in the flow graph is implemented by the union operator. Let $\alpha, \beta \in$ *SVS*, then the union operator $\vee$ of our data flow problem is defined as shown in Fig. 2. The union operator $\vee$ is a simple componentwise union of sets. Additionally, to ensure the well-definition of the operator, we map arising sets with cardinality greater than k to the special value $\perp$. We have proven, that for a fixed $k \in \mathbb{N}$, the set of states *SVS* in conjunction with this union operator constitutes a bounded semi-lattice with a one element and a zero element.

4.3. Semantic Functions

In the control flow graph chosen for the analysis, each node stands for a uniquely labeled program statement. Therefore we can unambiguously assign a *semantic function* to each of the nodes; this semantic function will be used to update the symbolic value sets assigned to each register. In Fig. 3 we specify some semantic functions used by our method. In this specification, α stands for a state before the execution of the semantic function, and α' for the corresponding state after the execution of the semantic function.

After the execution of an initialization point $R_{i,j}$, we have no knowledge about the defined value of register r_i. The main idea of our method is to describe the register content of r_i after such a definition as a symbolic value. As mentioned before, entry nodes of a procedure as well as load instructions are initialization points. The semantic function of an entry node *n* initializes the symbolic value set for each register r_i with its corresponding initialization point $R_{i,n}$. By doing so, after the execution of *n*, the symbolic value $R_{i,n}$ stands for the value which is stored in r_i before the execution of any procedure code.

The semantic function assigned to a load instruction initializes the symbolic value set of the register, whose value will be defined by the operation, similar to the description above of the corresponding initialization point. As opposed to entry nodes, such an initialization is only valid if the initialization point is safe. We call an initialization point *safe* if the corresponding statement is not part of a loop. In contrast, an initialization point inside a loop is called *unsafe*. The problem with unsafe initialization points is that the value of the affected register may change at each loop iteration. Therefore, we cannot make a safe assumption about its initialization value. To obtain a safe approximation in such a case the symbolic value set of the register is set to the special value $\perp$. In Fig. 3 we use the operator $\oplus$ which is an extension of the add operator for polynomials. The result of an application $A \oplus B, A, B \in \mathbf{SV_k}$, is a pairwise addition of the terms of A and B. To ensure the well-definition of the operator, resulting sets with cardinality greater than k will be mapped to $\perp$. Further, if one of the operands has the value $\perp$, the operator returns $\perp$. As we have proven that the semantic functions are monotone, the general iterative algorithm [10] can be used to solve our data flow problem.

5. Improvement of Value Set Propagation

Without limiting the cardinality of symbolic value sets our propagation algorithm may lead to infinite sets. Registers whose contents could change at each loop iteration are responsible for this phenomenon. The calculated symbolic value set for these registers comprises only the special value $\perp$. Such an inaccuracy in the analysis cannot be accepted in practice. Therefore, we propose an improvement of the symbolic value set propagation algorithm by using NSV registers.

5.1. NSV Registers

In this section we introduce the concept of non-symbolic value registers, hereafter called NSV registers. A *NSV register* of a loop G' is a register r_i used in G', which content

$\forall\, \alpha, \beta \in \mathit{SVS}, r_i \in \mathit{REGS}:$

$$(\alpha \vee \beta)(r_i) = \begin{cases} \alpha(r_i) \cup \beta(r_i) & if \quad \alpha(r_i), \beta(r_i) \in \mathbf{SV_k} \setminus \{\perp\} \\ & \quad\quad \wedge \mid \alpha(r_i) \cup \beta(r_i) \mid \leq k \\ \perp & otherwise \end{cases}$$

Figure 2. The union operator for symbolic value sets.

`n:  entry`

$\forall\, r_k \in \mathit{REGS} : \alpha'(r_k) := \{R_{k,n}\}$

`n:  mv a,%rj` *(copy $a \in \mathbb{Z}$ into register r_j)*

1. $\forall\, r_k \in \mathit{REGS} \setminus \{r_j\} : \alpha'(r_k) := \alpha(r_k)$
2. $\alpha'(r_j) := \{a\}$

`n:  add %ri,%rj,%rm` *(add value of r_i and r_j and store the result in r_m)*

1. $\forall\, r_k \in \mathit{REGS} \setminus \{r_m\} : \alpha'(r_k) := \alpha(r_k)$
2. $\alpha'(r_m) := \alpha(r_i) \oplus \alpha(r_j)$

`n:  ld [mem],%rj` *(load value from address mem into register r_j)*

1. $\forall\, r_k \in \mathit{REGS} \setminus \{r_j\} : \alpha'(r_k) := \alpha(r_k)$
2. $\alpha'(r_j) := \begin{cases} \{R_{j,n}\} & if \quad R_{j,n} \text{ is a safe initialization point} \\ \perp & otherwise \end{cases}$

Figure 3. Semantic functions for some instructions.

can change. The modified propagation algorithm works as follows:

1. First, we have to determine the NSV registers for all loops of the program. The sets of NSV registers contain among other things induction registers and registers which will be defined by a load instruction in G'.

2. Thereafter, for each NSV register r_i we insert additional nodes into the control flow graph. At the beginning of the loop body we attach a statement n' *: init* r_i, where n' is a unique and unused statement number. At the end of the loop body, and before each node of the control flow graph that can be reached after execution of the loop we insert a statement n' *: setbot* r_i.

3. After this, we perform the symbolic value set propagation on the modified control flow graph. Further, for the inserted nodes we have defined semantic functions which set the symbolic value set of r_i to the initialization point $R_{i,n'}$ (init) resp. to $\perp$ (setbot). Now we consider every initialization point as safe.

The improved version of our algorithm 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 NSV registers.

5.2. Determination of NSV Registers

In the following let G' be a loop and S a statement inside of 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]:

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 NSV registers of a loop G' in a simple way. For this, a register r_i is a NSV register in G' iff r_i is defined by a statement in G' that is not a loop invariant statement in G'.

Fig. 4 shows the results of an improved symbolic value set propagation for a simple program. The NSV registers of the loop are `%r1`, `%r2`, `%r3`, and `%r4`. For each NSV register an `init` instruction resp. `setbot` instructions is inserted into the program. As a consequence, the data flow algorithm terminates after the third iteration. The concept of NSV registers allows a more accurate analysis of memory references inside the loop. Without NSV registers the value of register `%r1` would have been set to $\perp$ eventually. In contrast, an improved symbolic value propagation always leads to proper values.

6. Data Dependence Analysis

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, for a specific statement and memory location, all statements where the value of this 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 true dependence between them, a def-def pair an output dependence, and an use-def pair an anti-dependence. For scalar variables the determination of reaching definitions can be performed by a well-known standard algorithm described in [1]. To use this algorithm for data dependence analysis of assembly code 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 refer always to the same storage object.

To achieve all this information we need a mechanism which checks whether the index expressions of two storage accesses *X* and *Y* could represent the same value. We solve this problem by applying a modified GCD test [23]. Therefore, 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.

For an example, we refer to Fig. 4. 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. This means, there is no loop-independent data dependence between these two instructions.

When the instructions are executed in different loop iterations, $R_{1,12}$ may have different values. The modified GCD test shows that both instructions may reference the same memory location. Therefore, we have to assume a loop-carried data dependence between instructions 5 and 8.

7. Implementation and Results

The method for determining of 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. 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:

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 be data dependent. 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 [19], but SALTO does not 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

		1. iteration	2. iteration
1	`mov 1,%r2`		
2	`.LL11:`		
12	`init %r1`	$\%\mathtt{r2} = \{1\}$	$\%\mathtt{r1} = \bot, \%\mathtt{r2} = \bot, \%\mathtt{r3} = \bot, \%\mathtt{r4} = \bot$
13	`init %r2`	$\%\mathtt{r1} = \{R_{1,12}\}, \%\mathtt{r2} = \{1\}$	$\%\mathtt{r1} = \{R_{1,12}\}, \%\mathtt{r2} = \bot$ $\%\mathtt{r3} = \bot, \%\mathtt{r4} = \bot$
14	`init %r3`	$\%\mathtt{r1} = \{R_{1,12}\}, \%\mathtt{r2} = \{R_{2,13}\}$	$\%\mathtt{r1} = \{R_{1,12}\}, \%\mathtt{r2} = \{R_{2,13}\}$ $\%\mathtt{r3} = \bot, \%\mathtt{r4} = \bot$
15	`init %r4`	$\%\mathtt{r1} = \{R_{1,12}\}, \%\mathtt{r2} = \{R_{2,13}\}$ $\%\mathtt{r3} = \{R_{3,14}\}$	$\%\mathtt{r1} = \{R_{1,12}\}, \%\mathtt{r2} = \{R_{2,13}\}$ $\%\mathtt{r3} = \{R_{3,14}\}, \%\mathtt{r4} = \bot$
3	`ld [%r1-40],%r3`	$\%\mathtt{r1} = \{R_{1,12}\}, \%\mathtt{r2} = \{R_{2,13}\}$ $\%\mathtt{r3} = \{R_{3,14}\}, \%\mathtt{r4} = \{R_{4,15}\}$	$\%\mathtt{r1} = \{R_{1,12}\}, \%\mathtt{r2} = \{R_{2,13}\}$ $\%\mathtt{r3} = \{R_{3,14}\}, \%\mathtt{r4} = \{R_{4,15}\}$
4	`add %r2,1,%r2`	$\%\mathtt{r1} = \{R_{1,12}\}, \%\mathtt{r2} = \{R_{2,13}\}$ $\%\mathtt{r3} = \{R_{3,3}\}, \%\mathtt{r4} = \{R_{4,15}\}$	$\%\mathtt{r1} = \{R_{1,12}\}, \%\mathtt{r2} = \{R_{2,13}\}$ $\%\mathtt{r3} = \{R_{3,3}\}, \%\mathtt{r4} = \{R_{4,15}\}$
5	`ld [%r1-80],%r4`	$\%\mathtt{r1} = \{R_{1,12}\}, \%\mathtt{r3} = \{R_{3,3}\}$ $\%\mathtt{r2} = \{(R_{2,13} + 1)\}, \%\mathtt{r4} = \{R_{4,15}\}$	$\%\mathtt{r1} = \{R_{1,12}\}, \%\mathtt{r3} = \{R_{3,3}\}$ $\%\mathtt{r2} = \{(R_{2,13} + 1)\}, \%\mathtt{r4} = \{R_{4,15}\}$
6	`cmp %r2,9`	$\%\mathtt{r1} = \{R_{1,12}\}, \%\mathtt{r3} = \{R_{3,3}\}$ $\%\mathtt{r2} = \{(R_{2,13} + 1)\}, \%\mathtt{r4} = \{R_{4,5}\}$	$\%\mathtt{r1} = \{R_{1,12}\}, \%\mathtt{r3} = \{R_{3,3}\}$ $\%\mathtt{r2} = \{(R_{2,13} + 1)\}, \%\mathtt{r4} = \{R_{4,5}\}$
7	`add %r3,%r4,%r3`	$\%\mathtt{r1} = \{R_{1,12}\}, \%\mathtt{r3} = \{R_{3,3}\}$ $\%\mathtt{r2} = \{(R_{2,13} + 1)\}, \%\mathtt{r4} = \{R_{4,5}\}$	$\%\mathtt{r1} = \{R_{1,12}\}, \%\mathtt{r3} = \{R_{3,3}\}$ $\%\mathtt{r2} = \{(R_{2,13} + 1)\}, \%\mathtt{r4} = \{R_{4,5}\}$
8	`st %r3,[%r1-40]`	$\%\mathtt{r1} = \{R_{1,12}\}, \%\mathtt{r4} = \{R_{4,5}\}$ $\%\mathtt{r2} = \{(R_{2,13} + 1)\}$ $\%\mathtt{r3} = \{(R_{3,3} + R_{4,5})\}$	$\%\mathtt{r1} = \{R_{1,12}\}, \%\mathtt{r4} = \{R_{4,5}\}$ $\%\mathtt{r2} = \{(R_{2,13} + 1)\}$ $\%\mathtt{r3} = \{(R_{3,3} + R_{4,5})\}$
9	`add %r1,4,%r1`	$\%\mathtt{r1} = \{R_{1,12}\}, \%\mathtt{r4} = \{R_{4,5}\}$ $\%\mathtt{r2} = \{(R_{2,13} + 1)\}$ $\%\mathtt{r3} = \{(R_{3,3} + R_{4,5})\}$	$\%\mathtt{r1} = \{R_{1,12}\}, \%\mathtt{r4} = \{R_{4,5}\}$ $\%\mathtt{r2} = \{(R_{2,13} + 1)\}$ $\%\mathtt{r3} = \{(R_{3,3} + R_{4,5})\}$
16	`setbot %r1`	$\%\mathtt{r1} = \{(R_{1,12} + 4)\}, \%\mathtt{r4} = \{R_{4,5}\}$ $\%\mathtt{r2} = \{(R_{2,13} + 1)\}$ $\%\mathtt{r3} = \{(R_{3,3} + R_{4,5})\}$	$\%\mathtt{r1} = \{(R_{1,12} + 4)\}, \%\mathtt{r4} = \{R_{4,5}\}$ $\%\mathtt{r2} = \{(R_{2,13} + 1)\}$ $\%\mathtt{r3} = \{(R_{3,3} + R_{4,5})\}$
17	`setbot %r2`	$\%\mathtt{r1} = \bot, \%\mathtt{r4} = \{R_{4,5}\}$ $\%\mathtt{r2} = \{(R_{2,13} + 1)\}$ $\%\mathtt{r3} = \{(R_{3,3} + R_{4,5})\}$	$\%\mathtt{r1} = \bot, \%\mathtt{r4} = \{R_{4,5}\}$ $\%\mathtt{r2} = \{(R_{2,13} + 1)\}$ $\%\mathtt{r3} = \{(R_{3,3} + R_{4,5})\}$
18	`setbot %r3`	$\%\mathtt{r1} = \bot, \%\mathtt{r2} = \bot, \%\mathtt{r4} = \{R_{4,5}\}$ $\%\mathtt{r3} = \{(R_{3,3} + R_{4,5})\}$	$\%\mathtt{r1} = \bot, \%\mathtt{r2} = \bot, \%\mathtt{r4} = \{R_{4,5}\}$ $\%\mathtt{r3} = \{(R_{3,3} + R_{4,5})\}$
19	`setbot %r4`	$\%\mathtt{r1} = \bot, \%\mathtt{r3} = \bot, \%\mathtt{r4} = \{R_{4,5}\}$ $\%\mathtt{r2} = \bot$	$\%\mathtt{r1} = \bot, \%\mathtt{r3} = \bot, \%\mathtt{r4} = \{R_{4,5}\}$ $\%\mathtt{r2} = \bot$
10	`ble .LL11`	$\%\mathtt{r1} = \bot, \%\mathtt{r3} = \bot, \%\mathtt{r4} = \bot$ $\%\mathtt{r2} = \bot$	$\%\mathtt{r1} = \bot, \%\mathtt{r3} = \bot, \%\mathtt{r4} = \bot$ $\%\mathtt{r2} = \bot$
20	`setbot %r1`	$\%\mathtt{r1} = \bot, \%\mathtt{r2} = \bot$ $\%\mathtt{r3} = \bot, \%\mathtt{r4} = \bot$	$\%\mathtt{r1} = \bot, \%\mathtt{r2} = \bot$ $\%\mathtt{r3} = \bot, \%\mathtt{r4} = \bot$
17	`setbot %r2`	$\%\mathtt{r1} = \bot, \%\mathtt{r2} = \bot$ $\%\mathtt{r3} = \bot, \%\mathtt{r4} = \bot$	$\%\mathtt{r1} = \bot, \%\mathtt{r2} = \bot$ $\%\mathtt{r3} = \bot, \%\mathtt{r4} = \bot$
21	`setbot %r3`	$\%\mathtt{r1} = \bot, \%\mathtt{r2} = \bot$ $\%\mathtt{r3} = \bot, \%\mathtt{r4} = \bot$	$\%\mathtt{r1} = \bot, \%\mathtt{r2} = \bot$ $\%\mathtt{r3} = \bot, \%\mathtt{r4} = \bot$
22	`setbot %r4`	$\%\mathtt{r1} = \bot, \%\mathtt{r2} = \bot$ $\%\mathtt{r3} = \bot, \%\mathtt{r4} = \bot$	$\%\mathtt{r1} = \bot, \%\mathtt{r2} = \bot$ $\%\mathtt{r3} = \bot, \%\mathtt{r4} = \bot$
11	`retl`	$\%\mathtt{r1} = \bot, \%\mathtt{r2} = \bot$ $\%\mathtt{r3} = \bot, \%\mathtt{r4} = \bot$	$\%\mathtt{r1} = \bot, \%\mathtt{r2} = \bot$ $\%\mathtt{r3} = \bot, \%\mathtt{r4} = \bot$

Figure 4. Symbolic value set propagation. Registers r_i that are not mentioned have the value $\{R_{i,0}\}$.

one in `gcc` [21] or the one used by Larus et. al. [20]. Systems that do some kind of value propagation, but only determine address-based dependences, are classified in level 3. In section 8 we will have a closer look at other techniques for value propagation. Our method is classified in level 4. As yet, we know of no other method which also determines value-based dependences. The table only contains those 39 procedures in which an improvement, i.e., less dependences, was noticeable from level 3 to level 4. Fig. 5 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. Fig. 5 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.

8. Related Work

So far, only some 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. [13], Böckle [4], and Ebcioğlu et. al. [15]. The approach presented by Ebcioğlu is implemented in the Chameleon compiler [16] 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

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%

Figure 5. 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.

technique as in the Bulldog compiler is used. If a register has multiple definitions, the algorithm described in [15] 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, 23]. 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.

9. 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 register and 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 [18] 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. As a first step, we could make assumptions about the use of global memory locations, and we could derive exact dependences.

Acknowledgments

We thank the referees for their comments which helped in improving this paper.

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

[11] 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.

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

[13] 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.

[14] V. Maslow. 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.

[15] 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.

[16] 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.

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

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

[19] 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.

[20] 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.

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

[22] D. W. Wall. Limits of instruction-level parallelism. Technical Report 93/6, Digital Western Research Laboratory, 250 University Avenue, Palo Alto, California, 1993.

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