A Graph–Free Approach to Data–Flow Analysis

Markus Mohnen

Lehrstuhl für Informatik II, RWTH Aachen, Germany
mohnen@informatik.rwth-aachen.de

Abstract. For decades, data–flow analysis (DFA) has been done using an iterative algorithm based on graph representations of programs. For a given data–flow problem, this algorithm computes the maximum fixed point (MFP) solution. The edge structure of the graph represents possible control flows in the program. In this paper, we present a new, graph–free algorithm for computing the MFP solution. The experimental implementation of the algorithm was applied to a large set of samples. The experiments clearly show that the memory usage of our algorithm is much better: Our algorithm always reduces the amount of memory and reached improvements upto less than a tenth. In the average case, the reduction is about a third of the memory usage of the classical algorithm. In addition, the experiments showed that the runtimes are almost the same: The average speedup of the classical algorithm is only marginally greater than one.

1 Introduction

Optimising compilers perform various static program analyses to obtain informations needed to apply optimisations. In the context of imperative languages, the technique commonly used is data–flow analysis (DFA). It provides information about properties of the states that may occur at a given program point during execution. Here, programs considered are intermediate code, e.g. three address code, register code, or Java Virtual Machine (JVM) code [LY97].

For decades, the *de facto* classical algorithm for DFA has been an iterative algorithm [MJ81, ASU86, Muc97] which uses a graph as essential data structure. The graph is extracted from the program, making explicit the possible control flows in the program as the edge structure of the graph. Typically, the nodes of the graph are basic blocks (BB), i.e. maximal sequences of straight–line code (but see also [KKS98] for comments on the adequacy of this choice). A distinct root node of the graph corresponds to the entry point of the program.

For a given graph and a given initial annotation of the root node, the algorithm computes an annotation for each of the nodes. Each annotation captures the information about the state of the execution at the corresponding program point. The exact relation between annotations and states depends on the data–flow problem. However, independently of the exact relation, the annotations computed by the algorithm are guaranteed to be the greatest solution of the consistency equations imposed by the data–flow problem. This result is known as the maximal fixed point (MFP) solution.

R. N. Horspool (Ed.): CC 2002, LNCS 2304, pp. 46–61, 2002.

In the context of BB graphs, there is a need for an additional post–processing of the annotations. Since each BB represents a sequence of instructions, the annotation for a single BB must be propagated to the instruction level. As a result of this post–processing, each program instruction is annotated.

The contribution of this paper is an alternative algorithm for computing the MFP solution. In contrast to the classical algorithm, our approach is *graph–free*: Besides a working set, it does not need any additional data structures (of course, the graph structure is always there *implicitly* in the program). The key idea is to give the program a more active role: While the classical approach transforms the program to a passive data object on which the solver operates, our point of view is that the program itself executes on the annotation.

An obvious advantage of this approach is the reduced memory usage. In addition, it is handy if there is already machinery for execution of programs available. Consequently, our execution–based approach is advantageous in settings where optimisations are done immediately before execution of the code. Here it saves effort to implement the analyses and it saves valuable memory for the execution. The most prominent example of such a setting is the Java Virtual Machine (JVM) [LY97]. In fact, the JVM specification *requires* that each class file is verified at linking time by a data–flow analyser. The purpose of this verification is to ensure that the code is well–typed and that no operand stack overflows or underflows occur at runtime.

In addition, certain optimisations cannot be done by the Java compiler producing JVM code. For instance, optimisation w.r.t. memory allocation like compile–time garbage collection (CTGC) can only be done in the JVM since the JVM code does not provide facilities to influence the memory allocation. CTGC was originally proposed in the context of functional languages [Deu97, Moh97] and then adopted for Java [Bla98, Bla99].

To validate the benefits of our approach, we studied the performance of the new algorithm in competition with the classical one, both in terms of memory usage and runtime. Therefore, we applied both to a large set of samples. The experiments clearly show that the memory usage of our algorithm is much better: Our algorithm always reduces the amount of memory and reached improvements upto less than a tenth. In the average case, the reduction is about a third of the memory usage of the classical algorithm. Moreover, the runtimes are comparable in the average case: Using the classical algorithm does not give a substantial speedup.

Structure of this article. We start by defining some basic notions. In Section 3 the classical, iterative algorithm for computing the MFP solution is discussed briefly. Our main contribution starts with Section 4 where we present the new execution algorithm, discuss its relation to the classical algorithm, and prove the termination and correctness. Experimental results presented in Section 5 give an estimation of the benefits our method. Finally, Section 6 concludes the paper.

2 Notations

In this section, we briefly introduce the notations that we use in the rest of the paper. Although we focus on abstract interpretation based DFA, our results are applicable to other DFAs as well.

The programs we consider are three–address code programs, i.e. non–empty sequences of instructions $I \in \mathrm{Instr}$. Each instruction I is either a jump, which can be conditional ($\mathsf{if}\ \psi\ \mathsf{goto\ n}$) or unconditional ($\mathsf{goto\ n}$), or an assignment ($\mathsf{x{:=}y{\circ}z}$). In assignments, x must be a variable, and y and z can be variables or constants. Since we consider intraprocedural DFA only, we do not need instructions for procedure calls or exits. The major point of this setting is to distinguish between instructions which cause the control flow to branch and those which keep the control flow linear. Hence, the exact structure is not important. Any other intermediate code, like the JVM code, is suitable as well.

To model program properties, we use lattices $L = \langle A, \sqcap, \sqcup \rangle$ where A is a set, and $\sqcap$ and $\sqcup$ are binary meet and join operations on A. Furthermore, $\bot$ and $\top$ are least and greatest element of the lattice. Often, finite lattices are used, but in general it suffices to consider lattices which have only finite chains.

The point of view of DFA based on abstract interpretation [CC77, AH87] is to replace the standard semantics of programs by an abstract semantics describing how the instructions operate on the abstract values A. Formally, we assume a monotone semantic functional $![.!] : \mathrm{Instr} \to (A \to A)$ which assigns a function on A to each instruction.

A data–flow problem is a quadruple $(P, L, ![.!], a_0)$ where $P = I_0 \ldots I_n \in \mathrm{Instr}^+$ is a program, L is a lattice, $![.!]$ is an abstract semantics, and $a_0 \in A$ is an initial value for the entry of P.

To define the MFP solution of a data–flow problem, we first introduce the notion of predecessors. For a given program $P = I_0 \ldots I_n \in \mathrm{Instr}^+$, we define the function $\mathrm{pred}_P : \{0, \ldots, n\} \to \mathfrak{P}(\{0, \ldots, n\})$ in the following way: $j \in \mathrm{pred}_P(i)$ iff either $I_j \in \{\mathsf{goto\ i}, \mathsf{if}\ \psi\ \mathsf{goto\ i}\}$, or $i = j + 1$ and $I_j \neq \mathsf{goto\ t}$ for some t. Intuitively, the predecessors of an instruction are all instructions which may be executed immediately before it.

The MFP solution is a vector of values $s_0, \ldots, s_n \in A$. Each entry s_i is the abstract value valid immediately before the instruction I_i. It is defined as the greatest solution of the equation system $s_i = \prod_{j \in \mathrm{pred}_P(i)} ![I_j!](s_j)$. The well–known fixed point theorem by Tarski guarantees the existence of the MFP solution in this setting.

Example 1 (Constant Folding Propagation). We now introduce an example, which we use as a running example in the rest of the paper. Constant folding and propagation aims at finding as many constants as possible at compile time, and replacing the computations with the constant values. In the setting described above, we associate with each variable and each program point the information if the variable is always constant at this point. For simplicity, we assume that the program only uses the arithmetic operations on integers. We define a set $C := \mathbb{Z} \uplus \{\top, \bot\}$ and a relation $c_1 \leq c_2$ iff (a) $c_1 = c_2$, (b) $c_1 = \bot$, or

(c) $c_2 = \top$. Intuitively, values can be interpreted in the following way: An integer means "constant value", $\top$ means "not constant due to missing information", and $\bot$ means "not constant due to conflict". The relation $\leq$ induces meet and join operations. Hence, $\langle C, \sqcap, \sqcup \rangle$ is a (non–finite) lattice with only finite chains. Fig. 1 shows the corresponding Hasse diagram.

The abstract lattice is defined in terms of this lattice. Formally, let X be the set of variables of a program P. By definition, X is finite. We define the set of abstract values as $\mathfrak{C} := X \to C$, the set of all functions mapping a variable to a value in C. Since X is finite, $\mathfrak{C}$ is finite as well. We obtain meet and join operations $\sqcap_{\mathfrak{C}}, \sqcup_{\mathfrak{C}}$ in the canonical way by argument–wise use of the corresponding operation on C. Hence, our lattice for this abstract interpretation is $\langle \mathfrak{C}, \sqcap_{\mathfrak{C}}, \sqcup_{\mathfrak{C}} \rangle$.

The abstract semantics $[\![.]\!]_{\mathfrak{C}}$: Instr $\to (\mathfrak{C} \to \mathfrak{C})$ is defined in the following way: For jumps, we define $[\![\mathsf{goto\ l}]\!]_{\mathfrak{C}}$ and $[\![\mathsf{if}\ \psi\ \mathsf{goto\ l}]\!]_{\mathfrak{C}}$ to be the identity, since jumps do not change any variable. For assignments, we define $[\![\mathsf{x{:=}y{\circ}z}]\!]_{\mathfrak{C}} := c \mapsto c'$, where $c' = c[\mathsf{x}/a]$, i.e. c' is the same function as c except at argument x. The new value is defined as

$$c'(\mathsf{x}) = a := \begin{cases} a_{\mathsf{y}} \circ a_{\mathsf{z}} & \text{if } \mathsf{y} = a_{\mathsf{y}} \in \mathbb{Z} \text{ or } c(\mathsf{y}) = a_{\mathsf{y}} \in \mathbb{Z} \\ & \text{and } \mathsf{z} = a_{\mathsf{z}} \in \mathbb{Z} \text{ or } c(\mathsf{z}) = a_{\mathsf{z}} \in \mathbb{Z} \\ \bot & \text{otherwise} \end{cases}$$

Intuitively, the value of the variable on the left–hand side is constant iff all operands are either constants in the code or known to be constants during execution.

For a data–flow problem, the initial value will be $a_0 = \bot$: At the entry, no variable can be constant. Fig. 1 shows an example for a program, the associated abstractions, the equation system, and the MFP solution. This example also demonstrates why it is necessary to use the infinite lattice C: The solution contains the constant '5' which is not found in the program.

Our presentation of these notions differs slightly from the presentation found in text books. Typically, data–flow problems are already formulated using an explicit graph structure. However, we want to point out that this is not a necessity. Furthermore, it allows us to formulate and prove the correctness of our algorithm without reference to the classical one.

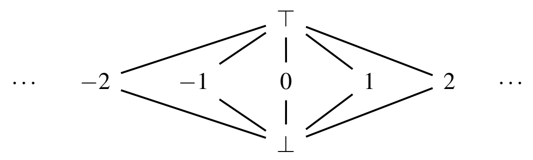


Fig. 1. Hasse diagram of $\langle C, \sqcap, \sqcup \rangle$

Program	Abstractions[a]	Equations	Solution
$I_0 = \mathsf{x} := 1$	$\mathsf{x}/1$	$s_0 = a_0$	$\mathsf{x}/\bot\ \mathsf{y}/\bot\ \mathsf{z}/\bot\ \mathsf{r}/\bot$
$I_1 = \mathsf{y} := 2$	$\mathsf{y}/2$	$s_1 =![I_0!]_{\mathfrak{c}}(s_0)$	$\mathsf{x}/1\ \mathsf{y}/\bot\ \mathsf{z}/\bot\ \mathsf{r}/\bot$
$I_2 = \mathsf{z} := 3$	$\mathsf{z}/3$	$s_2 =![I_1!]_{\mathfrak{c}}(s_1)$	$\mathsf{x}/1\ \mathsf{y}/2\ \mathsf{z}/\bot\ \mathsf{r}/\bot$
$I_3 = \mathsf{goto}\ 8$	(identity)	$s_3 =![I_2!]_{\mathfrak{c}}(s_2)$	$\mathsf{x}/1\ \mathsf{y}/2\ \mathsf{z}/3\ \mathsf{r}/\bot$
$I_4 = \mathsf{r} := \mathsf{y} + \mathsf{z}$	$\mathsf{r}/\begin{cases} c(\mathsf{y})+c(\mathsf{z}) & c(\mathsf{y}), c(\mathsf{z}) \in \mathbb{Z} \\ \bot & \text{otherwise} \end{cases}$	$s_4 =![I_8!]_{\mathfrak{c}}(s_8)$	$\mathsf{x}/\bot\ \mathsf{y}/2\ \mathsf{z}/3\ \mathsf{r}/\bot$
$I_5 = \mathsf{if}\ \mathsf{x} \leq \mathsf{z}\ \mathsf{goto}\ 7$	(identity)	$s_5 =![I_4!]_{\mathfrak{c}}(s_4)$	$\mathsf{x}/\bot\ \mathsf{y}/2\ \mathsf{z}/3\ \mathsf{r}/5$
$I_6 = \mathsf{r} := \mathsf{z} + \mathsf{y}$	$\mathsf{r}/\begin{cases} c(\mathsf{z})+c(\mathsf{y}) & c(\mathsf{y}), c(\mathsf{z}) \in \mathbb{Z} \\ \bot & \text{otherwise} \end{cases}$	$s_6 =![I_5!]_{\mathfrak{c}}(s_5)$	$\mathsf{x}/\bot\ \mathsf{y}/2\ \mathsf{z}/3\ \mathsf{r}/5$
$I_7 = \mathsf{x} := \mathsf{x} + 1$	$\mathsf{x}/\begin{cases} c(\mathsf{x})+1 & c(\mathsf{x}) \in \mathbb{Z} \\ \bot & \text{otherwise} \end{cases}$	$s_7 =![I_5!]_{\mathfrak{c}}(s_5) \sqcap ![I_6!]_{\mathfrak{c}}(s_6)$	$\mathsf{x}/\bot\ \mathsf{y}/2\ \mathsf{z}/3\ \mathsf{r}/5$
$I_8 = \mathsf{if}\ \mathsf{x} < 10\ \mathsf{goto}\ 4$	(identity)	$s_8 =![I_3!]_{\mathfrak{c}}(s_3) \sqcap ![I_7!]_{\mathfrak{c}}(s_7)$	$\mathsf{x}/\bot\ \mathsf{y}/2\ \mathsf{z}/3\ \mathsf{r}/5$

[a] For each abstraction only the modification x/y as abbreviation for $c \mapsto c[x/y]$ is given.

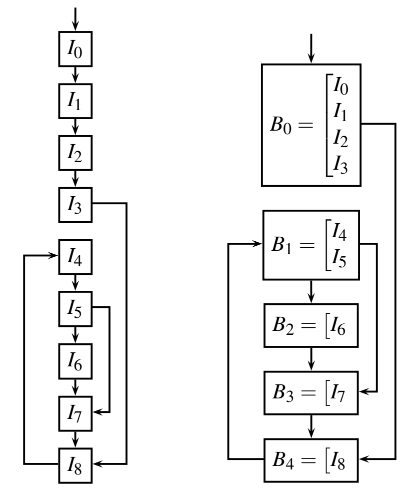
Fig. 2. Example for data–flow problem

The approach described so far can be generalised in two dimensions: Firstly, changing $\sqcap$ to $\sqcup$ results in *existential* data–flow problems, in contrast to *universal* data–flow problems: The intuition is that a property holds at a point if there is a single path starting at the point such that the property holds on this path. For existential data–flow problems, the least fixed point is computed instead of the greatest fixed point. Secondly, we can change predecessors pred_P to successors $\text{succ}_P : \{0, \ldots, n\} \to \mathfrak{P}(\{0, \ldots, n\})$ defined as $i \in \text{succ}_P(j) \iff j \in \text{pred}_P(i)$. The resulting class of data–flow problems are called *backward* problems (in contrast to *forward* problems), since the flow of information is opposite to the normal execution flow. Here, the abstract values are valid immediately after the corresponding instruction.

Altogether, the resulting taxonomy has four cases. However, the algorithms for all the cases have the same general structure. Therefore, we will consider only the forward and universal setting.

3 Classical Iterative Basic-Block Based Algorithm

This section reviews the classical, graph–based approach to DFA. To make the data–flow of program explicit, we define two types of flow graphs: single instruction (SI) graphs and basic block (BB) graphs. For a program $P = I_0 \ldots I_n$, we define the SI graph $\text{SIG}(P) := (\{I_0, \ldots, I_n\}, \{(I_j, I_i) \mid j \in \text{pred}_P(i)\}, I_0)$ with a node for each instruction, an edge from node I_j to node I_i iff j is predecessor of i, and root node I_0. Intuitively, the BB graph results from the SI graph by merging maximal sequences of straight–line code. Formally, we define the set of basic blocks as the unique partition of P: $\text{BB}(P) = \{B_0, \ldots, B_m\}$ iff (a) $B_j = I_{j_1} \ldots I_{j_n}$ with $j_{k+1} = j_k + 1$, (b) $\text{pred}_P(j_1) \neq \{(j-1)_n\}$ or $\text{succ}_P((j-1)_n) \neq \{j_1\}$, (c) $|\text{pred}_P(j_k)| = 1$ for $j_1 < j_k \leq j_n$, and (d) $I_{j_n+1} = I_{(j+1)_1}$, $I_{0_1} = I_0$, and $I_{m_n} = I_n$. The BB graph is defined as $\text{BBG}(P) := (\text{BB}(P), \{(B_j, B_i) \mid j_n \in \text{pred}_P(i_1)\}, B_0)$.



(a) SI graph (b) BB graph

Fig. 3. Examples for SI graph and BB graph

Example 2 (Constant Folding Propagation, Cont'd). In Fig. 3 we see the SI graph and the BB graph for the example program from the last section.

Obviously, for a given flow graph $G = (N, E, r)$, the usual notions of predecessors $\text{pred}_G : N \to \mathfrak{P}(N)$ and successors $\text{succ}_G : N \to \mathfrak{P}(N)$, defined as $n \in \text{pred}_G(n'), n' \in \text{succ}_G(n) :\iff (n', n) \in E$ coincide with the corresponding notions for programs.

For a given data–flow problem $(P, L, ![.!], a_0)$, an additional pre–processing step must be performed to extend the abstract semantics to basic blocks: We define $![.!] : Instr^+ \to (A \to A)$ as $![I_0 \dots I_n!] :=![I_n!] \circ \dots \circ ![I_0!]$.

The classical iterative algorithm for computing the MFP solution of a data–flow problem is shown in Fig. 4. In addition to the BB graph G it uses a working set W and an array a, which associates an abstract value with each node. The working set keeps all nodes which must be visited again. In each iteration a node is selected from the working set. At this level, we assume no specific strategy for the working set and consider this choice to be non–deterministic. By visiting all predecessors of this node, a new approximation is computed. If this approximation differs from the last approximation, the new one is used. In addition, all successors of the node are put in the working set. After termination of the main loop, the post–processing is done, which propagates the solution from the basic block level to the instruction level.

Example 3 (Constant Folding Propagation, Cont'd). For the example from the last section, Table 1 shows a trace of the execution of the algorithm. Each line shows the state of working set W, the selected node B, and the array $a[.]$ at

```
Input: Data–flow problem $(P, L, ![.!], a_0)$
          where $P = I_0 \dots I_n$, $L = \langle A, \sqcap, \sqcup \rangle$
Output: MFP solution $s_0, \dots, s_n \in A$
$G = (\mathrm{BB}(P), E, B_0) := \mathrm{BBG}(P)$
$a[B_0] := a_0$
for each $B \in \mathrm{BB}(P) - B_0$ do $a[B] := \top$
$W := \mathrm{BB}(P)$
while $W \neq \emptyset$ do
  choose $B \in W$
  $W := W - B$
  $new := a[B]$
  for each $B' \in \mathrm{pred}_G(B)$ do $new := new \sqcap ![B'!](a[B'])$
  if $new \neq a[B]$ then
     $a[B] := new$;
     for each $B' \in \mathrm{succ}_G(B)$ do $W := W + B'$
  end
end
for each $B \in \mathrm{BB}(P)$ do
  with $B = I_k \dots I_l$ do
    $s_k := a[B]$
    for $i := k + 1$ to $l$ do $s_i := ![I_{i-1}!](s_{i-1})$
  end
end
```

Fig. 4. Classical iterative algorithm for computing MFP solution

the end of the main loop. To keep the example brief, we omitted all cells which did not change w.r.t. the previous line and we have chosen the best selection of nodes. The resulting MFP solution is identical to the one in Fig. 1, of course.

In an implementation, the non–deterministic structure of the working set must be implemented in a deterministic way. However, both the classical algorithm described above and the new algorithm, which we describe in the next section, based on the concept of working sets. Therefore, we continue to assume that the working set is non–deterministic.

4 New Execution Based Algorithm

The new algorithm for computing the MFP solution (see Fig. 5) of a given data–flow problem is graph–free. The underlying idea is to give the program a more active role: The program itself executes on the abstract values. The program counter variable *pc* always holds the currently executing instruction. The execution of this instruction affects the abstract values for all succeeding instructions and it is propagated iff it makes a change. Here we see another difference w.r.t. the classical algorithm: While the *pc* in our algorithm identifies the instruction *causing* a change, the current node *n* in the classical algorithm

Table 1. Example Execution of classical iterative algorithm

W	B	$a[B_0]$	$a[B_1]$	$a[B_2]$	$a[B_3]$	$a[B_4]$
$\{B_1, B_2, B_3, B_4\}$	B_0	x/⊥ y/⊥ z/⊥ r/⊥	x/⊤ y/⊤ z/⊤ r/⊤	x/⊤ y/⊤ z/⊤ r/⊤	x/⊤ y/⊤ z/⊤ r/⊤	x/⊤ y/⊤ z/⊤ r/⊤
$\{B_1, B_2, B_3\}$	B_4					x/1 y/2 z/3 r/⊥
$\{B_2, B_3\}$	B_1		x/1 y/2 z/3 r/5			
$\{B_3\}$	B_2			x/1 y/2 z/3 r/5		
$\{B_4\}$	B_3				x/2 y/2 z/3 r/5	
$\{B_1\}$	B_4					x/⊥ y/2 z/3 r/⊥
$\{B_2\}$	B_1		x/⊥ y/2 z/3 r/5			
$\{B_3\}$	B_2			x/⊥ y/2 z/3 r/5		
$\emptyset$	B_3				x/⊥ y/2 z/3 r/5	

identifies the point where a change is *cumulated.* Note that the algorithm checks whether or not the instruction makes a change by the condition $new < s_{pc'}$ which is equivalent to $new \sqcap s_{pc'} = new$ and $new \neq s_{pc'}$.

Obviously, the execution cannot be deterministic: On the level of abstract values there is no way to determine which branch to follow at conditional jumps. Therefore, we consider both branches here. Consequently, we use a working set of program counters, just like the classical algorithm uses a working set of graph nodes. However, the new algorithm uses the working set in a more modest way that the classical: While the classical one chooses a new node from the working set in each iteration, the new one follows one *path* of computation as long as changes occur and the path does not reach the end of the program. This is done in the inner `repeat`/`until` loop. Only if this path terminates, elements are chosen from the working set in the outer `while` loop. In addition, the new algorithm tries to keep the working set as small as possible during execution of a path: Note that the instruction $W := W - pc$ is placed inside the inner loop. Hence, even execution of a path may cause the working set to shrink.

In comparison to the classical algorithm, our approach has the following advantages:

– It uses less memory: There is neither a graph to store the possible control flows in the program nor an associative array needed to store the abstract values at the basic block level.

```
Input: Data–flow problem $(P, L, ![.!], a_0)$
       where $P = I_0 \dots I_n$, $L = \langle A, \sqcap, \sqcup \rangle$
Output: MFP solution $s_0, \dots, s_n \in A$

$s_0 := a_0$
for $i := 1$ to $n$ do $s_i := \top$
$W := \{0, \dots, n\}$
while $W \neq \emptyset$ do
  choose $pc \in W$
  repeat
    $W := W - pc$
    $new := ![I_{pc}!](s_{pc})$
    if $I_{pc} = (\text{goto } l)$ then $pc' := l$
    else
      $pc' := pc + 1$
      if $I_{pc} = (\text{if } \psi \text{ goto } l)$ and $new < s_l$ then
         $W := W + l$
          $s_l := new$
      end
    end
    if $new < s_{pc'}$ then
        $s_{pc'} := new$
      $pc := pc'$
    else $pc := n + 1$
    end
  until $pc = n + 1$
end
```

Fig. 5. New execution algorithm for computing MFP solution

– The data locality is better. At a node, the classical algorithm visits all predecessors and potentially all successors. Since these nodes will typically be scattered in memory, the access to the abstract values associated with them will often cause data cache misses. In contrast, our algorithm only visits a node and potentially its successors. Typically, one of the successors is the next instruction. Since the abstract values are arranged in an array, the abstract value associated with the next instruction is the next element in the array. Here, the likelihood of cache hits is large. Recent studies show that such small differences in data layout can cause large differences in performance on modern system architectures [CHL99, CDL99].
– There is no need for pre–processing by finding the abstract semantics of a basic block $![I_0 \dots I_n!] := ![I_n!] \circ \dots \circ ![I_0!]$.
– There is no need for a post–processing stage, which propagates the solution from the basic block level to the instruction level.

Theorem 1 (Termination). *The algorithm in Fig. 5 terminates for all inputs.*

Proof. During each execution of the inner loop at least one $0 \leq i \leq n$ exists such that value of the variable s_i decreases w.r.t. the underlying partial order of the lattice L. Since L only has finite chains, this can happen only finitely many times. Hence, the inner loop always terminates.

Furthermore, the working set grows iff a conditional jump is encountered and the corresponding value s_l decreases. Just like above, this can happen only finitely many times. Hence, there is an upper bound for the size of the working set. In addition, during each execution of the outer loop, the working set shrinks at least by one element, the one chosen in the outer loop. Hence, the outer loop always terminates. □

Theorem 2 (Correctness). *After termination of the algorithm in Fig. 5, the values of the variables $s_0, \ldots, s_n$ are the MFP solution of the given data–flow problem.*

Proof. To prove correctness, we can obviously consider a modified version of the algorithm, where the inner loop is removed and nodes are selected from the working set in each iteration. In this setting, no program point will be ignored forever. Hence, we can use the results from [GKL+94]: The selection of program point is a fair strategy and the correctness of our algorithm directly follows from the theorem on chaotic fixed point iterations.

To do so, we have to validate one more premise of the theorem: We have to show that the algorithm computes $s_i = \prod_{j \in \text{pred}_P(i)} ![I_j!](s_j)$ for each program point $0 \leq i \leq n$. The algorithm can change s_i iff it visits a program point pc with $pc \in \text{pred}_P(i)$. Let s be the value of s_i before the loop and s' be the value after the loop. If we can show that $s' = s \sqcap ![I_{pc}!](s_{pc})$, we know that the algorithm computes the meet over all predecessors by iteratively computing the pairwise meet. To show that, we distinguish two cases:

1. If $![I_{pc}!](s_{pc}) = new < s$ then $s' = new = s \sqcap ![I_{pc}!](s_{pc})$.
2. Otherwise, we know that $![I_{pc}!](s_{pc}) = new \geq s$ since $![.!]$ is monotone and the initial value of s is the top element. Hence we also have $s' = s = s \sqcap ![I_{pc}!](s_{pc})$. □

Example 4 (Constant Folding Propagation, Cont'd). Table 2 shows an trace of the execution of the new algorithm for the constant folding propagation example. Each line shows the state of the working set and the approximations at the end of the inner loop, and the values of the program counter pc at the beginning and the end of the inner loop (written in the column pcs in the form $begin/end$).

During this execution, the algorithm loads the value of pc only three times from the working set: Once at the beginning and twice after reaching the end of the program ($pcs = 8/9$).

The adaption of the execution algorithm for the other three cases of the taxonomy of data–flow problems described at the end of Section 2 is straightforward: (a) Existential problems can simply be handled by replacing $<$ by $>$, and (b) backward problems require a simple pre–processing which inserts new pseudo instructions to connect jump targets with the corresponding jump instructions.

5 Experimental Results

To validate the benefits of our approach, we studied the performance of the new algorithm in competition with the classical one, both in terms of memory

Table 2. Example execution of new algorithm

W	pcs	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8
$\{1,\ldots,8\}$	0/1	x/⊥ y/⊥ z/⊥ r/⊥	x/1 y/⊤ z/⊤ r/⊤	x/⊤ y/⊤ z/⊤ r/⊤	x/⊤ y/⊤ z/⊤ r/⊤	x/⊤ y/⊤ z/⊤ r/⊤	x/⊤ y/⊤ z/⊤ r/⊤	x/⊤ y/⊤ z/⊤ r/⊤	x/⊤ y/⊤ z/⊤ r/⊤	x/⊤ y/⊤ z/⊤ r/⊤
$\{2,\ldots,8\}$	1/2			x/1 y/2 z/⊥ r/⊥						
$\{3,\ldots,8\}$	2/3				x/1 y/2 z/3 r/⊥					
$\{4,\ldots,8\}$	3/8									
$\{4,\ldots,7\}$	8/9					x/1 y/2 z/3 r/⊥				
$\{5,\ldots,7\}$	4/5						x/1 y/2 z/3 r/5			
$\{6,7\}$	5/6							x/1 y/2 z/3 r/5	x/1 y/2 z/3 r/5	
$\{7\}$	6/7									
$\emptyset$	7/8									x/2 y/2 z/3 r/5
$\{4\}$	8/9					x/⊥ y/2 z/3 r/⊥				
$\emptyset$	4/5						x/⊥ y/2 z/3 r/5			
$\{7\}$	5/6							x/⊥ y/2 z/3 r/5	x/⊥ y/2 z/3 r/5	
$\{7\}$	6/7									
$\emptyset$	7/8									x/⊥ y/2 z/3 r/5
$\emptyset$	8/9									

usage and runtimes. Prior to the presentation of the results, we discuss the experimental setting in more detail.

We have implemented the classical BB algorithm and our new execution algorithm for full Java Virtual Machine (JVM) code [LY97]. This decision was taken in view of the following reasons:

1. As already mentioned, we see the JVM as a natural target environment for our execution–based algorithm, since it already contains an execution environment and is sensitive to high memory overhead.
2. Except for native code compilers for Java [GJS96], all compilers generate the same JVM code as target code. Consequently, we get realistic samples independent of a specific compiler.
3. Java programs are distributed as JVM code, often available for free on the internet.

Although we omitted procedure/method calls from our model, we can handle full JVM code. For intraprocedural analysis, we assume the result of method invocations to be the top element of the lattice.

All these aspects allowed us to collect a large repository of JVM code with little effort. In addition to active search, we established a web site for donations of class files at `http://www-i2.informatik.rwth-aachen.de/~mohnen/CLASSDONATE/`. So far, we have collected 15,339 classes with a total of 98,947 methods. This large set of samples covers a wide range of applications, applets, and APIs. To name a few, it contains the complete JDK runtime environment (including AWT and Swing), the compiler generator ANTLR, the Byte Code Engineering Library, and the knowledge-based system Protégé. The classes were compiled by a variety of compilers: `javac` (Sun) in different version, `jikes` (IBM), `CodeWarrior` (Metrowerks), and `JBuilder` (Borland). In some cases, the class files were compiled to JVM code from other languages than Java, for instance from Ada using `Jgnat`.

In contrast to a hand–selected suite of benchmarks like SPECjvm98 [SPE], we do not impose any restrictions on the samples in the set: The samples may contains errors or even might not be working at all. In our opinion, this allows a better estimation of the "average case" a data–flow analyser must face in practice. Altogther, we consider our experiments suitable for estimating the benefits and drawbacks of our method.

```
import de.fub.bytecode.generic.*;
import Domains.*;

public interface JVMAbstraction {
  public Lattice getLattice();
  public Element getInitialValue(InstructionHandle ih);
  public Function getAbstract(InstructionHandle ih);
}
```

Fig. 6. Interface `JVMAbstraction`

However, we did not integrate our experiment in a JVM. Doing so would have fixed the experiment to a specific architecture since the JVM implementation depends on it. Therefore, we implemented the classical BB algorithm and our new execution algorithm in Java, using the Byte Code Engineering Library [BD98] for accessing JVM class files. The implementation directly follows the notions defined in Section 2: We used the *interface* concept of Java to model the concepts of lattices, (JVM) abstractions, and data–flow problems. For instance, Fig. 6 shows the essential parts of the interface `JVMAbstraction` which models JVM abstractions. Consequently, the algorithms do not depend on specific data–flow problems. In contrast, our approach allows to model any data–flow problem simply by providing a Java class which implements the interface `JVMAbstraction`.

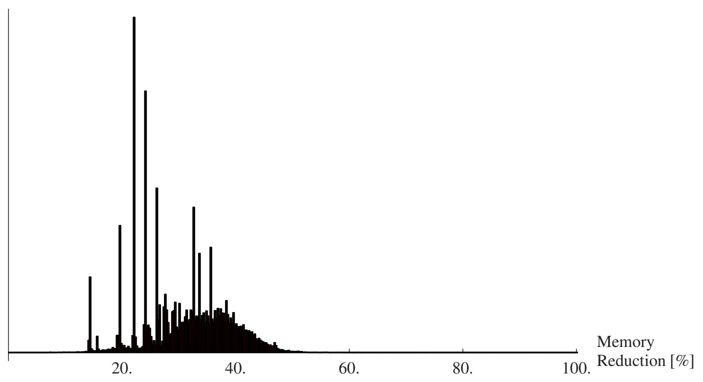


Fig. 7. Histogram of memory reduction

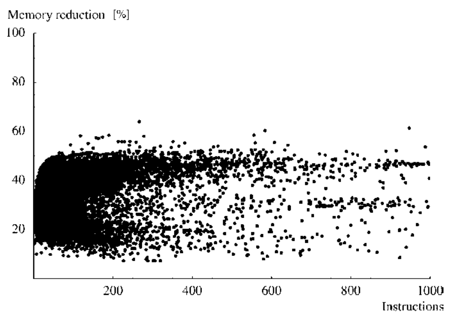
For the experiment, we implemented constant folding propagation, as described in the previous sections. All experiment were done on a system with Pentium III at 750 Mhz, 256 MB main memory running under Linux 2.2.16 and Sun JDK 1.2.2. For each of the 98,947 JVM methods of the repository, we measured memory usage and runtimes of both our algorithm and the classical algorithm. The working set was implemented as a stack.

Memory improvement. Given the number of bytes m_X allocated by our algorithm and the number of bytes m_C allocated by the classical algorithm, we compute the *memory reduction* as the percentage $m_X/m_C * 100$. In the resulting distribution, we found a maximal reduction of 7.28%, a minimal reduction of 74.61%, and an average reduction of 30.83%. Moreover, the median[1] is 31.28%, which is very close to the average. Hence, our algorithm always reduces the amount of memory and reached improvements upto less than a tenth! In the average case, the reduction is about a third. Fig. 7 shows a histogram of the distribution.

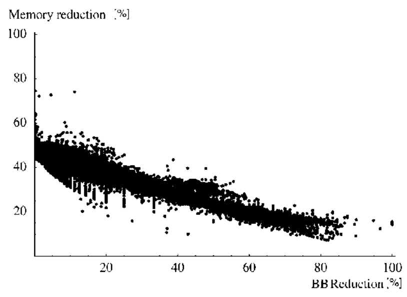
A study of the relation of number of instructions and memory reduction does not reveal a relation between those values. In Fig. 8(a) each point represents a method: The coordinates are the number of instructions on the horizontal axis and memory reduction of the vertical axis. We have restricted the plot to the interesting range up to 1,000 instructions: While the sample set contains methods with up to 32,768 instructions, the average of instructions per method is only 40.3546 and the median is only 11. Obviously, object–orientation has a measurable impact on the structure of program.

Surprisingly, there is a relation between the amount of reduction caused by BBs and memory reduction. One might expect that the classical algorithm is better for higher amounts of reduction cause by BBs. However, this turns out to

[1] The median (or central value) of a distribution is the value with the property that one half of the elements of the distribution is less or equal and the other half is greater or equal.



(a) Memory reduction vs. number of instructions



(b) Memory reduction vs. basic block reduction

Fig. 8. Memory reduction of new algorithm

be a wrong: Fig. 8(b) shows that the new algorithm reduces the memory even more for higher BB reductions.

Runtimes. For the study of runtimes, we use the speedup caused by the use of the classical algorithm: If t_C is the runtime of the classical algorithm and t_X is the runtime of our algorithm, we consider t_C/t_X to be the speedup. The distribution of speedups turned out to be a big surprise: Speedups vary from 291.2 down to 0.015, but the mean is 1.62, median is 1.33, and variance is only 7.49! Hence, for the majority of methods our algorithm performs as well as the BB algorithm. Fig. 9 shows a histogram of the interesting area of the distribution.

Again, relating speedup on one hand and number of instructions Fig. 10(a) on the other hand did not reveal a significant correlation. In addition, and not surprisingly, the speedup is higher for better BB reduction Fig. 10(b) .

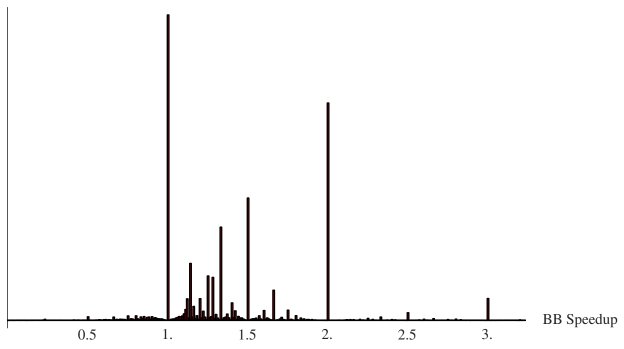
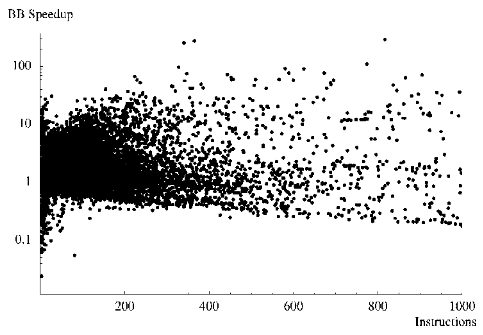
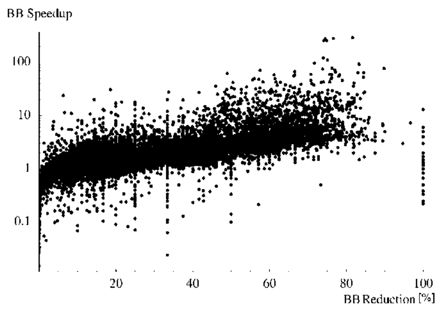


Fig. 9. Histogram of BB speedup



(a) Speedup vs. number of instructions



(b) Speedup vs. basic block reduction

Fig. 10. Speedup of classical algorithm

6 Conclusions and Future Work

We have shown that data–flow analysis can be done without explicit graph structure. Our new algorithm for computing the MFP solution of a data–flow problem is based on the idea of the program executing on the abstract values. The advantages resulting from the approach are less memory use, better data locality, and no need for pre–processing or post–processing stages. We validated these expectation by applying a test implementation to a large set of sample. It turned out that while the runtimes are almost identical, our approach always saves between a third and 9/10 of the memory used by the classical algorithm. In the average case, it saves two thirds of the memory used by the classical algorithm.

The algorithm is very easy to implement in settings where there is already a machinery for execution of programs available, for instance in Java Virtual Machines. In addition, the absence of the graph makes the algorithm easier to implement. In the presence of full JVM code, implementing BB graphs turned out to be trickier than expected. In fact, after having implemented both approaches, errors in the implementation of the BB graphs were revealed by the correct results of the new algorithm.

References

[AH87] S. Abramsky and C. Hankin. An Introduction to Abstract Interpretation. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 1, pages 63–102. Ellis Horwood, 1987. 48

[ASU86] A.V. Ahos, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986. 46

[BD98] B. Bokowski and M. Dahm. Byte Code Engineering. In C. H. Cap, editor, *Java-Informations-Tage (JIT)*, Informatik Aktuell. Springer–Verlag, 1998. See also at `http://bcel.sourceforge.net/`. 57

[Bla98] B. Blanchet. Escape Analysis: Correctness Proof, Implementation and Experimental Results. In *Proceedings of the 25th Symposium on Principles of Programming Languages (POPL)*. ACM, January 1998. 47

[Bla99] B. Blanchet. Escape Analysis for Object Oriented Languages: Application to Java. In *Proceedings of the 14th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, volume 34, 10 of *ACM SIGPLAN Notices*, pages 20–34. ACM, 1999. 47

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixed Points. In *Proceedings of the 4th Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM, January 1977. 48

[CDL99] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In PLDI'99 [PLD99], pages 13–24. 54

[CHL99] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-Conscious Structure Layout. In PLDI'99 [PLD99], pages 1–12. 54

[Deu97] A. Deutsch. On the Complexity of Escape Analysis. In *Proceedings of the 24th Symposium on Principles of Programming Languages (POPL)*, pages 358–371. ACM, January 1997. 47

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison Wesley, 1996. 56

[GKL$^+$94] A. Geser, J. Knoop, G. Lüttgen, O. Rüthing, and B. Steffen. Chaotic Fixed Point Iterations. Technical Report MIP-9403, Fakultät für Mathematik und Informatik, University of Passau, 1994. 55

[KKS98] J. Knoop, D. Koschützki, and B. Steffen. Basic-Block Graphs: Living Dinosaurs? In K. Koskimies, editor, *Proceedings of the 7th International Conference on Compiler Construction (CC)*, number 1383 in Lecture Notes in Computer Science, pages 65–79. Springer–Verlag, 1998. 46

[LY97] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley, 1997. 46, 47, 56

[MJ81] S. S. Muchnick and N. D. Jones. *Program Flow Analysis: Theory and Applications*. Prentice–Hall, 1981. 46

[Moh97] M. Mohnen. Optimising the Memory Management of Higher–Order Functional Programs. Technical Report AIB-97-13, RWTH Aachen, 1997. PhD Thesis. 47

[Muc97] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997. 46

[PLD99] *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI)*, SIGPLAN Notices 34(5). ACM, 1999. 61

[SPE] Standard Performance Evaluation Corporation. SPECjvm98 documentation, Relase 1.01. Online version at `http://www.spec.org/osg/jvm98/jvm98/doc/`. 57