Polytechnic
UNIVERSITY
Brooklyn · Long Island · Westchester

Using the Observer Design Pattern for Implementation of Data Flow Analyses

Gleb Naumovich

Department of Computer and Information Science

Technical Report
TR-CIS-2002-01
6/21/2002

Using the Observer Design Pattern for Implementation of Data Flow Analyses*

Gleb Naumovich
Department of Computer and Information Science
Polytechnic University
5 MetroTech Center
Brooklyn, NY 11201
gleb@poly.edu

Abstract

Data flow analysis is used widely in program compilation, understanding, design, and analysis tools. In data flow analysis, problem-specific information is associated with nodes and/or edges in the flow graph representation of a program or component and re-computed iteratively. A popular data flow analysis design relies on a worklist that stores all nodes and edges whose data flow information has to be re-computed. While this approach is straightforward, it has some drawbacks. First, the presence of the worklist makes data flow algorithms centralized, which may reduce effectiveness of parallel implementations of these algorithms. Second, the worklist approach is difficult to implement in a way that minimizes the amount of information passed between flow graph nodes.

In this paper, we propose to use the well-known Observer pattern for implementation of data flow analyses. We argue that such implementations are more object-oriented in nature, as well as less centralized, than worklist-based ones. We show that by adopting this Observer-based view, data flow analyses that minimize the amount of information passed between flow graph nodes can be implemented easier than by using the worklist view. We present experimental data indicating that for some types of data flow problems, even single-threaded implementations of Observer-based data flow analysis have better run times than comparable worklist-based implementations.

1 Introduction

Data flow analyses (DFAs) [11, 17] are widely used in static program analysis techniques. Examples include many compiler optimization techniques [1], program understanding techniques such as slicing [14, 27], and program verification tools (e.g. [3, 19, 23, 25]). In general, DFAs work on a graph representation of control and/or data flow in the program, propagating information specific to the problem along the edges of this graph. DFA information is associated with each node in the graph. This information is iteratively re-computed by the DFA algorithm.

Typically, DFA implementations use iterative algorithms. *Iterative search* DFA algorithms [1] re-compute the DFA information for all nodes in the graph on each iteration, until the DFA information stops changing. A potential drawback of iterative search algorithms is that for many nodes the DFA information may not change on a given iteration. *Worklist* DFA algorithms avoid this inefficiency by storing all nodes whose DFA information has to be re-computed in a worklist. On each iteration, such algorithms remove node n from the worklist and re-compute its DFA

*This research was partially supported by the National Science Foundation under Grant CCR-0093174.

information. If the DFA information for n changes on this iteration, all nodes that may be affected by the change in the DFA information of n are added to the worklist. The algorithm terminates when the worklist becomes empty.

Worklist algorithms have a centralized nature, which may lead to potential problems. Parallelization of worklist algorithms has to be structured around the worklist. Dwyer and Martin [7] use the replicated workers computation [2] to implement a parallel data flow algorithm for FLAVERS [6]. In their implementation, a pool of threads (workers) is used to re-compute the DFA information for a node. Each time a node is taken off the worklist, if an idle worker exists in the pool, this worker is given the task of re-computing the DFA information for this node. Clearly, this parallelization approach is centralized, which may make the analysis efficiency suffer. A parallel DFA by Lee et al. [8] does not have to rely on the worklist, but rather identifies statically the parts of the problem that will be handled by separate threads. The drawbacks of this approach are that additional analysis of the program is required and the degree of parallelization is fixed.

Another drawback of worklist DFA implementations is that the DFA information for a node may need to be fully re-computed a number of times. In many cases, it is possible to define a DFA that iteratively refines the DFA information for a flow graph node, instead of re-computing it from scratch. While it is possible to define such *iterative refinement* DFAs using a worklist, we show in this paper that such DFAs are more complex than their re-computing counterparts (although not necessarily more efficient, as our experimental data indicates).

In this paper we propose an alternative implementation of iterative DFAs based on the well-known Observer pattern [10]. The Observer pattern is widely used in object-oriented design to describe event-based notification. DFA can be naturally represented in the event-based formalism, where events that represent problem-specific DFA information are sent from one node of the flow graph to another. We argue that this technique is more intuitive than the worklist technique. We also argue that this technique de-centralizes data flow analysis, leading to a more effective parallelization. Finally, we describe experimental results of comparing Observer-based implementations of the MHP analysis [22] and the FLAVERS finite state verifier [6, 23] with a number of worklist-based implementations. These results suggest that even naïve single-thread Observer-based implementations of DFA can be efficient compared to worklist implementations.

The paper is organized as follows. Section 2 gives an overview of DFA and its typical worklist implementations, including an iterative refinement version. Section 3 introduces the Observer pattern and describes DFA based on it. Section 4 describes experiments with different implementations of the MHP algorithm and FLAVERS. Finally, Section 5 concludes and describes future work.

2 Data Flow Analysis Overview

In this section we give a general overview of data flow analysis and its worklist-based implementations. We use the MHP analysis for illustrations.

2.1 Data Flow Analysis

Masticola and Ryder give a generalized framework for data flow analysis in [17]. According to their definition, DFAs operate on a flow graph $G =< N, E >$, where N is the set of nodes and $E \subseteq N \times N$ is the set of edges. Conceptually, all possible values of DFA information that can be associated with flow graph nodes are organized in a lattice[1]. Formally, lattice L is a tuple $< V, \top, \bot, \sqsubseteq, \sqcap, \sqcup >$, where

- V is a set of possible *elements* of the lattice. For the purposes of notation, we treat L as a set, writing $l \in L$ to represent that l is an element of L.
- $\top, \bot \in V$ are unique *top* and *bottom* elements respectively.
- $\sqsubseteq$ is a partial order operation on the elements.

[1] A semi-lattice is sufficient in many situations [17].

- $\sqcap$ is a commutative and associative *meet* operation with the following properties: $\forall a, b \in V$, (1) $a \sqsubseteq b \Leftrightarrow a \sqcap b = a$, (2) $a \sqcap a = a$, (3) $(a \sqcap b) \sqsubseteq a$, (4) $a \sqcap \bot = \bot$, and (5) $a \sqcap \top = a$.
- Optionally, a commutative and associative *join* operation $\sqcup$ is defined and has the following properties: $\forall a, b \in V$, (1) $a \sqsubseteq b \Leftrightarrow a \sqcup b = b$, (2) $a \sqcup a = a$, (3) $a \sqsubseteq (a \sqcup b)$, (4) $a \sqcup \bot = a$, and (5) $a \sqcup \top = \top$.

A *power set* lattice is one of the most common forms of lattices used in practice. Elements of a power set lattice are subsets of a fixed set S. The top and bottom elements of a power set lattice are S and $\emptyset$ respectively, the partial order operation is the subset operation, the meet operation is set intersection, and the join operation is set union.

DFA associates elements of the lattice with flow graph nodes and propagates these elements from one node to another. We define *data flow information I* for node n to be a lattice element associated with n. A DFA is *forward flow* if DFA information is propagated in the direction of the flow graph edges and *backward flow* if DFA information is propagated in the direction opposite the flow graph edges. Bidirectional DFAs propagate information in both directions [18]. To avoid making discussion specific to forward-, backward-, or bidirectional analyses, we use the following terminology. Node n is *data flow dependent* on node m if data flow information of m affects data flow information of n. Let $Dep : N \rightarrow 2^N$ be a function of data flow dependencies among the flow graph nodes, i.e. it returns all nodes data flow dependent on the given node. Similarly, the inverse dependence function $Dep^{-1} : N \rightarrow 2^N$ returns all nodes on which the given node is data flow dependent. DFAs compute $I(n)$ using the I sets for the nodes in $Dep^{-1}(n)$. In this paper, we use *merge function Merge* to describe combining of DFA information. This function operates on a multi-set[2] of lattice elements and produces a lattice element. Either or both meet and join operations of the lattice can be used in defining *Merge*.

To define the flow of DFA information, *propagation function Prop*(n) is associated with each node n. This function specifies how the DFA information is computed for n, given the information from nodes on which n depends.

The most common form of DFAs are *iterative DFAs*. After problem-specific initialization, iterative DFA repeatedly re-compute information for the flow graph nodes, until a *fixed point* is reached, i.e. the DFA information associated with each node stops changing. A DFA is *monotone* if the DFA information associated with each node either never increases or never decreases (in terms of the partial order operation on the lattice) after the initialization. Monotone DFAs possess many desirable properties, including efficient complexity bounds [17]. Typically, DFAs compute conservative information, but can be imprecise. For example, a live variable analysis [1] computes all triples (v, s', s) such that variable v is defined in statement s' and there is a path from s' to statement s on which v is not re-defined. The variable analysis DFA is conservative in the sense that if an execution of the actual program exists on which v is defined in s' and not re-defined until s, the DFA will generate (v, s', s). The converse is not true, since it is possible that the analysis generates (v, s', s) while there are no actual executions of the programs on which v is not re-defined between s' and s. This imprecision is the direct result of the fact that the flow graph is not a precise representation of the program it models and the lattice may not precisely model the program information.

2.2 Example: May Happen in Parallel Analysis

We use *May Happen in Parallel (MHP)* analysis [22] to illustrate the concepts in this paper. This data flow based analysis conservatively computes, for each statement s in a concurrent program, all statements from other threads that can potentially be executed at the same time (or be interleaved) with s. We present the MHP algorithm specific to the Ada concurrency model from [22] (the MHP algorithm for the Java concurrency model appears in [24]).

The flow graph used by MHP analysis is the *Trace Flow Graph (TFG)* model, introduced in [6]. The nodes in this graph explicitly represent both statements local to Ada tasks and rendezvous between these tasks. Similarly, the edges in this graph represent both intra- and inter-task flow of control. Figure 1 shows code for three Ada tasks and the corresponding TFG. The initial and final node of the TFG (shown as triangles and labeled n_{init} and n_{fin} respectively) represent the points in the execution before any task started its execution and after all tasks finished their execution

[2]Multi-set is an unordered collection of elements. A multi-set can contain multiple instances of an element. In this paper, we overload the standard set notation to deal with multi-sets.

```
task body T1 is
begin
  a;
  T2.E;
end T1;

task body T2 is
begin
  accept T2;
  b;
end T2;

task body T3 is
begin
  c;
end T3;
```

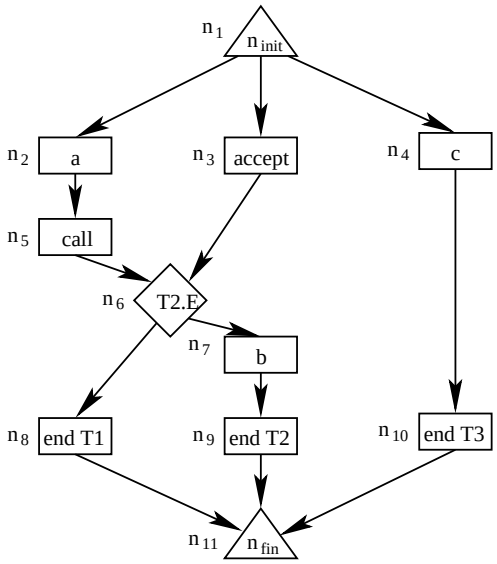

Figure 1: TFG example

respectively. Rectangular-shaped nodes represent non-communication statements in tasks, such as procedure calls to methods `a`, `b`, and `c`[3]. Finally, the diamond-shaped node represents the rendezvous between tasks `T1` and `T2` on entry `E` of `T2`. A communication node has exactly two predecessors, one from each task participating in the communication represented by this node.

Formally, a TFG is a tuple $< N, E, COM, LOCAL, n_{init}, n_{fin} >$, where N is the set of all nodes in this TFG, E is the set of all edges, COM is the set of nodes that represent task communications, $LOCAL$ is the set of nodes that represent non-communication statements in tasks, and n_{init} and n_{fin} are the initial and final node of the TFG respectively.

MHP analysis is forward-flow, so the dependence and inverse dependence functions for MHP analysis are defined simply as:

$$\forall n \in N, Dep(n) = Succs(n)$$

$$\forall n \in N, Dep^{-1}(n) = Preds(n)$$

The lattice for MHP analysis is a power set lattice based on the set of all TFG nodes. With each node n in the TFG we associate a set of TFG nodes I. If node $m \in I$, it means that the statement(s) represented by n may happen in parallel with the statement(s) represented by m (in the rest of the paper, we simply say "n may happen in parallel with m"). Different types of TFG nodes require different merge functions. For local nodes we use set union, since the semantics of transfer of control between two non-communication nodes in the same task are independent of the activity of other tasks. For communication nodes we use set intersection, since the semantics of task rendezvous require a task reaching an entry call (accept statement) to block until another task reaches the corresponding accept statement (entry call). Since each communication node c has exactly two local predecessors that represent these blocking statements, it may happen in parallel only with nodes that may happen in parallel with both of these local predecessors. Formally,

[3]We assume that these three procedures do not contain any task communication statements. In general, the MHP analysis inlines the communication statements from called subroutines.

the merge function is defined as follows[4]:

$$\forall n \in N, \mathit{Merge}_n(B) = \begin{cases} \bigcup_{\forall l \in B} l & \text{if } n \in \mathit{LOCAL} \\ l_1 \cap l_2, \text{ where } \{l_1, l_2\} = B & \text{if } n \in \mathit{COM} \wedge p_1 \in l_2 \\ \emptyset, \text{ where } \{l_1, l_2\} = B & \text{if } n \in \mathit{COM} \wedge p_1 \notin l_2 \end{cases}$$

Propagation functions are also different for local and communication nodes. The propagation function for a local node has to take into account the possibility that this node immediately follows a communication node. Communication nodes represent synchronization points and, after synchronization, the participating tasks can continue independently. Therefore, we add local successors of each reachable communication node to the MHP sets of each other.

Propagation functions for communication nodes keep track of whether, according to the MHP information associated with local predecessors of such nodes, the communication represented by the node is reachable. The MHP analysis considers a communication node reachable if the two local predecessors of this node may happen in parallel with each other. If a communication node is not reachable, the propagation function returns $\emptyset$ for this node. If a communication node is reachable, the propagation function for this node behaves as the identity function. Formally, the propagation function is defined as follows[5]:

$$\forall n \in N, \forall l \in L, \mathit{Prop}_n(l) = \begin{cases} l \cup \{m | \exists c \in \mathit{COM} : n, m \in \mathit{Succs}(c) \wedge \mathit{task}(n) \neq \mathit{task}(m)\} & \text{if } n \in \mathit{LOCAL} \\ l & \text{if } n \in \mathit{COM}, \end{cases}$$

where function *task* returns the identity of the task of the given local node.

MHP information is symmetric in nature, i.e. if $n \in I(m)$, then necessarily $m \in I(n)$. MHP analysis must enforce symmetry explicitly, since it may not be achieved via only the merge and propagation operations. The traditional data flow formulation [17, 18] does not accommodate the symmetry step of MHP analysis.

2.3 Worklist DFA Implementations

Data flow analyses typically are implemented as worklist algorithms. A worklist is a collection of flow graph nodes that have to be processed before the algorithm terminates. This collection is modified dynamically. Figure 2 gives the general form of a worklist implementation of data flow algorithms. In the first step, the worklist is initialized. This initialization is specific to the data flow problem being solved. For the MHP analysis, the initial DFA information is computed as follows. Initially, the *I* sets of all nodes are set to $\emptyset$, except for the successors of the initial node n_{init}. The *I* sets of such a successor s are initialized to contain all other successors of the initial node that belong to tasks other than that of s. Formally,

$$\forall n \in N, I(n) = \begin{cases} \emptyset & \text{if } n \notin \mathit{Succs}(n_{init}) \\ \mathit{Succs}(n_{init}) \setminus \mathit{nodesIn}(\mathit{task}(n)) & \text{if } n \in \mathit{Succs}(n_{init}) \end{cases}$$

For the example in Figure 1, initially, the *I* sets of all nodes are empty, except $I(n_2) = \{n_3, n_4\}$, $I(n_3) = \{n_2, n_4\}$, and $I(n_4) = \{n_2, n_3\}$.

In the second step of the algorithm, the worklist is initialized. For the MHP analysis, all successor nodes of nodes from $\mathit{Succs}(n_{init})$ (nodes n_5, n_6, and n_{10} from Figure 1) are placed on the worklist in this step.

After the initialization, the iterative part of the algorithm begins, where on each iteration a single node is taken off the worklist and processed. This processing consists of merging the data flow information coming into the node

[4] A communication node always has two predecessors (p_1 and p_2 in this equation, corresponding to lattice elements l_1 and l_2 respectively), so there are exactly two lattice elements in bag B.

[5] In addition to operations in this formula, when computing DFA information for a local node, MHP analysis checks a simple necessary condition for communication predecessors of this node to be reachable. For simplicity, we do not show this check here.

Algorithm 1 (DFA).

Input: *A flow graph $G =< N, E >$, lattice L, and a set of Merge and Prop functions.*
Output: *Data flow information $I(n)$ for each $n \in N$.*

(1) Initialize $I(n)$ for each $n \in N$ (analysis-specific)
(2) Initialize the worklist W (analysis-specific)
(3) While $W \neq \emptyset$
(4) Remove node n from W
(5) Compute $IN(n) = Merge_n(\{I(d) | \forall d \in Dep^{-1}(n)\})$
(6) Compute $I'(n) = Prop_n(IN(n))$
(7) If $I'(n) \neq I(n)$
// Add nodes that depend on n to the worklist:
(8) $W = W \cup Dep(n)$

Figure 2: A general worklist data flow algorithm

and then applying the propagation function to obtain the new version of the information associated with the node. In Figure 2, we use $I'(n)$ to refer to newly computed DFA information for node n and $I(n)$ to refer to DFA information for this node computed on previous iterations. In general, the order in which nodes are taken from the worklist may affect the efficiency of the analysis but does not affect precision for monotone problems. If, as a result of application of the propagation function, the information associated with the node changes, the nodes whose data flow information may be affected are placed on the worklist. The algorithm terminates when the worklist becomes empty. To illustrate a worklist implementation of the MHP analysis, consider one iteration of the algorithm. Let node n_{10} be taken off the worklist. First, incoming information for this node is computed. Since this is a local node with a single predecessor n_4, the incoming information for node n_{10} is equal to the current value of the *I* set of node n_4, $\{n_2, n_3\}$. Applying the propagation function yields the same value for the *I* set of node n_{10}. The symmetry step also inserts node n_{10} in the *I* sets of nodes n_2 and n_3. Since the previous value of the *I* set of node n_{10} was $\emptyset$, we add all successors of n_{10} (node n_{11}) to the worklist. Similarly, we add all successors of nodes n_2 and n_3, whose MHP information also changed, to the worklist. Since node n_6 is already in the worklist, after this iteration of the algorithm, the worklist is $\{n_6, n_{11}\}$.

2.4 Iterative Refinement DFA Algorithms

The straightforward worklist implementation of data flow analysis is intuitively inefficient because it may propagate the same information through an edge several times. Consider our MHP analysis example. In Section 2.3, we illustrated one step of the analysis on which node n_{10} was taken off the worklist. At a later point in the algorithm, node n_5 is inserted in the *I* set of node n_4 and therefore node n_{10} will be placed on the worklist. When, subsequently, node n_{10} is taken off the worklist, its *I* set is re-computed from scratch. Intuitively, it is more efficient to recognize that nodes n_2 and n_3 have already been propagated from node n_4 to node n_{10} and simply add to the *I* set of node n_{10} those nodes that have not yet been propagated.

To achieve this refinement of DFA information, we associate sets Δ with edges of the flow graph. At any point, for a given edge $e = (s, d)$, $\Delta(e)$ contains the difference of the current value of $I(s)$ and the value of $I(s)$ at the time when when $I(d)$ was computed last. To enable computing the difference of data flow information, we introduce the *difference operation* $\setminus$ on the lattice:

$$\forall l_1, l_2 \in L, l_1 \setminus l_2 = l, \text{ where}$$
$$(1)\ l \sqcup (l_1 \sqcap l_2) = l_1$$
$$(2)\ \forall l' \text{ that satisfy (1)}, l \sqcap l' = l \text{ (l is "minimal")}$$

Algorithm 2 (delta-DFA).

Input: *A flow graph $G =< N, E >$, lattice L, and a set of Merge and Prop functions.*
Output: *Data flow information $I(n)$ for each $n \in N$.*

(1) Initialize $I(n)$ for each $n \in N$ (analysis-specific)
(2) Initialize the worklist W (analysis-specific)
(3) Initialize $\Delta(e)$ for each $e \in E$ (analysis-specific)
(4) While $W \neq \emptyset$
(5) Remove a node n from W
(6) Compute $IN(n) =$
$Merge_n(\{\Delta(d, n) | \forall d \in Dep^{-1}(n)\})$
(7) Compute $I'(n) = Prop_n(I(n), IN(n))$
(8) $\forall s \in Dep(n)$, set $\Delta(n, s) = I'(n) \setminus I(n)$
(9) If $I'(n) \neq I(n)$
// Add nodes that depend on n to the worklist:
(10) $W = W \cup Dep(n)$

Figure 3: A general Δ-worklist-DFA algorithm

For a power set lattice, such as the one used by the MHP analysis, the difference operation coincides with set difference operation.

Each time node n is taken from the worklist during the algorithm, we use information in the Δ sets of edges connecting nodes on which n depends with n to compute an update for data flow information associated with n. We need to be able to combine the Δ sets with the I set of the node being processed. To do this, we alter function *Merge* to operate on information coming to a node through Δ sets of its adjacent edges. We also alter function *Prop* to combine freshly propagated DFA information for the node with the previously computed DFA information for this node. For MHP analysis, we have

$$\forall n \in N, Merge_n(B) = \begin{cases} \bigcup_{\forall l \in B} l & \text{if } n \in LOCAL \\ (l_1 \cap I(p_2)) \cup (l_2 \cap I(p_1)), \text{ where } \{l_1, l_2\} = B, \{p_1, p_2\} = Preds(n), & \\ \quad \text{and } l_1 \text{ corresponds to } p_1 \text{ and } l_2 \text{ corresponds to } p_2 & \text{if } n \in COM \end{cases}$$

$$\forall n \in N, \forall l_1, l_2 \in L,$$

$$Prop_n(l_1, l_2) = \begin{cases} l_1 \cup l_2 \cup \{m | \exists c \in COM : n, m \in Succs(c) \wedge task(n) \neq task(m)\} & \text{if } n \in LOCAL \\ l_1 \cup l_2 & \text{if } n \in COM \end{cases}$$

We call this type of DFAs Δ-*worklist-DFA*s and the re-computing type of DFAs from Section 2.3 *worklist-DFA*s. Figure 3 shows a general form of the Δ-worklist-DFA algorithm.

Note that Δ-worklist-DFA and worklist-DFA versions of the same analysis compute identical information for a given data flow problem. The difference between these two forms of algorithms lies in efficiency of computations. Δ-worklist-DFA algorithms have advantage over worklist-DFA algorithms in cases where efficient merging of DFA information for nodes is possible by the *Merge* and *Prop* functions. For example, the worst-case complexity of worklist-DFA and Δ-worklist-DFA algorithms for MHP analysis is $\mathcal{O}(|N|^4)$ and $\mathcal{O}(|N|^3)$ respectively. But although Δ-worklist-DFA algorithms may have better worst-case complexity than worklist-DFA algorithms, this does not mean

that Δ-worklist-DFA algorithms always run faster than worklist-DFA algorithms. In Section 4, we present experimental data for the MHP algorithm suggesting that in some cases in practice, worklist-DFA implementations may be more efficient than Δ-worklist-DFA implementations.

3 Observer-style Implementation of Data Flow Analysis

In this section, we propose using the Observer design pattern for implementing data flow analyses. In Section 3.1, we overview the Observer pattern and in Section 3.2 we describe the details of using this pattern to implement DFAs. Finally, in Section 3.3, we describe an Observer-based implementation of the MHP analysis.

3.1 Observer Pattern

The Observer pattern is a popular way for describing event-based control flow for object-oriented programs [10]. This pattern defines interactions between two types of objects, *observers* and *subjects*. On the high level, the observers must react to changes in the states of the subjects. Instead of repeatedly checking for changes in the states of the subjects, the observers first *register* with the subjects. Subsequently, whenever the state of a subject changes, this subject *notifies* all observers registered with it about the change. To reduce coupling between objects, in many situations where the Observer pattern can be applied, it is possible to avoid disclosing the identity of subjects to the observers. Instead of passing the identity of the changed subject in the notification, an event that describes the change itself can be passed. The Observer pattern is used especially widely in graphical user interface design. For example, an observer object may have as a subject a visible button in a user interface. Whenever this button is clicked, it notifies the observer object, which in turn can carry out some operations, such as displaying a dialog window.

3.2 Using the Observer Pattern to Implement Data Flow Analyses

Event-based notification is natural for describing DFAs. Nodes of the flow graph play roles of both observers and subjects. A node observes all nodes (possibly including itself) on whose DFA information its own DFA information may depend. When node n is notified of a change in the DFA information of a node it depends on, n immediately re-computes its DFA information. If this results in a change in the DFA information of n, then n notifies all of its observer nodes. This short and intuitive high-level description is particularly attractive when data flow analysis has to be explained to a novice not familiar with lattice and function space theory.

Similar to worklist-based DFA algorithms, Observer-based DFA algorithms may either fully re-compute the DFA information for node n each time n is notified of a change to one of nodes on which it depends or modify the DFA information for n incrementally. We refer to the first type of Observer-based algorithms *observer-DFA* and the second type Δ-*observer-DFA*. Figure 4 shows high-level pseudocode for the Δ-observer-DFA algorithm. The algorithm uses a set $N_0 \subseteq N$ of *kick-off* nodes, i.e. nodes that initialize the notification chains. This set is analysis-specific.

For generality, we deliberately do not specify the mechanics of notification calls. For example, notification can take the form of regular methods calls. Alternatively, a new thread can be spawned to handle each notification call or a thread pool [2] can be used.

Intuitively, Δ-observer-DFA algorithms propagate information among flow graph edges in smaller portions than the corresponding Δ-worklist-DFA algorithms. The reason for this is that when node n is processed by a worklist algorithm, DFA information from all nodes that n depends on is used. Alternatively, when the `notify` function is called for this node in a Δ-observer-DFA, only DFA information from a single notifying node on which n depends is used to update $I(n)$. Because of this fine granularity of Observer-based DFA algorithms, Δ-observer-DFA appears to be more natural than observer-DFA. Therefore, in this paper, we do not discuss or experiment with observer-DFA.

Observer-based view of DFAs arguably represents a better design than the worklist-based view. First, the Observer pattern uses an event-based notification mechanism. This mechanism naturally lends itself to description of DFAs, where changing information associated with node n in the flow graph should lead to changing information associated

Algorithm 3 (Observer-DFA).

Input: *A flow graph* $G =< N, E >$*, lattice* L*, a set of Merge and Prop functions, and kick-off nodes* N_0.
Output: *Data flow information* $I(n)$ *for each* $n \in N$.

Main
(1) Initialize $I(n)$ for each $n \in N$ (analysis-specific)
// kick-off the analysis
(2) $\forall n \in N_0$:
(3) `notifyObservers`$(n, I(n))$

notifyObservers Input: $n \in N, l \in V$
(4) $\forall d \in Dep(n)$:
(5) `notify`(d, l)

notify Input: $n \in N, l \in V$
(6) Compute $I'(n) = Prop_n(I(n), l)$
(7) If $I'(n) \neq I(n)$
(8) `notifyObservers`$(n, I(n))$

Figure 4: A general Δ-observer-DFA algorithm

with nodes dependent on n. Second, using the Observer pattern allows algorithm designers to use object-oriented design. It is natural, from the object-oriented view, that nodes are responsible for computing their own information and communicating it directly to other nodes. Third, using the Observer pattern provides good information hiding, since it is possible to set it up so that observer nodes do not know the identity of those nodes that notify them. This leads to more generic implementations of DFA frameworks that can be used to solve multiple data flow problems. Last but not least, Observer-based DFAs are de-centralized, while worklist-based DFAs are centered around the worklist. This is important for parallelization of DFAs. In a parallel implementation of a worklist algorithm, different threads of control that perform data flow computations must synchronize on a single worklist. A parallel implementation of an Observer-based algorithm does not have this restriction, although some synchronization may be necessary to make sure that information for a node is not modified while it is being used by some other node. Thus, parallel Observer-style implementations are likely to allow more parallel operations at the same time than the corresponding worklist implementations.

3.3 Observer-based implementation of the MHP analysis

Since we introduced the lattice for the MHP analysis, as well as *Dep* and *Prop* functions for it, in Section 2, we only need to give the set of kick-off nodes N_0 for the MHP analysis. Recall that in step (1) of the Δ-worklist-DFA algorithm for MHP analysis (which is exactly the same as the step (1) of the Δ-observer-DFA algorithm), DFA information of the successors of the initial node of the TFG is initialized. These nodes comprise the kick-off set for the Δ-observer-DFA algorithm for the MHP analysis: $N_0 = Succs(n_{init})$.

Note that the symmetry step has to be carried out in addition to the operations in the general Δ-worklist-DFA algorithm in Figure 4. This step can be implemented in Observer-style as well, so that whenever node m is added to the I set of node n through means other than symmetry, n notifies m about this event.

4 Experiments

We implemented a Δ-observer-DFA algorithm for MHP and FLAVERS [6] analyses and experimentally compared them with worklist-based implementations of these analyses. All implementations have identical precision in the sense that they compute the same information for each problem. All worklist algorithms used the same FIFO worklist. All implementation was done in Java. For each example, we ran each algorithm 5 times and averaged the run times for each version. Section 4.1 describes an experiment that compares the Δ-observer-DFA algorithm and several versions of worklist-DFA and Δ-worklist-DFA algorithms for MHP analysis. This experiment was done using mostly small concurrent Ada programs. In Section 4.2 we describe an experiment that compares the Δ-observer-DFA algorithm and a worklist-DFA algorithm on scalable versions of several examples. Section 4.3 describes a comparison of a Δ-observer-DFA algorithm with two versions of Δ-worklist-DFA algorithm for FLAVERS. Section 4.4 concludes with a discussion of the results.

4.1 Comparison of MHP Analysis Implementations on Mostly Small Programs

The examples for our first experiment with different implementations of MHP analysis came from the suite of 160 mostly small concurrent Ada programs, used in our experiments with the MHP algorithm in [22]. 132 of these were programs used by Masticola and Ryder to evaluate their non-concurrency analysis [20].

For MHP analysis, run time of its implementations depends on what data structures are used to implement sets of TFG nodes that represent DFA information. We constructed versions of both worklist-DFA and Δ-worklist-DFA for MHP analysis for different implementations of sets. We used four set implementations: `HashSet`, `TreeSet`, and `BitSet` from the standard Java package `java.util` and our own array-based implementation of a look-up table. As a result, in this experiment, we compared nine versions of MHP analysis on all 160 programs, including four versions for each of worklist-DFA and Δ-worklist-DFA and one Δ-observer-DFA implementation. We used a 1.33GHz Athlon Linux machine with 1Gb of memory (the maximal amount of memory JVM was allowed to use was set to 256Mb), running Sun JDK 1.3.1.

Our Δ-observer-DFA implementation is single threaded, with all notifications implemented as methods calls. One concern we had before the experiment was that call chains may be too long, slowing down the analysis and possibly exceeding the maximal size of the JVM call stack.

For clarity of presentation of the results, we separate the description of this experiment into three stages. In the first stage, we compare the four versions of worklist-DFA and select the version with the best performance. In the second stage, we compare the four versions of Δ-worklist-DFA and select the version with the best performance. Finally, we compare Δ-observer-DFA version with the two versions selected in the first two steps.

The results of comparing the four versions of worklist-DFA MHP analysis are displayed in Figure 5. We show relative running times for the four versions for each of the 30 programs compared. For each program, we select the version that ran the longest and take its running time to be the unit against which running times of other versions are compared. Therefore, for each program the bar extending to 1 on the Y axis is the version that took the longest. For example, for the first program, the `TreeSet`-based implementation took the longest, while the running time of the `BitSet`-based implementation was only about 9% of that of the `TreeSet`-based implementation. It is obvious from this figure that the `BitSet`-based implementation is the most efficient, as it outperforms all other implementations on all 30 programs. The longest running time of the `BitSet`-based implementation was 4.5 seconds (the first bar in Figure 5). Therefore, in our subsequent comparisons we only show the `BitSet`-based implementation of worklist-DFA MHP analysis.

Similar to worklist-DFA, we compared four different versions of Δ-worklist-DFA. The results of this comparison are shown in Figure 6. It is clear from this figure that the `BitSet`-based and `Lookup`-based implementations are better on average than the `HashSet`-based and `TreeSet`-based implementations. On average, the `BitSet`-based implementation is somewhat better than than the `Lookup`-based implementation. In our subsequent comparisons we only show the `BitSet`-based implementation of Δ-worklist-DFA MHP analysis.

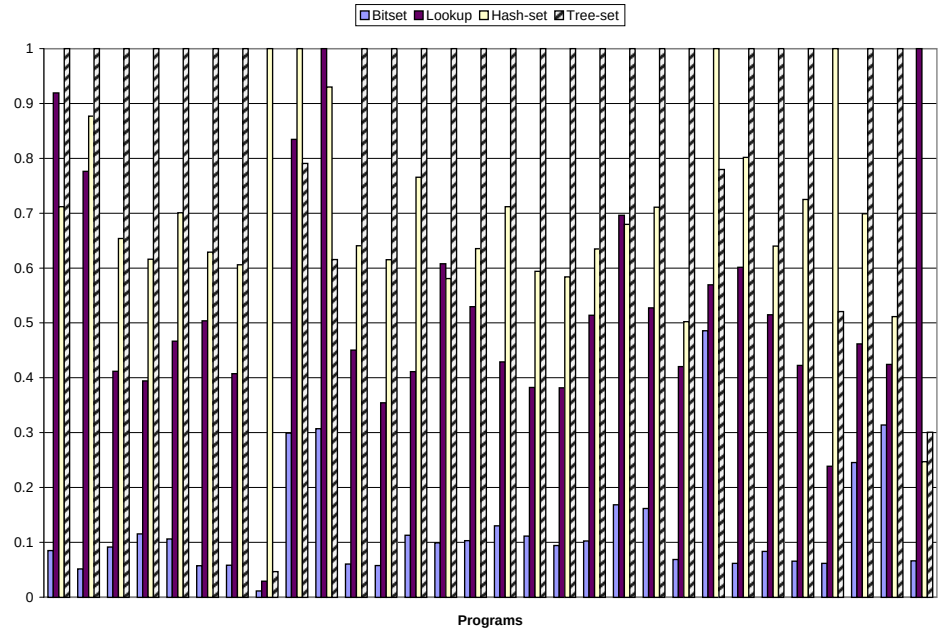

Figure 5: Results of comparing the four versions of worklist-DFA MHP analysis on 30 Ada programs

Finally, we compare the performance of our Δ-observer-DFA implementation with `BitSet`-based implementations of worklist-DFA and Δ-worklist-DFA. Figure 7 shows the results of this comparison. It turned out that for all of these programs, the Δ-worklist-DFA version took the longest. (We address the reasons the worklist-DFA version seems to perform better than the Δ-worklist-DFA version for MHP analysis in Section 4.4.) For each of the 30 programs, we show run time for worklist-DFA and Δ-observer-DFA versions as percentage of run time for Δ-worklist-DFA for the same program. Somewhat surprisingly for us, on average, the Δ-observer-DFA implementation performs better than the worklist implementations. In fact, the longest run time for the Δ-observer-DFA implementation is under one second, while run times for the worklist-DFA and Δ-worklist-DFA implementations on the same program are 4.5 and 10.5 seconds respectively.

The second surprising result of this comparison is that the Δ-worklist-DFA algorithm performed worse than the worklist-DFA algorithm. We address the causes of this in Section 4.4.

4.2 Comparison of MHP Analysis Implementations on Scalable Examples

Several well-known examples from finite state verification literature are scalable in the sense that the number of threads in the program can be varied. These examples, though contrived, give us the ability to see how well different implementations of DFAs scale. In this experiment, we compared only the `BitSet`-based worklist-DFA version with the Δ-observer-DFA version, since the `BitSet`-based Δ-worklist-DFA version turned out to be significantly inferior to both worklist-DFA and Δ-observer-DFA versions in the experiment discussed in Section 4.1. This experiment was performed on a Sun Enterprise 3500 with two 336MHz processors and 2Gb of memory (the full amount of memory was made available to JDK), running Solaris 2.6 and Sun JDK version 1.3.0 with HotSpot[6].

[6]The reason we used different platforms for different experiments was that we ran into JDK bugs on the Sun machine when attempting to run the experiments in Section 4.1 and also ran into JDK bugs on the Linux machine when attempting to run the experiments in Section 4.3. On those experiments that we ran on both platforms, timing data were consistent.

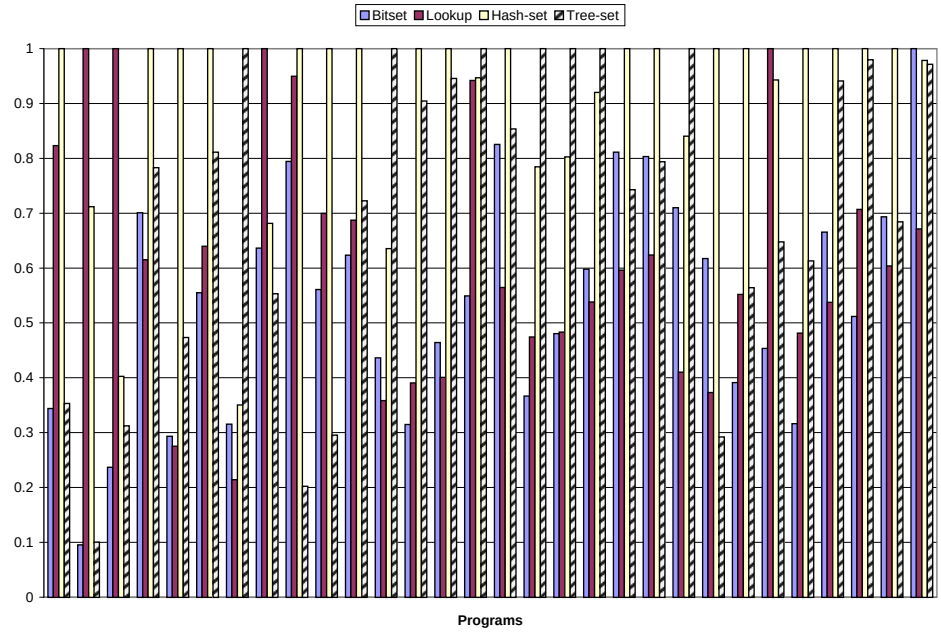

Figure 6: Results of comparing the four versions of Δ-worklist-DFA MHP analysis on 30 Ada programs

Figure 8 lists the scalable concurrent Ada programs used in this experiment and references figures that displays the results for these programs (all these figures are in Appendix A). We analyzed scalable versions of eight programs. Four programs are variations of the dining philosophers example. Although these are different versions of the same example, they exhibit different patterns of communications among tasks, which seems to affect the running performance of different versions of the MHP analysis significantly.

The Δ-observer-DFA implementation runs significantly faster than the `BitSet`-based worklist-DFA implementation on the `dpd`, `cyclic`, and `gas` examples. The `BitSet`-based worklist-DFA implementation runs significantly faster than the Δ-observer-DFA implementation on the `dph`, `dpfm`, `relay`, and `rw` examples. The two implementations exhibit almost the same performance on the `dps` example, with the `BitSet`-based worklist-DFA implementation being marginally faster. Unfortunately, for several large programs, Δ-worklist-DFA exceeded the maximal JVM call stack size.

4.3 Comparison of FLAVERS Analysis Implementations

FLAVERS is a finite state verification tool for checking properties of programs using Ada or Java model of concurrency [6, 23]. FLAVERS uses a data flow analysis algorithm for checking properties. Similar to MHP analysis, FLAVERS uses the TFG model to represent programs under analysis. Properties to be checked are represented as finite state automata (FSAs). In addition, FLAVERS allows the analyst to improve the analysis precision by modeling selected behaviors of the program components (such as individual program variables) as FSAs [6]. The data flow information that FLAVERS associates with each node in the TFG is a set of *tuples*. Each tuple contains a state for the property and a state for each of the constraints used in the analysis, thereby representing an approximate partial state of the program execution, with respect to the property. The merge operation for all nodes is set union. The propagation function replaces each tuple in the input set with a tuple obtained by applying the action represented to the node as a transition to all states in the tuple.

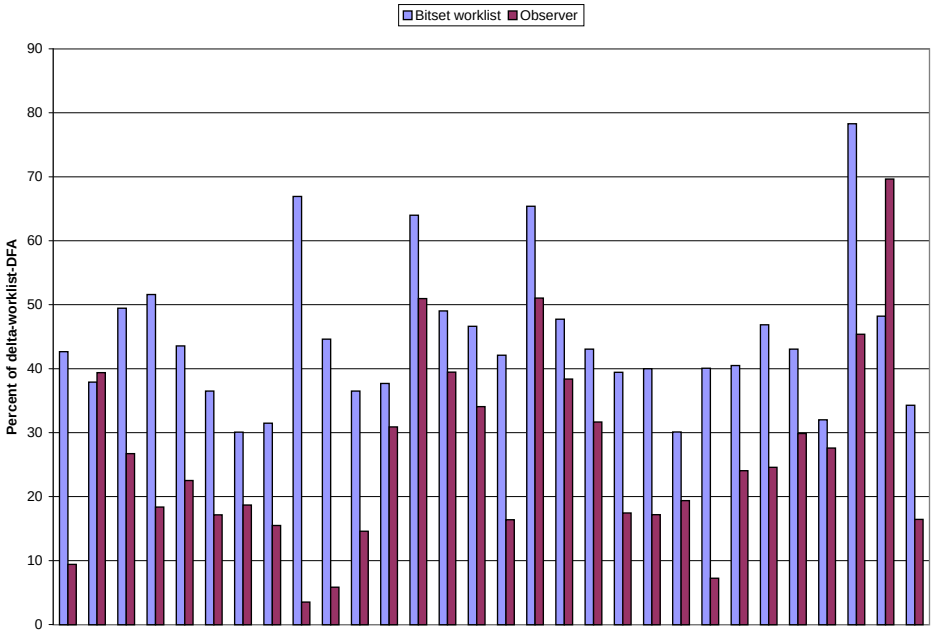

Figure 7: Results of comparing the Δ-observer-DFA implementation with `BitSet`-based implementations of worklist-DFA and Δ-worklist-DFA MHP analysis on 30 Ada programs

We produced a Δ-observer-DFA implementation of the verification algorithm of FLAVERS for Ada and compared it with two existing Δ-worklist-DFA versions[7]. The *pull* version is implemented similar to the general Δ-worklist-DFA algorithm in Figure 3. The *push* version, like the general Δ-worklist-DFA algorithm, makes sure that a unit of information (tuple) is propagated only once from node n to node m, where $m \in \mathit{Dep}(n)$. A set of "new" tuples $\mathit{NEW}(n)$ is associated with each node n in the TFG. When a node n is taken from the worklist, it computes $I'(n) = I(n) \cup \mathit{NEW}(n)$ and adds the tuples from its *NEW* set that were not already in its I set to the *NEW* sets of all its successors:

$$\forall d \in \mathit{Dep}(n), \mathit{NEW}'(d) = \mathit{NEW}(d) \cup (\mathit{NEW}(n) \setminus I(n))$$

Finally, $\mathit{NEW}(n)$ is set to empty. A detailed description of these versions and their experimental comparison for different modes of FLAVERS usage appear in [4].

We ran our Δ-observer-DFA implementation, as well as the push and pull versions, on a number of standard examples, including different sizes of gas station [12], Milner's cyclic scheduler [21], memory management [9], token ring simulation [5], and Chiron [15] examples, as well as well-known readers-writers, dining philosophers and relay examples. In total, we ran 208 experiments (some of them use different sizes of the same scalable program and others check different properties for the same program). This experiment was performed on the same platform as the one in Section 4.2.

The size of data flow problems being solved by FLAVERS in order to check properties of these examples is measured not only by the size of programs, but also by the amount of information modeled by the constraints. In the following diagrams, we order examples verified with different implementations of FLAVERS not by the size of their code, but rather by the running time of the push implementation. We do not show data for 152 programs for which running time of the push implementation was less than 0.1 seconds, as no significant differences in running

[7] Earlier undocumented experiments with FLAVERS indicated that in most situations, Δ-worklist-DFA implementations are more efficient than worklist-DFA implementations.

Name	Description	Figure
dps	the standard dining philosophers implementation, where the order in which philosophers pick forks is reversed for one of the philosophers, to avoid deadlock	12
dpd	the version that uses a dictionary to prevent deadlock (a philosopher using the dictionary cannot eat)	13
dph	the version that uses a host task to prevent deadlock	14
dpfm	the version that uses a fork manager to prevent deadlock	15
cyclic	the Ada implementation of Milner's cyclic scheduler [21]	16
gas	the gas station example [12]	17
relay	the relay example, where multiple tasks communicate indirectly via a single task	18
rw	a shared buffer implementation of the readers-writers example	19

Figure 8: Scalable examples used in the experiment

times between different versions of DFA were discovered for these programs. Figure 9 shows the results for programs on which the push implementation ran for more than 0.1 but less than 1 second. Figure 10 shows the results for programs on which the push implementation ran for more than 1 but less than 5 seconds. Figure 11 shows the results for programs on which the push implementation ran for more than 5 seconds.

In a number of cases, the Δ-observer-DFA implementation performed significantly worse than both push and pull implementations. On the other hand, in many cases the Δ-observer-DFA implementation was faster than the push and pull implementations. For the largest example, a version of the memory management program (the last group of data in Figure 11), the Δ-observer-DFA implementation was about 5 seconds slower than the pull implementation and about 7 seconds slower than the push implementation, but for the second largest example, a version of the token ring protocol, it ran about 10 seconds faster than the other two implementations. Note that groups of data 3-5 from the right of Figure 11 represent runs of the analyses on the same version of the relay example, with three different properties.

As evident from Figures 9 and 10, on several examples, the Δ-observer-DFA implementation ran significantly slower than the other two implementations. About an order of magnitude difference was observed on a version of the memory management example (this indicates that the property and constraints affect the applicability of the Δ-observer-DFA implementation, because the Δ-observer-DFA implementation performed fine on the memory management example when other properties were checked). The Δ-observer-DFA implementation was also significantly slower for the alternating bit protocol and handshake protocol examples. At present, we do not know the precise reason for this behavior.

4.4 Discussion

We view the results of our experiments with the observer-style implementations of data flow analysis as very encouraging. At the onset of the experiment, we were pessimistic about scalability of the technique, given that the size of program call stack is likely to be very large for large examples. The experimental results show that the extra time the JVM takes to maintain large call stacks seems to be offset by the efficient propagation of DFA information.

Comparing problems on which the Δ-observer-DFA versions for MHP analysis did better than the worklist-DFA and Δ-worklist-DFA versions, it seems that the Δ-observer-DFA version runs faster for programs with decentralized control, while the worklist versions runs faster for programs with centralized control. For example, all scalable examples from Section 4.2 on which the Δ-observer-DFA implementation ran faster than the other DFA implementations have decentralized nature, with many similar threads collaborating to achieve some task. The examples on which worklist-DFA implementations ran faster than the Δ-observer-DFA implementation tend to be centralized. For example, in dpd and dpfm, dining philosophers are synchronized through a central thread (dictionary and fork manager respectively). We believe that decentralized examples tend to have more parallelism, which in the case of MHP and

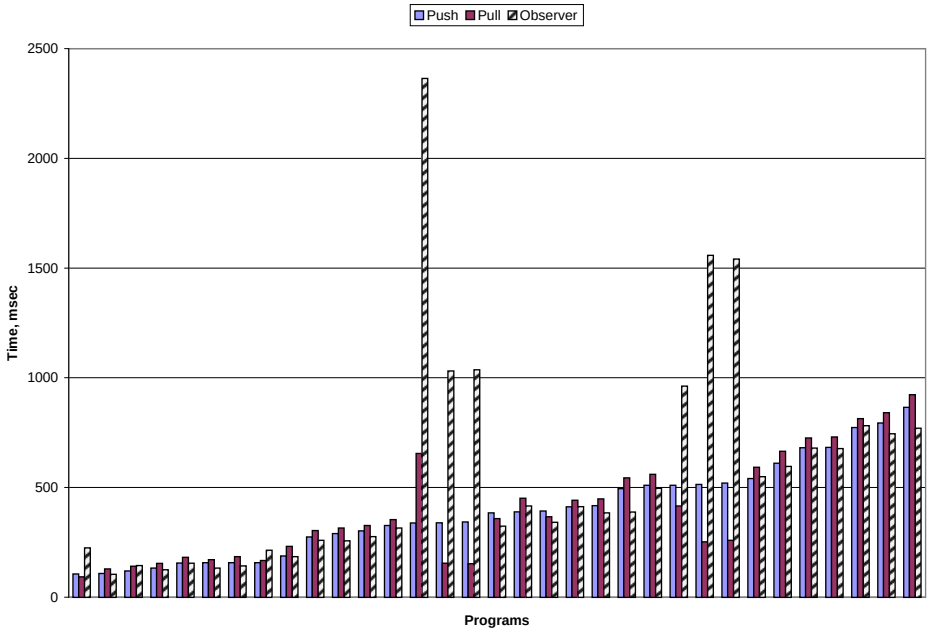

Figure 9: Running times for the three versions of FLAVERS for programs on which the push implementation took between 0.1 and 1 sec.

FLAVERS analyses enables Δ-worklist-DFA versions to have more balanced call chains than for centralized examples. More experiments are needed to check this hypothesis.

At first, it seems surprising that for MHP analysis, the `BitSet`-based worklist-DFA implementation consistently outperformed all Δ-worklist-DFA implementations. We believe that this can be explained by the fact that (1) sets of nodes that represent DFA information in MHP analysis are relatively small and (2) the `BitSet` class is implemented very efficiently. Because sets of nodes are relatively small, there is no significant difference in sizes of sets on which the worklist-DFA and Δ-worklist-DFA versions operate. But because, compared with the worklist-DFA version, the Δ-worklist-DFA version has to maintain additional data structures, the time it spends on maintaining these data structures is more than the time it saves on set operations. In the case of FLAVERS, sets of tuples associated with flow graph nodes are often large and so the Δ-worklist-DFA versions are able to provide savings over the worklist-DFA versions.

Understanding the experiment involving different implementations of FLAVERS is not straightforward. For some example programs, the Δ-observer-DFA version performed better than worklist-DFA implementations with one set of property and constraints but significantly worse with others. Clearly, not only the nature of the flow graph (e.g. centralized vs. decentralized), but also the nature of DFA information affects efficiency of the Δ-observer-DFA version. Understanding these dependencies is left for future work.

5 Conclusion and Future Work

In this paper, we described an object-oriented technique for implementing data flow analyses, by using the well-known Observer pattern. This technique is less centralized than traditional worklist implementations and therefore has better promise for parallelization. This technique is also more natural than the worklist-based ones for implementing data flow analyses in ways that minimize the amount of information that is propagated through the flow graph. We demon-

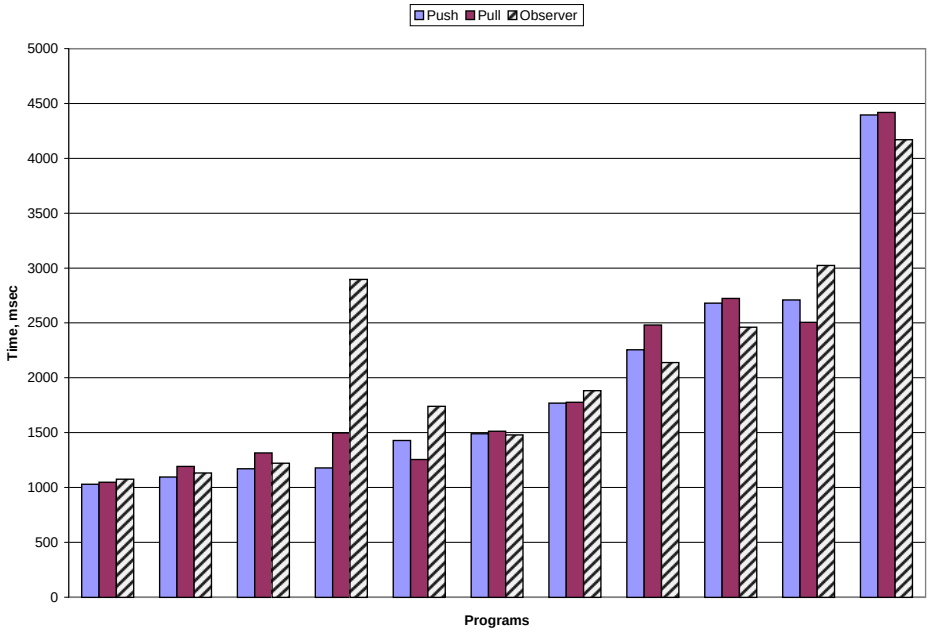

Figure 10: Running times for the three versions of FLAVERS for programs on which the push implementation took between 1 and 5 sec.

strated that even a straightforward single-thread implementation of Observer-based data flow analysis is efficient. An implementation of MHP analysis using the Observer pattern generally outperformed worklist versions of this analysis on small programs. The results were mixed for scalable versions of several examples, with the Observer implementation performing better on some examples and worse on others. With only a few exceptions, an implementation of the FLAVERS analysis using the Observer pattern performed comparably to the worklist versions of this analysis.

We plan to investigate carefully the cases where the Observer implementations performed significantly better or significantly worse than the worklist implementations. It is possible that some general features of flow graphs and data flow information can be identified that determine whether a data flow problem lends itself better to an Observer-based or worklist-based implementation. We will look into hybrid implementations, using both the worklist and Observer-style event notification.

Our future work includes experiments with parallelized implementations of data flow analyses. We will implement a distributed worker [2] version of worklist-based and observer-based data flow analyses and perform an empirical comparison.

Both DFAs in our experiments represent analyses done for software engineering purposes. We plan to experiment with other types of data flow analysis, including those used in compiler optimization, such as constant propagation and alias analysis [1]. We also plan to experiment with different platforms, e.g. using the PROLANGS Analysis Framework [26], a C-based DFA framework.

Finally, we are interested in data flow analyses that use more complex data flow information than sets of values associated with flow graph nodes. For example, points-to analysis [13] computes an approximation of the allocated portion of the heap. Flow-sensitive versions of points-to analysis (e.g. [16, 28]) associate points-to graphs with nodes of the flow graph of the program. It will be interesting to see if an observer-DFA implementation of points-to analysis can be defined in a natural and efficient way.

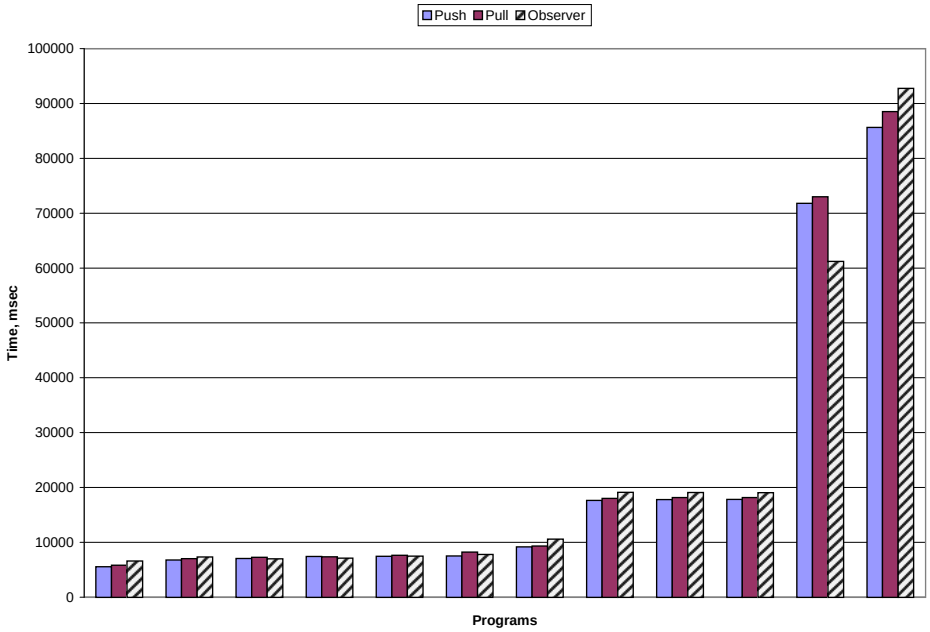

Figure 11: Running times for the three versions of FLAVERS for programs on which the push implementation took more than 5 sec.

References

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

[2] G. R. Andrews. *Concurrent Programming — Principles and Practice*. Benjamin/Cummins Publishing Company Ltd., 1991.

[3] H. Y. Chen, T. H. Tse, and T. Y. Chen. Automatic analysis of consistency between requirements and designs. *IEEE Transactions on Software Engineering*, 27(7), July 2001.

[4] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. The right algorithm at the right time: Comparing data flow analysis algorithms for finite state verification. In *Proceedings of the 23rd International Conference on Software Engeneering*, pages 37–46, May 2001.

[5] J. C. Corbett and G. S. Avrunin. Toward scalable compositional analysis. In *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 53–61, Dec. 1994.

[6] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 62–75, Dec. 1994.

[7] M. B. Dwyer and M. Martin. Practical parallelization : Experience with a complex flow analysis. Technical Report KSU CIS TR 99-4, Kansas State University, 1999.

[8] Y. fong Lee, T. J. Marlowe, and B. G. Ryder. Experiences with a parallel algorithm for data flow analysis. *The Journal of Supercomputing*, 5(2–3):163–188, Oct. 1991.

[9] R. Ford. Concurrent algorithms for real-time memory management. *IEEE Software*, pages 10–23, Sept. 1988.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1994. ISBN 0-201-63361-2.

[11] M. S. Hecht. *Flow Analysis of Computer Programs*. North-Holland, New York, 1977.

[12] D. P. Helmbold and D. C. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, Mar. 1985.

[13] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, June 2001.

[14] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, Jan. 1990.

[15] R. K. Keller, M. Cameron, R. N. Taylor, and D. B. Troup. User interface development and software environments: The Chiron-1 system. In *Proceedings of the 13th International Conference on Software Engineering*, pages 208–218, Oct. 1991.

[16] W. A. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 235–248, June 1992.

[17] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28(2):121–163, 1990.

[18] S. P. Masticola, T. J. Marlowe, and B. G. Ryder. Lattice frameworks for multisource and bidirectional data flow problems. *ACM Transactions of Programming Languages and Systems*, 17(5):777–803, Sept. 1995.

[19] S. P. Masticola and B. G. Ryder. A model of Ada programs for static deadlock detection in polynomial time. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 97–107, May 1991.

[20] S. P. Masticola and B. G. Ryder. Non-concurrency analysis. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 129–138, May 1993.

[21] R. Milner. *A Calculus of Communicating Systems*, volume 92. Springer-Verlag, Berlin, 1980.

[22] G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of the 6th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 24–34, Nov. 1998.

[23] G. Naumovich, G. S. Avrunin, and L. A. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *Proceedings of the 21st International Conference on Software Engineering*, pages 399–410, May 1999.

[24] G. Naumovich, G. S. Avrunin, and L. A. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. In *Proceedings of the joint 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 338–354, Sept. 1999.

[25] K. M. Olender and L. J. Osterweil. Interprocedural static analysis of sequencing constraints. *ACM Transactions on Software Engineering and Methodology*, 1(1):21–52, Jan. 1992.

[26] Rutgers University Programming Languages Research Group. PROLANGS. http://www.prolangs.rutgers.edu/public.html, 1999.

[27] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[28] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming*, pages 187–206, Oct. 1999.

A Timing results for the experiment in Section 4.2.

Figures 12, 13, 14, 15, 16, 17, 18, and 19 show the comparison of the Δ-worklist-DFA and Δ-observer-DFA implementations of MHP analysis on scalable examples introduced in Section 4.2.

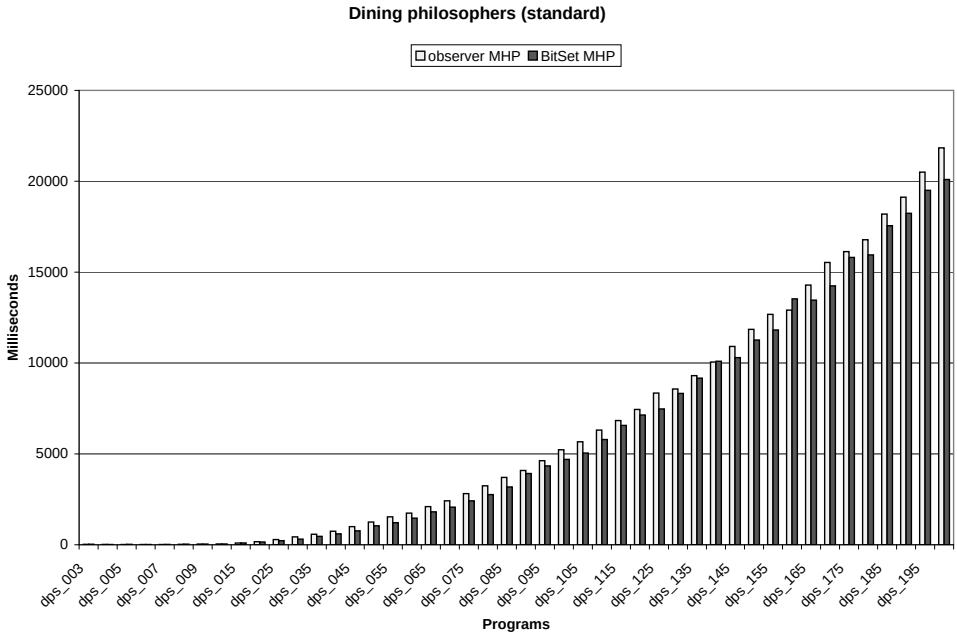

Figure 12: Results for `dps`

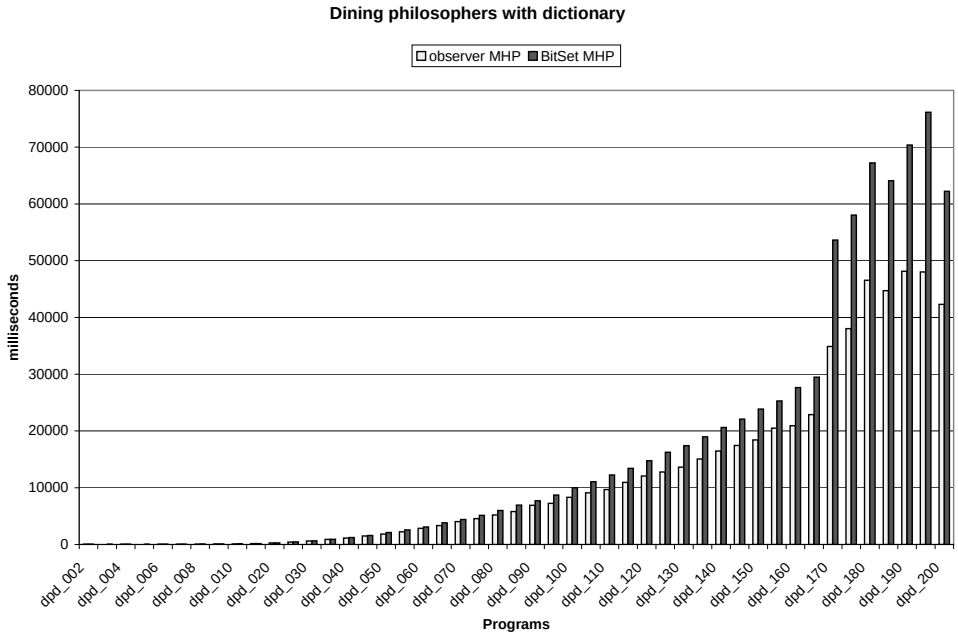

Figure 13: Results for `dpd`

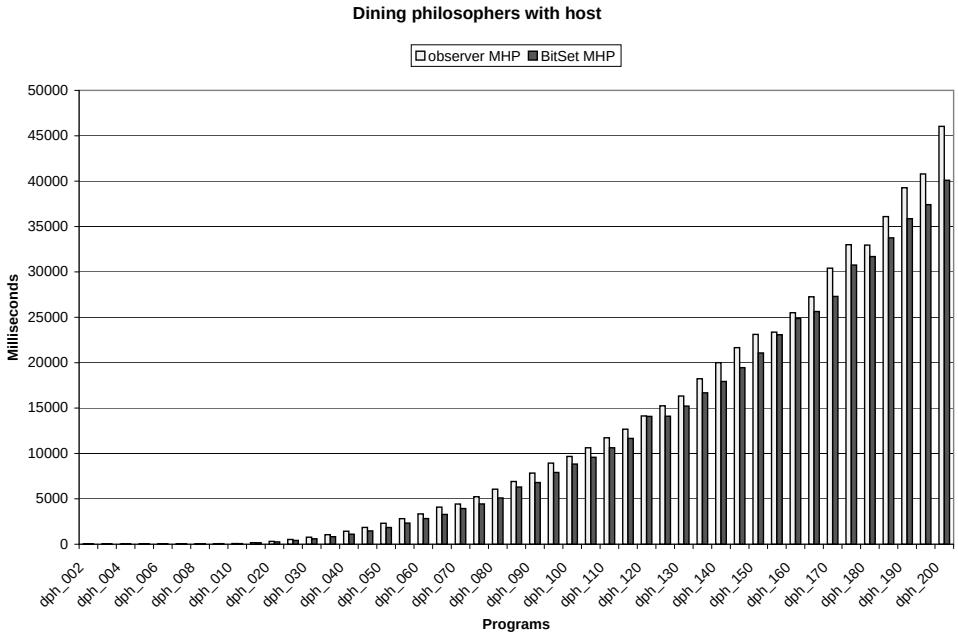

Figure 14: Results for `dph`

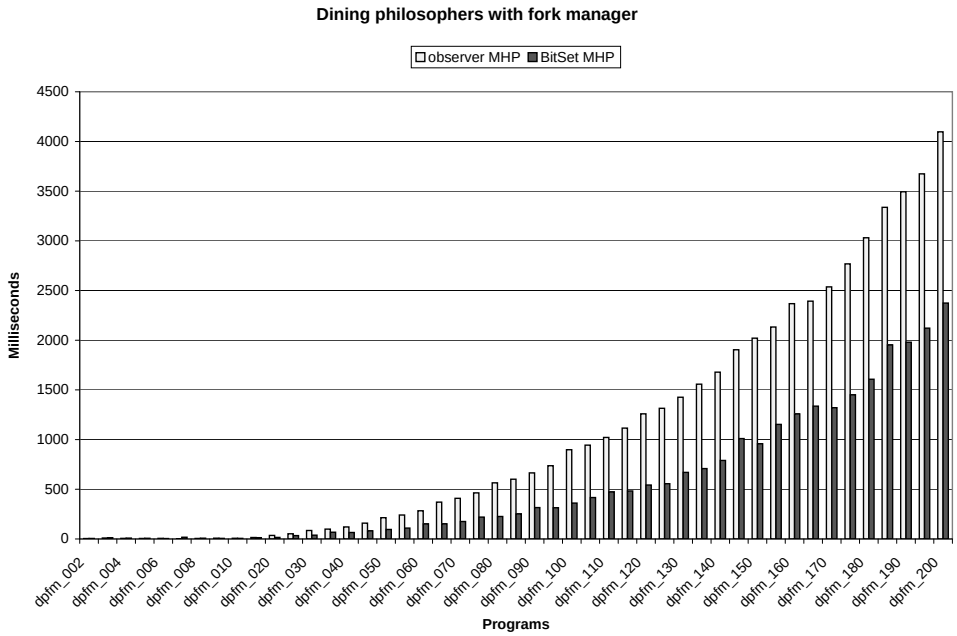

Figure 15: Results for `dpfm`

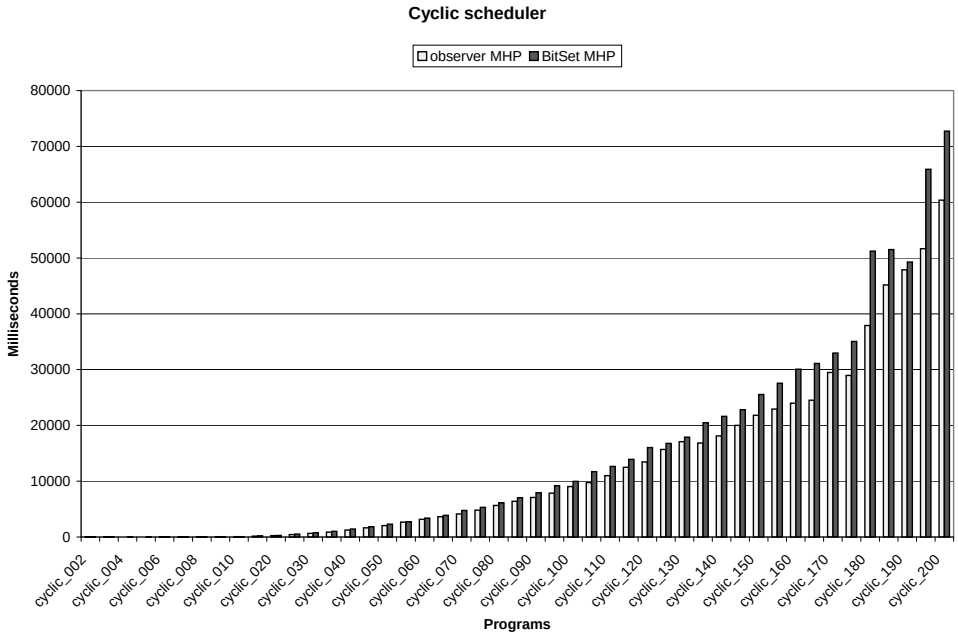

Figure 16: Results for `cyclic`

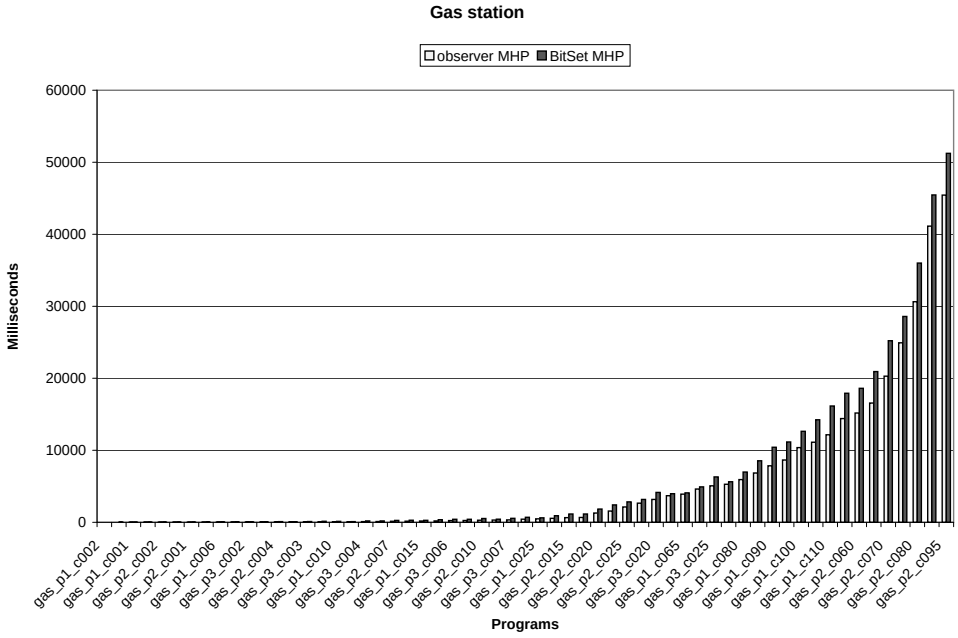

Figure 17: Results for `gas`

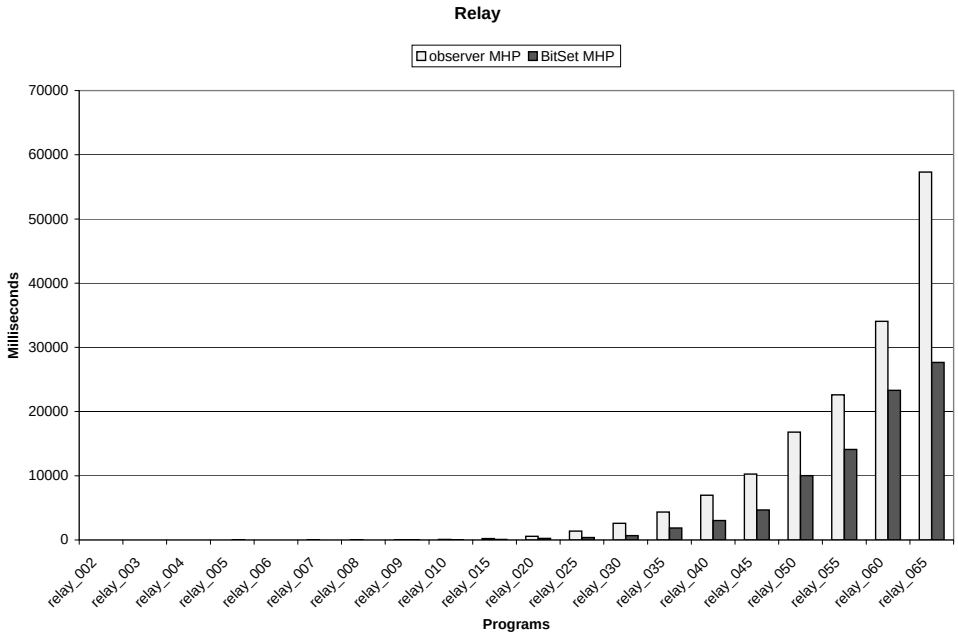

Figure 18: Results for `relay`

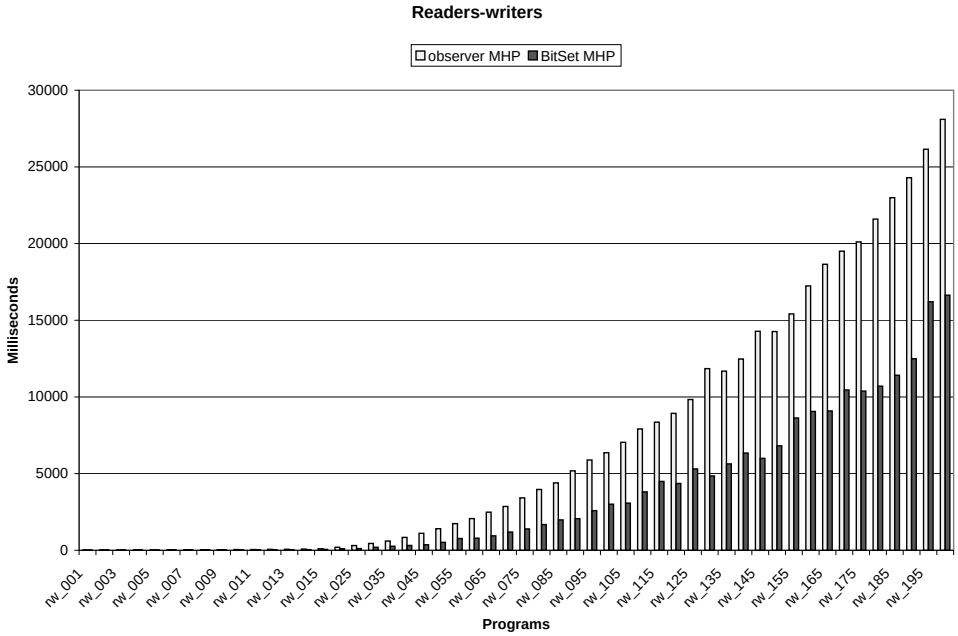

Figure 19: Results for `rw`