Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 176 (2007) 97–108
www.elsevier.com/locate/entcs

Data Flow Analysis as a General Concept for the Transport of Verifiable Program Annotations[1]

Wolfram Amme,[2] Marc-André Möller,[3] Philipp Adler[4]

Institut für Informatik
Friedrich-Schiller-Universität Jena
Jena, Germany

Abstract

Just-in-Time (JIT) compilation is frequently employed in order to speed-up the execution of platform-independent and dynamically extensible mobile code applications. Since the time required for dynamic compilation directly influences a program's execution time, JIT compilers usually utilize only simple and fast techniques for program analysis and optimization. To improve further the analysis and optimization process of such compilers program annotations can be used.
However, mostly all current annotation approaches suffer from the fact that the verification of transmitted program information is time consuming and therefore will not be carried out on the consumer side of a mobile code system. In this paper, we present a verifiable annotation technique that is based on a well known iterative data flow algorithm and which can be used for the transmission of all program information that can be derived through data flow analysis. Preliminary measurements of compilation and verification time indicate that the presented technique seems to be implementable and therefore could be used as an all-purpose transportation technique for safe program annotations.

Keywords: Data Flow Analysis, Program Annotation, Verifikation

1 Introduction

Platform-independent mobile code like Java bytecode often is executed using Just-in-Time compilation. Since mobile code, in particular Java bytecode, is usually unoptimized when it arrives at the runtime system a Just-in-Time (JIT) compiler often applies several analysis and optimization techniques to run the mobile program faster and more efficiently. Nevertheless, since analysis and optimization effort

[1] Partially supported by the DFG under grants AM-150/1-1 and AM-150/1-3.
[2] Email: amme@informatik.uni-jena.de
[3] Email: marcandm@informatik.uni-jena.de
[4] Email: phadler@informatik.uni-jena.de

doi:10.1016/j.entcs.2006.06.019

increases program execution time, most JIT compilers contain only fast and simple analysis methods.

Program annotations have been suggested to improve the code generation or verification process of a JIT compiler. The term program annotation is used as a synonym for code information added to the mobile code during its generation. This information can be used by the consumer side of a mobile system to speed-up optimizations or increase security of a given program. The main challenge after transferring mobile code to the runtime environment is the verification of the transmitted annotations. Since annotations are additional information which are derived from the program and do not belong to the underlying mobile code the verification of program annotations is complicated. Therefore, in most projects program annotations are assumed to be sound [7,13] and will not come under further scrutiny. However, if the code consumer is relying on the annotations but cannot prove their correctness, semantically incorrect transformation of the program code can occur and harmful behavior could be the result.

In this paper a verifiable program annotation technique is presented which conceptually is based on data flow analysis and that can be used for the transmission of program information which can be modelled through a data flow framework. Our technique derives information of a program on the producer side making use of a well-known general iterative data flow algorithm. Upon completion of an analysis, our algorithm adds parts of the derived data flow information called annotation points to the transmitted code format. On the consumer side the full data flow information is reconstructed from the transmitted annotation points by using a modified version of the same general data flow algorithm that has been used on the producer side. Non-accurate transport of program annotations like manipulation can be detected from this algorithm by the fact that a fixpoint of the considered data flow problem cannot be reconstructed from its annotation points.

In particular, the technique presented in this work has been developed for the annotation of SafeTSA programs. SafeTSA [1,2] is a safe mobile code format designed as an alternative to the Java Virtual Machine's bytecode language (JVML). SafeTSA safely and compactly represents programs in Static Single Assignment Form (SSA Form [4]) using novel encoding techniques [12]. The use of SSA Form simplifies the verification and code generation process and also allows for the natural and efficient application of producer-side platform-independent optimizations. As proof of concept, we have developed an entire system for the transport of SafeTSA programs that can be used for programs written in *Java* and for *IA32* and *PowerPC* target architectures.

The paper is structured as follows: Section 2 gives a brief introduction into monotone data flow analysis, and Section 3 describes the conceptual functioning of our program annotation technique. Implementation details and results are described in section 4. In Section 5, we discuss related work and Section 6 concludes with a summary.

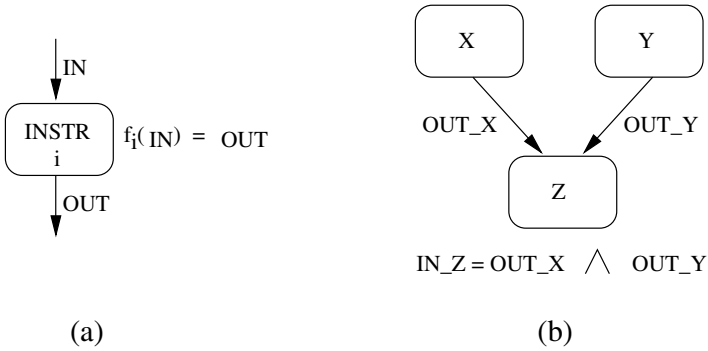

Fig. 1. (a) Effect of semantic functions and (b) the meet operator.

2 Monotone Data Flow Analysis

A data flow analysis gathers information for each instruction by iteratively propagating local computed data flow information through the control flow graph of a program. In principle, each data flow problem can be modelled using a data flow framework $(L, \wedge, F)$, where L is called the *data flow information set*, $\wedge$ is the *meet operator* (sometimes called the *union operator* in the literature), and F is the set of *semantic functions*.

The data flow information set of a data flow framework is a conceptual universe of objects upon which the analysis is working. A semantic function f_i corresponds directly to an instruction in the program and models the effect that an execution of the instruction has onto the incoming data flow information (see Figure 1 a). The meet operator implements joining paths in the control flow graph. Figure 1 b describes the function of the meet operator for a node of a control flow graph that can be reached from its two predecessor nodes. In the example the two data flow information items OUT_X and OUT_Y, each coming from different program paths, are merged into dataflow information IN_Z.

Figure 2 shows a general iterative algorithm that always terminates and yields the least fixpoint of a data flow framework if and only if the semantic functions are monotone [5] and $(L, \wedge)$ forms a bounded semi-lattice with a one element *1* and a zero element *0* [14]. Data flow frameworks that comply with these requirements of correctness are called monotone data flow frameworks (MDF). In the initial phase of the algorithm each instruction other than the start node is assigned to an outgoing data flow information that corresponds to the one element of the semi-lattice. The start node s of a method is assigned a special element $NULL$ that stands for the information that arrives at the start node from the different call points of the method. For intraprocedural analysis, that is what we are interested in, $NULL$ stands for no incoming information and therefore can be represented depending on the considered data problem by the one or zero element of the semi-lattice. In the iteration phase the algorithm derives for each instruction successively the outgoing data flow information from its direct predecessor nodes. The algorithm terminates and yields as a

[5] A semantic function f is called monotone iff for each a,b $\in L$ holds $f(a \wedge b) \leq f(a) \wedge f(b)$, whereby $\leq$ is the ordinary partial order given through the semi-lattice: $a \leq b \leftrightarrow a \wedge b = a$ for each a,b $\in L$.

```
1   OUT(s) := NULL
2   for every n ∈ N − {s}
3      do OUT(n):= 1 end for
4   do
5      stable := true
6      for every n ∈ N − {s} do
7         IN(n) := ⋀ OUT(pred(n))
8         NEW := f_n(IN(n))
9         if NEW ≠ OUT(n) then
10           OUT(n) := NEW
11           stable := false
12        end if
13     end for
14  while (!stable)
```

Fig. 2. The General Iterative Algorithm

result a safe solution of the data flow framework, if for each instruction no further data flow information can be derived.

3 Monotone Data Flow Analysis as a Basic for Verifiable Program Annotations

The simplest way for the construction of verifiable program annotations is to annotate each instruction with the data flow information item that the general iterative algorithm has derived for this instruction. Since the data flow information that is assigned to the start node of a program is known for each special data flow problem, on the consumer side the transmitted annotations can be verified by applying the general iterative data flow algorithm to these annotations. If the application of the algorithm is stable after one performed iteration, the program annotations represent a fixpoint of the corresponding monotone data flow framework and therefore must be identical to those which have been added to the mobile code format at the producer side. In contrast, if the application of the algorithm will lead to the execution of a further iteration, the annotations must have been modified during the transmission process.

3.1 Selection of Annotation Points

However, annotating each instruction with its corresponding data flow information item will result in unacceptable file sizes. To avoid such a large increase of transported program code in our approach only data flow information items will be added to those instructions which are essential for restoring the complete result of the considered data flow analysis.

Instructions that are essential for the restoration of the entire data flow information are called annotation points in our methodology. Depending on an efficient verification of transmitted program annotations, the selection of annotation points should guarantee that the restoration of data flow results could be integrated into the verification process during a single iteration of the general algorithm. In SafeTSA, which actually can be considered as an high level intermediate representation that prevents the appearance of irreducible control flow graphs, the requirement of a fast restoration and verification pass can be accomplished by the exclusive annotation of instructions with outgoing backward edges.

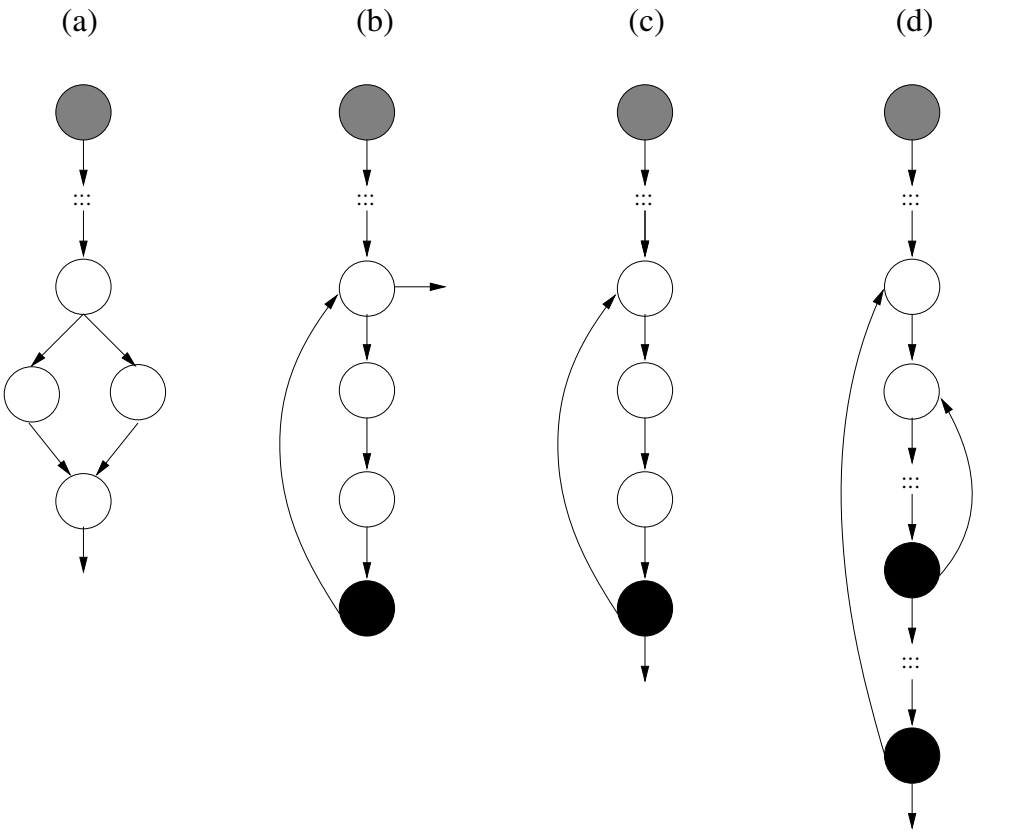

Fig. 3. Program (a) without a loop, (b) with a while loop, (c) with a repeat loop, and (d) with a nested loop.

Figure 3 shows the position of annotation points in the control flow graph for some sample programs. In the figure black colored nodes stand for annotation points and gray shaded nodes denote the start node of a control flow graph. For a program without a loop (Figure 3 a) data flow information items can be derived for each instruction during one iteration of the general algorithm, therefore the use of annotation points for such kind of programs is not necessary. In contrast, for programs that contain one or more loops (Figure 3 b, c and d) each instruction which has an outgoing edge to the entry point of a loop is becoming an annotation point.

3.2 Restoring and verifying data flow information

For the restoration of data flow information items first the start node of the control flow graph is set to *NULL*. Afterwards, comparable with the functioning of the general algorithm, the data flow information for each instruction will be iteratively calculated form its predecessor nodes. For assuring that the restoration of data flow information can be performed in one pass during the recovering process the nodes of the control flow graph will be traversed in reversal postorder. The use of this traversal order guarantees that always when an entry node of a loop is

```
1    OUT(s) := NULL
2       for every n ∈ N − {s} in rpostorder do
3           IN(n) := ⋀ OUT(pred(n))
4           NEW := f_n(IN(n))
5           if  n is an annotation point then
6             if(NEW ≠ OUT(n))
7                  stop
8           end if
9       end for
```

Fig. 4. Data Flow Algorithm based on Annotation Points

reached, the data flow information for the instructions inside the loop will be derived before data flow information is passed outside of the loop. If during the restoration process for each annotation point exactly the same data flow information item as the annotated item is calculated a fixpoint of the data flow framework must be derived and therefore the data flow information items restored from the annotation points at least stand for a safe solution of the data flow problem.

Figure 4 shows a modified version of the general iterative algorithm that is used in our approach for the restoration of data flow information and verification of annotation points. As an important difference to the general iterative algorithm, there is no enclosing `do-while` loop present in this algorithm, so it needs exactly one iteration for the recovering and verification process. Beginning with the start node for each instruction data flow information items are calculated in reversal postorder. This guarantees that always when the algorithm enters a loop, data flow information items within the loop body are determined until the annotation point of the loop is reached. At this point the derived data flow information is compared with the annotated ones. If they do not match, all annotations will be considered incorrect and the algorithm stops. If these comparisons yields true for all annotation points, both the annotations and all other data flow information items can be considered to be correct, i.e. that the data flow information items delivered by the algorithm build a safe approximation of the data flow problem.

In order to prove the correctness of our algorithm we establish the following theorem, which points out that if in the algorithm the correctness of an annotation point for a loop with incoming information K can be verified, then the item added to the annotation point must always be a safe approximation of the information delivered by the general iterative algorithm in the same situation.

Theorem 3.1 *Let $(L, \wedge, F)$ be a MDF, p an arbitrary loop, $f_1, \ldots, f_n \in F$ the semantic functions assigned to the instructions of p, Ap the annotation point of p and $A \in L$ the item that has been annotated to Ap. Then for each $K \in L$ it holds*

$$f_1 \circ \ldots \circ f_n(K \wedge A) = A \Longrightarrow A \leq f^{fix}(K), \text{ where}$$

$f^{fix}(K)$ stands for the item that would be derived for Ap applying the general iterative algorithm on p with incoming data flow information K.

Proof. Since $f_1 \ldots f_n$ are monotone, we replace $f_1 \circ \ldots \circ f_n$ by $f_{1 \cdots n}$, because the property of monotony also holds under function composition. From the monotonic-

ity property of $f_{1\cdots n}$ we obtain

$$f_{1\cdots n}(K \wedge A) \leq f_{1\cdots n}(K) \wedge f_{1\cdots n}(A)$$

Our assumption $f_{1\cdots n}(K \wedge A) = A$ leads to

$$A \leq f_{1\cdots n}(K) \wedge f_{1\cdots n}(A)$$

and therefore $A \leq f_{1\cdots n}(K)$
By adding K to both sides of the inequality, we get:

$$A \wedge K \leq K \wedge f_{1\cdots n}(K)$$

Since $f_{1\cdots n}$ is monotone we can expand the inequality to

$$\underbrace{f_{1\cdots n}(A \wedge K)}_{A} \leq f_{1\cdots n}(K \wedge f_{1\cdots n}(K))$$

Repeating the last two steps arbitrary often we get $A \leq f^{fix}(K)$. □

In the case that all annotation points have been accepted by our algorithm, correctness of data flow information items restored from the algorithm can be proved by a successive application of theorem 3.1. As a start, programs that can be handled by our algorithm shall be restricted to those which contain only loops without further inner loops. As shown in figure 5 a our algorithm first propagates the data flow information associated with the start node down to the first loop entry. Since K_i can be viewed as correct, because of its origin at the start node, theorem 3.1 yields that the information which is leaving the loop must be a safe approximation of the information delivered from the general iterative algorithm applied on input K_i. After verifying an annotation point in that way, its outcome, which now can be viewed as safe, will be propagated through the control flow graph until the next loop is reached. Thus, successive reapplications of theorem 3.1 eventually will show the correctness of the algorithm.

For a general argumentation inner loops are modelled using monotone functions. If a loop is nested, as shown in figure 5 b and c, the inner loop is considered as a function g. For a correct conclusion it has to be guaranteed that g preserves the semantic of the underlying loop. As before, for this purpose theorem 3.1 can be applied. Since the assumption $f_1 \circ \ldots \circ f_n(K_j \wedge A_j) = A_j$ particularly holds for inner loops it can be concluded that A_j is an approximation of the result which the general iterative algorithm would derive for the annotation point of the inner loop on input K_j. For this reason in our algorithm the outcome of a semantic function, that stands for the effect that the execution of an inner loop has onto data flow information, always can be considered as a safe approximation of the original data flow information. As a consequence, the restoration process performed by our algorithm delivers a safe solution even in the presence of inner loops.

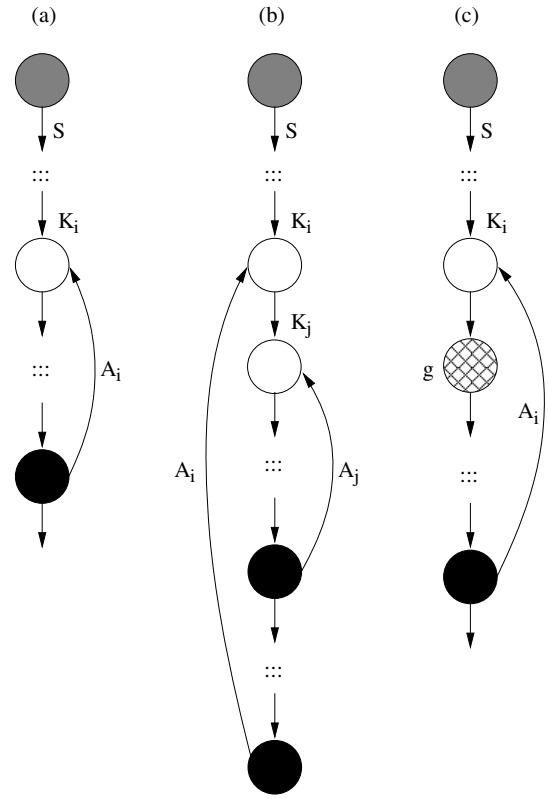

Fig. 5. (a) normal loop, (b) nested loop, (c) nested loop substituted by function g

4 Implementation and Preliminary Results

The presented concept of program annotation has been completely implemented into the *SafeTSA* system. In doing so to SafeTSA's producer side was added a general data flow algorithm that can be used for annotating SafeTSA programs with annotation points. Furthermore, a modified version of this data flow algorithm has been integrated into the consumer side of the system for performing the necessary restoration and verification process.

For hiding implementation details the interface *DataFlowInformation (Dfi)* has been constructed. This interface must be implemented respectively enlarged by a user on both sides of the SafeTSA system by adding appropriate fields and objects representing the data flow information of the considered data flow framework. Figure 6 depicts parts of the methods defined in *Dfi* and its invocation points in the general iterative algorithm on the producer side. To use the data flow algorithm for annotation purposes, the interface defines the methods *code* and *decode*. On the producer side *code* must be implemented as a method that is encoding a data flow information item to an annotation-adequate representation, while its counterpart must be implemented as a method that converts it back to its internal representation on the consumer side.

In order to empirically assess whether the presented annotation technique deliv-

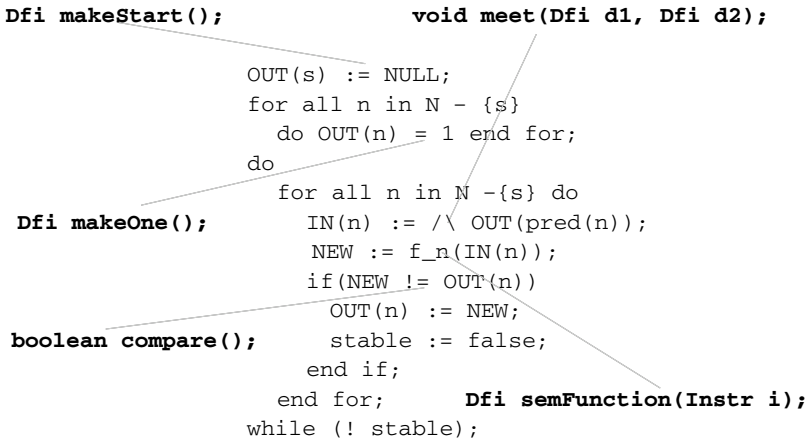

Fig. 6. Methods given by the interface `Dfi` and their invocation by the General Iterative Algorithm.

Benchmark	Number of instructions		Analysis time (sec)		File size
	total	annotation points	normal	annotation points	(Δ%)
Moldyn	1931	26	0,318	0,220	4,17
Euler	8709	46	3,370	2,016	0,65
Montecarlo	2371	12	0,502	0,254	0,12
RayTracer	1661	9	0,118	0,067	0,39
Search	1116	24	0,272	0,157	0,41

Fig. 7. Results of measurements.

ers the expected performance benefits, we built implementations of the interface *Dfi*, both on SafeTSA's producer and consumer sides, that can be used for annotation of dominance frontier information. The dominance frontier computes for each node n of a given control flow graph the nodes which are dominating [6] n. We ran a series of benchmarks in which we particularly compared the compilation times required for an ordinary data flow analysis and one that is based on our program annotation technique.

All results discussed in the following were obtained by running the benchmark programs from section 3 of the Java Grande Forum Sequential Benchmarks (JGF) [10]. The Java Grande Benchmarks were chosen because they were freely available in source code and seemed appropriate for measuring compilation of annotated and non-annotated programs. In order to get stable results from one run to the next we repeated each benchmark execution several times and took the best result.

Figure 7 depicts the number of annotation points, absolute compilation times

[6] A node k of a control flow graph dominates a node n iff on all paths beginning with the start node and ending in n an execution of k will be performed.

and increase in file sizes that we measured for the application of ordinary and annotation-based data flow analysis to the benchmark programs. Surprisingly, the measurements indicate that the number of annotation points that are needed for the restoration of data flow information is comparatively small compared to the total number of program instructions. As an example for the benchmark *MonteCarlo* only 12 annotation points out of 2371 instructions are needed. For an estimation of the additional file size overhead inserted through annotation points, we created annotated and non-annotated versions of the benchmark programs. The comparison of these differently encoded SafeTSA classes points out that the increase of file sizes due to the annotations in most of all cases is minimal and therefore negligible. Actually, the measurements showed that only for the benchmark *Moldyn* the file size for the annotated version increases observable (4,17%). For all other benchmarks the file size increment was between 0.12% to 0.65% in respect to non-annotated benchmark versions. Compilation times required for a determination of dominance frontiers that is based on program annotations already include the additional loading time that is needed to read in the data flow items added to the annotation points. The measurements show that the application of data flow analysis based on program annotations consistently resulted in considerable performance gains. For the benchmark program *Euler* our annotation algorithm ran even 1.354 sec (or 40%) faster than for the non-annotated version.

5 Related Work

Common use of JIT compilers lead to increased research for algorithms to speed-up code generation and program execution for mobile code. Usage of annotation techniques are a promising way to source out great parts of optimization work to the code producer. The gained information can be used by code consumers to reduce the overhead and time consumption of dynamically employed optimizations. Hence, it is not surprising that the first publications in this domain deal with program optimizations. In [3] and [8] annotation frameworks are introduced where the JIT compiler utilizes annotations generated by the Java front-end. These annotations carry information concerning optimizations. Thus, high-performance native code can be produced without performing costly analysis and transformations. One major problem with this approach is that these annotations are not verifiable and assumed to be sound. Nearly all other available annotation techniques like those presented in [7], [9] and [13] have this shortcoming, too. Annotations are simply transported as code attributes or similar without protection against manipulation. If a code consumer relies on provided but manipulated information semantically incorrect transformations may occur which can result in serious security issues. For example while transmitting information for bound check removal [13] or interprocedural side-effect optimizations [9] malicious code could lead to the elimination of checks which would fail in an unoptimized version. In the worst case computer attacks and data loss may be the consequence. With other annotations the threat is not as high as mentioned before. While transmitting helpful information for

program improvements like virtual register assignment [7] manipulation results in worse runtime behavior but otherwise program semantic is not altered. Nevertheless, manipulations made in this way are usable for denial-of-service attacks and other similar threats.

To reduce the overhead for executing mobile programs a reduction in verification time especially for constrained devices is appropriate. In [11] the verification process is split-up into two parts, one performed at the code producer and the other at the code consumer. At the code producer verification information is constructed and transmitted as a verification certificate, which can be understand as an annotation, together with the mobile code to the consumer. There, a lightweight verification consisting of a check of code and certificate is done, requiring less time and space than the normal verification algorithm. By construction these verification certificates are tamper-proof and verifiable but the technique is hard to generalize and difficult to apply to other domains.

Some newer research introduced in [5] and [6] specify a technique which transports annotations containing escape-analysis results in a safe and verifiable manner. The idea is to extend the underlying type system by an additional dimension representing "capturedness", the property if a reference escapes or not or possibly escapes. This annotation procedure is tamper-proof. If the state is changed to escape or possibly escapes, nothing more than an optimization chance is lost. The other way round the change of the type produces an erroneous program which is rejected. This annotation technique may be extended to other program information which can be represented as a type in the underlying type system. For annotations that can not be described as another type this procedure is not feasible.

6 Conclusion

Program annotations have been suggested to improve the code generation or verification process of a JIT compiler. Since annotations are additional information which are derived from the program and do not belong to the underlying mobile code the verification of program annotations is complicated and therefore often will not be carried out on the consumer side of a mobile code system.

In the paper we have introduced a general concept for the transport of safe and verifiable program annotations. Our method is adding parts of the result of a data flow analysis, that has been performed on the producer side, to the mobile code representation of a program. On the consumer side the entire results of the data flow analysis will be safely restored from the annotation points of the given program. Measurements that have been performed for the determination of dominance frontiers show that the space required for annotations is negligible and that compilation time is much faster than performing an ordinary data flow analysis at runtime.

Since the dominance frontier is a comparatively simple data flow problem the results in this paper should be considered as preliminary. Therefore, in future work we will concentrate on more complex data flow problems. On the one hand

the runtime of the general iterative algorithm would increase because of higher complexity of semantic functions, meet- and compare operators. On the other hand, we expect a higher workload for loading and decoding the annotations, because annotations of complicated data flow problems are more complex. However, for the expected case that the time overhead introduced through loading and decoding will comprise further a small part of the overall compilation time, our algorithm also should outperform an ordinary data flow analysis when using more complicated applications.

References

[1] Amme, W., N. Dalton, M. Franz and J. von Ronne, *SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form*, in: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'2001)*, ACM SIGPLAN Notices **36** (2001), pp. 137–147.

[2] Amme, W., J. von Ronne and M. Franz, *Quantifying the benefits of ssa-based mobile code*, in: *Proceedings of the International Workshop on Compiler Optimization Meets Compiler Verification (COCV'2005)*, Edinburgh, Scotland, 2005, pp. 101–115.

[3] Azevedo, A., A. Nicolau and J. Hummel, *Java annotation-aware just-in-time (AJIT) compilation system*, in: *Proceedings of the Conference on Java Grande (JAVA'1999)*, 1999, pp. 142–151.

[4] Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, *Efficiently computing static single assignment form and the control dependence graph*, ACM Transactions on Programming Languages and Systems **13** (1991), pp. 451–490.

[5] Franz, M., C. Krintz, V. Haldar and C. H. Stork, *Tamper proof annotations*, Technical Report 02-10, Department of Information and Computer Science, University of California, Irvine (2002).

[6] Hartmann, A., W. Amme, J. von Ronne and M. Franz, *Code annotation for safe and efficient dynamic object resolution*, in: J. Knoop and W. Zimmermann, editors, *Proceedings of the 2nd International Workshop on Compiler Optimization Meets Compiler Verification (COCV'2003)*, Warsaw, Poland, 2003, pp. 18–32.

[7] Jones, J. and S. N. Kamin, *Annotating Java class files with virtual registers for performance*, Concurrency: Practice and Experience **12** (2000), pp. 389–406.

[8] Krintz, C. and B. Calder, *Using annotations to reduce dynamic optimization time*, in: *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'2001)*, ACM SIGPLAN Notices **36.5** (2001), pp. 156–167.

[9] Le, A., O. Lhoták and L. J. Hendren, *Using inter-procedural side-effect information in JIT optimizations*, in: *Proceedings of the International Conference on Compiler Construction (CC'2005)*, 2005, pp. 287–304.

[10] Mathew, J. A., P. D. Coddington and K. A. Hawick, *Analysis and development of Java Grande Benchmarks*, in: *Proceedings of the Conference on Java Grande (JAVA'1999)* (1999), pp. 72–80.

[11] Rose, E. and K. H. Rose, *Lightweight bytecode verification*, in: *Proceedings of the Workshop on Formal Underpinnings of the Java Paradigm (OOPSLA'1998)*, 1998, p. 1.

[12] von Ronne, J., "A Safe and Efficient Machine-Independent Code Transportation Format Based on Static Single Assignment Form and Applied to Just-In-Time Compilation," Ph.D. thesis, University of California, Irvine, USA (2005).

[13] Yessick, D. E., *Removal of bounds checks in an annotation aware JVM* (2004).

[14] Zima, H. P. and B. Chapman, "Supercompilers for Parallel and Vector Computers," ACM Press Frontier Series, Addison-Wesley (ACM), Menlo Park, CA (New York), 1991, $39.95.