2014 14th International Conference on Computational Science and Its Applications

Certified Transformation for Static Single Assignment of SPMD Programs

Mohamed A. El-Zawawy*
*College of Computer and Information Sciences,
Al Imam Mohammad Ibn Saud Islamic University (IMSIU)
Riyadh, Kingdom of Saudi Arabia
*Department of Mathematics, Faculty of Science
Cairo University
Giza 12613, Egypt
Email: maelzawawy@cu.edu.eg

***Abstract*—A common program view adapted by most contemporary compilers is Static Single Assignment (SSA) which can be realized as transitional form (TF). In SSA, each variable is modified by exactly one assignment. SSA is based on splitting variables of the original program into many versions. Power constraints of sequential programming led parallel programming to be the main programming style today for performance boosting. The single program, multiple data (SPMD) style of parallelism is a prevalent model of parallel computing. A Proof-Carrying Code package typically consists of the original code, a proof that is checkable by a machine, and the code's correctness specification.**

A new technique for constructing a static single assignment form for SPMD programs is introduced in this paper. The proposed technique is in the form of a system of inference rules. The input of our proposed technique is a SPMD program and the output is a SSA form of the program. Judgment derivations in the proposed system are convenient choices for proof parts of a PCC packages. Therefore the resulting SSA form is certified in terms of proof-carrying code area of research.

I. INTRODUCTION

Static Single Assignment (SSA) [5], [32] is a common program view adapted by most contemporary compilers. Static Single Assignment is also considered a tool for compiler design. SSA forms can be realized as transitional form (TF) in which exactly one assignment modifies each variable. The main idea of building SSA forms is the splitting of variables of the original program into many versions. This is implemented via indicating new variables with the original names and new subscriptions. This style allows each definition to have its own history. Therefore, in the SSA form, use-definition sequences are clearly trackable. SSA forms are even adopted in phases of final-code generation. Compilers use SSA representations include LLVM, LibFirm, Java HotSpot, and LAO. Just-in-time compilers (with relatively longer compilation time) also benefit from advantages of SSA. Examples of Just-in-time compilers are LAO, Java HotSpot, and Mono.

Although historically, SSA main function was to enable building transformations of high-level program, SSA forms have attracted researcher attentions due to good characteristics (reducing computational complexities and enabling simple algorithms) that these forms enjoy. Based on SSA, different important program analyses and optimizations (like dead-code removal, pointer analysis, constant propagation, memory safety, and live-range analysis) have been studied and designed. Therefore this paper focuses on SSA forms.

Parallel programming [29] is a main programming style today for performance boosting due to power constraints of sequential programming. There are considerable intersections between challenges of parallel programming and that of large-scale machines programming. One of common challenges is that of involved assignments; distributed computers are organized in a hierarchical style. Of course, different types of communication charges among applications and distributed machines are allowed. This even complicates parallel-programs analysis in presence of involved assignments.

One prevalent model of parallel computing is that of single program, multiple data (SPMD) [25], [35] style of parallelism. The strength of this style comes from its ability to mix global simultaneity and cumulative communication operations with separate execution threads. There are many advantages of SPMD compared to relevant models. Efficient programming that is free of many common parallel errors is possible with SPMD which is as well a convenient style for constructions and maintenance of compiler optimization and analysis. The execution style of SPMD is typically locality-aware. This makes it straightforward to programmer to control data locality. Hence SPMD programs are typically scalable for large-scale computers. Therefore SPMD programs are the main objects of interest in this paper.

The Proof-Carrying Code (PCC) [36], [28] has a great impact on certifying compilers, theorem provers, and program justification tools. This is so as by the aid of PCC, it is now possible to construct softwares that are more trustworthy. A PCC package typically consists of the original code, a proof that is checkable by a machine, and the code's correctness specification. The proof is supposed to ensure that the code behavior is in line with the code specification. Therefore, before executing the code, the user of the code can employ a simple checker to test the safety proof. The technique used in this paper to construct the SSA form produces such proofs used in PCC.

This paper presents a new technique for constructing a static single assignment form for SPMD programs. The input of our proposed technique is a SPMD program and the output is a SSA form of the program. However the new technique has

978-1-4799-4264-0/14 $31.00 © 2014 IEEE
DOI 10.1109/ICCSA.2014.15

the form of a system of inference rules. This hence is enabling associating each produced SSA form of our system with a justification-proof for the correctness of the SSA form. This makes the SSA form certified. Applications like proof-carrying code are basically built on the use of certified transformations. Therefore results of the technique proposed in this paper are expected to have wide range of applications in proof-carrying code area of research.

Motivation

The paper is motivated by the need for an SSA-transformation technique for SPMD programs that produces machine-checkable and easily-transferable justification-proofs for each SSA-transformation towards an SSA-form for a given SPMD program.

Contributions

The contribution of this paper is a new technique for producing Static Single Assignment of SPMD programs. the technique has the form of a system of inference rules and hence its results are applicable to methods of Proof-Carrying Code.

Paper Outline

The rest of the paper is outlined as following. The langauge model, *SSA-ParLang*, used in the paper is shown in Section II which also introduces the details of the new technique proposed by this paper. A conclusion to the paper as well as suggestions for future work are introduced in Section IV.

II. CERTIFIED STATIC SINGLE ASSIGNMENT

This section presents a new technique for building Static Single Assignment (SSA) forms for single program multiply data (SPMD) programs. The proposed technique is a transformation one that transforms a given SPMD (in a classical form) into an equivalent program in a SSA form. The transformation has the from of annotations to the original program statements. The proposed technique is in the form of a type system (system of inference rules and set of types). Elements of the proposed system are the main contribution of this paper.

The model langauge, *SSA-ParLang*, used in this paper to present the new technique is shown in Figure 1. There are two syntactical categories of statements; *Stmt* and *AnnotStmt*. The former category is intended to capture the SPMD in its original form and the later category is meant to capture the SSA form produced by the technique presented in this paper. Because our transformation technique is an annotating one, the definition of the category *AnnotStmt* includes that of *Stmt*. Therefore the set of programs produced by *Stmt* is a subset of the set of programs produced by *AnnotStmt*.

Our technique inserts three different types of statements in the form of annotations to the original program. The first type of annotation is that uses the statement $x_i := fi(x_j, x_k)$. This statement is added to preserve single assignments at the junction points of control flow graphs (CFGs). The statement $x_i := fi(x_j, x_k)$ can be realized as pseudo assignment. While the symbol x_i stands for the new version of the variable x, the symbols x_j, x_k stand for original versions of the variable which are needed up to the junction point. The implementation of *fi* annotation is typically achieved in two steps. the first step is to insert the fi statement and then to do the variable subscribing. Typically the fi statements are placed on control borderlines of the CFG. Then the variable order of appearance in the the dominator tree determines their subscribes.

The other two types of statement annotations are related to operations of indirect memory-access which complicates the process of maintaining the use-definition chains. The source of complication is that aliasing pointers of scalar variables may create definitions. Therefore for the SSA technique, symbolic investigation would not be enough to determine single definition sites. In this paper we treat this problem in a way inspired by [21] and [11]. However our solution improves over similar approaches from many points of view including (a) refining places of insertions for new statements, (b) producing proofs (in the form of type derivations for each transformation process), and (c) employing the well-established concepts of typing theory to determine variables subscripts in a systematic way. We use two annotate statements to treat indirect memory-access; $x_i := md(x_j)$ and $mu(x_j)$. Details of conditions and locations concerning the insertion of these annotations depend on the type of the indirect memory-access statement. This is evident in inference rules of Figure 5 and 6.

As stressed earlier in many occasions, the approach we present has the form of a type system consisting of a set of SSA-types and a set of inference rules. The set of the types (SSA-types) are defined as follows.

Definition 1: A SSA-type is a map $T : \cup_{m \in \mathcal{M}} \mathrm{lVar}_m \rightharpoonup$ Integers.

A SSA-type is a partial map from the set of local variables of all machines to the set of integers. The set of inference rules are introduced in Figures 2, 3, 4, and 6. The proposed type system is syntax-directed. Therefore each syntactical category of the language syntax, *SSA-ParLang*, corresponds to a set of inference rules. Our system assumes that the input program is annotated with pointer information. This can be calculated using any of common algorithms for flow-sensitive pointer analysis. The pointer-analysis annotations have the forms of $S : P \rightarrow P'$ and $e : P \rightarrow A$, where P and P' denote the pre and post pointer types and A is the set of variables that e may alias to in a state of type P.

Judgements produced by the system have the forms $e : T \rightarrow T' \hookrightarrow e'$ and $\mathit{Stmt} \ni S : T \rightarrow T' \hookrightarrow S' \in \mathit{AnnotStmt}$. In these Judgement e' and S' are the transformations (annotated versions) of e and S, respectively. The former judgment reads as follows; (a) evaluating e at a state of type T (if ends) reaches a state of type T' and (b) executing e' at a state equivalent to that of executing e reaches (if ends) a state equivalent to that e reaches (if ends) at the end of its execution. Similarly $S : T \rightarrow T' \hookrightarrow S'$ reads as follows; (a) executing S at a state of type T (if ends) reaches a state of type T' and (b) executing S' at a state equivalent to that of executing S reaches (if ends) a state equivalent to that S reaches at the end of its execution. This can be formalized in the following theorem that assumes an appropriate operational semantics for constructs of *SSA-ParLang*.

Theorem 1: 1) Suppose that $e : T \rightarrow T' \hookrightarrow e'$. Suppose also the existence of an appropriate opera-

$$
\begin{array}{rcl}
 & & x \in \text{lVar},\ i_{op} \in I_{op},\ b_{op} \in B_{op},\ \text{and } m \in M \subseteq \mathcal{M} \\
V \in \text{VarDefs} & ::= & \text{int } x \mid d_1 d_2 \mid \epsilon. \\
l \in \text{Loc} & ::= & x \mid l \to y \mid [l]. \\
e \in \text{DistExpr} & ::= & l \mid e_1\ i_{op}\ e_2 \mid \&l \mid \text{allocate}() \mid \text{run } (e, m) \mid \\
 & & \text{convert(ptr } m, \text{int } m)\ e \mid \text{convert (int } m_j, \text{int } m_i))\ e. \\
S \in \text{Stmts} & ::= & l := e \mid \text{run } (S, m) \mid S_1; S_2 \mid \text{if } e \text{ then } S_t \text{ else } S_f \mid \text{while } e \text{ do } S_t. \\
AS \in \text{AnnotStmts} & ::= & l := e \mid \text{run } (S, m) \mid S_1; S_2 \mid x_i := fi(x_j, x_k) \mid x_i := md(x_j) \mid \\
 & & mu(x_j) \mid \text{if } e \text{ then } S_t \text{ else } S_f \mid \text{while } e \text{ do } S_t. \\
prog \in \text{Progs} & ::= & V; S.
\end{array}
$$

Fig. 1. Programming Language Model: *SSA-ParLang*

$$
\frac{}{\epsilon : T \to_V T \hookrightarrow \epsilon}\ (\epsilon)
\qquad
\frac{}{\text{int } x : T \to_V T[x \mapsto 1] \hookrightarrow \text{int } x_1}\ (\text{int})
$$

$$
\frac{d_1 : T \to_V T'' \hookrightarrow d_1' \qquad d_2 : T'' \to_V T' \hookrightarrow d_2'}{d_1 d_2 : T \to_V T' \hookrightarrow d_1' d_2'}\ (d_1 d_2^V)
$$

Fig. 2. Typing Rules for Static Single Assignment (SSA): Variable Types.

$$
\frac{T(x) = i}{x : T \to_l T[x \mapsto T(x) + 1] \hookrightarrow x_i}\ (\text{x}^l)
$$

$$
\frac{T(y) = i}{(l \to y) : T \to_l T[y \mapsto T(y) + 1] \hookrightarrow (l \to y_i)}\ (\to^l)
\qquad
\frac{l : T \to_l T' \hookrightarrow l'}{[l] : T \to_l T' \hookrightarrow [l']}\ ([l]^l)
$$

Fig. 3. Typing Rules for Static Single Assignment (SSA): Locations.

$$
\frac{T(x) = i}{x : T \to_e T \hookrightarrow x_i}\ (\text{x}^e)
\qquad
\frac{T(y) = i}{l \to y : T \to T \hookrightarrow l \to_e y_i}\ (\to^e)
\qquad
\frac{l : T \to_e T' \hookrightarrow l'}{[l] : T \to_e T' \hookrightarrow [l']}\ ([l]^e)
$$

$$
\frac{\begin{array}{c} e_1 : T \to_e T' \hookrightarrow e_1' \\ e_2 : T \to_e T' \hookrightarrow e_2' \end{array}}{(e_1\ i_{op}\ e_2) : T \to_e T' \hookrightarrow (e_1'\ i_{op}\ e_2')}\ (\text{l}^t)
\qquad
\frac{l : T \to_e T' \hookrightarrow l'}{\&l : T \to_e T' \hookrightarrow \&l'}\ (\&l^e)
$$

$$
\frac{}{\text{allocate}() : T \to_e T \hookrightarrow \text{allocate}()}\ (\text{allocate}^e)
\qquad
\frac{e : T \to_e T' \hookrightarrow e'}{\text{run}(e, m) : T \to_e T' \hookrightarrow \text{run}(e', m)}\ (\text{run}^e)
$$

$$
\frac{e : T \to_e T' \hookrightarrow e'}{\text{convert(ptr } m, \text{int } m)\ e : T \to_e T' \hookrightarrow \text{convert(ptr } m, \text{int } m)\ e'}\ (\text{con}_1^e)
$$

$$
\frac{e : T \to_e T' \hookrightarrow e'}{\text{convert (int } m_j, \text{int } m_i))\ e : T \to_e T' \hookrightarrow \text{convert (int } m_j, \text{int } m_i))\ e'}\ (\text{con}_2^e)
$$

Fig. 4. Typing Rules for Static Single Assignment (SSA): Distributed Expressions.

Fig. 5. Static Single Assignment (SSA) of SPMD: Technique Elements

$$\frac{\begin{array}{l} l \neq [\ldots] \qquad e \neq [\ldots] \\ l : T \rightarrow T'' \hookrightarrow_l l' \\ e : T'' \rightarrow_e T' \hookrightarrow e' \end{array}}{l := e : T \rightarrow_s T' \hookrightarrow l' := e'}\ (;^s_1)$$

$$\frac{\begin{array}{ll} l : T \rightarrow T'' \hookrightarrow_l l' & e : T'' \rightarrow_e T' \hookrightarrow e' \\ l := [e] : P \rightarrow_a P' & e : P \rightarrow_a A = \{x_1, \ldots, x_n\} \\ \forall x \in A.\ S_x = mu(x_{T'(x)}) & l \neq [\ldots] \end{array}}{l := [e] : T \rightarrow_s T' \hookrightarrow l' := [e']; S_{x_1}; \ldots; S_{x_n}}\ (;^s_2)$$

$$\frac{\begin{array}{ll} l : T \rightarrow T'' \hookrightarrow_l l' & e : T'' \rightarrow_e T' \hookrightarrow e' \\ [l] := e : P \rightarrow_a P' & l : P \rightarrow_a A = \{x_1, \ldots, x_n\} \\ \forall x \in A.\ S_x = (x_{T'(x)+1} := md(x_{T(x)})) & e \neq [\ldots] \end{array}}{[l] := e : T \rightarrow_s T'[x \mapsto T'(x) + 1 \mid x \in A] \hookrightarrow [l'] := e'; S_{x_1}; \ldots; S_{x_n}}\ (;^s_3)$$

$$\frac{\begin{array}{ll} l : T \rightarrow T'' \hookrightarrow_l l' & \\ [l] := e : P \rightarrow_a P' & e : T'' \rightarrow_e T' \hookrightarrow e' \\ \forall x \in A.\ S_x = (x_{T'(x)+1} := md(x_{T(x)})) & l : P \rightarrow_a A = \{x_1, \ldots, x_n\} \\ \forall y \in B.\ S_y = mu(y_{T'(y)}) & e : P \rightarrow_a B = \{y_1, \ldots, y_m\} \end{array}}{[l] := [e] : T \rightarrow_s T'[x \mapsto T'(x) + 1 \mid x \in A] \hookrightarrow S_{y_1}; \ldots; S_{y_m}; [l'] := e'; S_{x_1}; \ldots; S_{x_n}}\ (;^s_4)$$

$$\frac{\begin{array}{l} S_1 : T \rightarrow_s T'' \hookrightarrow S'_1 \\ S_2 : T'' \rightarrow_s T' \hookrightarrow S'_2 \end{array}}{S_1; S_2 : T \rightarrow_s T' \hookrightarrow S'_1; S'_2}\ (:=^s) \qquad \frac{S : T \rightarrow_s T' \hookrightarrow S'}{\text{run } (S, m) : T \rightarrow_s T' \hookrightarrow \text{run } (S', m)}\ (\text{run}^s)$$

$$\frac{\begin{array}{ll} e : T \rightarrow_s T_e \hookrightarrow e' & A = \{x_1, \ldots, x_n\} = \{x \mid T'(x) > T(x)\} \\ S_t : T_e \rightarrow_s T'' \hookrightarrow S'_t & \forall x \in A.\ S_x = x_{T'(x)+1} := fi(x_{T(x)}, x_{T'''(x)}) \\ S_f : T'' \rightarrow_s T''' \hookrightarrow S'_f & T' = T'''[x \mapsto T'''(x) + 1 \mid x \in A] \end{array}}{\text{if } e \text{ then } S_t \text{ else } S_f : T \rightarrow_s T' \hookrightarrow S_{x_1}; \ldots; S_{x_n} \text{if } e' \text{ then } S'_t \text{ else } S'_f}\ (\text{if}^s)$$

$$\frac{\begin{array}{ll} & A = \{x_1, \ldots, x_n\} \text{ is the set of varibales modified by } S_t \\ & T'' = T_e[x \mapsto T_e(x) + 1 \mid x \in A] \\ e : T \rightarrow_s T_e \hookrightarrow e' & \forall x \in A.\ S_x = x_{T(x)+1} := fi(x_{T(x)}, x_{T'''(x)}) \\ S_t : T'' \rightarrow_s T''' \hookrightarrow S'_t & \forall y \in A.\ S_y = y_{T'''(x)+1} := fi(x_{T(x)}, x_{T'''(x)}) \\ & T' = T'''[x \mapsto T'''(x) + 1 \mid x \in A] \end{array}}{\text{while } e \text{ do } S_t : T \rightarrow_s T' \hookrightarrow S_{y_1}; \ldots; S_{y_n}; \text{while } e' \text{ do } S'_t; S_{x_1}; \ldots; S_{x_n}}\ (\text{whl}^s)$$

Fig. 6. Typing Rules for Static Single Assignment (SSA): Statements.

tional semantics, *Sem-e*,to the distributed expressions of *SSA-ParLang*. Suppose in *Sem-e* that $e : s \rightarrow s'$. Then $[e]s \equiv [e']s$ and if s is of type T, then s' is of type T'.

2) Suppose $S : T \rightarrow T' \hookrightarrow S'$. Suppose also the existence of an appropriate operational semantics, *Sem-S*, to the statements of *SSA-ParLang*. Suppose in *Sem-s* that $S : s \rightarrow s'$. Then $[S]s \equiv [S']s$ and if s is of type T, then s' is of type T'.

The above theorem formalizes the soundness of our proposed technique and has a straightforward proof if the used semantics is a convenient one. Therefore the choice of the operational semantics affects the complexity of the theorem proof.

Our proposed technique works as following. Given a SPMD program $S \in$ *Stmt*, one uses the inference rules presented in this section to gradually build $S' \in$ *AnnotStmt* such as $S : \perp \rightarrow T' \hookrightarrow S'$. The symbol $\perp$ denotes the bottom type with an empty domain. Once S' is built, the type derivation of this judgment serves as a justification-proof for a PCC package.

Rules corresponding to distributed expressions are shown in Figure 2, 3, and 4. Comments are in order. The rule (x^l) creates a new version of the variable x using the index specified by the type T and increases the index of x in the post-type by 1. This is so as in this case x is encountered on the left-hand-side of some assignment statement and hence is assigned a value. On the other hand, the rule (x^e) creates a new version of the variable x using the index specified by the type T and does not modify the index of x in the post-type. This is so as in this case x is encountered on the right-hand-side of an assignment statement and hence is not modified. Similar explanations clarify remaining rules for distributed expressions.

Rules corresponding to statements are shown in Figure 5. Comments are in order. The rule $(;^s_2)$ treats the statement $l := [e]$. The preconditions of this rule include a transformation derivation for l and e and include as well pointer analysis for the statement $l := [e]$ and for the distributed expression e. This is so as the set of variables that e may alias to will be the bases for constructing the annotating statements $S_{x_1}; \ldots ; S_{x_n}$. The idea of the annotation is that these are the variables that may include pointers at that program point. The rule $(;_4^s)$ treats the statement $[l] := [e]$ which includes two types of indirect memory-access; reading and modifying. For the reading process we insert a sort of annotations before the statement and for modifying we insert a different sort of annotations after the statement. Similar explanations clarify remaining rules for statements.

III. RELATED WORK

To represent the programs data flow, many data structures, including SSA form, were proposed. As symmetric extension of SSA, One example of these data structures is present in [34] and presents static single information form (SSI) [3], [7]. The idea of SSI is to present, for every branch, new definitions in case of variable uses in many different branches in the control flow graph. Webs were presented in [27] as maximal collections of def-use sequences having a similar use.

Static Single Assignment form construction, as an example of program analysis and transformation, is based on recognizing, in a control-flow graph, the dominator tree. Of differing worst-case and average complexities, much research were carrier out to find these trees [8], [9], [19], [4]. On the control-flow graph, a common attribute of all these research is that they are all expressed in a single phase. Other research split the tree construction in two phases [20].

Program dependence graphs (PDGs) [13] can be used for software compiling, developing, and debugging. In PDGs, vertices and edges represent subprogram and dependencies, respectively. This facilitates realizing (as graph traversals) involved program analyses (such as slicing [22]). The effort of establishing PDGs mostly goes to constructing control and data dependencies. Dataflow analyses can be used to construct the def-use chains such as in [24]. Also interval analysis and directed methods [10], [33], [14] can be used for the same purpose. However this technique is not efficient enough to deal with languages with pointers and control flow that is unstructured.

Static Single Assignment is not classified as transport format. However, as transitional representation, most of the recent efficient virtual machines (JIT-based) use SSA form. More interestingly, SSA is indeed used as an encoding format in representations of mobile code formats [2], [1] that are inherently safe. An example of this use is SafeTSA [1]. The format of mobile code is typically self-consistent. This format can only represent programs that are well typed and formed. Therefore the use of SSA removes the need for dynamic verifications. However this use usually results not treating the Java class-file already existing. Some research [20], [1] speeds up code construction via presenting, in SSA-form, the code to the JIT.

To study programming-languages implementations that are high-level, SSA representation can also be used in byte-code compilation as in Marmot [18]. Such use typically pays more attention to code consumption and does not help (such as program reordering) in the code production process (via hinting for example).

Proof-carrying code (PCC) [30], [23] handles the need for mobile code annotation with proofs simply checkable by code consumer. Therefore PCC [31], [26] carries the code verification instead of the code consumer. In absence of PCC, using policies for public safety, the code producer establishes a justification condition and proves its correctness for the program in hand. Of course the justification is typically send to consumer together with the code. Therefore when the code is received, the consumer rechecks the justification condition and makes sure that the received justification indeed satisfies the claimed verification condition. Shipping the justification unfortunately typically results in abandoning the format of Java byte-code format.

Similar to PCC is the split verifier technique [12]. To reduce class loading time and relieving the burden from JVM, split verifier uses the data-flow analysis fixed-point to annotate the JVML [6], [37]. Therefore this way simplifies the verification as it becomes necessary only to make sure that the annotation is a logical fixed-point, achievable in in linear time. Some research used this idea to verify the constructed dominator trees

IV. CONCLUSION AND FUTURE WORK

This paper proposed a new method for establishing an SSA form for SPMD programs. The method basic components are inference rules. This makes its application relatively simple and trustful. The type derivations of the proposed method can be used in proof-carrying code area of research (PCC).

There are many directions for future work. Producing analyses techniques for different programming styles (including SPMD) using SSA forms is an interesting direction for future work. For SPMD programs, program analysis techniques can be designed on the program format produced by the technique proposed in this paper. If these analyses are designed using concepts of type systems, in the spirit of [17], [15], [16], they will have direct applications in PCC.

REFERENCES

[1] Wolfram Amme, Niall Dalton, Michael Franz, and Jeffery von Ronne. Safetsa: A type safe and referentially secure mobile-code representation based on static single assignment form. In Michael Burke and Mary Lou Soffa, editors, *PLDI*, pages 137–147. ACM, 2001.

[2] Wolfram Amme, Thomas S. Heinze, and Jeffery von Ronne. Intermediate representations of mobile code. *Informatica (Slovenia)*, 32(1):1–25, 2008.

[3] Davide Ancona and Giovanni Lagorio. Static single information form for abstract compilation. In Jos C. M. Baeten, Thomas Ball, and Frank S. de Boer, editors, *IFIP TCS*, volume 7604 of *Lecture Notes in Computer Science*, pages 10–27. Springer, 2012.

[4] Tom Bäckström. Computationally efficient objective function for algebraic codebook optimization in acelp. In Frédéric Bimbot, Christophe Cerisara, Cécile Fougeron, Guillaume Gravier, Lori Lamel, François Pellegrino, and Pascal Perrier, editors, *INTERSPEECH*, pages 3434–3438. ISCA, 2013.

[5] Gilles Barthe, Delphine Demange, and David Pichardie. A formally verified ssa-based middle-end - static single assignment meets compcert. In Helmut Seidl, editor, *ESOP*, volume 7211 of *Lecture Notes in Computer Science*, pages 47–66. Springer, 2012.

[6] Nadia Belblidia and Mourad Debbabi. A dynamic operational semantics for jvml. *Journal of Object Technology*, 6(3):71–100, 2007.

[7] Philip Brisk and Majid Sarrafzadeh. Interference graphs for procedures in static single information form are interval graphs. In Heiko Falk and Peter Marwedel, editors, *SCOPES*, volume 235 of *ACM International Conference Proceeding Series*, pages 101–110, 2007.

[8] Adam L. Buchsbaum, Loukas Georgiadis, Haim Kaplan, Anne Rogers, Robert Endre Tarjan, and Jeffery Westbrook. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM J. Comput.*, 38(4):1533–1573, 2008.

[9] Adam L. Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery Westbrook. *Corrigendum:* a new, simpler linear-time dominators algorithm. *ACM Trans. Program. Lang. Syst.*, 27(3):383–387, 2005.

[10] David Byers, Mariam Kamkar, and Ture Pålsson. Syntax-directed construction of value dependence graphs. In *ICSM*, pages 692–, 2001.

[11] Fred C. Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in ssa form. In Tibor Gyimóthy, editor, *CC*, volume 1060 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 1996.

[12] Ariel Cohen, Kedar S. Namjoshi, and Yaniv Sa'ar. Split: A compositional ltl verifier. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 558–561. Springer, 2010.

[13] David J. A. Cooper, Mun Wai Chan, Gautam Mehra, Peter Woodward, Brian R. von Konsky, Michael C. Robey, and Michael Harding. Using dependence graphs to assist manual and automated object oriented software inspections. In *ASWEC*, pages 262–269. IEEE Computer Society, 2006.

[14] Lin Du, Guorong Xiao, and Daming Li. A novel approach to construct object-oriented system dependence graph and algorithm design. *JSW*, 7(1):133–140, 2012.

[15] Mohamed A. El-Zawawy and Nagwan M. Daoud. New error-recovery techniques for faulty-calls of functions. *Computer and Information Science*, 5(3):67–75, May 2012.

[16] Mohamed A. El-Zawawy and Hamada A. Nayel. Partial redundancy elimination for multi-threaded programs. *IJCSNS International Journal of Computer Science and Network Security*, 11(10):127–133, October 2011.

[17] Mohamed A. El-Zawawy and Hamada A. Nayel. Type systems based data race detector. *IJCSNS International Journal of Computer Science and Network Security*, 5(4):53–60, July 2012.

[18] Robert P. Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: an optimizing compiler for java. *Softw., Pract. Exper.*, 30(3):199–232, 2000.

[19] Wojciech Fraczak, Loukas Georgiadis, Andrew Miller, and Robert Endre Tarjan. Finding dominators via disjoint set union. *J. Discrete Algorithms*, 23:2–20, 2013.

[20] Andreas Gal, Christian W. Probst, and Michael Franz. Structural encoding of static single assignment form. *Electr. Notes Theor. Comput. Sci.*, 141(2):85–102, 2005.

[21] Ming-Yu Hung, Peng-Sheng Chen, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq Kuen Lee. Support of probabilistic pointer analysis in the ssa form. *IEEE Trans. Parallel Distrib. Syst.*, 23(12):2366–2379, 2012.

[22] Paritosh Jain and Nitish Garg. A novel approach for slicing of object oriented programs. *ACM SIGSOFT Software Engineering Notes*, 38(4):1–4, 2013.

[23] Romain Jobredeaux, Heber Herencia-Zapana, Natasha A. Neogi, and Eric Feron. Developing proof carrying code to formally assure termination in fault tolerant distributed controls systems. In *CDC*, pages 1816–1821. IEEE, 2012.

[24] Jens Krinke. Advanced slicing of sequential and concurrent programs. In *ICSM*, pages 464–468. IEEE Computer Society, 2004.

[25] Xu Liu, Jianfeng Zhan, Kunlin Zhan, Weisong Shi, Lin Yuan, Dan Meng, and Lei Wang. Automatic performance debugging of spmd-style parallel programs. *J. Parallel Distrib. Comput.*, 71(7):925–937, 2011.

[26] Hans-Wolfgang Loidl, Kenneth MacKenzie, Steffen Jost, and Lennart Beringer. A proof-carrying-code infrastructure for resources. In *LADC*, pages 127–134. IEEE Computer Society, 2009.

[27] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[28] Preeti Mudliar, Sharon Strover, and Kenneth Flamm. Outside looking in: shaping access and use of pccs. In Gary Marsden and Julian May, editors, *ICTD (2)*, pages 104–107. ACM, 2013.

[29] Peter Pacheco. *An Introduction to Parallel Programming*. Elsevier, 2011. 1 edition (2011).

[30] Frank Pfenning, Luís Caires, and Bernardo Toninho. Proof-carrying code in a session-typed process calculus. In Jean-Pierre Jouannaud and Zhong Shao, editors, *CPP*, volume 7086 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2011.

[31] Heidar Pirzadeh, Danny Dubé, and Abdelwahab Hamou-Lhadj. An extended proof-carrying code framework for security enforcement. *Transactions on Computational Science*, 11:249–269, 2010.

[32] Subhajit Roy and Y. N. Srikant. The hot path ssa form: Extending the static single assignment form for speculative optimizations. In Rajiv Gupta, editor, *CC*, volume 6011 of *Lecture Notes in Computer Science*, pages 304–323. Springer, 2010.

[33] Josep Silva, Salvador Tamarit, and César Tomás. System dependence graphs in sequential erlang. In Juan de Lara and Andrea Zisman, editors, *FASE*, volume 7212 of *Lecture Notes in Computer Science*, pages 486–500. Springer, 2012.

[34] Jeremy Singer. Static single information from a functional perspective. In Stephen Gilmore, editor, *Trends in Functional Programming*, volume 4 of *Trends in Functional Programming*, pages 63–78. Intellect, 2003.

[35] Ajit Singh and Vincent Van Dongen. An integrated performance analysis tool for spmd data-parallel programs. *Parallel Computing*, 23(8):1089–1112, 1997.

[36] Daniel Wonisch, Alexander Schremmer, and Heike Wehrheim. Programs from proofs - a pcc alternative. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 912–927. Springer, 2013.

[37] Hamdi Yahyaoui, Mourad Debbabi, and Nadia Tawbi. A denotational semantic model for validating jvml/cldc optimizations under isabelle/hol. In *QSIC*, pages 348–355. IEEE Computer Society, 2007.