Frequent Statement and De-reference Elimination for Distributed Programs

Mohamed A. El-Zawawy[1,2]

[1] College of Computer and Information Sciences,
Al Imam Mohammad Ibn Saud Islamic University (IMSIU)
Riyadh, Kingdom of Saudi Arabia
[2] Department of Mathematics, Faculty of Science, Cairo University
Giza 12613, Egypt
maelzawawy@cu.edu.eg

Abstract. This paper introduces a new approach for the analysis of *frequent statement and de-reference elimination* for distributed programs run on parallel machines equipped with hierarchical memories. The address space of the language studied in the paper is globally partitioned. This language allows programmers to define data layout and threads which can write to and read from other thread memories.

Simply structured type systems are the tools of the techniques presented in this paper which presents three type systems. The first type system defines for program points of a given distributed program sets of calculated (*ready*) statements and memory accesses. The second type system uses an enriched version of types of the first type system and determines which of the specified statements and memory accesses are used later in the program. The third type system uses the information gather so far to eliminate unnecessary statement computations and memory accesses (the analysis of *frequent statement and de-reference elimination*).

Two advantages of our work over related work are the following. The hierarchical style of concurrent parallel computers is similar to the memory model used in this paper. In our approach, each analysis result is assigned a type derivation (serves as a correctness proof).

Keywords: *Ready statement and memory access* analysis, *semi-expectation* analysis, *frequent statement and de-reference elimination* analysis, certified code, distributed programs, semantics of programming languages, operational semantics, type systems.

1 Introduction

Distributed programming is about building a software that has concurrent processes cooperating in achieving some task. For a problem specification, the type, number, and the way of interaction of processes needed to solve the problem is decided beforehand. Then a supercomputer can be computationally simulated by a group of workstations to carry different processes. A group of supercomputers can in turn be combined to provide a computing power greater than that provided by any single machine. This enormous computing power provided by distributed systems is why the distributed-programming style [1,20,30] is quite important and attractive. Among examples of distributed programming languages (DPLs), based on machines having multi-core processors and using partitioned-global model, are Titanium which is based on Java, Unified Parallel C (UPC), Chapel, and X10.

Recomputing a non-trivial statement and re-accessing a memory location are waste of time and power if the value of the statement and the content of the location have not been changed. The purpose of *frequent statement and de-reference elimination* analysis is to save such wasted power and time. This is an interesting analysis because it involves connecting statement and de-reference calculations to program points where these calculation values may be reused. The analysis also requires doing changes to the program points at the ends of these connections. Such changes to program points have to be done carefully so that they do not destroy the compositionality. Our approach to treat this analysis is a type system [6,4,15,5,11] built on a combination of two analyses; one of them builds on the results of the other one.

For different programming languages, in previous work [14,3,6,4,15,5,8,9,7,16,10], we have proved that the type-systems style is certainly an adaptable approach for achieving many static analyses. This paper proves that this style is flexibly useful to the involved and important problem of frequent statement and de-reference elimination of distributed programs.

This paper introduces a new technique for frequent statement and de-reference elimination for distributed programs run on hierarchical memories. Simply structured type systems are the main tools of our technique. The proposed technique is presented using a language (the appendix- Figure 3) equipped with basic commands for distributed execution of programs and for pointer manipulations. The single program multiply data (*SPMD*) model is the execution archetypal used in this paper. On different data of different machines this archetypal runs the same program. The analysis of *frequent statement and de-reference elimination* for distributed programs is achieved in three steps each of which is done using a type system. The first of these steps achieves *ready statement and memory access* analysis. The second step deals with *semi-expectation* analysis and builds on the type system of the first step. The third type systems takes care of the analysis of *frequent statement and de-reference elimination* and is built on the type system of the second step.

Motivation

The left-hand-side of Figure 1 presents a motivating example of our work. We note that lines 4 and 6 de-reference $a * b$ which has already been de-referenced in line 2 with no changes to values of a and b in the path from 2 to 6. This is a waste of computational power and time (accessing a secondary storage). One objective of the research in this paper is avoid such waste by transforming the program into that in the right-hand-side of the figure. This is not all; we need to do that in a way that provides a correctness proof for each such transformation. We adopt a style (type systems) that provides these proofs (type derivations).

```
0.                                          l := c*d; k := a*b;
1. x := a*b + c*d;                          x := k + l;
2. x := convert (*(a*b), 2);                x := convert (*k, 2);
3. y := transmit c*d from 3;                y := transmit l from 3;
4. if (*(a*b) = *(c*d))                     if (*k = *l)
5.     then y := transmit *(c*d) from 2;        then y := transmit *(c*d) from 2;
6.     else x := convert (*(a*b), 2);           else x := convert (*(a*b), 2);
```

Fig. 1. A motivating example

Contributions

Contributions of this paper are new techniques, in the form of type systems, for:

1. The analysis of *ready statement and memory access* for distributed programs.
2. The analysis of *semi-expectation* for distributed programs.
3. The analysis of *frequent statement and de-reference elimination* of distributed programs.

Organization. Organization in the rest of the paper is as following. Section 2 presents the type system achieving the analysis of *ready statement and memory access* for distributed programs. The analysis of *semi-expectation* as an enrichment of the type system presented in Section 2 is outlined in Section 3. The main type system carrying the analysis of *frequent statement and de-reference elimination* is contained in Section 4. Related and future work are briefed in Section 5. An appendix to the paper briefly reviews our language syntax, memory model, and operational semantics which are presented in more details in [7].

2 Ready Statement and Memory Access Analysis

If the value of a statement and the content of a memory location have not been changed, then the compiler should not recompute the statement or re-access the location. The purpose of *Frequent Statement and De-reference Elimination* is to save the wasted power and time involved in these repeated computations. This is not a trivial task; compared to other program analyses, it is a bit complex. This task is done in stages. The first stage is to analyze the given program to recognize *ready* statements and memory locations.

The analysis of *ready* statements and memory locations calculates for every program point the set of statements and memory locations that are *ready* at that point in the sense of Definition 1. This section presents a type system (*ready* type system) to achieve this analysis for distributed programs.

Definition 1. *1. At a program point pt, a statement S is ready if each computational path to pt:*
(a) contains an evaluation of S at some point (say pt') and
(b) does not modify S (changing value of any of S's variables) between pt' and pt.

2. *At a program point pt, a memory location l is ready if each computational path to pt:*
   (a) reads l at some point (say pt′) and
   (b) does not modify content of l between pt′ and pt.

The *ready* analysis is a forward analysis that takes as an input a set of statements and memory locations (the *ready* set of the first program point). It is sensible to let this set be the empty set. The set of types of our *ready* type system has the form: *points-to-types* $\times$ $\mathcal{P}(Stmt^+ \cup gAddrs)$, where

1. $Stmt^+$ is the set of nontrivial statements (Figure 3 – the paper appendix),
2. *gAddrs* is the set of global addresses on our machine. This set is defined precisely in the appendix of this paper, and
3. *points-to-types* is a set of points-to types (typically have the form of maps from the union of variables and global addresses to the power set of global addresses [6,4,15,5]).

The subtyping relation has the form $\leq_p \times \supseteq$ where $\leq_p$ is the order relation on the points-to types and $\supseteq$ is the order relation on $\mathcal{P}(Stmt^+ \cup gAddrs)$. A state on an execution path is of type $rs \in \mathcal{P}(Stmt^+ \cup gAddrs)$ if all elements of *rs* are *ready* at this state according to Definition 1. Judgments of the *ready* type system have the form $S : (p, rs) \rightarrow_m (A', p', rs')$. The symbols p and p' denote the points-to types of the pre and post states of executing S. The set A' denotes the set of addresses that S may evaluate to. We assume that all such pointer information are given along with the statement S. Techniques like [6,4,15,5] are available to compute the pointer information. For a given statement along with pointer information and a *ready* pre-type *rs*, we present a type system to calculate a post *ready*-type rs' such that $S : (p, rs) \rightarrow_m (A', p', rs')$. The type derivation of this typing process is a proof for the correctness of the *ready* information. The meaning of the judgment is that if elements of *rs* are *ready* before executing S, then elements of rs' are *ready* after executing S.

The inference rules of the *ready* type system are as follows:

$$\frac{n : p \rightarrow_m (A', p')}{n : (p, rs) \rightarrow_m (A', p', rs)} \qquad \frac{\begin{array}{c} S_1 : (p, rs) \rightarrow_m (A'', p'', rs'') \\ S_2 : (p'', rs'') \rightarrow_m (A', p', rs') \end{array}}{S_1\ i_{op}\ S_2 : (p, rs) \rightarrow_m (\emptyset, p', rs' \cup \{S_1\ i_{op}\ S_2\})}\,(i^r_{op}) \qquad \frac{x : p \rightarrow_m (A', p')}{x : (p, rs) \rightarrow_m (A', p', rs)}$$

$$\frac{\begin{array}{c} S_1 : (p, rs) \rightarrow_m (A'', p'', rs'') \\ S_2 : (p'', rs'') \rightarrow_m (A', p', rs') \end{array}}{S_1\ b_{op}\ S_2 : (p, rs) \rightarrow_m (\emptyset, p', rs' \cup \{S_1\ b_{op}\ S_2\})}\,(b^r_{op}) \qquad \frac{*S : p \rightarrow_m (A', p') \qquad S : (p, rs) \rightarrow_m (A'', p'', rs'')}{*S : (p, rs) \rightarrow_m \begin{cases} (A', p', rs'' \cup \{*S, g\}), & A'' = \{g\}; \\ (A', p', rs'' \cup \{*S\}), & |A''| \neq 1. \end{cases}}\,(*^r)$$

$$\frac{}{skip : (p, rs) \rightarrow_m (\emptyset, p, rs)} \qquad \frac{x := S : p \rightarrow_m (A', p') \qquad S : (p, rs) \rightarrow_m (A'', p'', rs'')}{x := S : (p, rs) \rightarrow_m (A', p', rs'' \setminus \{S \in Stmt \mid x \in free(S)\})}\,(:=^r)$$

$$\frac{\begin{array}{c} S_1 \leftarrow S_2 : p \rightarrow_m (A', p') \\ S_1 : (p, rs) \rightarrow_m (A_1, p_1, as_1) \end{array} \quad S_2 : (p_1, as_1) \rightarrow_m (A_2, p_2, as_2)}{S_1 \leftarrow S_2 : (p, rs) \rightarrow_m (A', p', as_2 \setminus (A' \cup \{S \mid S : p \rightarrow_m (A_s, \_) \ \&\ A_s \cap A' \neq \emptyset\}))}\,(\leftarrow^r)$$

$$\frac{\begin{array}{c} S_1 : (p, rs) \rightarrow_m (A'', p'', rs'') \\ S_2 : (p'', rs'') \rightarrow_m (A', p', rs') \end{array}}{S_1; S_2 : (p, rs) \rightarrow_m (A', p', rs')}\,(seq^r) \qquad \frac{\begin{array}{c} S : (p, rs) \rightarrow_m (A'', p'', rs'') \\ S_t : (p, rs'') \rightarrow_m (A', p', rs') \\ S_f : (p, rs'') \rightarrow_m (A', p', rs') \end{array}}{if\ S\ then\ S_t\ else\ S_f : (p, rs) \rightarrow_m (A', p', rs')}\,(if^r)$$

$$\frac{S_1[S_2/x] : (p,rs) \rightarrow_m (A',p',rs')}{(\lambda x.S_1)S_2 : (p,rs) \rightarrow_m (A',p',rs')}\,(appl^r) \qquad \frac{\begin{array}{c} S : (p,rs) \rightarrow (A'',p'',rs') \\ S_t : (p'',rs') \rightarrow_m (A',p',rs) \end{array}}{while\ S\ do\ S_t : (p,rs) \rightarrow_m (A',p',rs')}\,(whl^r)$$

$$\frac{S : (p,rs) \rightarrow_m (A',p',rs')}{\lambda x.S : (p,rs) \rightarrow_m (A',p',rs')}\,(abs^r) \qquad \frac{fd(name) : (p,rs) \rightarrow_m (A',p',rs')}{name : (p,rs) \rightarrow_m (A',p',rs')}\,(name^r)$$

$$\frac{(\lambda x.S')S : (p,rs) \rightarrow_m (A',p',rs')}{letrec\ x = S\ in\ S' : (p,rs) \rightarrow_m (A',p',rs')}\,(letrec^r) \qquad \frac{new_l : p \rightarrow_m (A',p')}{new_l : (p,rs) \rightarrow_m (A',p',rs)}\,(new^r)$$

$$\frac{\begin{array}{c} convert\ (S,n) : p \rightarrow_m (A',p') \\ S : (p,rs) \rightarrow_m (A,p'',rs') \end{array}}{convert\ (S,n) : (p,rs) \rightarrow_m (A',p',rs')}\,(convert^r) \qquad \frac{\begin{array}{c} transmit\ S_1\ from\ S_2 : p \rightarrow (A',p') \\ S_2 : (p,rs) \rightarrow_m (A_2,p_2,as_2) \\ S_1 : (p_2,as_2) \rightarrow_m (A_1,p_1,rs') \end{array}}{transmit\ S_1\ from\ S_2 : (p,rs) \rightarrow_m (A',p',rs')}\,(trans^r)$$

$$\frac{\begin{array}{c} (p'_1,rs'_1) \leq (p_1,as_1) \\ S : (p_1,as_1) \rightarrow_m (p_2,as_2) \\ (p_2,as_2) \leq (p'_2,rs'_2) \end{array}}{S : (p'_1,rs'_1) \rightarrow_m (p'_2,rs'_2)}\,(csq^r) \qquad \frac{Defs : \emptyset \curvearrowright fd \quad S : (p,rs) \rightarrow_m (A',p',rs')}{Defs : S : (p,rs) \rightarrow_m (A',p',rs')}\,(prg^r)$$

Comments on the inference rules are in order. We note that numbers, variables, and the allocating statement (*new*) do not affect the *ready* pre-type. In line with semantic rules (i^r_{op}) and (b^r_{op}) (the paper's appendix), nontrivial arithmetic and boolean statements and their nontrivial sub-statements are made *ready*. The direct assignment rule $(:=^r)$ expresses that after executing the assignment the sub-statements of r.h.s. become *ready* and that all statements involving x become *unready* as the value of x may become different. The rule $(*^r)$ reflects the fact that the statement $*S$ becomes *ready* after executing the de-reference. Moreover if S evaluates to a single address according to the underlying pointer analysis, then this address becomes *ready* as well. However if S evaluates to a large set of addresses (more than one), then we are not sure which of these addresses is the concerned one and hence can not conclude any readiness information about addresses. The rule $(\leftarrow^r)$ adds the sub-statements of S_1 and S_2 to the *ready* pre-type. Since the content of address referenced by S_1 is possibly changed after executing the statement, all statements involving de-referencing this address are removed from the set of *ready* items. Remaining rules are self-explanatory. The Boolean statements *true* and *false* have inference rules similar to that of n.

All in all, the information provided by type derivations obtained using this and the following type systems is classified into two sorts. The first sort is about knowing the program point at which a particular statement becomes *ready*. The second sort of information is about the program point at a which a pre-computed value of a *ready* statement can be replaced with the statement.

Now we recall the assumption that our distributed system consists of $|M|$ machines. For a given statement S and a given machine m, the type system given above calculates for each program point of S, the set of *ready* items. The following rule can be used to combine the information calculated for each machine to get a new *ready* information for each program point. The new *ready* information is valid on any of the $|M|$ machines.

$$\frac{\forall m \in M.\ S : (\sup\{p, p_j \mid j \neq i\}, \sup\{rs, rs_j \mid j \neq i\}) \rightarrow_m (A_m, p_m, rs_m)}{S : (p,rs) \rightarrow_M (\cup_i A_i, \sup\{p_1,\ldots,p_n\}, \sup\{rs_1,\ldots,rs_n\})}\,(main\text{-}rs)$$

The rule above supposes we have a suitable notion for join of pointer types.

It is not hard to prove the soundness of the above type system:

Theorem 1. *Suppose that* $(S,\delta) \rightarrow (V,\delta')$, $S : (p,rs) \rightarrow (A',p',rs')$ *and the items of* rs *are ready at the point corresponding to* δ *on the execution path. Then the items of* rs' *are ready at the point corresponding to* δ' *on the execution path.*

3 Semi-expectation Analysis

The aim of frequent statement elimination is to introduce new variables to accommodate values of frequent statements and reusing these values rather than recomputing the statements. Analogously, the aim of frequent de-references elimination is to introduce new variables to accommodate values of frequent de-references and reusing these values rather than re-accessing the memory. The information gathered so far by the *ready* type system introduced in the previous section is not enough to achieve frequent statements and de-references elimination. We need to enrich the *ready* information, assigned to each program point, with a new information called *semi-expectable* information:

Definition 2.
1. *At a program point* p*, a statement* S *is semi-expectable if there is a computational path from* p *that:*
   (a) *contains an evaluation of* S *at some point (say* p'*), where* S *is ready at* p'*, and*
   (b) *does not evaluate* S *between* p' *and* p*.*
2. *At a program point* p*, a memory location* l *is semi-expectable if each computational path to* p*:*
   (a) *reads* l *at some point (say* p'*) where* l *is ready at* p'*, and*
   (b) *does not read* l *between* p' *and* p*.*

The *semi-expectation* analysis is a backward analysis that takes as an input a set of statements and memory locations (the *semi-expectable* set of the last program point). It is sensible to let this set be the empty set. The following example gives an intuition for the previous definition:

$$if\ (\ldots)\ then\ a := y+t\ else\ b := *r;\ c := (y+t)/*r.$$

Neither the statement $y+t$ nor the statement $*r$ is *ready* after the if statement because they are not computed in all branches. Hence it is not true to replace these statements with variables towards optimizing the last statement of the example. The job of the type system presented in this section is to provide us with this sort of information. More precisely, as the statements $y+t$ and $*r$ are not *ready* after the if statement, the second statement of the example does not make them *semi-expectable*.

The *semi-expectation* analysis assigns for each program point the set of items that are *semi-expectable*. The analysis is based on the *readiness* analysis and is backward. The set of types of the *semi-expectation* type system has the form: *points-to-types* $\times$ $\mathcal{P}(Stmt^+ \cup gAddrs) \times \mathcal{P}(Stmt^+ \cup gAddrs)$. The subtyping relation has the form $\leq_p \times \supseteq \times \supseteq$. A state on an execution path is of type $se \in \mathcal{P}(Stmt^+ \cup gAddrs)$ if all elements of se are *semi-expectable* according to Definition 2. Judgments of the *semi-expectation* type system have the form $S : (p,rs,se) \rightarrow_m (A',p',rs',se')$. For a given statement along with pointer information, readiness information, and a *semi-expectation* type se', we present a type system to calculate a pre *semi-expectable*-type se such that $S : (p, rs, se) \rightarrow_m (A', p', rs', se')$. The type derivation of this typing process is a proof for the correctness of the *semi-expectable* information. The meaning of the judgment is that if elements of se' are *semi-expectable* after executing S, then elements of se must have been *semi-expectable* before executing S.

The inference rules of the *semi-expectation* type system are as follows:

$$\frac{n : p \rightarrow_m (A', p')}{n : (p, rs, se) \rightarrow_m (A', p', rs, se)} \qquad \frac{x : p \rightarrow_m (A', p')}{x : (p, rs, se) \rightarrow_m (A', p', rs, se)}$$

$$\frac{S_1 : (p, rs, se) \rightarrow_m (A'', p'', rs'', se'') \quad S_2 : (p'', rs'', se'') \rightarrow_m (A', p', rs', se')}{S_1\ i_{op}\ S_2 : (p, rs, se \cup (rs \cap \{S_1\ i_{op}\ S_2\})) \rightarrow_m (\emptyset, p', rs' \cup \{S_1\ i_{op}\ S_2\}, se')}\ (i^e_{op})$$

$$\frac{S_1 : (p, rs, se) \rightarrow_m (A'', p'', rs'', se'') \quad S_2 : (p'', rs'', se'') \rightarrow_m (A', p', rs', se')}{S_1\ b_{op}\ S_2 : (p, rs, se) \rightarrow_m (\emptyset, p', rs' \cup \{S_1\ b_{op}\ S_2\}, se')}\ (b^e_{op})$$

$$\frac{*S : p \rightarrow_m (A', p') \qquad S : (p, rs, se) \rightarrow_m (A'', p'', rs'', se'')}{*S : (p, rs, se \cup (re \cap \{*S, g\})) \rightarrow_m \begin{cases} (A', p', rs'' \cup \{*S, g\}, se''), & A' = \{g\}; \\ (A', p', rs'' \cup \{*S\}, se''), & |A'| \neq 1. \end{cases}}\ (*^e)$$

$$\frac{}{skip : (p, rs, se) \rightarrow_m (\emptyset, p, rs, se)} \qquad \frac{x := S : p \rightarrow_m (A', p') \qquad S : (p, rs, se) \rightarrow_m (A'', p'', rs'', se'')}{x := S : (p, rs, se) \rightarrow_m (A', p', rs'' \setminus \{S \in Stmt \mid x \in free(S)\}, se'')}\ (:=^e)$$

$$\frac{\begin{array}{c} S_1 \leftarrow S_2 : p \rightarrow_m (A', p') \\ S_1 : (p, rs, se) \rightarrow_m (A_1, p_1, as_1, se_1) \end{array} \quad S_2 : (p_1, as_1, se_1) \rightarrow_m (A_2, p_2, as_2, se_2)}{S_1 \leftarrow S_2 : (p, rs, se) \rightarrow_m (A', p', as_2 \setminus (A' \cup \{S \mid S : p \rightarrow_m (A_s, \_)\ \&\ A_s \cap A' \neq \emptyset\}), se_2)}\ (\leftarrow^e)$$

$$\frac{\begin{array}{c} S_1 : (p, rs, se) \rightarrow_m (A'', p'', rs'', se'') \\ S_2 : (p'', rs'', se'') \rightarrow_m (A', p', rs', se') \end{array}}{S_1 ; S_2 : (p, rs, se) \rightarrow_m (A', p', rs', se')}\ (seq^e) \qquad \frac{\begin{array}{c} S : (p, rs, se) \rightarrow_m (A'', p'', rs'', se'') \\ S_t : (p, rs'', se'') \rightarrow_m (A', p', rs', se') \\ S_f : (p, rs'', se'') \rightarrow_m (A', p', rs', se') \end{array}}{if\ S\ then\ S_t\ else\ S_f : (p, rs, se) \rightarrow_m (A', p', rs', se')}\ (if^e)$$

$$\frac{S_1[S_2/x] : (p, rs, se) \rightarrow_m (A', p', rs', se')}{(\lambda x.S_1)S_2 : (p, rs, se) \rightarrow_m (A', p', rs', se')}\ (appl^e)$$

$$\frac{S : (p, rs, se) \rightarrow (A'', p'', rs', se') \quad S_t : (p'', rs', se') \rightarrow_m (A', p', rs, se)}{while\ S\ do\ S_t : (p, rs, se) \rightarrow_m (A', p', rs', se')}\ (whl^e)$$

$$\frac{fd(name) : (p, rs, se) \rightarrow_m (A', p', rs', se')}{name : (p, rs, se) \rightarrow_m (A', p', rs', se')}\ (name^e) \qquad \frac{S : (p, rs, se) \rightarrow_m (A', p', rs', se')}{\lambda x.S : (p, rs, se) \rightarrow_m (A', p', rs', se')}\ (abs^e)$$

$$\frac{(\lambda x.S')S : (p, rs, se) \rightarrow_m (A', p', rs', se')}{letrec\ x = S\ in\ S' : (p, rs, se) \rightarrow_m (A', p', rs', se')}\ (letrec^e) \qquad \frac{new_l : p \rightarrow_m (A', p')}{new_l : (p, rs, se) \rightarrow_m (A', p', rs, se)}\ (new^e)$$

$$\frac{convert\ (S, n) : p \rightarrow_m (A', p') \qquad S : (p, rs, se) \rightarrow_m (A, p'', rs', se')}{convert\ (S, n) : (p, rs, se) \rightarrow_m (A', p', rs', se')}\ (convert^e)$$

$$\frac{\begin{array}{c} transmit\ S_1\ from\ S_2 : p \rightarrow (A', p') \\ S_2 : (p, rs, se) \rightarrow_m (A_2, p_2, as_2, se_2) \end{array} \quad S_1 : (p_2, as_2, se_2) \rightarrow_m (A_1, p_1, rs', se')}{transmit\ S_1\ from\ S_2 : (p, rs, se) \rightarrow_m (A', p', rs', se')}\ (trans^e)$$

$$\frac{\begin{array}{c}(p'_1,rs'_1,se'_1) \le (p_1,as_1,se_1)\\ S:(p_1,as_1,se_1) \rightarrow_m (p_2,as_2,se_2)\\ (p_2,as_2,se_2) \le (p'_2,rs'_2,se'_2)\end{array}}{S:(p'_1,rs'_1,se_1) \rightarrow_m (p'_2,rs'_2,se'_2)}\ (csq^e) \qquad \frac{\mathit{Defs}:\emptyset \curvearrowright fd \quad S:(p,rs,se) \rightarrow_m (A,p',rs',se')}{\mathit{Defs}:S:(p,rs,se) \rightarrow_m (A,p',rs',se')}\ (prg^e)$$

Some comments on the inference rules are in order. In the rule (i^e_{op}), given the post type se', we calculate the pre-type se'' for the statement S_2. Then the resulting pre-type is used as a post-type for the statement S_1 to calculate the pre-type se. In line with Definition 2, the arithmetic statement $S_1\ i_{op}\ S_2$ is added to se only if it belongs to rs. Similar explanations illustrate the rule $(*^e)$. The remaining rules mimic the rules of the *ready* type system.

Now we recall the assumption that our distributed system consists of $|M|$ machines. For a given statement S and a given machine m, the type system given above calculates for each program point of S, the set of *semi-expectable* items. Now the following rule can be used to combine the information calculated for each machine to get a new *semi-expectable* information for each program point. The new *semi-expectable* information is valid on any of the $|M|$ machines.

$$\frac{\forall m \in M.\ S:(\sup\{p,p_j \mid j \neq i\},\sup\{rs,rs_j \mid j \neq i\},se_m) \rightarrow_m (A_m,p_m,rs_m,\inf\{se',se_j \mid j \neq i\})}{S:(p,rs,\inf\{se_1,\ldots,se_{|M|}\}) \rightarrow_M (\cup_i A_i,\sup\{p_1,\ldots,p_{|M|}\},\sup\{rs_1,\ldots,rs_{|M|}\},se')}\ (\mathit{main\text{-}rs})$$

The difference in the way that this rule treats the *semi-expectable* information and the way *ready* information is treated is explained by the fact that the *ready* analysis is forward while the the *semi-expectation* analysis is backward.

It is not hard to prove the soundness of the above type system:

Theorem 2. *Suppose that* $(S,\delta) \rightarrow (V,\delta')$, $S:(p,rs,se) \rightarrow (A',p',rs',se')$ *and the items of* se' *are semi-expectable at the point corresponding to* δ' *on the execution path. Then the items of* se *are semi-expectable at the point corresponding to* δ *on the execution path.*

4 Frequent Statement and De-reference Elimination

This section presents a type system that is an enrichment of the type system presented in the previous section. The type system of this section achieves the frequent statement and de-reference elimination. The type system uses a function $sn : S^+ \rightarrow \mathit{Stmt\text{-}names}$ that assigns each nontrivial statement a name. These names are meant to carry values of frequent statements and de-references. The judgments of our type system have the form $S:(p,rs,se) \rightarrow_m (A',p',rs',se') \Rrightarrow (ns,S')$. The type information (p,rs,se) and (A',p',rs',se') were calculated by the previous type system. S' is the optimization of S and ns is a sequence of assignments that links optimized statements with the names of their un-optimized versions. The inference rules for the frequent statements and de-references elimination are as follows:

$$\frac{n:p \rightarrow_m (A',p')}{n:(p,rs,se) \rightarrow_m (A',p',rs,se) \Rrightarrow (skip,n)} \qquad \frac{x:p \rightarrow_m (A',p')}{x:(p,rs,se) \rightarrow_m (A',p',rs,se) \Rrightarrow (skip,x)}$$

$$\frac{*S \in as \quad *S : p \rightarrow_m (A',p') \quad S : (p,rs,se) \rightarrow_m (A'',p'',rs'',se'') \Rrightarrow (ns,S')}{*S : (p,rs,se \cup (re \cap \{*S,g\})) \rightarrow_m \begin{cases} (A',p',rs'' \cup \{*S,g\},se''), \ A' = \{g\}; \\ (A',p',rs'' \cup \{*S\},se''), \quad |A'| \neq 1. \end{cases} \Rrightarrow (ns, sn(*S))} \; (*^f_1)$$

$$\frac{*S \notin as \quad *S \in se' \quad *S : p \rightarrow_m (A',p') \quad S : (p,rs,se) \rightarrow_m (A'',p'',rs'',se'') \Rrightarrow (ns,S')}{\begin{array}{c} *S : (p,rs,se \cup (re \cap \{*S,g\})) \rightarrow_m \begin{cases} (A',p',rs'' \cup \{*S,g\},se''), \ A' = \{g\}; \\ (A',p',rs'' \cup \{*S\},se''), \quad |A'| \neq 1. \end{cases} \\ \Rrightarrow (ns; sn(*S) := *S', sn(*S)) \end{array}} \; (*^f_2)$$

$$\frac{*S \notin as \quad *S \notin se' \quad *S : p \rightarrow_m (A',p') \quad S : (p,rs,se) \rightarrow_m (A'',p'',rs'',se'') \Rrightarrow (ns,S')}{*S : (p,rs,se \cup (re \cap \{*S,g\})) \rightarrow_m \begin{cases} (A',p',rs'' \cup \{*S,g\},se''), \ A' = \{g\}; \\ (A',p',rs'' \cup \{*S\},se''), \quad |A'| \neq 1. \end{cases} \Rrightarrow (ns, *S')} \; (*^f_3)$$

$$\frac{S_1 \ i_{op} \ S_2 \in re \quad \begin{array}{l} S_1 : (p,rs,se) \rightarrow_m (A'',p'',rs'',se'') \Rrightarrow (ns_1,S'_1) \\ S_2 : (p'',rs'',se'') \rightarrow_m (A',p',rs',se') \Rrightarrow (ns_2,S'_2) \end{array}}{\begin{array}{r} S_1 \ i_{op} \ S_2 : (p,rs,se \cup (rs \cap \{S_1 \ i_{op} \ S_2\})) \rightarrow_m (\emptyset,p',rs' \cup \{S_1 \ i_{op} \ S_2\},se') \\ \Rrightarrow (ns_1;ns_2, sn(S_1 \ i_{op} \ S_2)) \end{array}} \; (i^f_{op(1)})$$

$$\frac{\begin{array}{ll} S_1 \ i_{op} \ S_2 \notin as & S_1 : (p,rs,se) \rightarrow_m (A'',p'',rs'',se'') \Rrightarrow (ns_1,S'_1) \\ S_1 \ i_{op} \ S_2 \in se' & S_2 : (p'',rs'',se'') \rightarrow_m (A',p',rs',se') \Rrightarrow (ns_2,S'_2) \end{array}}{\begin{array}{r} S_1 \ i_{op} \ S_2 : (p,rs,se \cup (rs \cap \{S_1 \ i_{op} \ S_2\})) \rightarrow_m (\emptyset,p',rs' \cup \{S_1 \ i_{op} \ S_2\},se') \\ \Rrightarrow (ns_1;ns_2; sn(S_1 \ i_{op} \ S_2) := (S'_1 \ i_{op} \ S'_2), sn(S_1 \ i_{op} \ S_2)) \end{array}} \; (i^f_{op(2)})$$

$$\frac{\begin{array}{ll} S_1 \ i_{op} \ S_2 \notin as & S_1 : (p,rs,se) \rightarrow_m (A'',p'',rs'',se'') \Rrightarrow (ns_1,S'_1) \\ S_1 \ i_{op} \ S_2 \notin se' & S_2 : (p'',rs'',se'') \rightarrow_m (A',p',rs',se') \Rrightarrow (ns_2,S'_2) \end{array}}{\begin{array}{r} S_1 \ i_{op} \ S_2 : (p,rs,se \cup (rs \cap \{S_1 \ i_{op} \ S_2\})) \rightarrow_m (\emptyset,p',rs' \cup \{S_1 \ i_{op} \ S_2\},se') \\ \Rrightarrow (ns_1;ns_2, S'_1 \ i_{op} \ S'_2) \end{array}} \; (i^f_{op(3)})$$

$$\frac{}{skip : (p,rs,se) \rightarrow_m (\emptyset,p,rs,se) \Rrightarrow (skip,skip)}$$

$$\frac{x := S : p \rightarrow_m (A',p') \quad S : (p,rs,se) \rightarrow_m (A'',p'',rs'',se'') \Rrightarrow (sn,S')}{x := S : (p,rs,se) \rightarrow_m (A',p',rs'' \setminus \{S \in Stmt \mid x \in free(S)\},se'') \Rrightarrow (skip, ns; x := S')} \; (:=^f)$$

$$\frac{\begin{array}{l} S_1 : (p,rs,se) \rightarrow_m (A_1,p_1,as_1,se_1) \Rrightarrow (ns_1,S'_1) \\ S_2 : (p_1,as_1,se_1) \rightarrow_m (A_2,p_2,as_2,se_2) \Rrightarrow (ns_2,S'_2) \end{array} \quad S_1 \leftarrow S_2 : p \rightarrow_m (A',p')}{\begin{array}{c} S_1 \leftarrow S_2 : (p,rs,se) \rightarrow_m (A',p',as_2 \setminus (A' \cup \{S \mid S : p \rightarrow_m (A_s,\_) \ \& \ A_s \cap A' \neq \emptyset\}),se_2) \\ \Rrightarrow (skip, ns_1;ns_2;S'_1 \leftarrow S'_2) \end{array}} \; (\leftarrow^f)$$

$$\frac{\begin{array}{l} S_1 : (p,rs,se) \rightarrow_m (A'',p'',rs'',se'') \Rrightarrow (ns_1,S'_1) \\ S_2 : (p'',rs'',se'') \rightarrow_m (A',p',rs',se') \Rrightarrow (ns_2,S'_2) \end{array}}{S_1;S_2 : (p,rs,se) \rightarrow_m (A',p',rs',se') \Rrightarrow (ns_1;ns_2,S'_1;S'_2)} \; (seq^f)$$

$$\frac{S : (p,rs,se) \rightarrow_m (A'',p'',rs'',se'') \Rrightarrow (ns,S') \quad \begin{array}{l} S_t : (p,rs'',se'') \rightarrow_m (A',p',rs',se') \Rrightarrow (ns_t,S'_t) \\ S_f : (p,rs'',se'') \rightarrow_m (A',p',rs',se') \Rrightarrow (ns_f,S'_f) \end{array}}{if \ S \ then \ S_t \ else \ S_f : (p,rs,se) \rightarrow_m (A',p',rs',se') \Rrightarrow (skip, ns; if \ S' \ then \ ns_t;S'_t \ else \ ns_f;S'_f)} \; (if^f)$$

$$\frac{(\lambda x.S_1)[S_2/x] : (p,rs,se) \rightarrow_m (A',p',rs',se') \Rrightarrow (ns,S')}{(\lambda x.S_1)S_2 : (p,rs,se) \rightarrow_m (A',p',rs',se') \Rrightarrow (ns,S')} \; (appl^f)$$

$$\frac{S : (p,rs,se) \rightarrow (A'',p'',rs',se') \Rrightarrow (ns,S') \quad S_t : (p'',rs',se') \rightarrow_m (A',p',rs,se) \Rrightarrow (ns_t,S'_t)}{while \ S \ do \ S_t : (p,rs,se) \rightarrow_m (A',p',rs',se') \Rrightarrow (skip, ns; while \ S' \ do \ (ns_t;S'_t;ns))} \; (whl^f)$$

$$\frac{fd(name) : (p,rs,se) \rightarrow_m (A',p',rs',se') \Rrightarrow (ns,S')}{name : (p,rs,se) \rightarrow_m (A',p',rs',se') \Rrightarrow (ns,S')} \; (name^f)$$

$$\frac{S : (p,rs,se) \rightarrow_m (A',p',rs',se') \Rrightarrow (ns,S')}{\lambda x.S : (p,rs,se) \rightarrow_m (A',p',rs',se') \Rrightarrow (skip, ns; \lambda x.S')} \; (abs^f)$$

$$\frac{new_l : p \rightarrow_m (A', p')}{new_l : (p, rs, se) \rightarrow_m (A', p', rs, se) \Rrightarrow (skip, new_l)}\ (new^f)$$

$$\frac{(\lambda x.S')S : (p, rs, se) \rightarrow_m (A', p', rs', se') \Rrightarrow (ns, S'')}{letrec\ x = S\ in\ S' : (p, rs, se) \rightarrow_m (A', p', rs', se') \Rrightarrow (ns, S'')}\ (letrec^f)$$

$$\frac{convert\ (S, n) : p \rightarrow_m (A', p') \qquad S : (p, rs, se) \rightarrow_m (A, p'', rs', se') \Rrightarrow (ns, S')}{convert\ (S, n) : (p, rs, se) \rightarrow_m (A', p', rs', se') \Rrightarrow (skip, ns; convert\ (S', n))}\ (convert^f)$$

$$\frac{\begin{array}{l} transmit\ S_1\ from\ S_2 : p \rightarrow (A', p') \\ S_2 : (p, rs, se) \rightarrow_m (A_2, p_2, as_2, se_2) \Rrightarrow (ns_2, S'_2) \\ S_1 : (p_2, as_2, se_2) \rightarrow_m (A_1, p_1, rs', se') \Rrightarrow (ns_1, S'_1) \end{array}}{transmit\ S_1\ from\ S_2 : (p, rs, se) \rightarrow_m (A', p', rs', se') \Rrightarrow (ns_1; ns_2, transmit\ S'_1\ from\ S'_2)}\ (trans^f)$$

$$\frac{\begin{array}{l} (p'_1, rs'_1, se'_1) \leq (p_1, as_1, pa_1) \\ (p_2, as_2, pa_2) \leq (p'_2, rs'_2, se'_2) \end{array} \quad S : (p_1, as_1, pa_1) \rightarrow_m (p_2, as_2, pa_2) \Rrightarrow (ns, S')}{S : (p'_1, rs'_1, pa_1) \rightarrow_m (p'_2, rs'_2, se'_2) \Rrightarrow (ns, S')}\ (csq^f)$$

$$\frac{Defs : \emptyset \curvearrowright fd \quad S : (p, rs, se) \rightarrow_m (A, p', rs', se') \Rrightarrow (ns, S')}{Defs : S : (p, rs, se) \rightarrow_m (A, p', rs', se') \Rrightarrow Defs : ns; S'}\ (prg^f)$$

We note the following on the inference rules. A big deal of optimization is achieved by the three rules for $*S$. These rules are $(*^f_1)$, $(*^f_2)$, and $(*^f_3)$. The rule $(*^f_1)$ takes care of the case where $*S$ is *ready* and is replaceable by its name under the function *sn*. The rule $(*^f_2)$ treats the case where $*S$ is *semi-expectable* and is not *ready* before calculating the statement. In this case, a statement name of $*S$ is used. The rule $(*^f_3)$ considers the case where $*S$ is neither *semi-expectable* at the program point after execution nor *ready* before calculating the statement. In this case, the statement $*S$ does not get changed. Similarly, the three rules $(i^f_{op(1)})$, $(i^f_{op(2)})$, and $(i^f_{op(3)})$ treat different cases for arithmetic statements. The Boolean statements are treated with rules quite similar to that of arithmetic statements. The rule (whl^f) reuses frequent sub-statements of the guard. This is done via adding *ns* in the positions clarified in the rule. Remaining rules of system are self-explanatory.

For expressing the soundness, we introduce the following definition:

Definition 3. *Suppose that δ is a state defined on the set of locations, Loc (Definition 4). Suppose also that δ_* is a state defined on $Loc \cup Stmt\text{-}names$. The expression $\delta \equiv_{se} \delta_*$ denotes the fact that δ and δ_* are equivalent with respect to the semi-expectation type se. More precisely $\delta \equiv_{se} \delta_*$ iff:*

1. $\forall j \in Loc.\ \delta(j) = \delta_*(j)$, and
2. $\forall S \in se.\ (S, \delta) \rightsquigarrow_m (v, \delta') \Longrightarrow \delta_*(sn(S)) = v$.

The soundness of frequent statements and de-references elimination means that the original and optimized programs are equivalent in the following sense:

– The states of the two programs coincide on the *Loc*, and
– If a statement is both *ready* and *semi-expectable*, then its semantics in the original-program state equals the value of its corresponding name in optimized-program state.

This gives an intuition to the previous definition. The following soundness theorem is proved by a structure induction.

Theorem 3. *Suppose that* $S : (p, rs, se) \rightarrow_m (A', p', rs', se') \Rrightarrow (ns, S')$ *and* $\delta \equiv_{se} \delta_*$. *Then*

- $(S, \delta) \rightsquigarrow_m (v, \delta') \Longrightarrow \exists \delta'_*.\ \delta' \equiv_{se'} \delta'_*$ *and* $(S', \delta_*) \rightsquigarrow_m (v, \delta'_*)$.
- $(S', \delta_*) \rightsquigarrow_m (v, \delta'_*) \Longrightarrow \exists \delta'.\ \delta' \equiv_{se'} \delta'_*$ *and* $(S, \delta) \rightsquigarrow_m (v, \delta')$.

5 Related and Future Work

The techniques of common sub-expression elimination (CSE) [18,17] are among the most closed to our work. In [27], a type system for CSE of the *while* language is introduced. The work presented in our paper can be realized as a generalization of that presented in [27]. The generality of our work is evident in our language model which is much richer with distributed and pointer commands. Consequently, the operational semantics that we measure the soundness of our system against is much more involved than that used in [27]. Using new opportunities appearing while scheduling of control-intensive designs, the work in [25] introduces a technique that dynamically eliminates CSE. To optimize polynomial expressions (important for applications like domains, computer graphics, and signal processing), the paper [19] generalizes algebraic techniques originally designed for multilevel logic synthesis. This generalization uses factoring and eliminating common subexpressions.

The association of a correctness proof with each result of the static analysis is important and needed by applications like proof-carrying code and certified code. The work presented in this paper has the advantage over most related work of constructing these proofs. Adding to the value of using type systems, the proofs constructed in our proposed approach have the form of type derivations. The work in [6,4,15,5,8,9,7,16] presents many examples of other static analyses that are in the form of type systems.

In [21], a technique for flow-insensitive pointer analysis of programs that run on parallel and hierarchical machines and that share memory is introduced. Via a two-level hierarchy, [22] and [23] present constraint-based approaches to evaluate locality information and sharing attributes of references. Our language model is a generalization of models presented in [21,22].

Much research acclivities [21,24] was devoted to analyze distributed programs. This is motivated by the importance of distributed programming as a main stream of programming today. The examining and capturing of causal and concurrent relationships are among important issues to many distributed systems applications. In [28], an analysis that examines the source code of each process constructs an inclusive graph, POG, of the possible behaviors of systems. Data racing bugs [2] can be a side effect of the parallel access of cores of a multi-core process to a physically distributed memory. In [2] a technique, called DRARS, is proposed for avoidance and replay of this data race. Parallel programs on DSM or multi-core systems, can be debugged using DRARS. The classical problems of satisfiability decidability and algorithmic decidability are approached in [29] on the distributed-programs model of message sending. In this work, distributed programs are represented by communicating via buffers.

Mathematical domains (sets) and maps between domains mathematically represent data structures and programs in the area of denotational semantics. For future work, it is interesting to study the possibility of translating concepts of frequent statement and de-reference elimination to the side of denotational semantics [14,3]. This translation has the impact of easing achieving theoretical studies about frequent statement and de-reference elimination. Established theoretical results may then be translated back to the side of data structures and programs. Similarly, we also intend to test the gains of applying the type systems approach on the problems treated by our work in [10,12,13].

6 Conclusion

This paper introduces a new technique for the analysis of *frequent statement and de-reference elimination* for distributed programs running on parallel machines equipped with hierarchical memories. Type systems are the tools of the techniques presented in this paper which presents three type systems. The first type system defines for program points of a given distributed program sets of calculated (*ready*) statements and memory accesses. The second type system determines which of the ready statements and memory accesses are used later in the program. The third type system eliminates unnecessary statement computations and memory accesses.

References

1. Barpanda, S.S., Mohapatra, D.P.: Dynamic slicing of distributed object-oriented programs. IET Software 5(5), 425–433 (2011)
2. Chiu, Y.-C., Shieh, C.-K., Huang, T.-C., Liang, T.-Y., Chu, K.-C.: Data race avoidance and replay scheme for developing and debugging parallel programs on distributed shared memory systems. Parallel Computing 37(1), 11–25 (2011)
3. El-Zawawy, M.A.: Semantic spaces in Priestley form. PhD thesis, University of Birmingham, UK (January 2007)
4. El-Zawawy, M.A.: Flow sensitive-insensitive pointer analysis based memory safety for multithreaded programs. In: Murgante, B., Gervasi, O., Iglesias, A., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2011, Part V. LNCS, vol. 6786, pp. 355–369. Springer, Heidelberg (2011)
5. El-Zawawy, M.A.: Probabilistic pointer analysis for multithreaded programs. ScienceAsia 37(4), 344–354 (2011)
6. El-Zawawy, M.A.: Program optimization based pointer analysis and live stack-heap analysis. International Journal of Computer Science Issues 8(2), 98–107 (2011)
7. El-Zawawy, M.A.: Abstraction analysis and certified flow and context sensitive points-to relation for distributed programs. In: Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A.M.A.C., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2012, Part IV. LNCS, vol. 7336, pp. 83–99. Springer, Heidelberg (2012)
8. El-Zawawy, M.A.: Dead code elimination based pointer analysis for multithreaded programs. Journal of the Egyptian Mathematical Society 20(1), 28–37 (2012)
9. El-Zawawy, M.A.: Heap slicing using type systems. In: Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A.M.A.C., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2012, Part III. LNCS, vol. 7335, pp. 592–606. Springer, Heidelberg (2012)
10. El-Zawawy, M.A.: Recognition of logically related regions based heap abstraction. Journal of the Egyptian Mathematical Society 20(2) (September 2012)

11. El-Zawawy, M.A.: Detection of probabilistic dangling references in multi-core programs using proof-supported tools. In: Murgante, B., et al. (eds.) ICCSA 2013, Part III. LNCS, vol. 7973, Springer, Heidelberg (2013)
12. El-Zawawy, M.A., Daoud, N.M.: Dynamic verification for file safety of multithreaded programs. IJCSNS International Journal of Computer Science and Network Security 12(5), 14–20 (2012)
13. El-Zawawy, M.A., Daoud, N.M.: New error-recovery techniques for faulty-calls of functions. Computer and Information Science 5(3), 67–75 (2012)
14. El-Zawawy, M.A., Jung, A.: Priestley duality for strong proximity lattices. Electr. Notes Theor. Comput. Sci. 158, 199–217 (2006)
15. El-Zawawy, M.A., Nayel, H.A.: Partial redundancy elimination for multi-threaded programs. IJCSNS International Journal of Computer Science and Network Security 11(10), 127–133 (2011)
16. El-Zawawy, M.A., Nayel, H.A.: Type systems based data race detector. IJCSNS International Journal of Computer Science and Network Security 5(4), 53–60 (2012)
17. Gopalakrishnan, S., Kalla, P.: Algebraic techniques to enhance common sub-expression elimination for polynomial system synthesis. In: DATE, pp. 1452–1457. IEEE (2009)
18. Ho, H., Szwarc, V., Kwasniewski, T.A.: Low complexity reconfigurable dsp circuit implementations based on common sub-expression elimination. Signal Processing Systems 61(3), 353–365 (2010)
19. Hosangadi, A., Fallah, F., Kastner, R.: Optimizing polynomial expressions by algebraic factorization and common subexpression elimination. IEEE Trans. on CAD of Integrated Circuits and Systems 25(10), 2012–2022 (2006)
20. Seragiotto Jr., C.: Thomas Fahringer. Performance analysis for distributed and parallel java programs with aksum. In: CCGRID, pp. 1024–1031. IEEE Computer Society (2005)
21. Kamil, A., Yelick, K.A.: Hierarchical pointer analysis for distributed programs. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 281–297. Springer, Heidelberg (2007)
22. Liblit, B., Aiken, A.: Type systems for distributed data structures. In: POPL, pp. 199–213 (2000)
23. Liblit, B., Aiken, A., Yelick, K.A.: Type systems for distributed data sharing. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 273–294. Springer, Heidelberg (2003)
24. Lindberg, P., Leingang, J., Lysaker, D., Khan, S.U., Li, J.: Comparison and analysis of eight scheduling heuristics for the optimization of energy consumption and makespan in large-scale distributed systems. The Journal of Supercomputing 59(1), 323–360 (2012)
25. Nicolau, A., Dutt, N.D., Gupta, R., Savoiu, N., Reshadi, M., Gupta, S.: Dynamic common sub-expression elimination during scheduling in high-level synthesis. In: ISSS, pp. 261–266. IEEE Computer Society (2002)
26. Onbay, T.U., Kantarci, A.: Design and implementation of a distributed teleradiaography system: Dipacs. Computer Methods and Programs in Biomedicine 104(2), 235–242 (2011)
27. Saabas, A., Uustalu, T.: Program and proof optimizations with type systems. Journal of Logic and Algebraic Programming 77(1-2), 131–154 (2008); The 16th Nordic Workshop on the Prgramming Theory (NWPT 2006)
28. Simmons, S., Edwards, D., Kearns, P.: Communication analysis of distributed programs. Scientific Programming 14(2), 151–170 (2006)
29. Toporkov, V.V.: Dataflow analysis of distributed programs using generalized marked nets. In: DepCoS-RELCOMEX, pp. 73–80. IEEE Computer Society (2007)
30. Truong, H.L., Fahringer, T.: Soft computing approach to performance analysis of parallel and distributed programs. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 50–60. Springer, Heidelberg (2005)

Appendix: Memory Model, Language, and Operational Semantics

This appendix briefly reviews our language syntax, memory model, and operational semantics which are presented in more details in [7]. Hierarchical [24,26] memory models are typically used in parallel computers (Figure 2).

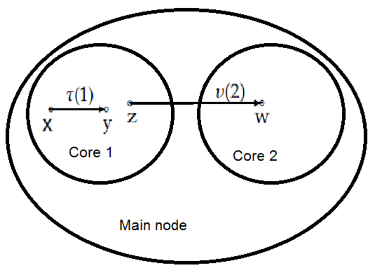

Fig. 2. Hierarchy memory

In PGAS languages, each pointer is assigned the memory-hierarchy level that the pointer may reference. Two-levels hierarchy is shown in Figure 2 in which τ is a core pointer and may point to addresses on core 1 and υ is a node pointer. These pointer domains are represented via assigning a width (number) to each pointer. Increasing hierarchy levels is the trend in hardware research. Hence benefiting from the hierarchy [24,26] is very important for software.

$$\begin{aligned}
name &::= \textit{'string of characters'}.\\
S \in Stmts &::= n \mid true \mid false \mid x \mid S_1\ i_{op}\ S_2 \mid S_1\ b_{op}\ S_2 \mid *S \mid skip \mid name \mid x := S \mid S_1 \leftarrow S_2 \mid\\
&\quad\ S_1;S_2 \mid if\ S\ then\ S_t\ else\ S_f \mid while\ S\ do\ S_t \mid \lambda x.S \mid S_1 S_2 \mid letrec\ x = S\ in\ S' \mid\\
&\quad\ new_l \mid convert\ (S,n) \mid transmit\ S_1\ from\ S_2.\\
Defs &::= (name = S);Defs \mid \varepsilon.\\
Program &::= Defs : S.
\end{aligned}$$

where
$x \in lVar$, an infinite set of variables, $n \in \mathbb{Z}$ (integers), $i_{op} \in \mathbb{I}_{op}$ (integer-valued binary operations), and $b_{op} \in \mathbb{B}_{op}$ (Boolean-valued binary operations).

Fig. 3. The programming language *while*$_d$

Figure 3 presents the syntax of $While_d$ [21,22] used to present the results of this paper. $While_d$ has pointer, parallelism, and function constructions [24,26] and supports the SPMD parallelism model in which the same code is executed on all machines. The hierarchy height of memory is denoted by h. Therefore, pointer widths are in [1,h]. The language uses a fixed set of variables, *lVar*, each of which is machine-private. For each program, the semantics specifies a map *fd* of the domain *Function-defs*:

$$fd \in \mathit{Function\text{-}defs} = \text{'}\mathit{strings\ of\ characters}\text{'} \longrightarrow \mathit{Stmts}.$$

This map is defined using the following inference rules:

$$\frac{}{\varepsilon : fd \curvearrowright fd}\,(fd_1) \qquad \frac{\mathit{Defs} : fd[\mathit{name} \mapsto S] \curvearrowright fd'}{(\mathit{name} = S);\mathit{Defs} : fd \curvearrowright fd'}\,(fd_2)$$

The semantics introduces transition relations $\rightsquigarrow_m$; between pairs of statements and states and pairs of values and states. The following definition introduces components used in the inference rules of the semantics.

Definition 4. *1. The set of global variables, denotes by gVar, is defined as* $gVar = \{(m,x) \mid m \in M, x \in lVar\}$.

2. *The set of global addresses, denotes by gAddrs, is defined as* $gAddrs = \{g = (l,m,a) \mid l \in L, m \in M, a \in lAddrs\}$.
3. $loc \in Loc = gAddrs \cup gVar$.
4. $v \in Values = \mathbb{Z} \cup gAddrs \cup \{true, false\} \cup \{\lambda x.S \mid S \in Stmt\}$.
5. $\delta \in States = Loc \longrightarrow Values$.

The symbols M, W, *and lAddrs denote the set of machines labels (integers), the set of width* $\{1,\ldots,h\}$, *and the set of local addresses located on each single machine, respectively. The set of labels of allocation sites is denoted by L.*

The semantics judgments have the form $(S,\delta) \rightsquigarrow_m (v,\delta')$ meaning that executing S on the machine m and in the state δ produces the value v and modifies δ to δ'. We have $\delta[x \mapsto v] \Longleftrightarrow \lambda y.$ if $y = x$ then v else $\delta(y)$.

The inference rules of our semantics are as follows:

$$(n,\delta) \rightsquigarrow_m (n,\delta) \qquad (\mathit{true},\delta) \rightsquigarrow_m (\mathit{true},\delta) \qquad (\mathit{false},\delta) \rightsquigarrow_m (\mathit{false},\delta) \qquad (x,\delta) \rightsquigarrow_m (\delta(x),\delta)$$

$$(\lambda x.S,\delta) \rightsquigarrow_m (\lambda x.S,\delta)(\mathit{abs}) \qquad \frac{(S_1,\delta) \rightsquigarrow_m (n_1,\delta'') \quad (S_2,\delta'') \rightsquigarrow_m (n_2,\delta')}{(S_1\ i_{op}\ S_2,\delta) \rightsquigarrow_m \begin{cases} (n_1\ i_{op}\ n_2,\delta'), & n_1\ i_{op}\ n_2 \in \mathbb{Z}; \\ \mathit{abort}, & \text{otherwise.} \end{cases}}\,(\mathit{int\text{-}stmt})$$

$$\frac{(S_1,\delta) \rightsquigarrow_m (b_1,\delta'') \quad (S_2,\delta'') \rightsquigarrow_m (b_2,\delta')}{(S_1\ b_{op}\ S_2,\delta) \rightsquigarrow_m \begin{cases} (b_1\ b_{op}\ b_2,\delta'), & b_1\ b_{op}\ b_2 \text{ is a Boolean value;} \\ \mathit{abort}, & \text{otherwise.} \end{cases}}\,(\mathit{bo\text{-}stmt}) \qquad (\mathit{skip},\delta) \rightsquigarrow_m (0,\delta)$$

$$\frac{(S,\delta) \rightsquigarrow_m (g,\delta')}{(*S,\delta) \rightsquigarrow_m \begin{cases} (\delta'(g),\delta'), & g \in gAddrs; \\ \mathit{abort}, & \text{otherwise.} \end{cases}}\,(\mathit{de\text{-}ref}) \quad \frac{(S,\delta) \rightsquigarrow_m \mathit{abort}}{(x := S,\delta) \rightsquigarrow_m \mathit{abort}} \quad \frac{(S,\delta) \rightsquigarrow_m (v,\delta')}{(x := S,\delta) \rightsquigarrow_m (v,\delta'[x \mapsto v])} \quad \frac{(S_1,\delta) \rightsquigarrow_m \mathit{abort}}{(S_1;S_2,\delta) \rightsquigarrow_m \mathit{abort}}$$

$$\frac{(S_1,\delta) \rightsquigarrow_m \mathit{abort} \text{ or for } v \notin gAddrs.\ (S_1,\delta) \rightsquigarrow_m (v,\delta'')}{(S_1 \leftarrow S_2,\delta) \rightsquigarrow_m \mathit{abort}}\,(\leftarrow_1) \quad \frac{\begin{array}{c}(S_1,\delta) \rightsquigarrow_m (v,\delta'') \\ (S_2,\delta'') \rightsquigarrow_m \mathit{abort}\end{array}}{(S_1 \leftarrow S_2,\delta) \rightsquigarrow_m \mathit{abort}}\,(\leftarrow_2) \quad \frac{(S_1,\delta) \rightsquigarrow_m \mathit{abort}}{(S_1;S_2,\delta) \rightsquigarrow_m \mathit{abort}}$$

$$\frac{\begin{array}{c}(S_1,\delta) \rightsquigarrow_m (g,\delta'') \\ (S_2,\delta'') \rightsquigarrow_m (v,\delta''') \\ g \in gAddrs\end{array}}{(S_1 \leftarrow S_2,\delta) \rightsquigarrow_m (v,\delta'''[g \mapsto v])}\,(\leftarrow_3) \quad \frac{\begin{array}{c}(S_1,\delta) \rightsquigarrow_m (v_1,\delta'') \\ (S_2,\delta'') \rightsquigarrow_m (v_2,\delta')\end{array}}{(S_1;S_2,\delta) \rightsquigarrow_m (v_2,\delta')} \quad \frac{\begin{array}{c}(S_1,\delta) \rightsquigarrow_m (v_1,\delta'') \\ (S_2,\delta'') \rightsquigarrow_m \mathit{abort}\end{array}}{(S_1;S_2,\delta) \rightsquigarrow_m \mathit{abort}} \quad \frac{\begin{array}{c}(S,\delta) \rightsquigarrow_m (\mathit{true},\delta'') \\ (S_t,\delta'') \rightsquigarrow_m \mathit{abort}\end{array}}{(\mathit{if}\ S\ \mathit{then}\ S_t\ \mathit{else}\ S_f,\delta) \rightsquigarrow_m \mathit{abort}}$$

$$\frac{\begin{array}{c}(S,\delta) \rightsquigarrow_m (\mathit{true},\delta'')\\ (S_t,\delta'') \rightsquigarrow_m (v,\delta')\end{array}}{(\mathit{if}\ S\ \mathit{then}\ S_t\ \mathit{else}\ S_f,\delta) \rightsquigarrow_m (v,\delta')} \quad \frac{\begin{array}{c}(S,\delta) \rightsquigarrow_m (\mathit{false},\delta'')\\ (S_f,\delta'') \rightsquigarrow_m \mathit{abort}\end{array}}{(\mathit{if}\ S\ \mathit{then}\ S_t\ \mathit{else}\ S_f,\delta) \rightsquigarrow_m \mathit{abort}} \quad \frac{\begin{array}{c}(S,\delta) \rightsquigarrow_m (\mathit{false},\delta'')\\ (S_f,\delta'') \rightsquigarrow_m (v,\delta')\end{array}}{(\mathit{if}\ S\ \mathit{then}\ S_t\ \mathit{else}\ S_f,\delta) \rightsquigarrow_m (v,\delta')}$$

$$\frac{(S,\delta) \rightsquigarrow_m \mathit{abort}}{(\mathit{if}\ S\ \mathit{then}\ S_t\ \mathit{else}\ S_f,\delta) \rightsquigarrow_m \mathit{abort}} \quad \frac{(S,\delta) \rightsquigarrow_m \mathit{abort}}{(\mathit{while}\ S\ \mathit{do}\ S_t,\delta) \rightsquigarrow_m \mathit{abort}} \quad \frac{(S,\delta) \rightsquigarrow_m (\mathit{false},\delta'')}{(\mathit{while}\ S\ \mathit{do}\ S_t,\delta) \rightsquigarrow_m (\mathit{skip},\delta)}$$

$$\frac{\begin{array}{c}(S,\delta) \rightsquigarrow_m (\mathit{true},\delta'')\\ (S_t,\delta'') \rightsquigarrow_m \mathit{abort}\end{array}}{(\mathit{while}\ S\ \mathit{do}\ S_t,\delta) \rightsquigarrow_m \mathit{abort}} \quad \frac{\begin{array}{c}(S,\delta) \rightsquigarrow_m (\mathit{true},\delta'')\\ (S_t,\delta'') \rightsquigarrow_m (v'',\delta'')\\ (\mathit{while}\ S\ \mathit{do}\ S_t,\delta'') \rightsquigarrow_m (v',\delta')\end{array}}{\mathit{while}\ S\ \mathit{do}\ S_t : (\delta,p) \rightsquigarrow_m (v',\delta')} \quad \frac{\begin{array}{c}(S,\delta) \rightsquigarrow_m (\mathit{true},\delta'')\\ (S_t,\delta'') \rightsquigarrow_m (v'',\delta'')\\ (\mathit{while}\ S\ \mathit{do}\ S_t,\delta'') \rightsquigarrow_m \mathit{abort}\end{array}}{(\mathit{while}\ S\ \mathit{do}\ S_t,\delta) \rightsquigarrow_m \mathit{abort}}$$

$$\frac{\begin{array}{c}(S_1,\delta) \rightsquigarrow_m (\lambda x.S_1',\delta'')\\ (S_1'[S_2/x],\delta'') \rightsquigarrow_m (v,\delta')\end{array}}{(S_1S_2,\delta) \rightsquigarrow_m (v,\delta')}\ (\mathit{appl}) \quad \frac{\begin{array}{c}(S,\delta) \rightsquigarrow_m (v,\delta'')\\ (S'[v/x],\delta'') \rightsquigarrow_m (v',\delta')\end{array}}{(\mathit{letrec}\ x = S\ \mathit{in}\ S',\delta) \rightsquigarrow_m (v',\delta')}\ (\mathit{letrec}) \quad \frac{\begin{array}{c}a \in \mathit{lAddrs}\\ a \text{ is fresh on } m\end{array}}{(\mathit{new}_l,\delta) \rightsquigarrow_m ((l,m,a),\delta[(l,m,a) \mapsto \mathit{null}])}$$

$$\frac{\begin{array}{c}(S,\delta) \rightsquigarrow_m (g = (l,m',a),\delta')\\ \mathrm{hdist}(m,m') \leq n\end{array}}{(\mathit{convert}(S,n),\delta) \rightsquigarrow_m (g,\delta')}\ (\mathit{conv}) \quad \frac{(\mathit{fd}(\mathit{name}),\delta) \rightsquigarrow_m v,\delta')}{(\mathit{name},\delta) \rightsquigarrow_m (v,\delta')}\ (\mathit{name}) \quad \frac{\begin{array}{c}(S_2,\delta) \rightsquigarrow_m (m',\delta'')\\ m' \in M \quad (S_1,\delta'') \rightsquigarrow_{m'} (v,\delta')\end{array}}{(\mathit{transmit}\ S_1\ \mathit{from}\ S_2,\delta) \rightsquigarrow_m (v,\delta')}\ (\mathit{trans})$$

$$\frac{\begin{array}{l}\theta : \{1,2,\ldots,|M|\} \to M\\ (S,\delta) \rightsquigarrow_{\theta(1)} (v_1,\delta_1) \rightsquigarrow_{\theta(2)} (v_2,\delta_2) \rightsquigarrow_{\theta(3)} \ldots \rightsquigarrow_{\theta(|M|)} (v_{|M|},\delta_{|M|})\end{array}}{(\mathit{Defs} : S,\delta) \rightsquigarrow_M (v_{|M|},\delta_{|M|})}\ (\mathit{main\text{-}sem})$$

The semantics of running a program *Defs* : *S* on the distributed systems is treated by the rule (*main-sem*).