Support of Probabilistic Pointer Analysis in the SSA Form

Ming-Yu Hung, Peng-Sheng Chen, *Member*, *IEEE*, Yuan-Shin Hwang, Roy Dz-Ching Ju, *Senior Member*, *IEEE*, and Jenq-Kuen Lee

Abstract—Probabilistic pointer analysis (PPA) is a compile-time analysis method that estimates the probability that a points-to relationship will hold at a particular program point. The results are useful for optimizing and parallelizing compilers, which need to quantitatively assess the profitability of transformations when performing aggressive optimizations and parallelization. This paper presents a PPA technique using the static single assignment (SSA) form. When computing the probabilistic points-to relationships of a specific pointer, a pointer relation graph (PRG) is first built to represent all of the possible points-to relationships of the pointer. The PRG is transformed by a sequence of reduction operations into a compact graph, from which the probabilistic points-to relationships of the pointer can be determined. In addition, PPA is further extended to interprocedural cases by considering function related statements. We have implemented our proposed scheme including static and profiling versions in the Open64 compiler, and performed experiments to obtain the accuracy and scalability. The static version estimates branch probabilities by assuming that every conditional is equally likely to be true or false, and that every loop executes 10 times before terminating. The profiling version measures branch probabilities dynamically from past program executions using a default workload provided with the benchmark. The average errors for selected benchmarks were 3.80 percent in the profiling version and 9.13 percent in the static version. Finally, SPEC CPU2006 is used to evaluate the scalability, and the result indicates that our scheme is sufficiently efficient in practical use. The average analysis time was 35.59 seconds for an average of 98,696 lines of code.

Index Terms—Compiler, pointer analysis, control flow graph (CFG), static single assignment (SSA) form

1 Introduction

Pointer analysis is a compiler analysis technique that statically estimates the possible runtime values of a pointer. Because of the dynamic association property of pointers in programs, it is difficult for compilers to know where pointers may point to in general. The absence of such knowledge makes conservative assumptions about pointer information, which can impede aggressive optimizations. Considerable efforts on this topic have led to the development of many intra and interprocedural pointer analysis algorithms [1], [2], [3], [4], [5], [6], [7]. The computational cost of gathering pointer information is also an important consideration. Hind and Pioli [8], [9] summarized that the complexity of existing methods was from almost linear [2] to doubly exponential [10]. The efficiency was also reported by measuring the analysis time and memory consumption of the pointer analyzers and their clients. In order to reduce the cost of pointer analysis, a static single assignment (SSA)-based approach is proposed in this paper. The built-in explicit use-definition (use-def) chains in SSA help the analyzer to track related locations or addresses at a lower cost.

The pointer information gathered by the traditional points-to analysis techniques [1], [2], [3], [4], [5] can be categorized into two classes: *definitely-points-to* relationships and *possibly-points-to* relationships. Such information may not be precise enough to direct certain optimization, since the information does not describe how likely that *possibly-points-to* relationships hold, and hence quantitative descriptions are needed for modern compiler optimizations. Probabilistic pointer analysis (PPA) techniques have been proposed to quantify points-to relationships [11]. PPA can facilitate the compiler to determine whether it is beneficial to perform certain optimizations, such as speculative multithreading execution [12], data speculation [7], data prefetching [13], and transactional memory [14], as these optimizations will show a profit when specific points-to relationships hold with high or low probabilities. PPA is useful for speculative multithreading execution for threads with pointer-induced data dependencies. PPA provides the essential information for a compiler to estimate the likelihood of dependencies so that it can maximize the number of threads for parallel execution, but minimize the chance of dependence violations between threads. Therefore, it can help a compiler to achieve speedups by executing speculative threads when the possibilities of conflicts are low, and can avoid slowdown by turning off thread speculation if the possibilities are high. In a distributed environment, PPA can also be applied to manage data distributions. Each points-to location has probabilities assigned to it by pointers, so the reuse and use frequencies of locations can be deduced

- *M.-Y. Hung and J.-K. Lee are with the Department of Computer Science, National Tsing Hua University, No. 101, Section 2, Kuang-Fu Road, Hsinchu 30013, Taiwan. E-mail: myhung@pllab.cs.nthu.edu.tw, jklee@cs.nthu.edu.tw.*
- *P.-S. Chen is with the Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi 621, Taiwan. E-mail: pschen@cs.ccu.edu.tw.*
- *Y.-S. Hwang is with the Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, 43 Keelung Road, Section 4, Taipei 106, Taiwan. E-mail: shin@csie.ntust.edu.tw.*
- *R.D.-C. Ju is with Advanced Micro Devices, Inc., 1 AMD Place, MS 25, PO Box 3453, Sunnyvale, CA 94088. E-mail: roy.ju@amd.com.*

Manuscript received 4 Aug. 2011; revised 2 Dec. 2011; accepted 12 Feb. 2012; published online 17 Feb. 2012.
Recommended for acceptance by M. Kandemir.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2011-08-0514.
Digital Object Identifier no. 10.1109/TPDS.2012.73.

from PPA information. This information can be used by the optimizer to arrange a better memory layout for a pointer program that bridges the latency gap between internal and external memory [15].

PPA was pioneered to report quantitative points-to relationships by providing a set of transfer functions of data flow analysis for the usage of each pointer [6], [7], [11]. Assume that the probability for each points-to pair is required when collecting data flow information. When this data flow analysis converges, the equations containing the unknown probability for each points-to pair must be solved to obtain probabilistic information. However, the normally large amount of unknown relationships collected by a data flow analysis makes the compiler analysis of previous PPA techniques impractical for large programs. The complexity of data-flow PPA is $O(V_{df}N_{df}) + O(E_{df}^3)$ where V_{df} is the number of nodes in CFG, N_{df} is the maximum number of points-to relationship, and E_{df} is the number variables in the equation [6]. Da Silva and Steffan [16] also proposed a method of probabilistic pointer analysis based on sparse transformation matrices. In their algorithm the pointer information was modeled as a points-to matrix, and transformation matrices recorded the influences of points within a statement or a set of statements. Without the use-def chains of SSA, this method must collect information about all of the pointers even when information is only needed for one or few of them. In this paper, we present a PPA scheme based on the SSA form. PPA using the SSA form quantifies each points-to relationship as a probability. A *possibly-points-to* relationship is represented by a probability between 0 and 100 percent. This SSA-based approach significantly reduces the complexity of PPA to $O(N) + O(VE)$ where N is the number of identifications (IDs) in SSA, V is the number of related pointers, and E is the number of use-def chains connecting V by utilizing the explicit information about use-def chains in the SSA form.

Since PPA analyses pointers, it must be extended to have the ability to deal with indirect operators. This was achieved in the present study by proposing a set of algorithms that compute the probabilistic points-to relationships of a specific pointer. The algorithms use the memory SSA information to connect related pointers as a graph and to use the edge weights from the execution frequencies of the CFG to associate the probabilities with points-to relationships. In addition *PPA* is further extended to interprocedural cases by adding function calls in the program representation. Here, we also present running examples to illustrate our proposed algorithms.

To our best knowledge, this is the first study to devise a scheme for probabilistic pointer analysis based on the SSA form. We have implemented it in the global optimization (WOPT) phase of the Open64 compiler, and have developed two versions for obtaining edge frequencies: static assignment and profiling feedback, respectively. Our experimental results show that the probability distributions of points-to relationships are almost all in extreme high or low regions, and the compiler can confidently make decisions about whether or not to perform aggressive pointer-induced optimizations for the pointers in these regions. The average errors were measured using pointer related benchmarks, Olden and CHJL [6]; these were 3.80 percent in the profiling version and 9.13 percent in the static version by comparing with actual runtime executions. SPEC CPU2006 benchmark [17] is further used to evaluate the scalability of PPA, and the result also demonstrates that our scheme is sufficiently efficient for practical use. The average analysis time was 35.59 seconds for the benchmark with an average of 98,696 lines of code.

This paper makes the following contributions:

1. Points-to relationships are presented by quantitative probability instead of qualitative information, *may* or *must*.
2. Compared with previous work, proposed algorithms improve the analysis speed and complexity.
3. Olden and CHJL are used to evaluate the accuracy and SPEC CPU2006 benchmarks are used for scalability evaluation of PPA in the SSA form.

The remainder of this paper is organized as follows: Section 2 defines the problem and terminologies. Section 3 presents our proposed PPA algorithm and also the running examples to illustrate our proposed algorithms. The experimental results are described in Section 4. Finally, Section 5 lists related work, and conclusions are drawn in Section 6.

2 Problem Specifications and Terminologies

The goal of SSA-based probabilistic pointer analysis is to compute the probabilities of points-to relationships for a pointer, p, at a program point, s.

2.1 Problem Specifications

For each points-to relationship, assume that p points to a location, v, denoted as a tuple $\langle p,\ v\rangle$. The *probability function*, $\mathcal{P}(s,\ \langle p,\ v\rangle)$, computes the probability that pointer p points to v at s according to the following equation:

$$\mathcal{P}(s,\ \langle p,\ v\rangle) \stackrel{\text{def}}{=} \begin{cases} \dfrac{N(s,\ \langle p,\ v\rangle)}{N(s,\perp)} & \text{if N}(s,\perp) \neq 0 \\ 0 & \text{otherwise,} \end{cases}$$

where $N(s,\perp)$ is the number of times that s has executed, and $N(s,\ \langle p,\ v\rangle)$ denotes the number of times the points-to relationship $\langle p,\ v\rangle$ holds at s [18].

The probability function can be overloaded to compute the probabilities for the set of points-to relationships, if the set is represented by a vector. Specifically, if V is the set of points-to relationships at s, the probability function for V at s will be

$$\mathcal{P}(s,\ V) \stackrel{\text{def}}{=} \{\mathcal{P}(s,\ \langle p,\ v\rangle) \mid \langle p,\ v\rangle \in V\}.$$

Such an overloaded probability function returns a vector, the ith element of which contains the result of the probability function for the ith points-to relationship in V.

Because we are only concerned about pointers, assuming that p is the analysis target, the possible statements in a program are listed in Table 1. *Points-to Location* points to one or more locations, and the prefix, "*affine-*", means the array index or offset is composed of $f(x) = ax + b$. *Pointers Aliased* is an alias between pointers with an affine offset or it aliases with a function pointer. If multilevel pointers are considered, one-level-higher pointer points to one-level-lower point is a kind of *Pointers Aliased*, because the dereferencing

TABLE 1
Program Normalization in C Style

Points-to Location	$p = \&v$, $p = malloc()$, $p = \&A[affine\text{-}index]$, $p = A$
Pointers Aliased	$p = q + (affine\text{-}offset)$, $p = function\ pointer$, $r = \&p$
Updated by Function	$p = foo()$
Pointer Parameter	pointer-type foo(type $*p$, ...)
Indirect Store	$*p = ..., p \rightarrow ... = ...$
Indirect Load	$... = *q, ... = q \rightarrow ...$

p and q are one-level pointers, r is a two-level pointer, v is a scalar variable, and A is an array of scalars.

one-level-higher pointer is aliased with one-level-lower pointer (*r and p are aliased). *Updated by Function* indicates that the foo function returns a pointer to p, and *Pointer Parameter* means that pointer p in a callee function aliases with other pointers at caller sites, and the last two statements, *Indirect Store* and *Indirect Load* access the pointers indirectly, when dereferencing a pointer by * or $\rightarrow$. However, indirect accesses make the SSA form more complicated, because the SSA constructor cannot determine the define site only from the symbolic information. This problem is discussed in Section 2.2.

Location naming is defined as (base, offset). The *base* denotes the base address of a location, and *offset* is the distance from the base address in bytes. For example, the location naming of an integer-type array, $A[3]$, is represented as $(\&A[0], 12)$. The offset part can also be an affine function that represents a dynamic access. For example, $(\&A[0], (2i + 3) \times 4)$ is the location naming of $A[2i + 3]$. If the storage locations are created by functions, the base part of its naming is the function name with a line number in the source code. Take the following code segment, for example. Pointer p points to the location which is named $mallocS2$, and q points to the location, $mallocS3$.

```
foo(int *p, int *q) { //S1
  p = malloc(sizeof (int)); //S2
  q malloc(sizeof (int)); //S3
}
```

2.2 Static Single Assignment Form

In compiler design, the SSA form [19], [20] is an intermediate representation (IR) in which every variable is assigned exactly once. Existing variables in the original IR are split into multiple versions, and new variables are typically indicated by the original name with a subscript, so that each definition gets its own version. As a result, use-def chains are explicitly tracked in the SSA form. Many efficient optimizations have been developed based on SSA, such as dead code elimination [21], constant propagation [22], and live range computation [23].

In order to maintain single assignments at the confluence points in a control flow graph (CFG), SSA introduces the Φ-function, which is represented as a *pseudoassignment* and it takes the form $V_k = \Phi(V_m, V_n, \ldots, V_i)$. V_k denotes the new version of V, while the operands on the right-hand side (RHS) denote the old versions that are live until a confluence point. When constructing the SSA form, usually Φ-*placement* is performed first, followed by variable renaming. Φs must be inserted in the dominance frontiers of the nodes containing defitions in the CFG, and then variables are numbered according to their order in the dominator tree. Cytron et al. [24] proposed an algorithm and presented experimental results to show that the construction time of SSA usually increases linearly with the size of the original program.

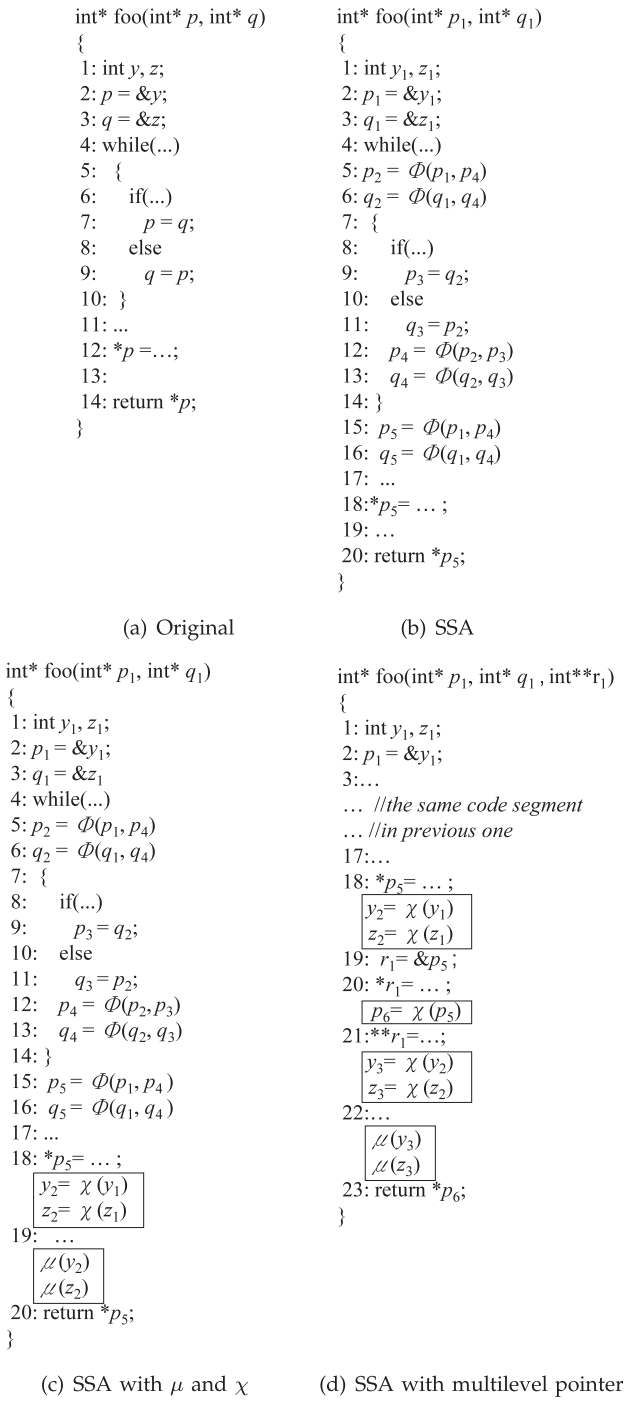


Fig. 1. Examples of SSA and of SSA with μ and χ.

Fig. 1 shows an example and its SSA form. This CFG has three dominance frontiers: the beginning and end of the *while loop*, and the end of the if-then-else structure. Therefore, variables with multiple definitions from incoming edges are placed in Φ at these three points. Let the pointer analyzer target the dereferencing pointer, p, at line 14 in the original

code; it is named as p_5 under the SSA representation at line 20. The explicit definition site of p_5 can be found directly at line 15. Φ is used to determine the related pointers, p_1 and p_4, from its operands. Later, p_1 points to the location of y_1, while p_4 is another Φ at line 12. Because p_1 has already found the *Points-to Location* that is the address of y_1, it is terminated. On the loop back edge, p_4 is defined based on its Φ operands, p_2 and p_3. Similarly, following the built-in use-def chains in SSA, p_3 aliases with q_2 at line 9. Another branch is a Φ at line 5. When a series of traces are applied following the use-def chains, the analyzer will eventually find p_5 pointing to the locations of y_1 and z_1. The above simple example shows the benefit of the explicit use-def chain. The detail of the algorithm is given in Section 3.

2.3 Memory SSA

Maintaining the explicit use-def chain is not straightforward when indirect memory operations are considered. Because a definition is made by an aliasing pointer of a scalar variable, the SSA constructor cannot exactly determine the single definition site from symbolic checking only. This paper solves this problem based on the method proposed by Chow et al. [25], which is an efficient method to model indirect accesses in SSA. The main concept is to introduce two annotations, *MayUse*:μ and *MayDef*:χ, which describe the behavior of indirect access operators. For example, if a pointer p may alias with a scalar, a. "$a_{i+1} = \chi(a_i)$" is associated after an indirect store statement such as $*p = \cdots$ This means that the indirect store may define a_i as a new version, a_{i+1}, in the SSA form. While, "$\mu(a_i)$" is associated before an indirect load statement such as $\ldots = *p$, meaning that indirect load statement may use the aliased variable, a_i. The versions of the variables in μ and χ are renamed simultaneously in the variable numbering phase of SSA construction. In order to maintain the explicit use-def chains of the example shown in *SSA with μ and χ* of Fig. 1, "$y_2 = \chi(y_1)$" and "$z_2 = \chi(z_1)$" must be associated after line 18; "$\mu(y_2)$" and "$\mu(z_2)$" must be associated before line 20. Because p aliases with u and v, u and v may be defined and used at these two places. This model treats scalar variables and pointers identically using three annotations: Φ, μ, and χ, where Φ records the related variables through the incoming edges of the CFG, χ records the related variables that are assigned by indirect memory operations, and μ represents the liveness of the variables. If there are multilevel pointers, SSA construction is applied multiple times and the candidates are computed from scalar variables to the pointers with more levels. χ in this model can protect the one-level-lower pointers or scalar variables from being potentially defined by the pointers that are one level higher. Assuming that the definition site of y_2 is queried at line 19, then the only definition site that can be found is the χ annotation at line 18, "$y_2 = \chi(y_1)$." This χ annotation guides the analyzer not only to track the operand, y_1, in the χ, but also the statement, $*p_5 = \cdots$, that is associated with this χ at line 18. The example, *SSA with multilevel pointer*, in Fig. 1 shows that the program contains a two-level pointer, r_1. Because r_1 points to p_5 at line 19, the χ annotation, "$p_6 = \chi(p_5)$," is associated with the statement, $*r_1 = \cdots$, at line 20. When the analysis involves p_6, this annotation can guide the analyzer to analyze not only the operand, p_5, but also the associated statement, $*r_1 = \cdots$. Similarly, the statement, $**r_1 = \cdots$, at line 21 can potentially define y_2 and z_2, so "$y_3 = \chi(y_2)$" and "$z_3 = \chi(z_2)$" are associated with this statement. Because SSA is constructed in the order of low level to high level, the constructor can know what kinds of χ annotations must be associated with the indirect defining statement at Line 21. Multilevel pointer is usually considered for precision [26], while the limitation of single level pointer is usually made for speed [27].

TABLE 2
Notation of the PPA Algorithm

Term	Description
CFG	Control flow graph of a program
SSA	Static single assign form of a program
G	PRG:(V, E)
u_o	Analysis candidate
G'	Reduced PRG:(V', E')
V_p	Eliminated nodes
$w(E)$	Probabilities of aliasing or points-to relationships
μ, χ	Annotation for *MayUse* and *MayDef*

3 Probabilistic Pointer Analysis

3.1 Main Algorithm

Before the algorithm is introduced, the notation used in the algorithm is presented in Table 2. The main algorithm of PPA in SSA is listed in Algorithm 1. The analysis candidate pointer, u_0, is passed to the *Backtrace* function. Meanwhile, the analysis program is based on CFG and SSA, and they are the global data which can be accessed and modified by *Backtrace* and *ReduceGraph* functions. In this function u_0 propagates through the related pointers until all of the related pointers and locations are found, and a PRG is produced by continuously updating the global PRG: G stored in *PPA*. The PRG is a directed weighted cyclic graph: $G = (V, E)$. The root node of the PRG is an analysis target, u_0, the leaf nodes are the points-to locations, and the other nodes in $V(G)$ are the related pointers. $E(G)$ connects the related pointers or locations between $V(G)$. The weights of $E(G)$ correspond to the probabilities of aliasing and points-to relationships. However, since the PRG is not a compact representation of pointer information, the *ReduceGraph* function returns a directed weighted cyclic graph with a compact shape: $G' = (V', E')$, where G' is reduced from G by merging the nodes, $V_p = V(G) - \{u_i \mid outdegree(u_i) = 0 \vee indegree(u_i) = 0\}$, and eliminating the corresponding edges that connect the merged nodes. It means all of the nodes in G except an analysis target and points-to locations are eliminated. Finally, $G' \subseteq G$. Detail of the *Backtrace* and *ReduceGraph* functions is presented in Sections 3.2 and 3.3, respectively.

3.2 *Backtrace* Algorithm

The *Backtrace* function is listed in Algorithm 2, and it calls the *BacktraceChi*, *BacktracePhi*, *BacktraceCallee*, *BacktraceCaller*, and *BacktraceMu* functions listed in Algorithms 3, 4, 5, 6, and 7, respectively. Because of the benefit of the SSA form with Φ, μ, and χ, not only can explicit definition sites be

found in linear time from χ and Φ, but indirect uses also can be obtained from μ. The analysis complexity is lower when using the use-def chains in SSA than in an iterative data flow analysis. SSA-based points-to analysis only updates pointer information at Φ, χ, and definition sites. However an iterative data-flow analysis always updates pointer information at every point of a program. Take the program in Fig. 1, for example, the data-flow analyzer spends three iterations for every statement in order to make the pointer information of p at line 12 converge. During the phase of collecting the pointer information of p, there are also four symbolic probabilities assumed, and then the data-flow analyzer needs to solve the polynomial equation to obtain the values of these four symbolic probabilities. As a result, PPA information is resolved [6].

Algorithm 1: PPA(u_0, CFG, SSA)

Input: An analysis candidate: u_0. A CFG: CFG of an analysis program. A SSA form: SSA of an analysis program.
Result: A compact directed weighted acyclic graph: $G' = (V', E')$.
Global Data: A CFG: CFG, a SSA form: SSA, and a PRG: $G = (V, E)$. The statements of the $CFG : S$.

```
1 begin
2   G= ∅ ;
3   V = V ∪ {u0};
4   Backtrace(u0); // Updates G
5   return ReduceGraph(G) ;
6 end
```

The main concept of the *Backtrace* function is maintaining pointer relationships by updating a global PRG: G when encountering χ, Φ, and the statements in definition sites. During back-tracing, χ and Φ lead to points-to relationships with probabilities lower than 100 percent since these two annotations contain multiple definition sites. The first situation is encountering a χ associated with a statement. This means that the indirect store statement may update the pointer on the RHS of the χ annotation, so not only this indirect store statement but also the pointer on the RHS of the χ annotation needs to be considered. The *BacktraceChi* function is used to address the situation where the probability of the dereferencing pointer in the indirect store statement is calculated first; this is called $mayDefProbability$, which is assigned to the weight of the corresponding edge in the PRG. Meanwhile, the probability of aliasing to the variable on the RHS of χ is $1.0 - mayDefProbability$, because it is not defined by the indirect store statement. Irrespective of whether or not the potential defining statement will take effect, two parts of semantics should be maintained by propagating *Backtrace* functions, so u_{md} and u_χ are passed to the *Backtrace* function. The second situation is encountering a Φ-function, which means the definition site may come from multiple different versions. The different execution frequencies mean that the definition probabilities vary between the Φ operands. The *BacktracePhi* function is called to analyze all possible definition sites, and the weights of edges are the execution frequencies in the CFG.

Algorithm 2: Backtrace(u)

Input: u is a traced pointer.
Result: The PRG: $G = (V, E)$ is generated by updating the global G itself, and the location naming is defined as $(base, offset)$.
Used Data: The global data in $PPA()$.
Local Data: base, offset, thisOffset.

```
1  begin
2    if u is visited then return;
3    Set u visited;
4    Follow the use-def chain of u in SSA, and find the
     only one def-site of u: s ∈ S;
5    if type of (s) = χ annotation then
     BacktraceChi(u, s);
6    else if type of (s) = Φ function then
     BacktracePhi(u);
7    else if type of (s) = Real statement then
8      switch (s) do
9        case Points-to Location          // p = &v, ..
10         E=E ∪ {(u, u_rb)} | u_rb ∈ base address at
           RHS of s;
11         w(u, u_rb)= 1.0 ;
12         V = V ∪ {u_rb};
13         base = u_rb;
14         offset = offset + thisOffset | thisOffset =
           offset at RHS ;
15         break;
16       case Pointers Aliased
         // p = q + (affine-offset), ..
17         E=E ∪ {(u, u_rb)} | u_rb ∈ aliased pointer at
           RHS of s;
18         w(u, u_rb)= 1.0;
19         V = V ∪ {u_rb};
20         offset = offset + thisOffset | thisOffset =
           offset at RHS ;
21         Backtrace(u_rb);
22         break;
23       case Updated by Function    // p = foo(), ..
24         BacktraceCallee(u, s) ;
25         break;
26       case Pointer Parameter  // foo(type * p, ..), ..
27         BacktraceCaller(u, s);
28         break;
29       case Indirect Load              // p = *q, ..
30         BacktraceMu(u, s) ;
31   return;
32 end
```

Besides χ and Φ, the *Backtrace* function also deals with pointer update statements. This function uses different strategies for each kind of assignment listed in Table 1. The first case, *Points-to Location*, listed at line 9, is already pointing to a location, and therefore the analysis is terminated for this branch of back-tracing after the $base$ and $offset$ of a location naming are stored. The following case, *Pointer aliased*, listed at line 16, is simply aliasing to another pointer; therefore, more back-traces should be performed until locations are found. The offset part needs to be maintained before propagating Backtrace functions. Then, *Updated by Function* and *Pointer Parameter* cases are dealt with by the *BacktraceCallee* and *BacktraceCaller* functions listed at line 23 and

26, respectively. These two algorithms describe how to deal with the pointers aliased across procedures with the aid of a call graph, and this extends the analysis to an interprocedural one. When the pointer is *Updated by Function*, the return statement of the callee is traced in order to identify the related pointer. Meanwhile, the pointer is passed as *Pointer Parameter*, the explicit definition sites as parameters of callers are found, and the analysis pointer is transformed as the pointers at the caller site. In the final case, *Indirect Load*, the pointer is updated by an indirect load at line 29, so all possible may-use variables in μ annotations are traced. In *Backtrace*, the only situation in Table 1 not encountered is *Indirect Store*, because it is considered according to the attached χ annotation rather than the statement itself.

Algorithm 3: BacktraceChi(u, s)

Input: u is a traced pointer, and $s \in S$.
Result: Update the PRG: $G = (V, E)$ in $Backtrace\ function$.
Used Data: The global data in $PPA()$.
Local Data: mayDefProbability.

```
1  begin
2      mayDefProbability = w(u_is, u) of PPA(u_is) | u_is
       is the indirect store pointer in s ;
3      u_md = the dereference pointer of u_is;
4      E=E ∪ {(u, u_md)};
5      w(u, u_md)=mayDefProbability;
6      E=E ∪ {(u, u_χ)} | u_χ ∈ operand in χ ;
7      w(u, u_χ)= 1.0 - mayDefProbability;
8      V = V ∪ {u_md} ;
9      V = V ∪ {u_χ} ;
10     Backtrace(u_md) ;
11     Backtrace(u_χ) ;
12 end
```

Algorithm 4: BacktracePhi(u)

Input: u is a traced pointer.
Result: Update the PRG: $G = (V, E)$ in $Backtrace\ function$.
Used Data: The global data in $PPA()$.

```
1 begin
2     foreach u_φ ∈ operands in Φ do
3         E=E ∪ {(u, u_φ)} ;
4         w(u, u_φ) = frequency from u_φ to u in CFG ;
5         V = V ∪ {u_φ} ;
6         Backtrace(u_φ) ;
7 end
```

3.2.1 Example of Backtrace Algorithm

The related data structures of PPA are shown in Fig. 2 for an analysis target, p_5. Fig. 2a is the SSA representation of the source code shown in Fig. 1a; meanwhile, the numbers labeled on the nodes denote the IDs of basic blocks (BBs), and the numbers on the dotted lines are the back-trace orders. For sparse occurrences of related pointers, the built-in use-def chains in SSA make the back-trace step effective. PPA does not need to visit all nodes of the CFG, and the back-trace step follows the dotted lines. The execution frequencies are statically assigned or given by profiling, and are assigned to each edge of the CFG, and hence each back-trace step can retrieve the edge frequency. A PRG is produced after all related nodes are visited by the *Backtrace* function, as shown in Fig. 2b. Each node in the PRG is assigned a unique number by globally numbering each version of a variable in SSA. The weights of the edges in the PRG indicate the probabilities of aliasing or points-to relationships between connected nodes. However, the PPA information is difficult to determine from the PRG, so the PRG must be reduced as shown in Fig. 2c, which shows that the probabilities of p_5 pointing to the addresses of y_1 and z_1 are 75 and 25 percent, respectively.

Algorithm 5: BacktraceCallee(u, s)

Input: u is a traced pointer, and $s \in S$.
Result: Update the PRG: $G = (V, E)$ in $Backtrace\ function$.
Used Data: The global data in $PPA()$.

```
1 begin
2     u_rv ∈ return value of callee in s ;
3     E = E ∪ {(u, u_rv)} ;
4     w(u, u_rv) = 1.0;
5     V = V ∪ {u_rv} ;
6     Backtrace(u_rv) ;
7 end
```

Algorithm 6: BacktraceCaller(u, s)

Input: u is a traced pointer, and $s \in S$.
Result: Update the PRG: $G = (V, E)$ in $Backtrace\ function$.
Used Data: The global data in $PPA()$.

```
1 begin
2     U_cp ∈ corresponding parameters of caller in s ;
3     foreach u_cp ∈ U_cp do
4         E=E ∪ {(u, u_cp)} ;
5         w(u, u_cp) = calling frequency ;
6         V = V ∪ {u_cp} ;
7         Backtrace(u_cp);
8 end
```

Algorithm 7: BacktraceMu(u, s)

Input: u is a traced pointer, and $s \in S$.
Result: Update the PRG: $G = (V, E)$ in $Backtrace\ function$.
Used Data: The global data in $PPA()$.

```
1 begin
2     foreach u_μ ∈ operands in μ is associated with the
      statement, s do
3         E=E ∪ {(u, u_μ)};
4         w(u, u_μ) = frequency from u_μ to u in CFG;
5         V = V ∪ {u_μ};
6         Backtrace(u_μ);
7 end
```

In this example, we assume that the probabilities of a branch-taken and a branch-not-taken are both 50 percent, and that the probability of leaving a loop is 10 percent. For p_5 as the analysis target, the detail of generating a PRG is as follows: p_5 is first passed to the *Backtrace* function. Because the definition site type of p_5 is a Φ at BB 7 in Fig. 2a, all the operands, p_1 and p_4, in the Φ are propagated to the *Backtrace*

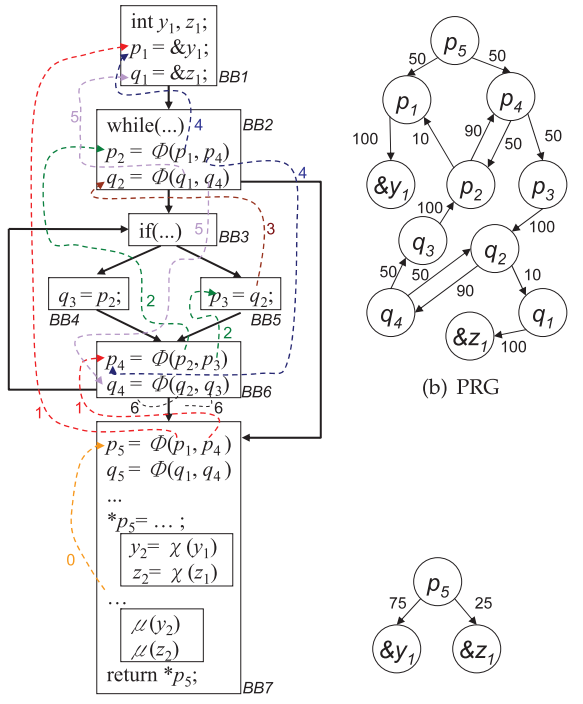


Fig. 2. Example: from SSA to PPA information.

function. Meanwhile, in Fig. 2b, node p_5 connecting $p1$ and $p4$ is maintained. According to the CFG, p_1 and p_4 at BB 1 and BB 6 are diverged by an *if while* condition, so the *weights* of (p_5, p_1) and (p_5, p_4) are statically assigned 50 percent both in the PRG. Next, the definition site type of p_1 is *Points-to Location* at BB 1, so the analyzer terminates this branch and p_1 points to the address of y_1 with 100 percent. On the other hand, the definition site of p_4 is a Φ at BB 6. Similarly, all of the operands, p_2 and p_3, are passed to the *Backtrace* function. According to the CFG, these two nodes are diverged by an *if*; therefore the *weights* of (p_4, p_2) and (p_4, p_3) are statically assigned 50 percent both in the PRG. p_2 is defined by a Φ at BB 2, while p_3 encounters *Pointers Aliased* at BB 5. Because the definition site statement of p_3 indicates that p_3 aliases q_2, q_2 is passed to the *Backtrace* function. Until now, the live branches are p_2 and q_2. For p_2, both of the operands, p_1 and p_4, have been visited, so no propagating back-traces are needed. According to the CFG, these two nodes are diverged by a *while loop*; therefore the *weights* of (p_2, p_1) and (p_2, p_4) are statically assigned 10 and 90 percent, respectively, in the PRG. For q_2, the type of the definition site is a Φ whose operands are q_1 and q_4, and none of its operands has been visited. For q_1, it encounters *Points-to Locations* at BB 1. q_1 points to the location of z_1 with 100 percent, and its trace branch is terminated. For q_4, the definition site is back to BB 6, and this involves q_2 and q_3. Furthermore, q_2 has been visited while q_3 is aliased with p_2. Finally, p_2 has been visited and there are no more live back-trace branches. Therefore, the *Backtrace* function successfully generates a PRG as shown in Fig. 2b.

3.3 ReduceGraph Algorithm

After the related pointer information is collected using the *Backtrace* algorithm, a graph reduction method is adopted to summarize the PPA information. The method for reducing the graph is listed in Algorithm 8. The goal of the *ReduceGraph* function is to make the points-to information more compact by producing a reduced graph. Only an analysis node, u_0, and the nodes which indicate points-to locations will remain, because all nodes in V_p are eliminated.

Algorithm 8: ReduceGraph(G)

Input: A PRG: $G = (V, E)$ is produced by *Backtrace*.
Result: A directed weighted cyclic graph: $G' = (V', E')$.
Local Data: V_p=$V(G) - \{u_i \mid outdegree(u_i) = 0 \vee indegree(u_i) = 0\}$.

```
1  begin
2    while V_p ≠ ∅ do
3      if (u_y, u_y) ∈ E(G) then
4        foreach u_z is adjacent to u_y do
           w(u_y, u_z) = w(u_y, u_z) / Σ_{V_z} w(u_y, V_z);
5
6          E = E − {(u_y, u_y)};
7      else
8        for u_i ∈ V_p is adjacent to u_0 do
9          foreach u_k | u_i is adjacent to u_k, u_j is
           adjacent to u_i, u_k ≠ u_0 do
10           E = E ∪ {(u_k, u_j)};
11           w(u_k, u_j) = w(u_k, u_i) * w(u_i, u_j);
12           E = E − {(u_k, u_i)};
13         foreach u_j do
14           E = E ∪ {(u_0, u_j)};
15           w(u_0, u_j) = w(u_0, u_i) * w(u_i, u_j);
16           E = E − {(u_0, u_i)};
17           E = E − {(u_i, u_j)};
18         V_p = V_p − {u_i} ;
19   return G' = G
20 end
```

First, the reducer checks if there are any self-cycle edges in PRG, which are caused by Φ-functions in loop structures of SSA. If there is a cycle, line 5 in the *ReduceGraph* algorithm adjusts the self-cycle weight, $w(u_y, u_y)$, proportionally. The derivation of proportioning $w(u_y, u_y)$ is in (1). The exponent part in (1) indicates that the program exits this loop at a given iteration. Assume that irrespective of how many iterations are executed, the program will eventually exit the loop, where $n \to \infty$. Moreover, because

$$\sum_{V=V_z \cup u_y} w(u_y, V) = 1 \wedge w(u_y, u_y) < 1,$$

under this assumption, the result

$$\frac{w(u_y, u_z)}{\sum_{V_z} w(u_y, V_z)}$$

is derived. An example of the reduction process is shown in Fig. 3a. The $w(u_y, u_y)$ values are distributed into the edges that are connected to the other nodes. When the proportioning equation is applied, the PRG is simplified as a root

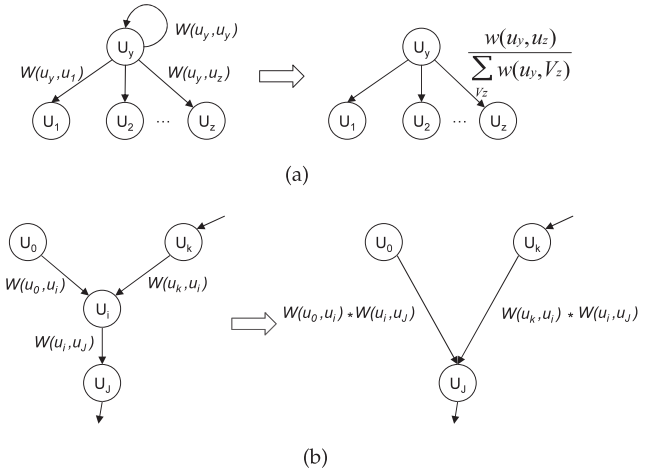


Fig. 3. A visual example of the *ReduceGraph* function.

connecting a sequence of aliased nodes, some of which point to location nodes.

If there are no self-cycles in G, lines 9 to 18 of Algorithm 8 indicate how to eliminate nodes and edges, while also adding new edges and calculating their own $weights$. An example of this process is shown in Fig. 3b, where u_i is eliminated. Before u_i is eliminated, (u_k, u_j) and (u_0, u_j) are added to E. The $weights$ of these edges must also be calculated. The $weight$ of (u_k, u_j) is obtained by multiplying the two respective $weights$ of the edges that connect (u_k, u_i) and (u_i, u_j). $w(u_k, u_j)$ means the probability that u_k aliases u_i and also that u_i aliases or points to u_j. u_i and the edges associated with u_i can be eliminated after the $weights$ of new edges have been updated.

$$\begin{aligned} w(u_y, u_z) &= w(u_y, u_z) + w(u_y, u_z)w(u_y, u_y) \\ &\quad + \cdots w(u_y, u_z) + w(u_y, u_y)^{n-1} \\ &= \frac{w(u_y, u_z)(1 - w(u_y, u_y)^n)}{1 - w(u_y, u_y)} \\ &\quad \|n \to \infty \wedge w(u_y, u_y) < 1\| \\ &= \frac{w(u_y, u_z)}{1 - w(u_y, u_y)} \\ &= \frac{w(u_y, u_z)}{\sum_{V_z} w(u_y, V_z)}. \end{aligned} \tag{1}$$

The PRG can always be reduced, because when there are no self-cycles, the *ReduceGraph* algorithm eliminates a node, $u_i \in V_p$, in each iteration until there no more such nodes remain in the PRG. Also no new nodes are created during the reduction process. u_i in the algorithm can be eliminated except when it exists as a self-cycle, so self-cycles in a PRG are checked first. A node with a self-cycle means it contains loop information, and therefore (1) is proposed for eliminating a node without losing loop information.

3.3.1 Example of ReduceGraph Algorithm

The reduction processes for the source code in Fig. 2a are shown in Fig. 4. The $weights$ indicate the probabilities in percentage while the nodes are named by unique variables in the SSA form. First, Fig. 4a is generated by the *Backtrace* algorithm. The processing between two subfigures is

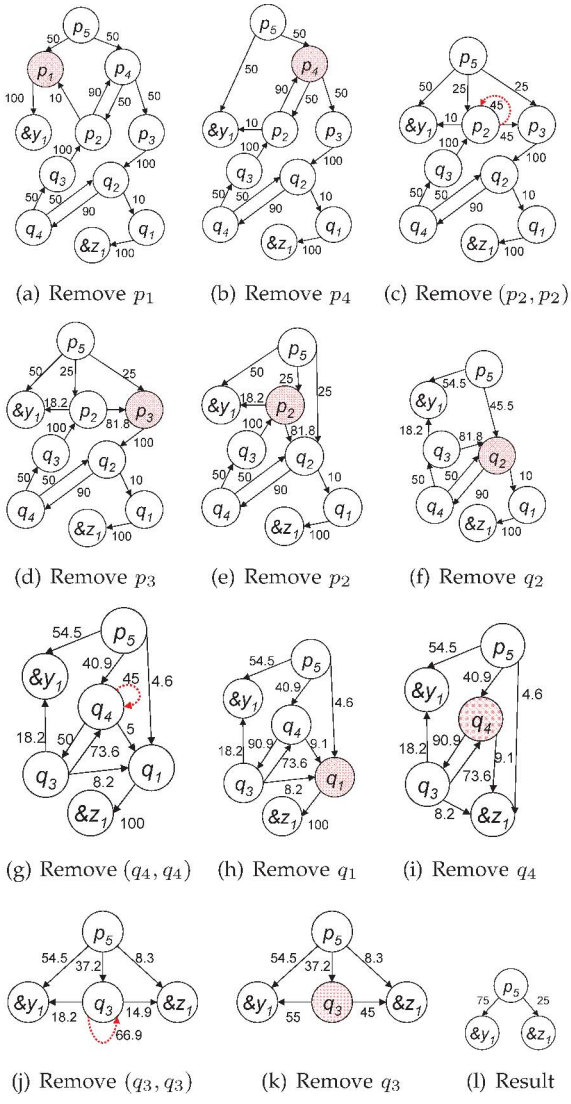


Fig. 4. Example of the processes involved in reducing a PRG.

highlighted. For example, Figs. 4a and 4b remove node p_1 and the edges associated with it, because there are no self-cycle edges in the PRG. Furthermore, edges (p_2, &y_1) and (p_5, &y_1) are created, and their weights are maintained. The detail of this process is implemented at lines 9 to 18 of Algorithm 8. The processing between Figs. 4c and 4d eliminates a self-cycle edge (p_2, p_2), and the probability, 45 percent, is divided into the edges of two adjacent nodes, nodes &y_1 and p_3, as shown from lines 4 to 6 of Algorithm 8. The reduced PRG obtained after a series of node reductions and self-cycle edge eliminations is shown in Fig. 4l. This reduced graph shows that a pointer variable named p_5 in the SSA form points to two locations, &y_1 and &z_1, with the probabilities of 75 and 25 percent, respectively.

3.4 Complexity

3.4.1 Backtrace Algorithm

Back-trace steps are stopped only when the visited node has been traced or is determined a location, so there are no duplicate nodes in a PRG produced by the *Backtrace* algorithm. This implies that there are fewer back-trace steps than the number of variable IDs in an SSA form. With

the aid of SSA representation, each back-trace step can be completed in $O(1)$. Assuming that N is the number of IDs in SSA, the complexity of the *Backtrace* algorithm is $O(N)$.

3.4.2 ReduceGraph Algorithm

In the *ReduceGraph* algorithm, a node u_i can be eliminated at each iteration until an empty V_p, and there are no duplicate nodes in the obtained PRG. Assuming that V is the number of nodes V_p in a PRG, then the number of eliminating iterations is V. Assuming that E is the number of edges in a PRG and the complexity of each iteration is $O(E)$. Consequently, the complexity of the *ReduceGraph* algorithm is $O(VE)$.

3.5 Interprocedural Analysis

Interprocedural analysis [28] was performed based on the Formal Bound Sets (FBS) problem [29]. The FBS problem gives a set of pairs (A, B) for each function Q, where A is a formal parameter of function P that calls another function Q directly or indirectly, and B is a formal parameter of Q such that B is bound to A along some call chain starting at P. First, a call graph is built in which each function is uniquely represented by a node and each call site by an edge. An initial flow function is applied to each edge in call graph, and later, ε, is applied to handle more general flow functions. Two rules $E1$ and $E2$ are applied to the FBS to find the closure of recursive function calls, and correspond to function composition. Finally, the flow function can gather the parameter bundle between each pair of parameters.

Algorithms 5 and 6 were developed so that this PPA framework could implement interprocedural pointer analysis. The *BacktraceCallee* function gathers the related pointer information in the callee when the analysis pointer is updated by a function. The return value of the callee is considered, and the *Backtrace* function is propagated if the traced variable has not yet been a points-to location. Meanwhile, the *BacktraceCaller* function gathers the related pointer information in callers when the analysis pointer is at the callee site. All corresponding parameters of callers are involved in this situation, and the weights of connected edges depend on the calling frequencies. Similarly, if the corresponding parameter is not a location, the *Backtrace* function is propagated. The χ annotation also plays an important role when performing interprocedural analysis, because χ records the may-define information for pointers and keeps track of side-effect pointer information of functions.

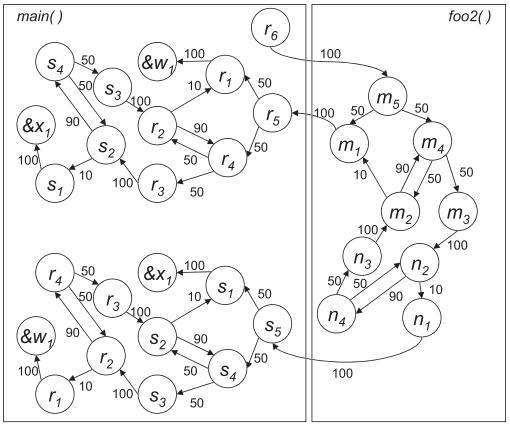
3.6 Interprocedural Example

An example of interprocedural PPA is shown in Fig. 5. The $main$ function in Fig. 5a is similar to the foo function in the example shown in Fig. 1c, but line 18 is replaced by a new statement, $r_6 = foo2(r_5, s_5)$. Also, the $foo2$ function in Fig. 5a is similar to the foo function in Fig. 1c, the main difference is that lines 2 and 3 of foo are removed. These minor modifications do not affect the shape of the PRG, so the produced PRG in the previous example can be reused to demonstrate interprocedural PPA.

If the analysis target is r_6 in the $main$ function, the interprocedural PRG is constructed by the *Backtrace* algorithm listed in Algorithm 2. Because "$r_6 = foo2(r_5, s_5)$" is inserted at line 18, r_6 in the $main$ function aliases with m_5 in the foo function, and nodes r_6 and m_5 have a 100 percent

```
int main(void)                    int* foo2(int* m1, int* n1)
{                                 {
 1: int w1, x1,*r1, *s1;          1: while(...)
 2: r1 = &w1;                     2: m2 = Φ(m1, m4)
 3: s1 = &x1;                     3: n2 = Φ(n1, n4)
 4: while(...)                    4: {
 5: r2 = Φ(r1, r4)                5:    if(...)
 6: s2 = Φ(s1, s4)                6:       m3 = n2;
 7: {                             7:    else
 8:    if(...)                    8:       n3 = m2;
 9:       r3 = s2;                9:    m4 = Φ(m2, m3)
10:    else                      10:    n4 = Φ(n2, n3)
11:       s3 = r2;               11: }
12: r4 = Φ(r2, r3)               12: m5 = Φ(m1, m4)
13: s4 = Φ(s2, s3)               13: n5 = Φ(n1, n4)
14: }                            14: ...
15: r5 = Φ(r1, r4)               15: return m5;
16: s5 = Φ(s1, s4)               }
17: ...
18: r6 = foo2(r5, s5);
19:   …
      μ(w2)
      μ(x2)
20: return *r6;
}
```

(a) SSA with μ and χ



(b) PRG generated from interprocedural *Backtrace*

Fig. 5. Example of interprocedural PPA, transforming the source into a PRG.

relationship according to the *BacktraceCallee* algorithm listed in Algorithm 5 shown in Fig. 5b. The shape of the PRG in the $foo2$ functions is similar to foo's shown in Fig. 2b, except that nodes m_1 and n_1 are aliased with r_5 and s_5, respectively, instead of pointing to &y_1 and &z_1. The relationship across procedures is maintained by the *BacktraceCaller* algorithm listed in Algorithm 6. Finally, the back-trace steps starting from r_5 and s_5 in the $main$ function are similar to Fig. 2, so the interprocedural PRG is constructed as shown in Fig. 5b.

After the *Backtrace* function constructs the PRG, the *ReduceGraph* function is called to reduce it. Because the shapes of the sub-PRGs in the $main$ and $foo2$ functions are similar to foo's, the detail of the reduction processes is the same as that in Fig. 4. The PRG shown in Fig. 5b can be transformed into Fig. 6a using the steps listed in Fig. 4. Algorithm 8 then eliminates in the order of nodes m_5, m_1, and n_1, producing a PRG with the shape as shown in Fig. 6b. Additional reduction steps are applied, and the

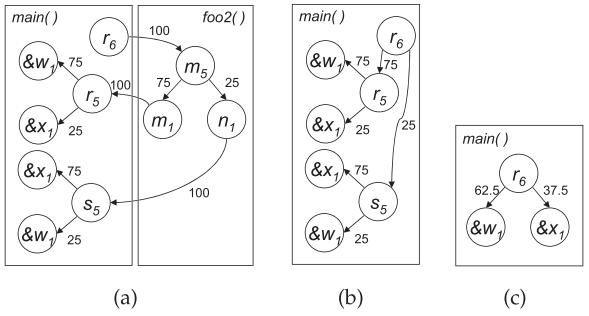


Fig. 6. Processes of reducing the interprocedural PRG.

interprocedural PPA information is produced as shown in Fig. 6c, which indicates that r_6 in the *main* function points to the addresses of w_1 and x_1 with the probabilities of 62.5 and 37.5 percent, respectively.

4 Experimental Results

4.1 SSA-Based PPA Framework

A block diagram of our PPA framework is presented in Fig. 7. The input programs are written in the C language. The programs are analyzed during the global optimization (WOPT) phase of Open64, which is a state-of-the-art compiler for Fortran, C, and C++, and its IR is named WHIRL. Different high- and low-level optimization modules interact with each other via WHIRL, and five levels of WHIRL are created. For example, the WOPT phase of Open64 deals with Mid-WHIRL, and it contains the essential information for the CFG and SSA with μ and χ. PPA is developed at the level of Mid-WHIRL for obtaining essential information. PPA is implemented using the *SSA* object in $COMP\_UNIT$ class and is called by the $Pre\_Optimizer$ function. The results are stored in $Opt\_main.cxx$. When PPA is implemented in the $COMP\_UNIT$ class, PPA can directly obtain the class members, the CFG and SSA. The edges of the CFG are assigned with one of two kinds of execution frequencies: one from static assignment and the other from edge profiling. Static assignment means that different types of edges are given different probabilities; for instance, outgoing edges without any branches are always assigned a probability of 100 percent, and branches that are taken or not have probabilities of 50 percent each, and the probability of leaving a loop is 10 percent. The execution frequencies from edge profiling are obtained by the instrumentation module of Open64 with the default inputs of benchmarks. After obtaining the basic information, WHIRL, SSA, and CFG are collected, and the algorithms proposed in Section 3 are implemented in order to obtain the PPA results, including *Backtrace* and *ReduceGraph* functions. The output of the *Backtrace* function is a PRG, which is the input to the *ReduceGraph* function. The output of the *ReduceGraph* function is a reduced PRG, which is the PPA results. PPA was implemented in the C front end of Open64-4.2.0 running on a machine with a 3-GHz, 64-bit x86 processor and 6 GB of RAM, and the OS kernel was Linux 2.6.29. Open64 analysis modules interact via the same IR, which allows PPA to possibly use the results of the other analysis modules that are done before PPA. Similarly, the results of PPA also can be fed back into WHIRL, and the compiler can pass PPA results to other analysis or optimization phases.

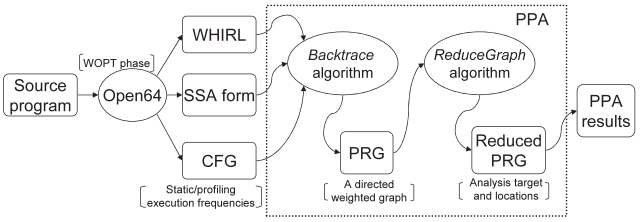


Fig. 7. SSA-based PPA framework.

4.2 PPA Accuracy and Scalability

With the environment described in Section 4.1, PPA provides two sets of experimental results: 1) the quantification of *possibly-points-to* relationships into probability distributions and a comparison of the precision during runtime, and 2) the analysis size and speed.

4.2.1 Accuracy

The accuracy evaluation was performed using pointer related benchmarks, Olden and CHJL. Olden is written in C and employs dynamically allocated data structures. The structures are usually organized into lists or trees. Olden is a popular benchmark for these types of studies [30], [31]. CHJL includes kernel routines from several benchmarks such as GCC, McGill, Netlib, and SPEC92. Because CHJL was used to evaluate data-flow PPA [6] before, it is chosen in this paper to do a direct comparison.

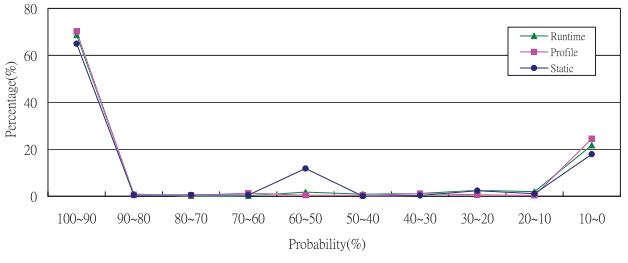
Table 3 lists program sizes, analysis targets, and PRG sizes of the programs in the CHJL benchmark. The analysis targets are pointer dereferences, and dereferencing a pointer means that the implicit data being pointed to are used in a load or store operation. Analyzing the points-to relationships of dereferenced pointers can help compiler to do aggressive optimizations, which utilize the knowledge of the probabilities of the data being used. This is why dereferenced pointers were assigned to be the analysis targets in this experiment. The results show that the mean number of dereferenced pointers was 73.9 of the benchmark with an average of 659 lines of code. The graph size is the number of edges in the PRG, which is produced by back-tracing SSA. Due to the sparsity of SSA, the mean size of the PRG was only 6.8. Finally, the execution time which is spent by static and profiling versions was a mean of 0.26 seconds for the benchmark with an average of 659 lines of code. The estimated time includes constructing and reducing all of the PRGs. This represents a significant speed improvement over previous methods [6], [7], [11]. The analysis time was proportional to the number of targets and the size of the PRG instead of the program size. However, there were slight disproportions between the benchmarks, which were caused by the reproduced self-cycles during the processes of reducing a PRG. For example, the *health* benchmark consumes more time because it produces self-cycles more often when nodes are eliminated. Moreover, *health* has the densest usage of pointers, because there are usually two or even three dereferences in a statement.

The average probabilities for all points-to relationships as estimated by the PPA in SSA form include the static and profiling distributions, and the actual pointer relationships at runtime are shown in Fig. 8a. For most of the benchmarks, the high or low probabilities account for most of the points-to relationships, and these points-to relationships with high or

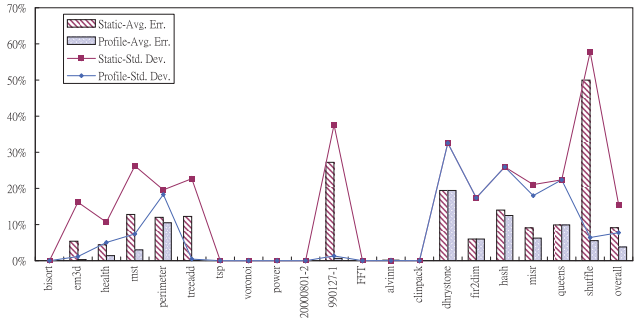
TABLE 3
Program Size, Execution Time, and Precision

Program	Program size (line)	Analysis targets	PRG size (edge)	Analysis time (second)	Average error (%)		Standard deviation (%)	
					Static	Profiling	Static	Profiling
bisort	632	174	17.33	0.53	0%	0%	0%	0%
em3d	906	99	4.14	0.05	5.39%	0.32%	16.17%	1.15%
health	728	675	10.08	3.59	4.4%	1.41%	10.65%	5.03%
mst	593	46	10.35	0.13	12.8%	3.01%	26.24%	7.41%
perimeter	478	109	25.54	0.26	12.01%	10.51%	19.6%	18.31%
treeadd	266	16	4.19	0.01	12.25%	0.25%	22.68%	0.46%
tsp	583	179	9.26	0.30	0%	0%	0%	0%
voronoi	1349	16	3.25	0.03	0%	0%	0%	0%
power	816	34	1.21	0.07	0%	0%	0%	0%
20000801-2	40	2	3.5	0.01	0%	0%	0%	0%
990127-1	31	8	3.38	0.01	27.24%	0.64%	37.60%	1.13%
FFT	962	12	4.25	0.04	0%	0%	0%	0%
alvinn	272	10	5.5	0.01	0.15%	0.15%	0.02%	0.02%
clinpack	1385	3	1.33	0.01	0%	0%	0%	0%
dhrystone	1895	9	2.56	0.02	19.44%	19.44%	32.57%	32.57%
fir2dim	151	16	9.9	0.01	5.99%	5.99%	17.40%	17.40%
hash	250	26	3.96	0.03	14.01%	12.52%	25.96%	24.99%
misr	276	26	8.42	0.03	9.09%	6.27%	21.08%	18.01%
queens	850	16	3.75	0.03	9.91%	9.91%	22.39%	22.39%
shuffle	718	2	4	0.01	50%	5.56%	57.74%	6.42%
Overall	659	73.9	6.80	0.26	9.13%	3.80%	15.05%	7.76%

low probabilities are the most important information for compiler optimizations. Knowledge of the extreme low and high values can guide a compiler to make better decisions regarding whether or not to apply aggressive optimizations. The largest differences between the static and profiling distributions are in the buckets of 0-10 percent and 50-60 percent. This is because the execution frequency of an edge in the CFG is assigned only according to the edge type in the static version, and for an *if* structure it is always given an equal probability, 50 percent, under each edge. However, for the profiling version, the execution frequency of an edge is according to edge profiling. Thus, in the static version there are no edges with exactly 0 percent points-to relationships in this experiment, and it obtains more cases in 50-60 percent.



(a) Distributions of points-to relationships



(b) Precision of points-to relationships

Fig. 8. Distribution and precision of points-to relationships.

Table 3 and Fig. 8b indicate that PPA with edge profiling is more accurate than PPA with static assignments of edge frequencies, when comparing with runtime results. The average error and standard deviation were calculated as follows:

$$Avg.Error = \frac{\sum_{i=1}^{n} |P_{estimated}(i) - P_{runtime}(i)|}{n},$$

and

$$Std.\ Deviation = \sqrt{\frac{\sum_{i=1}^{n} (P_{estimated}(i) - P_{runtime}(i))^2}{n}},$$

where $P_{estimated}(i)$ and $P_{runtime}(i)$ are the estimated and runtime profiled probabilities of the ith points-to relationship. The static and profiling average errors were 9.13 and 3.80 percent, and the standard deviations were 15.05 and 7.76 percent, respectively.

All of the dereferenced pointers in *bisort*, *tsp*, *voronoi*, *power*, *20000801-2*, *FFT*, *and clinpack* were from a single corresponding definition site, so their distributions had a probability of 100 percent.

4.2.2 Scalability

Since the PPA in SSA form has an advantage over the data-flow PPA on speed, we choose SPEC CPU2006 for the scalability evaluation. Because the considered statements listed in Table 1 are in a C style, Table 4 shows that PPA analyzes the integer benchmarks which are written in the C language in SPEC CPU2006. The average evaluated program size is 98,696 lines, the average number of analysis targets is 2,378.4, the average size of the PRG is 12.67, and the analysis time is 35.59 seconds on average. The result shows that the

TABLE 4
The Scalability: Program Size, Analysis Size, and Execution Time

Program	Program size (line)	Analysis targets	PRG size (edge)	Analysis time (second)
400.perlbench	155432	9428	31.14	89.45
401.bzip2	8293	153	2.19	0.3
403.gcc	517483	10334	13.98	291.42
429.mcf	2685	101	31.46	0.12
445.gobmk	197215	625	20.16	1.09
456.hmmer	35992	141	10.24	0.13
458.sjeng	13847	50	2.48	0.03
462.libquantum	4357	59	3.46	0.05
464.h264ref	51578	2891	10.64	2.14
999.specrand	74	2	1	0.01
Overall	98696	2378.4	12.67	35.59

analysis time is proportional to the number of analysis targets and the size of the PRG instead of program size. The number of analysis targets is dependent on the density of dereferenced pointers in a program, and the size of each PRG is limited by the factoring ability of SSA form, which lets the analyzer consider only the related information instead of whole program. The result also indicates that our scheme is sufficiently efficient for practical use. Compare *403.gcc* and *464.h264ref*, both of which have similar average PRG sizes: 13.98 and 10.64. Due to the large variation of the PRG size in *403.gcc*, the analysis time is 291.4 seconds (*464.h264ref* is 2.1 seconds). This is not proportional to the numbers of analysis targets: 10,334 and 2,891. PPA spent 93 percent of the analysis time to deal with top 10 large PRGs, because of a huge tree data structure which implements *Abstract Syntax Tree* (AST) and *Register Transfer Language* (RTL) in *403.gcc*. When an analysis target is involved in AST or RTL, the generated PRG is much bigger in *403.gcc*, and the average size of these ten PRGs was 1,412.21, which is much larger than the average size. However, PPA spends less analysis time on small PRGs, because of the variation of the PRG sizes in *464.h264ref*. Similarly, *400.perlbench* spends less analysis time, because the average size top 10 large PRGs is 692.32. The result was consistent with the graph reduction complexity, $O(VE)$, in Section 3.4.

5 Related Work

Previous papers on pointer analysis have proposed either aliases or points-to relationships [1], [2], [3], [4], [5], which categorized aliases or points-to information into *definitely-points-to* and *possibly-points-to* relationships instead of considering quantitative information, such as our proposed study. From the precision aspect, pointer analysis has been divided into intra- or interprocedural and flow-sensitive or -insensitive approaches. The flow-sensitive approach [1], [2], [3] considers the order of statements in a program, and the interprocedural approach [4], [5], [6], [7] considers the usage of pointers across functions. For more precise interprocedural information, pointer analysis also can be applied to the state-of-the-art interprocedural analysis algorithms [32], [33]. The present study applies flow-sensitive and interprocedural techniques simultaneously.

The previous work that is most closely related to the present study is that of Chen et al. [6], [7], [11]. They provided a set of transfer functions of data flow analysis for the usage of each pointer. When the data flow information was converged, polynomial equations were solved to obtain probabilistic information. However, the amount of information collected by data flow analysis usually increases exponentially, which makes solving polynomial equations inefficient. Furthermore, an optimization-directed analysis did not consider the behaviors of all of the pointers. This led to PPA in SSA due to its sparse representation being proposed to address these issues. PPA in SSA can speed up the analysis time, because of improving the order of complexity. The complexity of *Backtrace* and *ReduceGraph* in this paper is $O(N) + O(VE)$, respectively, which were discussed in Section 3.4 while the complexity of formulating and solving the equations in the data-flow PPA is $O(V_{df}N_{df}) + O(E_{df}^3)$ where V_{df} is the number of nodes in CFG, N_{df} is the maximum number of points-to relationships, and E_{df} is the number variables in the equations [6]. Comparing the complexity of PPA in SSA with data-flow PPA, because $O(VE)$ is equivalent to $O(V_{df}N_{df})$, and $O(N)$ has less complexity than $O(E_{df}^3)$, data-flow PPA is more complex than PPA in SSA. Da Silva and Steffan [16] proposed a method of pointer analysis based on sparse transformation matrices. In their algorithm the pointer information was modeled as a points-to matrix, and transformation matrices recorded the influences of pointers within a statement or a set of statements. The information was recorded in transformation matrices, and the points-to matrix could be adjusted between statements or BBs. Nevertheless, without the use-def chains of SSA, this method must collect information for all of the pointers even when information is only needed for one of them. Hardekopf and Lin [34] also applied SSA to pointer analysis; however, there was no probabilistic pointer information reported. In their algorithm, only the top-level pointers could get the benefit of SSA, and the others were subject to data flow analysis which had higher complexity.

6 Conclusion

This paper has presented an SSA-based probabilistic pointer analysis technique. The goal of this analysis technique is to compute the probabilities of points-to relationships for the pointers in the SSA form, and to reduce the analysis time of PPA. PPA algorithms were developed to accomplish the ability of quantifying *possibly-points-to* relationships, and the experimental results have shown the distributions and precisions of their occurrences. The analysis also uses explicit use-def chains in the SSA form, and the experimental results have shown that the analysis scheme was sufficiently efficient for practical use. Future studies should first investigate the applications of PPA and increase the precision of certain specific data structures, since precision should be considered under targeted optimizations. For example, the precision of array accesses through pointers is more important for cache prefetching, while speculative execution more concerns about the precision of pointers between loops. Second, it is now practical to investigate more optimization problems with PPA, since PPA information can be gathered quickly based on SSA. Finally, the precision of pointer analysis may vary between coarse- and fine-grained naming of memory locations. How to achieve precise pointer analysis with fine-grained memory naming is another issue for future studies.

REFERENCES

[1] M. Shapiro and S. Horwitz, "Fast and Accurate Flow-Insensitive Points-to Analysis," *Proc. 24th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '97),* pp. 1-14, 1997.

[2] B. Steensgaard, "Points-to Analysis in Almost Linear Time," *Proc. 23rd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '96),* pp. 32-41, 1996.

[3] J.-D. Choi, M. Burke, and P. Carini, "Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects," *Proc. 20th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '93),* pp. 232-245, 1993.

[4] A. Deutsch, "Interprocedural May-Alias Analysis for Pointers: Beyond K-Limiting," *Proc. ACM SIGPLAN 1994 Conf. Programming Language Design and Implementation (PLDI '94),* pp. 230-241, 1994.

[5] M. Emami, R. Ghiya, and L.J. Hendren, "Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '94),* pp. 242-256, 1994.

[6] P.-S. Chen, Y.-S. Hwang, R.D.-C. Ju, and J.K. Lee, "Interprocedural Probabilistic Pointer Analysis," *IEEE Trans. Parallel Distributed Systems,* vol. 15, no. 10, pp. 893-907, Oct. 2004.

[7] P.-S. Chen, M.-Y. Hung, Y.-S. Hwang, R.D.-C. Ju, and J.K. Lee, "Compiler Support for Speculative Multithreading Architecture with Probabilistic Points-to Analysis," *Proc. Ninth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '03),* pp. 25-36, 2003.

[8] M. Hind, "Pointer Analysis: Haven't We Solved This Problem Yet?" *Proc. ACM SIGPLAN-SIGSOFT Workshop Program Analysis for Software Tools and Eng. (PASTE '01),* pp. 54-61, 2001.

[9] M. Hind and A. Pioli, "Which Pointer Analysis Should I Use?" *Proc. ACM SIGSOFT Int'l Symp. Software Testing and Analysis (ISSTA '00),* pp. 113-123, 2000.

[10] M. Sagiv, T. Reps, and R. Wilhelm, "Solving Shape-Analysis Problems in Languages with Destructive Updating," *ACM Trans. Programming Language Systems,* vol. 20, no. 1, pp. 1-50, 1998.

[11] Y.-S. Hwang, P.-S. Chen, J.K. Lee, and R.D.-C. Ju, "Probabilistic Points-to Analysis," *Proc. Int'l Workshop Languages and Compilers for Parallel Computing (LCPC '01),* pp. 290-305, 2001.

[12] J.G. Steffan, C.B. Colohan, A. Zhai, and T.C. Mowry, "A Scalable Approach to Thread-Level Speculation," *Proc. 27th Ann. Int'l Symp. Computer Architecture,* pp. 1-12, 2000.

[13] S.P. Vanderwiel and D.J. Lilja, "Data Prefetch Mechanisms," *ACM Computing Surveys,* vol. 32, pp. 174-199, June 2000.

[14] M. Herlihy and J.E.B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," *Proc. 20th Ann. Int'l Symp. Computer Architecture (ISCA '93),* pp. 289-300, 1993.

[15] X. Dong, N.P. Jouppi, and Y. Xie, "Pcramsim: System-Level Performance, Energy, and Area Modeling for Phase-Change Ram," *Proc. Int'l Conf. Computer-Aided Design (ICCAD '09),* pp. 269-275, 2009.

[16] J. Da Silva and J.G. Steffan, "A Probabilistic Pointer Analysis for Speculative Optimizations," *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII),* pp. 416-425, 2006.

[17] J.L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Computer Architecture News,* vol. 34, pp. 1-17, Sept. 2006.

[18] G. Ramalingam, "Data Flow Frequency Analysis," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '96),* pp. 267-277, 1996.

[19] B. Alpern, M.N. Wegman, and F.K. Zadeck, "Detecting Equality of Variables in Programs," *Proc. 15th ACM SIGPLAN-SIGACT Symp. Principles of programming languages (POPL '88),* pp. 1-11, 1988.

[20] B.K. Rosen, M.N. Wegman, and F.K. Zadeck, "Global Value Numbers and Redundant Computations," *Proc. 15th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '88),* pp. 12-27, 1988.

[21] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Trans. Programming Language Systems,* vol. 13, no. 4, pp. 451-490, 1991.

[22] M.N. Wegman and F.K. Zadeck, "Constant Propagation with Conditional Branches," *ACM Trans. Programming Language Systems,* vol. 13, no. 2, pp. 181-210, 1991.

[23] M. Gerlek, M. Wolfe, and E. Stoltz, "A Reference Chain Approach for Live Variables," Technical Report CSE 94-029, Oregon Graduate Inst., 1994.

[24] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck, "An Efficient Method of Computing Static Single Assignment Form," *Proc. 16th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '89),* pp. 25-35, 1989.

[25] F.C. Chow, S. Chan, S.-M. Liu, R. Lo, and M. Streich, "Effective Representation of Aliases and Indirect Memory Operations in SSA Form," *Proc. Sixth Int'l Conf. Compiler Construction (CC '96),* pp. 253-267. 1996,

[26] Y. Cui, L. Li, and S. Yao, "Inclusion-Based Multi-Level Pointer Analysis," *Proc. Int'l Conf. Artificial Intelligence and Computational Intelligence (AICI '09),* vol. 2, pp. 204-208, 2009.

[27] M. Das, "Unification-Based Pointer Analysis with Directional Assignments," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '00),* pp. 35-46, 2000.

[28] V.C. Sreedhar, G.R. Gao, and Y.-F. Lee, "A New Framework for Elimination-Based Data Flow Analysis Using DJ Graphs," *ACM Trans. Programming Language Systems,* vol. 20, no. 2, pp. 388-435, 1998.

[29] M. Burke, "An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data-Flow Analysis," *ACM Trans. Programming Language Systems,* vol. 12, no. 3, pp. 341-395, 1990.

[30] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren, "Supporting Dynamic Data Structures on Distributed Memory Machines," *ACM Trans. Programming Languages and Systems,* vol. 17, no. 2, pp. 233-263, Mar. 1995.

[31] M.C. Carlisle and A. Rogers, "Software Caching and Computation Migration in Olden," *Proc. Fifth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPOPP '95),* pp. 29-38, 1995.

[32] T. Reps, S. Horwitz, and M. Sagiv, "Precise Interprocedural Dataflow Analysis via Graph Reachability," *Proc. 22nd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '95),* pp. 49-61, 1995.

[33] I. Dillig, T. Dillig, and A. Aiken, "Sound, Complete and Scalable Path-Sensitive Analysis," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '08),* pp. 270-280, 2008.

[34] B. Hardekopf and C. Lin, "Semi-Sparse Flow-Sensitive Pointer Analysis," *Proc. 36th Ann. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL '09),* pp. 226-238, 2009.

Ming-Yu Hung received the BS and MS degrees in computer science from National Tsing Hua University, Taiwan in 2002 and 2004, respectively. He is currently working toward the PhD degree in Department of Computer Science, National Tsing Hua University, Taiwan. His research interests include program analysis, parallel programming, and compiler optimization.

Peng-Sheng Chen received the BS degree in computer science from National Tsing Hua University, Taiwan, in 1995, the MS degree in computer science and Information Engineering from National Cheng-Kung University in 1997, and the PhD degree in computer science from National Tsing Hua University in 2005. He is currently an assistant professor in the Department of Computer Science and Information Engineering, National Chung-Cheng University, Taiwan. His research interests include parallel programming, optimizing compilers, and computer architectures. He is a member of the IEEE.

Yuan-Shin Hwang received the BS and MS degrees in electrical engineering from the National Tsing Hua University, Hsinchu, Taiwan in 1987 and 1989, respectively, and the MS and PhD degrees in computer science in 1994 and 1998 from the University of Maryland at College Park. He is an associate professor in the Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan. His research interests include parallel and distributed computing, parallel architectures, parallelizing compilers, and programming languages.

Roy Dz-Ching Ju received the BS degree in electrical engineering from National Taiwan University in 1984 and the MS and PhD degrees in electrical and computer engineering from the University of Texas at Austin in 1988 and 1992, respectively. He has been a compiler architect and a fellow at Advanced Micro Devices since 2006. He has been driving the optimization technology for high-performance, cross-platform compilation for AMD multicore CPU and GPU architectures. He is currently the compiler architect for the Fusion System Architecture project. Prior to joining AMD, he made a brief stop at Google, Inc., working on a distributed system and Google Desktop. From 1999 to 2005, he was a senior researcher and the manager of Compiler Technology group at the Programming System Lab in the Microprocessor Technology Labs, Intel Corp. He led an effort in innovating programming and compiler technology for multicore architectures, including a multicore network processor. He had been the architect of an IA-64 open source research compiler (Open Research Compiler), which provided an infrastructure for compiler and architecture research on IA-64 to the research and open source communities. He was with the Hewlett-Packard Company from 1994 to 1999, where he was a project lead in designing and developing an optimizing compiler for IA-64. He worked at IBM from 1992 to 1994 in developing a then state-of-the-art Fortran 90 optimizing compiler. His primary research interests include compiler optimizations, optimization for memory hierarchy, program analysis, computer architecture, multicore systems, and parallel processing. He currently holds 24 US patents and has published more than 60 journal and conference papers in various areas, including array language optimizations, compilation for instruction-level parallelism, cache optimization, coarse-grained parallelization, etc. He has served on the program committees of a number of conferences, such as MICRO-33, MICRO-35, PLDI '01, MSP '02, CGO '03, CGO '04, CGO '05, LCTES '05, COCV '05, Open64 Workshop '08 and Open64 Workshop '09, EUC '09 and EUC '10, PPoPP '11, and PACT '11. He was a general co-chair of CGO '07. He is a senior member of the IEEE and the IEEE Computer Society.

Jenq-Kuen Lee received the BS degree in computer science from National Taiwan University in 1984. He received the MS and PhD degrees in 1991 and 1992, respectively, in computer science from Indiana University. He is now a professor in the Department of Computer Science at National Tsing-Hua University, Taiwan, where he joined the Department in 1992. He was a key member of the team who developed the first version of the pC++ language and SIGMA system while at Indiana University. He was a recipient of the most original paper award in ICPP '97 with the paper entitled "Data Distribution Analysis and Optimization for Pointer-Based Distributed Programs." His supervised PhD student received the distinguished dissertation award as honorable mention by IICM, 1999. He received an achievement award from MOE of Taiwan for University and Industrial joint research, 2001. He received Google research award 2009. In addition, he is a recipient of Taiwan MOEA economic contribution award (Deep Plow Award), 2010. His current research interests include optimizing compilers, computer networks, embedded multicore compilers and systems, and computer architectures.