Modular Interprocedural Pointer Analysis Using Access Paths: Design, Implementation, and Evaluation

Ben-Chung Cheng* Wen-mei W. Hwu†

*Department of Computer Science
†Department of Electrical and Computer Engineering
University of Illinois
Email:{bccheng, hwu}@crhc.uiuc.edu

Abstract

In this paper we present a modular interprocedural pointer analysis algorithm based on access-paths for C programs. We argue that access paths can reduce the overhead of representing context-sensitive transfer functions and effectively distinguish non-recursive heap objects. And when the modular analysis paradigm is used together with other techniques to handle type casts and function pointers, we are able to handle significant programs like those in the SPECcint92 and SPECcint95 suites. We have implemented the algorithm and tested it on a Pentium II 450 PC running Linux. The observed resource consumption and performance improvement are very encouraging.

1 Introduction

Interprocedural pointer analysis is a critical component of accurate static memory disambiguation and has been addressed by many researchers over the past 20 years. The analysis goal has also evolved from detecting aliases among formal parameters for Fortran programs to aliases among multi-level pointer dereferences in C, C++ and Java. However, interprocedural pointer analysis is still not mature on the whole, since the feasibility of a fully functional interprocedural pointer analysis algorithm which can accommodate realistic large programs has not been demonstrated in the literature. Unlike function inlining [1], register allocation [2], and SSA analysis [3], interprocedural pointer analysis has not been integrated into commercial compilers.

The key issues that need to be addressed by an interprocedural pointer analysis algorithm include the following. We will demonstrate later that our algorithm indeed raises the applicability of interprocedural pointer analysis to the next level.

Flow sensitivity. A pointer analysis algorithm is flow-sensitive if it considers the effects of pointer assignments with respect to statement orders. The benefit is that a later assignment may kill earlier definitions of the same pointer. However, most algorithms only allow strong updates for scalar pointer variables [4, 5, 6]. In reality, most scalar pointers are rarely redefined. When they are redefined frequently, usually they are used in loops to traverse arrays or linked lists, but such data structures are often treated conservatively as a single aggregate element. As a result, the accuracy yielded is not proportional to the time spent, and earlier work reports that little benefit is observed on most studied benchmarks [7].

Context sensitivity. A context-sensitive interprocedural pointer analysis algorithm distinguishes different caller contexts for a common callee so that alias information carried by one caller will not be leaked into the other. The necessity of performing context-sensitive pointer analysis for some benchmarks shows little or no precision benefit [8]. However, for those where it is beneficial, we propose an inexpensive context-sensitive algorithm.

Whole-program or modular analysis. The most straightforward way to maintain context sensitivity is to analyze a whole program following the interprocedural control-flow graph [9, 4], but memory usage and analysis time are costly for this method. Partial transfer functions, as proposed by Wilson and Lam [5], use memoization to avoid the redundant analysis of a subset call-graph. The recent modular analysis proposed by Chatterjee et al. [6] further reduces the memory requirement since only method bodies belonging to the same strongly connected components (SCC) [10] in a C++ program need to be memory resident simultaneously. Since real programs contain large SCCs, it is desirable to further reduce the memory requirement for analyzing a single SCC.

Storage- or path-based representation. A static algorithm often needs to use an abstract notation to represent run-time accessed memory locations. Storage-based representation uses extended variable names for physical memory locations [9, 4, 5, 6]. To avoid ambiguity, it is often required that a single memory location cannot be represented by more than one storage name. Due to different aliases among formal parameters, more than one version of transfer functions, either separately maintained [5], or collectively maintained but differentiated by alias contexts [6], are required. Access paths [9, 11], on the other hand, simply represent physical memory locations by how they are accessed from an initial variable in a store-less model. As long as the length of access paths can be bound in the presence of recursive data structures, a context-independent representation of the summary transfer function and an easier way to produce

*Currently employed with Sun Microsystems, Palo Alto, CA 94303.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI 2000, Vancouver, British Columbia, Canada.
Copyright 2000 ACM 1-58113-199-2/00/0006...$5.00.

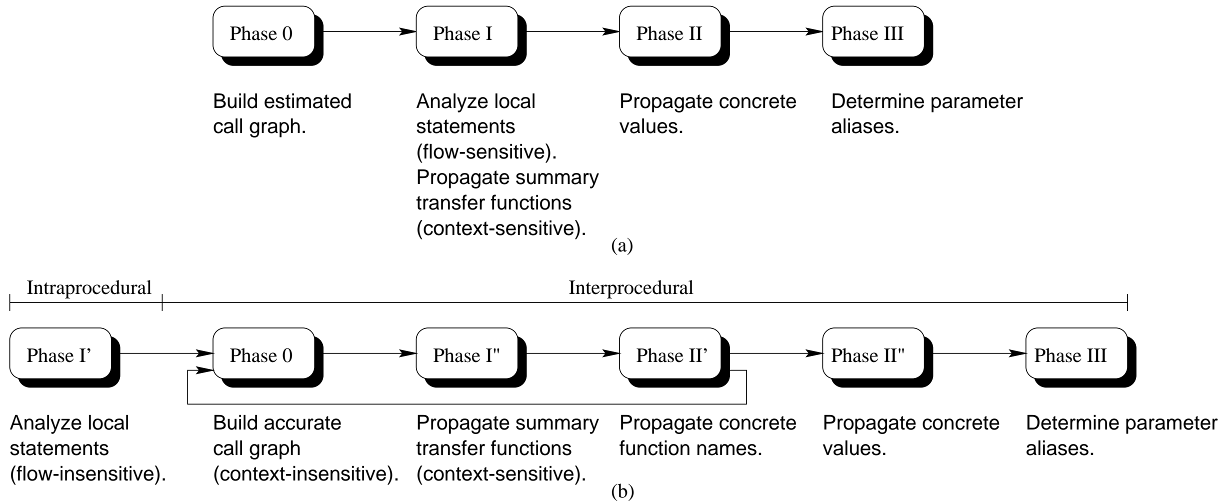


Figure 1: Modular interprocedural pointer analysis: (a) *RCI* algorithm for a subset of C++, (b) proposed modular analysis for complete C.

unique names for heap objects can be enabled. However, the literature does not explain how summary transfer functions are to be maintained in access paths.

Partial or complete language features. A compiler needs a full coverage of all language features in order to perform correct optimizations. The work proposed by Wilson and Lam [5] is the first to cover all C features, but its applicability is limited by the memory requirement of whole-program analysis. On the other hand, many flow- and context-insensitive algorithms are asymptotically faster [12, 13, 14], but handling realistic C features may have a drastic impact on the complexity. For example, without handling structures and unions, Steensgaard's algorithm has almost linear complexity [13]; when aggregate fields are considered, the complexity becomes exponential [15].

Effectiveness study. Most earlier algorithms are evaluated by resource consumptions, pointer resolutions, and statistics of improved optimization opportunities [16, 17, 18, 19]. The work by Wilson and Lam in [5] uses pointer analysis results to guide loop parallelizations for one SPEC benchmark, whereas the work by Diwan et al. [19] uses type-based alias analysis to guide redundant load elimination and loop-invariant load migration for a set of Modula-3 programs. Cooper and Lu [17] study the effect of register promotion for variables in the presence of pointers, but indirect accesses to heap objects and fields of aggregates are not handled. In this paper, interprocedural pointer analysis results are used to guide much more aggressive memory access optimizations including redundant load/store elimination, loop-invariant memory access migration, and load/store scheduling [28]. These optimizations are more aggressive because C is not a type-safe language, and both stack- and heap-based memory accesses can be optimized even for code regions containing function calls. Since a formal verification for the correctness of pointer analysis results is still absent, using the analysis results to guide aggressive optimizations provides a pragmatic alternative for verification.

In this paper, we propose a fully functional interprocedural pointer analysis algorithm which can handle widely used C programs. Our design emphasizes the practical aspect of interprocedural pointer analysis, adopting the modular and flow-insensitive paradigm. The propagation of summary transfer functions is conducted in a context-sensitive but inexpensive manner using access paths and points-to analysis [4]. In accommodating realistic C features, our algorithm handles function pointers by starting with an underestimated call graph which is augmented iteratively along the course of interprocedural pointer analysis [4]. We also incorporate the offset representation for structure/union fields into access paths to handle aliases caused by unions and type casts [5, 20]. In addition, we demonstrate that the number of names representing recursive heap objects can be bound and the location of acyclic heap objects can be disambiguated using access paths. Experiments were conducted in the IMPACT compiler framework [21] using the complete SPECcint92 and SPECcint95 benchmark suites including *126.gcc*, which contains 205,583 lines of code and 2001 functions. Instead of using a high-end workstation, we used a Pentium II 450 PC with 256MB of memory running Linux Red Hat 6.0 to demonstrate the feasibility of our algorithm. With no more than 10 minutes of analysis time and no more than 238 MB of memory usage, our analysis can deliver an average performance improvement of 42% and a peak improvement of 110% over programs compiled without pointer analysis and memory access optimizations. These numbers demonstrate both the efficiency and effectiveness of our algorithm.

The rest of this paper is organized as follows. Section 2 details the proposed modular interprocedural pointer analysis algorithm. Section 3 evaluates the complexity of the algorithm. Section 4 presents empirical results. Section 5 reviews related work. Section 6 concludes the paper.

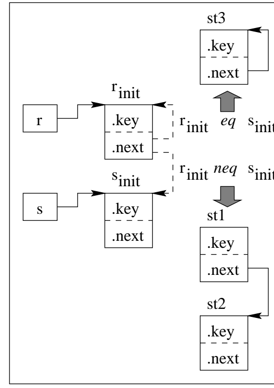
2 Modular Interprocedural Pointer Analysis

Modular interprocedural analysis is a technique that decomposes the whole program into smaller modules so that only a small number of modules need to be resident in memory simultaneously. Our algorithm consists of two major stages: the intraprocedural stage and the interprocedural stage. In

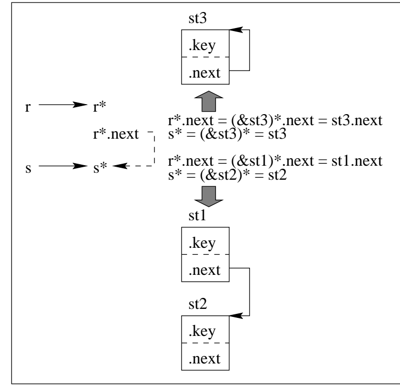
```
typedef struct S {
 int *key;        /* 0_3 */
 struct S *next; /* 4_7 */
} S;
main()
{
     void (*fn2)(S*, S*);
     S st1, st2, st3;
S1:  fn1(&fn2, fn3, fn4);
S2:  (*fn2)(&st1, &st2);
S3:  (*fn2)(&st3, &st3);
}
```

```
fn1((**fa)(S*, S*),
    (*fb)(S*, S*),
    (*fc)(S*, S*))
{
    (**temp)(S*, S*);
S4: temp = fa;
S5: *temp = fb;
}
fn3(S *r, S *s)
{
S6:  r->next = s;
}
fn4(S *r, S *s)
{
S7:  s->next = r;
}
```

(a)



(b)



(c)

Figure 2: Storage- and path-based transfer functions (shown by dashed edges): (a) code, (b) two versions distinguished by alias contexts, (c) one generic version applicable to both calling contexts.

the intraprocedural stage, each function is analyzed as an isolated compilation module where formal parameters, callee return values [1], and global variables are all assumed to have unknown values. Indirectly accessed locations through unknown pointers are represented by access paths. By the end of the intraprocedural stage, a summary behavior of each function is calculated, including a set of memory locations accessible across function boundaries, a set of call-site names, and a set of pointer definitions involving pointers accessible across function boundaries. The third set is considered as the *summary transfer function* and represented by points-to relations. In the interprocedural stage, we perform bottom-up propagation of summary transfer functions along the call graph. In the presence of function pointers, we also need to perform top-down propagation of function names along the partially resolved call graph, since some indirect call-sites may receive concrete function names through parameters. Because the transfer function of a just-resolved indirect callee may define function pointers used somewhere else in the program, the bottom-up and top-down propagation needs to be performed iteratively until a fixed point is reached. The aliases among formal parameters are then calculated after top-down propagation of concrete values along the complete call graph.

At a high-level, the idea behind our algorithm is similar to *Relevant Context Inference* (*RCI*), proposed by Chatterjee et al. [6], since we both use modular analysis to reduce the resource requirements. However, several significant improvements proposed in this paper distinguish our work, as highlighted in Figure 1. First, *RCI* covers a subset of C++ constructs where many general C features including explicit address operator (&), function pointers, structure members, and general pointer assignments are excluded. Our work covers complete C features and can generate safe analysis results to guide code optimizations. Secondly, *RCI* uses an over-estimated call graph. For an indirect call-site, all functions with names taken and identical signatures are considered as possible callees. In *132.ijpeg*, a SPECcint95 benchmark, there are 188 functions with names taken and the majority of them have the same signature. Furthermore, there are 641 indirect call-sites in the program. Therefore the estimated call graph will be very inaccurate, which wastes time in propagating summary transfer functions and produces spurious results. Lastly, *RCI* uses *alias contexts* to lazily enumerate potential aliases among parameters in order to distinguish transfer functions for different calling contexts. In our method, access paths enable a context-independent representation of transfer functions so that the memory used to represent summary functions is reduced.

Consider the example shown in Figure 2a. When an estimated call graph is used, `*fn2` may invoke both `fn3` and `fn4` since both their names are taken and their signatures match. In fact, only `fn3` can be called through `*fn2`, so propagating the summary transfer function of `fn4` would be spurious and unnecessary. Figure 2b shows how summary transfer functions are represented for `fn3` in *RCI*. r_{init} and s_{init} represent the objects pointed by r and s, respectively. The upper dashed edge indicates that $r_{init}.next$ points to r_{init} when `r` and `s` receive the same concrete value, while the lower dashed edge shows that $r_{init}.next$ points to s_{init} when `r` and `s` receive different concrete values. We believe the enumeration is unnecessary since the caller knows better than the callee whether or not identical values are passed to different parameters. As long as the abstract memory name in the transfer function faithfully represents how the memory location is accessed through formal parameters or global variables in a context-independent manner, the caller can easily determine where the location is after replacing the formal parameter with a concrete value. As shown in Figure 2c, by using postfix access paths [11], it is straightforward for call-site `S2` to derive that `r*` stands for `st1` and `s*` stands for `st2`, and for call-site `S3` to derive that both `r*` and `s*` stand for `st3`. Therefore both points-to relations (`st1.next`, `st2`) and (`st3.next`, `st3`) can be derived from a single summary points-to relation (`r*.next`, `s*`).

2.1 Intraprocedural Analysis Stage

Much of the advantage of the proposed algorithm comes from using access paths to represent memory locations. Described in regular expressions, the grammar of an access path is $\mathbf{v}(\mathbf{fd}|\mathbf{d})^*(\mathbf{f}|\epsilon)$, where $\mathbf{v}$ is a variable or call-site name, $\mathbf{f}$ is of the form "`.so_eo`" which denotes the starting and end offsets of a field in a structure/union, and $\mathbf{d}$ denotes a step of dereference and is symbolically shown as "`*`" elsewhere in this paper when there is no confusion with the closure symbol in regular expressions. Conceptually, an access path is simply an encoded postfix sequence of dereference and

[1] For simplicity we will collectively use the term *parameters* to mean function return values.

offset operations taken to reach the final destination. Unless the contents of intermediate pointers in an access path are known, an access path is not bound with any particular locations, but simply leads to the denoted locations.

Definition 2.1 (Construction of access paths) *As defined by the following rules, AP denotes the function that recursively determines postfix access paths for C expressions that involve memory accesses:*

1. *AP(v) =* **v***, where v is a variable.*
2. *AP(exp()) = AP(exp)*_**n()***, where exp is a call-site, either direct or indirect, and n is a unique id.*
3. *AP(*exp) = AP(exp)* if exp is not a function name.*
4. *AP(*exp) = AP(exp) if exp is a function name.*
5. *AP(exp[index]) = AP(exp), where exp is of array type and exp is not a formal parameter.*
6. *AP(exp[index]) = AP(exp)*, where exp is an arbitrary pointer or a formal parameter of array type.*
7. *AP(exp* **op** *exp1) = AP(exp), where* **op** *is an arbitrary binary operator and exp is a pointer type variable.*
8. *AP(exp→field) = AP(exp)**.**so_eo**.
9. *AP(exp.field) = AP(exp)***.so_eo**.
10. *AP(exp.field1.field2) = AP(exp)***.so3_eo3***, where* **so3 = so1+so2** *and* **eo3 = so1+eo2**. ◇

Instead of generating all possible access paths from any variables, access paths are generated lazily from observed C expressions in each function. As recursively defined in Definition 2.1, except for operators "`.`" and "`->`" which always add a constant offset to a pointer, we omit other offset manipulation operators, resulting in a coarse but safe access path for linearly accessed locations. That is, we do not aggressively differentiate elements in an array. It has been shown in the literature that using the starting offset (`so`) and end offset (`eo`) to represent field accesses can effectively resolve aliases caused by unions [5] and type casts [20]. We integrate both techniques by coalescing back-to-back fields as shown in Rule 10. For example, given a nested structure field access `s1.s2.f1`, assuming the size of `s2` is 100 bytes and `f1` occupies the first 4 bytes in `s2`, `((struct * S2) (&s1))->f1` is an alias which also accesses the same location. Without Rule 10, two access paths are encoded as `s1.0_99.0_3` and `s1.0_3`, respectively. With Rule 10, a normalized access path can be generated as `s1.0_3` and therefore aliases can be easily detected. Without explicit pointer assignments, access paths using byte offsets can resolve aliases caused by type casts and pointer arithmetics. How to resolve aliases in the presence of arbitrary pointer assignments will be discussed momentarily.

The points-to relation proposed by Emami et al. [4] is also adopted in this paper to represent the data-flow facts of pointer assignments. The original points-to relation is a three-tuple of the form `(p, t, P|D)` where `p` and `t` are two storage names representing physical pointer and target memory locations, respectively. The third operand, `P|D`, specifies whether the pointer *possibly* or *definitely* points to the target. In the points-to notation defined for this paper, both `p` and `t` are represented in access paths, and the `P|D` attribute is not used since only possible points-to relations are generated.

Given a pointer assignment `lhs = rhs` where both `lhs` and `rhs` are pointer-type C expressions and rhs is not NULL, the first step of determining the corresponding points-to relations is to construct the corresponding access paths for `lhs` and `*rhs`, noticing the dereference operator added to the `rhs` expression. Due to the effects of earlier pointer assignments, `lhs` and `*rhs` may have aliases. For example, given a pointer `p` and a prior pointer assignment `q = &p`, both access paths, `p`, `q*`, are aliases. Similarly, given a pointer `r` and a prior pointer assignment `r = &i`, `r*` and `i` are also aliases. As a result, any of the following four statements can cause `p` to point to `i`: `p = &i`, `p = r`, `*q = &i`, or `*q = r`.

Definition 2.2 (Right-most access path) *Given a direct access to a variable, its right-most access path is simply the variable's name. If a memory location is accessed indirectly, its right-most access path is contrived based on the access paths of the pointers that appear as the very first RHS operand in a sequence of pointer assignments that propagate the address of the indirectly accessed memory location.* ◇

One way to correctly represent the effects of the above set of pointer assignments is to create four points-to relations using the cross products of all aliases of the LHS pointer and all aliases of the RHS target as (`p`, `i`), (`p`, `r*`), (`q*`, `i`), and (`q*`, `r*`). However, the complete enumeration is unnecessary since there are ways to transform an access path into a normalized form based on the observations that aliases are caused by pointer assignments, and pointers must be initialized before they can be used. So for every pointer dereference, there must be one or a small number of *right-most access paths*, as explained in Definition 2.2, which denote the accessed memory location whose addresses are assigned to the dereferenced pointer through an arbitrary number of pointer assignments. As long as all encoded access paths from C expressions can be normalized to the right-most access paths, fewer access paths are ended up and therefore fewer points-to relations need to be maintained. Definition 2.3 shows how to use the encoded access path from a C expression and existing points-to relations to find the entire set of right-most access paths, where Definition 2.4 shows how to add points-to relations based on right-most access paths for a pointer assignment. Notice that the evaluated result is a set of access paths instead of a singular path due to that a pointer may have more than one definition. This is either caused by conditional definitions made to a pointer or the flow-insensitive nature of the pointer analysis algorithm.

Definition 2.3 (Finding the right-most forms of an access path) *Under a set of points-to relations S_{PTR}, the evaluation of an access path is recursively defined as below:*

1. *$EVAL(v \text{ or } (\&v)*, S_{PTR}) = \{v \mid v \text{ is a variable}\}$*
2. *$EVAL(v*, S_{PTR}) = \{v \mid v \text{ is a function name}\}$*
3. *$EVAL(\alpha*, S_{PTR}) = \{\beta \mid \exists\theta \in EVAL(\alpha, S_{PTR}), (\theta, \beta) \in S_{PTR}\} \cup \{\theta* \mid \exists\theta \in EVAL(\alpha, S_{PTR})$, θ does not point to any location and θ has fewer than k prefix paths with common types of $\alpha*\} \cup \{\gamma \mid \exists\theta \in EVAL(\alpha, S_{PTR})$, θ has k prefix paths with common types of $\alpha*$ and γ is the longest one$\}$*
4. *$EVAL(\alpha\mathbf{f}, S_{PTR}) = \{\delta \mid \exists\theta \in EVAL(\alpha, S_{PTR}), \delta = \theta\mathbf{f}\}$* ◇

Definition 2.4 (Path-based points-to relations) *Given a pointer assignment lhs = rhs in function fn where both lhs and rhs are pointer-type C expressions and rhs is not NULL, let S_{PTR} be the set of points-to relations already added for fn. For every $\pi \in EVAL(AP(lhs), S_{PTR})$ and $\tau \in EVAL(AP(*rhs), S_{PTR})$, points-to relation (π, τ) is added to S_{PTR}.* ◇

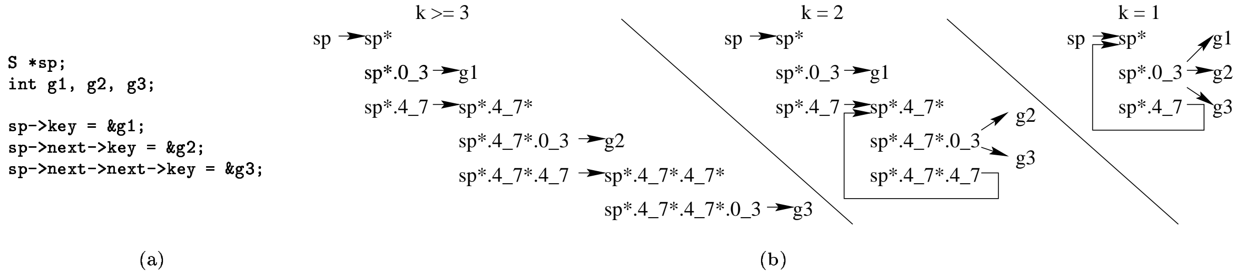


Figure 3: Using access paths to represent recursive data structures: (a) code, (b) different points-to relations controlled by different values of k.

Definition 2.3 defines a closure function EVAL that evaluates the contents of intermediate pointers in an access path by referencing existing points-to relations to find the set of right-most aliases of the input path. Rules 1 and 2 are the terminal cases where no evaluation is made on simple variables and function names. Rule 3 has three cases. The β-set discovers the denoted locations of $\alpha*$ by determining the denoted locations of α first, then for elements found in EVAL(α, S_{PTR}), their points-to locations are the ones denoted by $\alpha*$. If a pointer is not initialized but there is a potential dereference off the pointer, it is addressed by the following two cases. The $\theta*$-set simply creates a trivial target $\theta*$ for a uninitialized pointer θ, if $\alpha*$ is not of a recursive type, or no more than k instances of such a recursive type have been denoted by prefixes of θ. A trivial points-to relation (θ, $\theta*$) is also added. Otherwise the longest prefix of θ with the same C type of $\alpha*$ is reused, as specified by the γ set. k is a *recursive-sensitivity* parameter which differentiates the first k objects in a linked list. This is similar to the *k-limiting* approach used in [9], but we only control the length of access paths involving recursive data types. Consider the example in Figure 3. The larger the value of k, the better the resolution is, at the expense of more access paths and points-to relations. In our experiments, we assume the most conservative case with $k = 1$. Finally, Rule 4 simply refines locations denoted by α with field offsets.

Consider pointer assignments `S4` and `S5` from Figure 2a. The purpose of variable `temp` is designed to show that the indirect modification to `*fa` can be captured by Definitions 2.4 and 2.3. After `S4` is analyzed, points-to relation (`temp`, `fa*`) is added. When analyzing `S5`, due to the previously added points-to relation, `temp` is an initialized pointer, so EVAL(`temp*`, {(`temp`, `fa*`), ...}) is `{fa*}`. On the other hand, the result of EVAL(`fb*`, S_{PTR}) is `{fb*}` since `fb` is a formal parameter which is assumed as uninitialized in the intraprocedural phase. Therefore the points-to relation added for `S5` is (`fa*`, `fb*`).

Figure 4 shows the pseudo code of the intraprocedural stage. For library functions, we approximate their safe summary transfer functions with template statements. For example, points-to relation (`p1*`, `p2**`) can be derived to represent the transfer function of `memcpy` by analyzing:

```
memcpy(void *p1, void *p2, int n) {
    *((char **) p1) = *((char **) p2);
}
```

Currently there are 186 library functions modeled by template statements in the IMPACT compiler. They cover all the library functions invoked by SPECcint92, SPECcint95, MediaBench [22], and many Unix utility programs.

```
Intraprocedural_Pointer_Analysis(fn)
{
    S_PTR(fn) = ∅
    do {
        Apply EVAL(AP(*exp), S_PTR(fn)) for every pointer-type
            actual parameter exp found in fn
        for (each pointer assignment "lhs = rhs" ∈ fn)
            Apply Definition 2.4 to construct points-to relations
        for (each structure/union assignment "lhs = rhs" ∈ fn)
            Apply Definition 2.4 on "lhs.f = rhs.f" for every
                pointer-type field f
    } while (new access paths or points-to relations are added)
}
```

Figure 4: Pseudo code of the intraprocedural pointer analysis stage.

2.2 Interprocedural Analysis Stage

The interprocedural analysis stage of our algorithm has three iterative phases followed by two acyclic phases as shown in Figure 1b. The fundamental tasks performed in these phases are similar to *RCI*, so we use the same major phase numbers with minor annotations. The summarized behavior of each function is the only information needed in the interprocedural stage. In phase 0, depth-first search (DFS) is initiated from function `main` to construct the call graph. Along with DFS, the call graph is partitioned into strongly connected components and viewed as a directed acyclic graph SCC-DAG. If the program has no indirect calls, the iterative phases only need to be entered once and the complete call graph can be built in the first invocation of phase 0. Otherwise, for the example in Figure 2, once points-to relation (`fn2, fn3`) is propagated to `main`, which will happen in phase I", evaluating the access path associated with indirect call-site `*fn2` in phase 0 of the next iteration will reveal the callee, since EVAL(`fn2*`, {(`fn2`, `fn3`), ...}) = `{fn3}`.

In phase I", summary transfer functions are propagated along the SCC-DAG in a reverse topological order. As in [6], summary transfer functions in the same SCC are first propagated iteratively until a fixed point is reached. However, since our method is flow-insensitive, we only need to keep the summary behavior of the caller instead of the full function body and control-flow information of all functions in a SCC simultaneously in memory. As will be shown in the experiments, the largest SCC in benchmark *126.gcc* holds 391 functions [2]. This can introduce memory problems when full instead of summarized information is memory-resident when analyzing a SCC.

[2] The largest benchmark studied in [6] has no more than 230 methods and the largest SCC has no more than 4 methods.

Definition 2.5 (Propagating summary transfer functions) *Let ($r\alpha$, $s\beta$) be a points-to relation in function fn's summary transfer function where r, s are parameters or global variables, and α, β are two suffix access paths. The propagated points-to relations of ($r\alpha$, $s\beta$) to caller fm are:* $\{(\delta, \theta) \mid \exists\delta \in EVAL(a\alpha, S_{PTR}(fm))$ *and* $\exists\theta \in EVAL(b\beta, S_{PTR}(fm))\}$ *where a = r if r is a global variable. Otherwise a is the corresponding actual parameter in fm. The same relation holds between b and s.* ◇

Not every points-to relation in a function's summary behavior belongs in the function's summary transfer function. For example, trivial points-to relations do not count since they do not represent explicit pointer assignments. Given a points-to relation ($r\alpha$, $s\beta$), only when r and s are parameters or global variables and $r\alpha* \neq s\beta$ will ($r\alpha$, $s\beta$) be qualified as an entry in the summary transfer function. To propagate such a points-to relation from the callee to the caller while maintianing context-sensitivity is straightforward. As shown in Definition 2.5, the initial step is to replace the formal parameter with the corresponding actual parameter. Then by referencing the points-to relations contained in the summary behavior of the caller function, the suffix access path guides the search for locations in the caller denoted by the access path constructed in the callee. For example, to propagate the points-to relation (`fa*`, `fb*`) from `fn1` to `main`, `fa` is first replacd by `&fn2`, then EVAL((`&fn2`)*, $S_{PTR}(main)$) = {fn2}. Similarly, `fb` is replaced with `fn3` and EVAL(`fn3*`, $S_{PTR}(main)$) = `{fn3}`. So the points-to relation propagated to `main` from `fn1` is (`fn2`, `fn3`). Besides this example, it is also possible that the propagated points-to relation is represented by the caller's formal parameters or global variables, meaning that the caller's summary transfer function is augmented and will be reported to grand callers. Since the formal parameters of the caller are still considered uninitialized in the interprocedural stage, the augmented transfer function is still context-independent.

The next interesting question is how we handle dynamically allocated objects and propagate their existence across function boundaries. Instead of generating pseudo variable names using synthesized call-site paths, we name heap objects directly allocated from calling `malloc` by call-site names which are unique within each function, as shown by Rule 2 in Definition 2.1; and we name heap objects indirectly allocated from calling wrapper functions by access paths derived from actual parameters, which guarantee that unique names are always generated for objects allocated for different calling contexts. The remaining problem is how we detect the potential dependence among indirect accesses to heap objects allocated for the same calling context. Consider the example shown in Figure 5. In function `fn5`, there are three integer pointers, and through calling `my_malloc`, two instances of heap-based integer objects are allocated. Before propagating the transfer function of `my_malloc` to `fn5`, `p1`, `p2`, and `p3` are assumed to point to disjoint locations as denoted by `p1*`, `p2*`, and `p3*` in Figure 5b [3]. However, in this example `p2` and `p3` point to the same location, and disregarding this fact may cause WAW hazards for statements `S3` and `S4`. There are two options to represent the dependence as either creating an explicit object like `H` in Figure 5c, or adding at least one of the two dashed edges from `p2` to `p3*` or from `p3` to `p2*`. In our algortihm, we choose the latter one because it requires no special representation for heap objects. We use the concept of *extended access path* (*EAP*) to detect the dependence, whose algorithm is shown in Figure 6.

[3]The statement that causes the points-to relation is annotated along each edge.

```
Determine_Extended_Access_Path(fn)
{
    S = ∅
    for (each access path α found in fn's summary behavior) {
        S = S ∪ {α} if α initiates from a parameter or global
            variable
        EAP(α) = α
    }
    do {
        Let γ be the first element in S and remove γ from S
        if (∃ β such that (γ, β) ∈ S_PTR(fn) and EAP(β) is not
            defined) {
            EAP(β) = EAP(γ)*
            S = S ∪ {β}
        }
        if (∃ β.so_eo and EAP(β.so_eo) is not defined) {
            EAP(β.so_eo) = EAP(β).so_eo
            S = S ∪ {β.so_eo}
        }

    } while (S ≠ ∅)
}
```

Figure 6: Pseudo code of EAP construction.

The basic idea behind the algorithm is that if a callee function allocates a heap object for the caller to use, the object must be reachable by conducting DFS following points-to relations initiated from parameters and global variables. An EAP can be considered as a reverse-engineered access path obtained from DFS and indicates a potential way for the object to be accessed from a parameter or a global varible. The first EAP found for an access path is chosen as the persistent name to be viewed by the caller. For example, `malloc_1()*` and `malloc_2()*` represent two heap objects allocated in `my_malloc`. `malloc_1()*` has a unique EAP as `q1**` but `malloc_2()*` may be assigned as `q2**` or `q3**`, depending on whether DFS initiates from `q2` or `q3` first. In the particular case where DFS is performed for `q2` before `q3`, the EAP of `malloc_2()*` is `q2**`, and the points-to relation from `q3*` to `malloc_2()*` is considered in `my_malloc`'s transfer function as (`q3*`, `q2**`). After propagating that from `my_malloc` to `fn5`, the dashed edge from `p3` to `p2*` will be added. Otherwise if DFS is performed for `q3` before `q2`, (`p2`, `p3*`) will be added. It does not matter which one is actually added, as long as EVAL(`p2*`, S_{PTR}(fn5)) ∩ EVAL(`p3*`, $S_{PTR}(fn5)$) $\neq \emptyset$ is true since it asserts the dependence of two memory accesses.

In fact, the constituent points-to relations in the summary transfer function can be identified solely by EAPs. If an access path's EAP is not defined, it means that the denoted location is not accessible by the caller, therefore points-to relations with EAP-less access paths are not considered as part of the transfer function. The revised version of Definition 2.5, which propagates transfer functions as well as detects dependence among accesses to heap objects, is given below.

Definition 2.6 (Propagating summary transfer functions using EAPs) *Let (γ, λ) be a points-to relation in fn. It is part of fn's transfer function if* $EAP(\gamma) \neq \emptyset$ *and* $EAP(\lambda) \neq \emptyset$. *Assuming* $\gamma = r\alpha$ *and* $\lambda = s\beta$, *r and s must be parameters or global variables. The propagated points-to relations of (γ, λ) to caller fm are:* $\{(\delta, \theta) \mid \exists\delta \in EVAL(a\alpha, S_{PTR}(fm))$ *and* $\exists\theta \in EVAL(b\beta, S_{PTR}(fm))\}$. *The remaining conditions are identical to those in Definition 2.5.* ◇

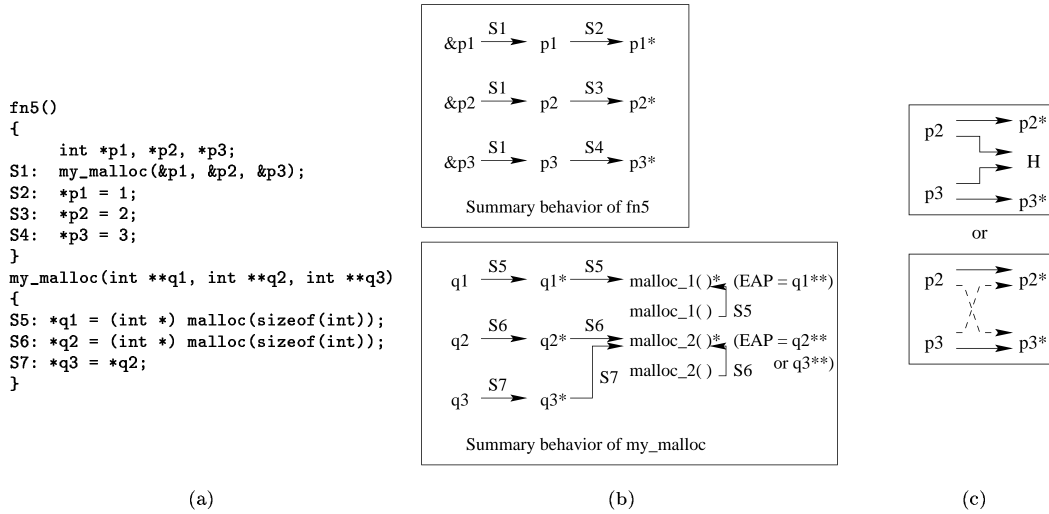


Figure 5: Handling heap objects: (a) code, (b) points-to relations in functions' summary behavior, (c) representing dependence among accesses to heap objects.

```
Interprocedural_Pointer_Analysis(prog)
{
    do {
        Resolve function pointers for each indirect call-site
        Use DFS to compose SCC-DAG for reachable functions
            ∈ prog
        for (each SCC ∈ prog in bottom-up order) {
            Determine EAP for access paths of each function
                ∈ the SCC
            Iteratively propagate points-to relations within the
                SCC if the SCC has more than one function
            Re-determine EAP for each function if new points-to
                relations are received
            Propagate the summary transfer function of the SCC
                to its caller SCCs
        }
        for (each SCC ∈ prog in top-down order) {
            Iteratively propagate function names within the SCC
                if the SCC has more than one function
            Propagate function names from the SCC to its
                callee SCCs
        }
    } while (call graph is changed in the previous iteration)
    for (each SCC ∈ prog in top-down order) {
        Iteratively propagate concrete values within the SCC if
            the SCC has more than one function
        Propagate concrete values from the SCC to its callee SCCs
    }
    Determine aliases among parameters
}
```

Figure 7: Pseudo code of the interprocedural pointer analysis stage.

Phase II' performs a subset of work done in the phase II analysis in *RCI* since we only propagate function names. Regardless of whether a function pointer is defined by local statements (handled in phase I'), side-effects of invoked functions (handled in phase I"), or concrete values passed by callers (handled by phase II'), the accurate call graph can be iteratively built. However, because function names are accumulated in the callee, the call graph built in phase 0 is context-insensitive. When the call graph is finalized, phases II" propagates all concrete values in the top-down manner along SCC-DAG. The result of phase II" will be used in phase III in order to determine potential parameter aliases. Due to space contraints, details will be omitted in this paper since they are very similar to [6]. The pseudo code of the interprocedural stage is shown in Figure 7.

3 Complexity Analysis

In this section we discuss the complexity of our algorithm using a set of parameters. F is the number of user and library functions contained in the program. E is the number of call sites in the program. V_f is the sum of variables and call-sites in a single function. T is the number of data types declared in the program, where k is the number of unique instances distinguished for linked lists. Although C is not well-typed, T is still a finite number. f is the total number of fields with distinct starting and end offsets in all structures/unions. S is the number of pointer assignments in a function, and M is the number expressions that result in memory accesses in a function.

First we demonstrate that the number of access paths stemmed from a variable is bound by $O(f^{kT})$. Since an intermediate path can be casted into a structure type, any path can be widened by field operations, but the length of each path is controlled by the number of types in the program and number of unique instances of linked lists. Therefore the space complexity to store access paths in the intraprocedural phase for a single function is $O(V_f f^{kT})$, and for the whole program in the interprocedural phase is $O(FV_f f^{kT})$. The space complexity to store points-to relations in the intraprocedural phase for a single function is therefore $O(V_f^2 f^{2kT})$, and $O(FV_f^2 f^{2kT})$ in the interprocedural phase. Both the algorithms in the intraprocedural and interprocedural phase will terminate because the number of access paths, points-to relations, and types are finite.

As for the time complexity, generating an access path for a C expression takes $O(l)$ steps, as l is a small and fixed number of dereference and field accesses associated with each expression. Evaluating an access path using Definition 2.1 takes $O(V_f^l f^{lkT})$ steps. Since the our intraprocedural algorithm evaluates every memory access and pointer assignment, the time complexity for each iteration is $O((M+S)V_f^l f^{lkT})$. And since the algorithm is flow-insensitive, the time for analyzing each function can be bound by $O(S(M+$

Table 1: Benchmark characteristics.

Benchmark	Lines	Fns	D. Fns	Libs	Sts/Uns	M. Flds	SCCs	M. SCC
008.espresso	14838	361	62	24	20	19	317	2
023.eqntott	12053	62	7	21	4	17	76	1
026.compress	1503	16	2	24	1	14	38	1
072.sc	8639	179	13	53	13	31	208	7
085.cc1	90857	1452	84	44	88	126	1258	148
099.go	29246	372	26	11	5	20	353	1
124.m88ksim	19092	252	25	36	81	138	261	3
126.gcc	205583	2001	238	45	125	229	1421	363
129.compress	1934	24	12	5	1	14	17	1
130.li	7597	357	9	27	14	44	67	309
132.ijpeg	29290	473	231	18	90	161	258	3
134.perl	26874	277	14	72	34	61	292	23
147.vortex	67205	923	331	33	86	292	576	33

$S)V_f^l f^{lkT})$. In the interprocedural phase, transferring an access paths along the call graph is also bound by $O(V_f^l f^{lkT})$. Using the DFS to partition the call graph into SCC-DAG takes $O(F+E)$ steps, where E may be different from iteration to iteration when indirect function pointers are resolved. In the worst case, E can be bound by $O(F^2)$. Using DFS to determine EAPs takes $O(V_f^2 f^{2kT})$ steps. Resolving an indirect call site also takes the time of $O(V_f^l f^{lkT})$, so assuming there are i indirect call-sites in the program and the height of the call graph is h, the complexity of the interprocedural algorithm can be bound as $O(h((E+i)V_p^l f^{lkT} + (V_p^2 f^{2kT}) + (F+E)))$.

The worst-case complexity occurs when all pointers, including pointer variables and fields, are casted so that they can point to all possible data types. Empirical results indicate that the worst-case complexity is not observed for the studied benchmarks.

Table 2: Resource requirements of conducting interprocedural pointer analysis.

Benchmark	Intra. T. (Sec)	Inter. T. (Sec)	Inter. Mem. (MB)	Total (Sec)
008.espresso	5.66	20.20	14.26	25.86
023.eqntott	0.75	0.99	2.81	1.54
026.compress	0.12	0.48	1.93	0.60
072.sc	2.42	3.15	6.34	5.57
085.cc1	33.61	277.67	79.22	311.28
099.go	2.82	0.21	8.13	3.03
124.m88ksim	2.36	3.54	8.09	5.90
126.gcc	76.68	520.85	238.00	597.53
129.compress	0.09	0.10	1.26	0.19
130.li	2.33	149.59	30.58	151.92
132.ijpeg	7.34	99.16	29.97	106.50
134.perl	14.43	468.33	51.14	482.76
147.vortex	15.08	166.90	74.80	181.98

4 Experiments

The proposed modular interprocedural pointer analysis algorithm has been implemented and tested in the IMPACT compilation framework [21]. To quantify the algorithm's quality, the analysis result is used to guide aggressive optimization routines including redundant load/store elimination, loop-invariant location promotion, and load/store scheduling. Similar experiments have been reported in [17, 23], but our optimizer is much more aggressive due to fully resolved pointers and function side-effects. Instead of using traditional C benchmarks adopted by earlier work, the SPECcint92 and SPECcint95 benchmark suites are chosen to demonstrate that our analysis is capable of handling much more significant C programs.

4.1 Analysis Costs

In this section, the analysis costs for the SPECcint92 and SPECcint95 benchmarks are reported. As listed in Table 1, some general characteristics of each benchmark are shown, including the numbers of lines, functions, dead functions, invoked library functions, structures/unions, number of unique fields in structures/unions, strongly connected components (SCC), and functions contained in the largest SCC. The number of lines directly affects the time spent in the intraprocedural stage, whereas the number of functions plays an important role in determining the complexity in the interprocedural stage. Removing dead functions can speedup the compilation process and improve the run-time performance. In addition to inherited dead functions in the program, function inlining introduce more dead functions to many benchmarks. Since the proposed interprocedural pointer analysis algorithm also considers the summarized behavior of library functions, the effective number of functions being analyzed is the number of total functions minus the number of dead functions plus the number of library functions, whose number ranges from 5 to 72 in these SPEC programs. The number of structures/unions determines the number of unique types in the programs. Together with the number of unique fields considered from all aggregate data types, they determine the number of derivable access paths from a variable. Since benchmark *126.gcc* has 125 aggregate declarations and 229 unique byte-offset representations for fields, as will be shown shortly after, analyzing this benchmark takes the longest time and the largest amount of memory. When each SCC has exactly one function, the total number of functions contained in the program plus the number of invoked library functions minus dead functions will be equal to the number of SCCs, as found in *023.eqntott*, *026.compress*, *099.go* and *129.compress*. Due to the common recursive feature in language processing programs, *085.cc1*, *126.gcc* and *130.li* have large SCCs. The proposed modular analysis technique in Section 2 can reduce the memory usage in analyzing a single SCC since only the summary behavior of each function needs to be maintained.

Table 2 shows the resource consumption statistics collected from a personal computer equipped with an Intel Pentium II processor running at 450 MHz with 256MB of

RAM. The underlying operating system is Linux RedHat 6.0, where all IMPACT modules are compiled using GNU C version 2.91.66 as the native compiler with the "-O" option turned on. The elapsed time is shown in seconds excluding the disk I/O time and the memory usage is shown in mega-bytes (MB). The time in the *Intraprocedural stage* column sums up the analysis time spent on each function. The high-water mark of memory usage in the intraprocedural stage is not shown since each function is analyzed in isolation and therefore the memory usage is no different than other function-level analysis. The *Interprocedural stage* columns show the elapsed analysis time in the interprocedural stage and the high-water mark of memory usage. The sum of analysis time spent in both stages is shown in the last column of Table 2.

As expected, analyzing *126.gcc* consumes the most memory and longest time due to its size (2001 functions/205,583 lines of code), complex structure declarations, and intensive usage of type casts. Due to similar characters found in *085.cc1* and *147.vortex*, they also require more resources than other benchmarks. The conventional wisdom holds that interprocedural pointer analysis is an expensive task which requires a high-end computer system with a huge amount of memory to accomplish. With the statistics shown in Table 2, it is demonstrated that the proposed modular interprocedural pointer analysis algorithm indeed raises the applicability of pointer analysis to the next level. Not only that the analyzed programs are one step closer to real-world programs, but also the system requirements can be met by a personal computer. In the following two subsections, it will be shown that the new algorithm is also accurate and can provide significant performance improvements.

4.2 Accuracy Measurement

In the literature, there are several metrics defined to measure the accuracy of pointer analysis. One common metric is to count the number of targets that each pointer points to, and a general rule to judge the accuracy of a pointer analysis algorithm is that the closer the number of targets is to 1, the better the analysis result. However, the target-per-pointer metric may be misleading when used alone since the most trivial pointer analysis algorithm can simply assume that all pointers point to the same target, causing the algorithm to be falsely judged as perfect. So a symmetric metric which measures the number of pointers that point to the same target is usually used to avoid the biased measurement. Similarly, a better algorithm should also have the pointer-per-target metric reasonable.

Table 3 shows the statistics of the target-per-pointer metric and the pointer-per-target metric observed on rightmost access paths after merging the interprocedural pointer analysis results to each function. Each benchmark has gone through function inlining which allows 60% of code growth. Numbers in Table 3a reflect the numbers of outgoing points-to relations carried by pointer-type access paths, where numbers in Table 3b columns reflect the opposite. Different targets passed to pointer-type formal parameters from different calling contexts are considered as a single object since the multiple instances of targets are the results of the programming style and therefore are irrelevant with the accuracy of pointer analysis. The average numbers of targets resolved for pointer paths range from 1.00 to 1.62, indicating that most pointers are resolved to a single target path. The average numbers of pointer-per-target range from 1.31 to 3.20, showing that there are usually more than one pointer pointing to the same target. Given hundreds or thousands of pointers in each program and the low numbers shown by both metrics, the new modular interprocedural pointer analysis algorithm is very accurate.

The most accurate assessment for flow sensitivity is to conduct both sensitive and insensitive analyses on the same programs [7]. However, it takes a lot of overhead to implement both methods in the same compiler, and it has never been shown in the literature that a flow-sensitive pointer analysis algorithm can handle the complete SPEC suites. Therefore, comparison is conducted indirectly in this paper.

Figure 8 classifies intraprocedural pointer definitions in-

Table 3: Resolutions of interprocedural pointer analysis: (a) target per pointer, (b) pointer per target.

Benchmark	1	2	3	≥ 4	Avg.
008.espresso	3071	436	40	24	1.16
023.eqntott	310	115	0	7	1.31
026.compress	34	1	1	0	1.08
072.sc	662	46	13	6	1.12
085.cc1	9231	1659	384	189	1.26
099.go	312	0	0	0	1.00
124.m88ksim	622	49	3	15	1.15
126.gcc	16147	3679	705	451	1.31
129.compress	37	2	0	0	1.05
130.li	1576	243	50	4	1.19
132.ijpeg	3897	766	135	23	1.23
134.perl	2873	691	340	435	1.62
147.vortex	6768	121	41	5	1.03

(a)

Benchmark	1	2	3	≥ 4	Avg.
008.espresso	1341	329	253	282	1.76
023.eqntott	165	30	31	37	1.77
026.compress	21	6	2	2	1.52
072.sc	291	88	36	40	1.62
085.cc1	2527	617	607	980	2.01
099.go	159	24	5	21	1.46
124.m88ksim	562	88	35	16	1.29
126.gcc	4481	1092	795	1852	2.00
129.compress	36	9	0	0	1.20
130.li	588	239	58	126	1.73
132.ijpeg	3166	829	201	144	1.38
134.perl	1205	477	206	543	2.04
147.vortex	4848	671	214	347	1.35

(b)

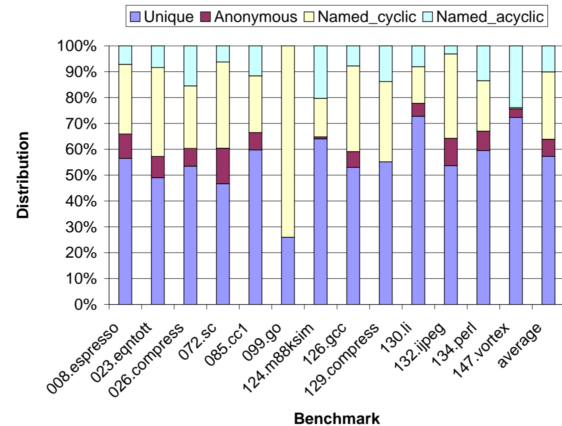


Figure 8: Distribution of pointer definitions.

to four categories as *unique*, *anonymous*, *named_cyclic*, and *named_acyclic*. The unique category represents pointers that are defined no more than once in each function. For example, if a local pointer variable is initialized after being declared and never redefined in a function, it belongs to the unique category. Also, if a pointer-type parameter is defined by the caller and passed to the callee for dereferences only, the pointer belongs to the unique category as well. As shown in Figure 8, on average over 57% of pointers are not redefined. The anonymous category, which accounts for 7% of pointer definitions on average, covers pointers that are not simple variables but are defined for multiple times. The named_cyclic portion, accounting 26% of pointer definitions, covers pointer variables which are defined in loops, where the named_acyclic category represents pointer variables that have more than one definition in acyclic code regions. For pointers that are redefined, a flow-sensitive pointer analysis algorithm can update the target information if the pointer is a variable and the definitions happen in acyclic code regions. The study shows that there are only about 10% of pointer definitions that potentially could be better resolved by a flow-sensitive algorithm. As will be shown in the next section, the additional performance improvement provided by these 10% of pointers is fairly small compared to that provided by static memory disambiguation as a whole, arguing the worth of trading flow-sensitivity for lower complexity.

4.3 Performance Improvements

The results of interprocedural pointer analysis are used to guide three memory access optimizations as mentioned before. The potential benefits of these memory access optimizations are evaluated on a simulated processor which can issue 8 instructions per cycle in order. This processor has 4 memory ports, 4 integer ALUs, 4 floating point units, 2 branch units, and 64 integer and floating point registers. The memory system consists of a 64K direct-mapped, non-blocking data cache with 64 byte block size. The data cache is write-through with no write allocate and has a hit latency of 2 cycles and a miss penalty of 4 cycles if hit in the second-level cache. The second-level cache is a unified 1024K, 4-way associative cache with a miss penalty of 100 cycles. The instruction cache is assumed to be perfect since the goal of the experiments is to understand the data access behavior. The branch prediction scheme is a 1K-entry BTB with 2 bit counters. The instruction set architecture and instruction latencies used match those of the HP PA-7100 microprocessor, where integer operations besides loads have 1-cycle latency [24].

Three sets of parameters in the IMPACT compiler are configured to generate three versions of programs with different levels of sophistication in memory disambiguation. The *base* version assumes all memory instructions are ambiguous and all function calls have pessimistic side-effects. Such pessimistic assumptions completely prevent register promotion and scheduling [4]. The *standard* version employs memory disambiguation information derived intraprocedurally. Memory disambiguation of this level, which is commonly used by commercial compilers without interprocedural pointer analysis, can effectively disambiguate accesses to spill locations, local variables, global variables, and indirect accesses with equivalent base registers and offsets, but revert to ambiguous relations for other combinations. The precise side-effects

[4]For temporary variables inserted by the compiler and local variables whose addresses are never taken, they are always promoted to registers nevertheless.

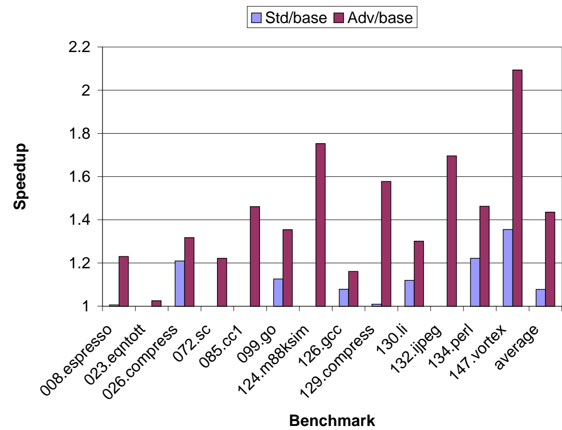


Figure 9: Performance improvements enabled by memory disambiguation.

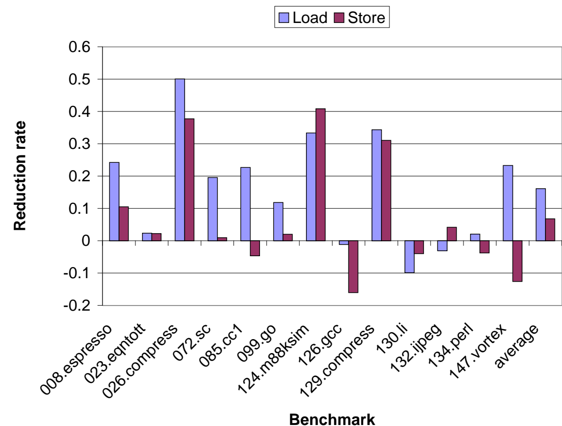


Figure 10: Reduction rates of load and store instructions.

of library function calls can be modeled, but the pessimistic side-effects are still assumed for user functions. The *advanced* version utilizes the complete memory disambiguation information generated by interprocedural pointer analysis and uses the memory disambiguation information to guide memory access optimizations. Side-effects of all functions can be accurately modeled in this version. Except for the differences mentioned above, all three versions are compiled through inlining up to 60% of code increase [25], classical optimizations [26], and superblock optimization [27]. The training input sets are used for all levels of experiments in this paper.

Figure 9 plots the speedups in execution time of the *standard* and *advanced* versions over the base version observed on the 8-issue processor. This figure indicates that memory optimizations with memory disambiguation performed intraprocedurally can provide an average speedup of 1.07. With much more optimization opportunities enabled by better disambiguated memory accesses, the average speedup is boosted to 1.43. The additional performance improvement from the standard version to the advanced version, which has a ratio about 600%, reflects the advancement of performance improvements derived from the compile-time memory disambiguation technology.

To better understand the merits of advanced static mem-

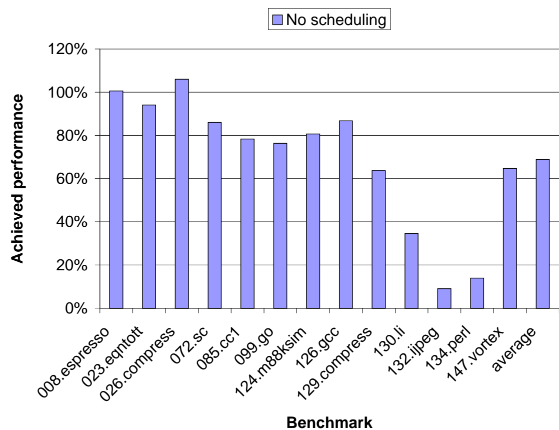


Figure 11: Ratio of the performance improvements enabled by scheduling only versus full memory access optimizations.

ory disambiguation, the run-time behaviors of the standard and advanced versions of code are further studied. The left bar of each benchmark in Figure 10 shows the reduction of dynamic loads between the standard and advanced versions. As these numbers show, as much as 40% of dynamic loads for *026.compress* can be eliminated, with an average reduction rate of 16%. Attributing the minor variance (± 3%) in speedup numbers to the normal deviation of detailed simulation, loads in benchmarks *023.eqntott*, *126.gcc*, *130.li*, *132.ijpeg*, and *134.perl* are not eliminated in a significant manner. The right bar of each benchmark in Figure 10 stands for the percentage of stores that are eliminated. For some benchmarks, the numbers of loads and stores are actually increased due to two reasons. The first reason is many load instructions are speculatively executed. For example, consider the code sequence "`branch-store-load`". Without interprocedural pointer pointer analysis, the relation between the store and the load may be ambiguous and therefore the load is only executed after the branch is not taken. If the memory disambiguation information indicates that the store and the load are independent and there are empty memory slots above the branch, the load can be executed speculatively before the branch. If the branch is not taken, the load instruction has been issued and its latency can be hidden. Otherwise an unnecessary load is executed whose result will be discarded. The second reason is the increased register pressure due to more memory contents are held in registers, which may increase both the numbers of load and store instructions because of register spilling and refilling. As shown in Figure 9, significant performance improvements are still obtained even though there are extra memory instructions.

Figure 11 shows the contribution of register promotion to the overall performance improvement with memory disambiguation. The 100% speedup is derived from the performance difference between the *standard* version and the *advanced* version, where the achieved performance improvement shown by each bar is obtained by disabling load/store scheduling. Comparing Figure 10 and 11, it shows that register promotion and scheduling both can provide significant performance improvements. Benchmarks like *008.espresso* and *026.compress* obtain all of their performance improvements from register promotion. Without speculatively executed load instructions which may cause extra cache misses, the performance is even slightly better. On the other hand, for benchmark like *132.ijpeg* with a negative load-reduction rate, the performance speedup between the *advanced* and the *standard* version is as high as 1.70, where only 10% of the total speedup is achieved when scheduling is disabled.

Additional experimental results can be found in [28], including the estimate of performance loss due to spurious analysis results and the impact of reduced machine resources to the benefit of static memory disambiguation. Case studies from SPEC code fragments are also included to exemplify the strength and weakness of the proposed modular interprocedural pointer analysis algorithm.

5 Related Work

In addition to the discussion in Section 1 and a thorough study of *RCI* in Section 2, we will summarize more related work in this section. Landi et al. [9] use the may-alias representation to perform flow- and context-sensitive interprocedural pointer analysis. This is the pioneer work in improving the accuracy of pointer analysis using the interprocedural control-flow graph (ICFG). The analyzed language is simplified from C, and memoization is not used. Emami et al. [4] propose the points-to representation to model the storage shape graph. Since their analysis is flow-sensitive, their points-to relations can be classified as *definite* or *possible*, while we only generate *possible* relations. Their algorithm is not designed to aggressively disambiguate heap objects, and memoization is not used either. They also handle a subset of C, but function pointers are included. Choi et al. [29] propose a sparse representation for the alias pairs, and they use the invocation path towards memory allocation sites to name heap objects. However, experimental results and implementation details are not provided. To look beyond tradition applications of pointer analysis, Rugina and Rinard [30] perform pointer analysis for multithreaded programs.

Many algorithms use types to perform alias analysis [19, 31, 13, 32]. Deutsch [11] uses symbolic access paths to analyze the alias relations among accesses to recursive structures on well-typed programs. In our algorithm, although the access path notation is also postfix, we use low-level offsets for fields. Types are also used differently since they are not directly used to resolve aliases due to type casts in C, but used to control the length of access paths in the presence of recursive data structures. The flow-insensitive analysis proposed by Andersen [14] has cubic worse-case running time, and the work from Fähndrich et al. [33] shows that Andersen's method can be practical for large programs. Shapiro et. al [18] propose a hybrid method with configurable complexity between Steensgaard's and Anderson's algorithms. Hasti et al. [34] propose the idea of iteratively improving the accuracy of pointer analysis in SSA form [3], but empirical result is not available. Zhang et al. [35] and Stocks et al. [36] balance the complexity by switching between various algorithms. The largest benchmark handled has around 60,000 lines of code. Our algorithm is the first to demonstrate that a flow-insensitive analysis accommodating complete C features can handle programs of more than 200,000 lines of code.

Besides resolving aliases for general pointer dereferences, a lot of researchers have focused on obtaining detailed descriptions for heap objects [37, 38, 39, 40, 41, 11]. Our algorithm can only provide a safe approximation but not a detailed profile.

6 Conclusions

In this paper we have presented an efficient and effective modular interprocedural pointer analysis algorithm for C programs. Our algorithm can calculate flow-insensitive and context-sensitive summary transfer functions based on the accurate call graph, therefore shorter analysis time and better analysis results can be achieved when the program contains a large number of indirect function calls. The low-level representation of structure/union fields in access paths handles both aliases caused by type casts and unions.

The experimental results demonstrate that our method can efficiently handle much larger programs than other interprocedural pointer analysis algorithms. Reasonable time and memory usage suggest that our algorithm has nice scalability, and thus is more feasible for implementation in production environments. The performance study on the complete SPECcint92 and SPECcint95 benchmark suites also shows that the proposed interprocedural pointer analysis can provide significant speedup.

Future work includes the evaluation of the proposed algorithm on a broader range of real applications. Although handling the SPEC benchmarks is an important milestone towards practical interprocedural pointer analysis, their size is still a fraction of many real world applications. We are also actively studying the relations between aggressive static memory optimizations and data speculation with hardware supports.

7 Acknowledgments

The authors would like to thank all the members of the IMPACT research group for their assistance, especially Dan Connors and Erik Nystrom for their design and implementation of many memory access optimization routines, and Marie Conte and Hillery Hunter for their careful review. The valuable comments from the anonymous reviewers significantly improved the quality of the paper. This research has been supported by the National Science Foundation (NSF) under grant CCR-9629948 and the Intel Foundation Fellowship.

References

[1] R. Allen and S. Johnson, "Compiling C for vectorization, parallelization, and inline expansion," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 241–249, June 1988.

[2] F. Chow and J. Hennessy, "Register allocation by priority-based coloring," in *Proceedings of the SIGPLAN 1984 Symposium on Compiler Construction*, pp. 222–232, June 1984.

[3] 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 Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, October 1991.

[4] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 242–256, June 1994.

[5] R. P. Wilson and M. S. Lam, "Effective context-sensitive pointer analysis for c programs," in *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pp. 1–12, June 1995.

[6] R. Chatterjee, B. G. Ryder, and W. A. Landi, "Relevant context inference," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 133–146, January 1999.

[7] M. Hind and A. Pioli, "Assessing the effects of flow-sensitivity on pointer alias analyses," in *Lecture Notes in Computer Science*, Springer-Verlag, 1998. Proceedings from the 5th International Static Analysis Symposium.

[8] E. Ruf, "Context-insensitive alias analysis reconsidered," in *Proceedings of the ACM SIGPLAN 95 Conference on Programming Language Design and Implementation*, pp. 13–22, June 1995.

[9] W. Landi and B. G. Ryder, "A safe approximate algorithm for interprocedural pointer aliasing," in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 235–248, June 1992.

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms.* The MIT Press and McGraw-Hill, 1992.

[11] A. Deutsch, "Interprocedural may-alias analysis for pointers: Beyond k-limiting," in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 230–241, June 1994.

[12] M. Burke, P. Carini, J. D. Choi, and M. Hind, "Flow-insensitive interprocedural alias analysis in the presence of pointers," in *Lecture Notes in Computer Science, 892* (K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, eds.), pp. 234–250, Springer-Verlag, 1995. Proceedings from the 7th Workshop on Languages and Compilers for Parallel Computing.

[13] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 32–41, January 1996.

[14] L. O. Andersen, *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994.

[15] B. Steensgaard, "Points-to analysis by type inference in programs with structures and unions," in *Lecture Notes in Computer Science, 1060* (T. Gyimothy, ed.), pp. 136–150, Springer-Verlag, 1996. Proceedings from the International Conference on Compiler Construction.

[16] R. Ghiya and L. J. Hendren, "Putting pointer analysis to work," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 121–133, January 1998.

[17] K. D. Cooper and J. Lu, "Register promotion in c programs," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pp. 308–319, June 1997.

[18] M. Shapiro and S. Horwitz, "Fast and accurate flow-insensitive points-to analysis," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 1–14, January 1997.

[19] A. Diwan, K. McKinley, and J. E. B. Moss, "Type-based alias analysis," in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pp. 106–117, June 1998.

[20] S. Yong, S. Horwitz, and T. Reps, "Pointer analysis for programs with structures and casting," in *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pp. 91–103, May 1999.

[21] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 266–275, May 1991.

[22] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 330–335, December 1997.

[23] R. Bodik, R. Gupta, and M. L. Soffa, "Load-reuse analysis: Design and evaluation," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pp. 64–76, May 1999.

[24] Hewlett-Packard Company, Cupertino, CA, *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 1990.

[25] B. Cheng, "A profile-driven automatic inliner for the impact compiler," Master's thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1997.

[26] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1986.

[27] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.

[28] B. Cheng, *Compile-Time Memory Disambiguation for C Programs.* PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 2000.

[29] J. D. Choi, M. G. Burke, and P. Carini, "Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects," in *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pp. 232–245, January 1993.

[30] R. Rugina and M. Rinard, "Pointer analysis for multithreaded programs," in *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pp. 77–90, May 1999.

[31] M. C. Rinard and P. C. Diniz, "Commutativity analysis: A new analysis framework for parallelizing compilers," in *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, June 1996.

[32] E. Ruf, "Partitioning dataflow analyses using types," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1997.

[33] M. Fähndrich, Z. S. J. S. Foster, and A. Aiken, "Partial online cycle elimination in inclusion constraint graphs," in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pp. 85–96, June 1998.

[34] R. Hasti and S. Horwitz, "Using static single assignment form to improve flow-insensitive pointer analysis," in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pp. 97–105, June 1998.

[35] S. Zhang, B. G. Ryder, and W. A. Landi, "Experiments with combined analysis for pointer aliasing," in *Proceedings of the 1998 Workshop on Program Analysis for Software Tools and Engineering*, June 1998.

[36] P. A. Stocks, B. G. Ryder, W. Landi, and S. Zhang, "Comparing flow and context sensitivity on the modifications-side-effects problem," in *International Symposium on Software Testing and Analysis*, pp. 21–31, March 1998.

[37] N. D. Jones and S. S. Muhnick, "A flexible approach to interprocedural data flow analysis and programs with recursive data structures," in *Conference Record of the 9th ACM Symposium on Principles of Programming Languages*, January 1982.

[38] J. R. Larus and P. N. Hilfinger, "Detecting conflicts between structure accesses," in *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 21–34, July 1988.

[39] L. J. Hendren and A. Nicolau, "Parallelizing programs with recursive data structures," *IEEE Transactions on Parallel and Distributed System*, vol. 1, pp. 35–47, January 1990.

[40] D. R. Chase, M. Wegman, and F. K. Zadeck, "Analysis of pointers and structures," in *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 296–310, June 1990.

[41] L. Hendren, J. Hummel, and A. Nicolau, "Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs," in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 249–260, June 1992.