COMPILE-TIME MEMORY DISAMBIGUATION FOR C PROGRAMS

BY

BEN-CHUNG CHENG

B.S., National Taiwan University, 1992
M.S., University of Illinois, 1997

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2000

Urbana, Illinois

COMPILE-TIME MEMORY DISAMBIGUATION FOR C PROGRAMS

Ben-Chung Cheng, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 2000
Wen-mei W. Hwu, Advisor

Static memory disambiguation is a compile-time technique which determines whether or not two memory instructions access the same memory location at run-time. With fully disambiguated memory accesses, instruction scheduling and code optimization can be conducted in a much more aggressive manner. As the speed gap between the processor and memory widens, the value of optimized memory accesses will scale accordingly. To disambiguate memory accesses beyond scalar local variables, a critical task in an optimizing compiler is to perform interprocedural pointer analysis in order to handle indirect memory accesses through pointers in the presence of function calls with side-effects. Interprocedural pointer analysis has long been considered as too expensive to afford. Therefore unlike function inlining, register allocation, and SSA analysis, interprocedural pointer analysis has not been integrated into commercial compilers.

The core of this dissertation is a new modular interprocedural pointer analysis algorithm which significantly improves the feasibility of static memory disambiguation. The proposed algorithm can handle much larger C programs with realistic language features. Due to its modular feature, the amounts of time and memory requirements are greatly reduced. The dissertation also evaluates the effectiveness of the proposed algorithm on the complete SPECcint92 and SPECcint95 benchmark suites. Significant performance improvements are observed from these pointer-intensive programs.

DEDICATION

To my family.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Professor Wen-mei Hwu, for his guidance and support during my studies. His sharp sense always can correct my research flaws at the earliest stage and keep my stream of though on the right track. It is a great honor and pleasure to have an advisor like him.

Next, I would like to extend my gratitude to the other members of my dissertation committee, Professor David Padua, Professor Constantine Polychronopoulos, and Professor Laxmikant Kale. Their suggestions significantly improved the quality of this work.

For different periods of my studies at Illinois, there are different people for me to express my gratitude. During my initial stage as a graduate student, I would like to thank Liang-Chuan Hsu and Grant Haab the most. With Liang-Chuan's experience, I could pass my comprehensive and oral qualifying examinations within the shortest amount of time; and with Grant's kindness to answer all my tedious questions regarding IMPACT, I quickly built a solid background to initiate my research. In my middle stage as a graduate student, I really appreciate Carole Dulong and Jim Pierce who gave me an internship opportunity at Intel Corporation. The project I worked there during the summer of 1997 eventually evolved into my PhD research topic. As I am getting closer to graduation, I really cherish the support and friendship from Le-Chun Wu. Whenever I felt frustrated in making progress for my research, I always could sense the same or even greater level of frustration from him. A brief visit to his office always gave me enough joy and energy to get back to my own problems.

The research presented in this this dissertation would not have been possible without the support of the members of the IMPACT research group, both past and present. Dan Connors has always been there to answer my questions regarding the backend of the IMPACT compiler. I also owe thanks to John Gyllenhaal, Brian Deitrich, Dan Lavery, David August, Teresa Johnson, Rick Hank, and David Gallagher for answering numerous IMPACT compilation questions that cannot be handled by Dan Connors. Other members including Marie Conte, Matt Merten, John Sias, Erik Nystrom, Andy Trick, Ron Barnes, Hillery Hunter, and Hong-Soek Kim also provided valuable assistance to my software enhancements and presentation practices. I would also like to thank my office mate, Sabrina Hwu, who made my stay in the group as warm as at home. Other than research partners, I also cherish the happy memories from my campus buddies, including Tai-Yi Huang, Ching-Han Tsai, Yu-Ming Chang, Hao-Hua Chu, Chi-Li Sung, Hao-Chung Kuo, Chien-Wei Li, Mei-Yen Tsai, Jane Wang, and Christine Jiau.

Finally, I would like express the most sincere gratitude to my parents, Wei-Tung and Ching-Wen Cheng, and my wife, Szu-Wen Kuo, for their endless love and support in all aspects. My wife's passion for knowledge and achievements in research have always been my models ever since our undergraduate days, leading me through many difficult challenges. I am truly blessed to have a lovely wife who is also the greatest companion in study and work.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A wide-issue superscalar processor cannot sustain its peak speed unless the memory system can provide data at the same rate as they are consumed by the processor. Since the performance gap between the processor and the memory keeps growing, the memory access latency has an even greater impact on performance, motivating the need for techniques that either eliminate memory instructions or at least tolerate the latency of load instructions.

Various hardware techniques that can hide the latency of load instructions have been investigated in the past. In addition to improvements made on the cache designs, reducing the load latency by accessing the memory earlier than the normal pipeline stage using a speculatively generated effective addresses can successfully reduce the cache hit latency by one or two cycles [1, 2, 3, 4, 5, 6]. Similarly, the concept of value prediction [7, 8] has introduced machine models that exceed the limit imposed by data dependences by predicting the outcome values of instructions. All these speculative mechanisms will not eliminate load instructions contained in the program. Instead, they may increase the total amount of memory traffics if mis-prediction prevails. In general, the effectiveness of these dynamic approaches is more significant if the underlying programs are less aggressively optimized.

To address the memory latency problem, another set of techniques use the compiler to aggressively transform load and store instructions under the guidance of static memory disambiguation information. The most effective way is to perform register promotion so that instructions communicate through registers instead of the memory. If register promotion is not possible due to hazardous memory instructions or function calls, a less ambitious approach is to schedule load instructions as ahead of the original position as possible until a potentially conflicting store is reached. In this way either the entire memory latency or at least part of it can be hidden. Without interprocedural pointer analysis, the compiler can only disambiguate direct accesses to local variables. For indirect accesses or accesses to global variables, they can only be disambiguated in a very limited code scope, which may be even smaller than a basic block. Given a program with intensive usage of pointers and function calls, analysis in such a scope can only provide modest performance gains. Therefore interprocedural pointer analysis plays a pivotal role in determining the quality of static memory disambiguation, which in turn determines the aggressiveness of memory access optimizations. However, conventional wisdom holds that interprocedural pointer analysis is not practical to be conducted in commercial compilers, so dynamic memory disambiguation mechanisms are persuaded as alternative approaches to tolerate the memory latency [9, 10].

This dissertation addresses the memory latency problem using aggressive static memory disambiguation. An efficient and effective interprocedural pointer analysis algorithm for C programs is proposed to demonstrate that static memory disambiguation can be practically integrated into an optimizing compiler. With interprocedural pointer analysis, the compiler can optimize programs in a much larger scope. Without any architectural modifications, the total amount of memory traffics can be reduced by the compiler while remaining memory instructions in the program can be scheduled in a much wider range to tolerate the memory latency. To demonstrate that the new interprocedural pointer analysis algorithm does raise the applicability of static memory disambiguation to the next level, the complete suites of SPECcint92 and SPECcint95 [11] are analyzed in the experiments. The SPEC suites are the most commonly used benchmarks to evaluate the performance of modern computer systems. Due to their sizes and many realistic C features, they are often conservatively optimized by pure static methods, leaving many novel architectural designs guided by sub-optimal program traces. Through aggressive memory access optimizations, significant performance improvements are observed which not only demonstrate the effectiveness of the proposed algorithm but also provide new insights to these benchmarks so that their optimized behaviors can be understood.

1.1 Contribution

The contributions of this dissertation are described by the following aspects.

- An efficient and effective interprocedural pointer analysis algorithm is proposed. The challenges faced by an interprocedural pointer analysis algorithm often include the control of resource usage and the coverage of realistic language features. To address the first challenge, the proposed algorithm uses the modular [12] and flow-insensitive [13, 14] paradigm to shorten the analysis time and lessen the memory consumption. To overcome the second challenge, the proposed algorithm handles all C features including unions, type casts, and function pointers. Consequently, the analysis result is safe and can be used to guide aggressive optimizations.

- A systematic way to exploit optimization opportunities using memory disambiguation information is demonstrated. With fully disambiguated memory accesses, the dependency between a pair of memory instructions can be accurately defined as *dependent*, *independent*, and *ambiguous*. These classifications are referenced by optimization routines to transform the program. However, a certain degree of conservative assumptions are inevitable for any static analysis algorithms including the proposed one. Therefore memory address profiling [15] is conducted in this dissertation to estimate the performance benefit that would be available if the conducted static memory disambiguation were perfect. Experimental results indicate that the extra performance gain is negligible compared to the performance improvements enabled by the new interprocedural pointer analysis algorithm, meaning a significant degree of accuracy has been achieved.

- An in-depth understanding of the usage of function pointers is provided. In addition to improving the performance the programs, static memory disambiguation information can also improve the productivity of programmers. Among many programming tools, a call graph extractor is a very important one since it helps the programmer understand the overall structure of the program. However, in the presence of function pointers, the complete and accurate call graph cannot be easily constructed. This dissertation presents a comprehensive study of the usage of function pointers found in the SPECcint92 and SPECcint95 suites. Experience from these benchmarks argues that function pointers can be as complicated as regular pointers. That is, an accurate call graph extractor would not be available unless full scale interprocedural pointer analysis is performed.

1.2 Overview

This dissertation consists of eight chapters. Chapter 2 explains various issues involved in interprocedural pointer analysis in detail. A comprehensive review of previous work is also discussed in this chapter.

Chapter 3 and Chapter 4 present the intraprocedural and interprocedural stages of the proposed modular interprocedural pointer analysis algorithm, respectively. Given a C program, each function is first analyzed as an isolated compilation unit where parameters and global variables are temporarily assumed to have uninitialized values. The analysis conducted in this stage will summarize the intraprocedural behavior of a function in a flow-insensitive manner, including how it can affect memory accesses in the caller and callee functions, and how its memory accesses can be affected by the caller and callee functions. The summarized behavior of each function is the only information to be processed in the next stage, the interprocedural stage. Because of the significant size reduction in the summarized representation than the full function body, much larger programs can be handled by the proposed modular algorithm.

Chapter 5 details how to utilize static memory disambiguation to guide memory access optimizations, including redundant load/store elimination, loop-invariant code migration, and instruction scheduling. Chapter 6 classifies the usage of function pointers using the SPEC suites. Comprehensive case studies extracted from representative benchmarks are shown to illustrate the intricacy of pointer manipulations. Resolutions of function pointers are presented to quantify the accuracy of interprocedural pointer analysis.

Chapter 7 quantitatively compares the performance of programs compiled with and without interprocedural pointer analysis. Different architectural configurations and optimization routines are evaluated to understand the relationship between machine resources and compiler capabilities. A quantitative upper-bound analysis using memory address profiling is also presented in this chapter. Finally, Chapter 8 presents conclusions and future research suggestions.

CHAPTER 2

OVERVIEW OF STATIC MEMORY DISAMBIGUATION

Interprocedural pointer analysis is a critical component of accurate static memory disambiguation and has been addressed by many researchers over the past twenty years. The analysis goal has 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 [16], register allocation [17], and SSA analysis [18], interprocedural pointer analysis has not been integrated into commercial compilers.

2.1 Background Information

The key issues that need to be addressed by an interprocedural pointer analysis algorithm include the following. It will be demonstrated later that the proposed modular interprocedural pointer analysis algorithm indeed raises the applicability of interprocedural pointer analysis to the next level.

2.1.1 Flow sensitivity

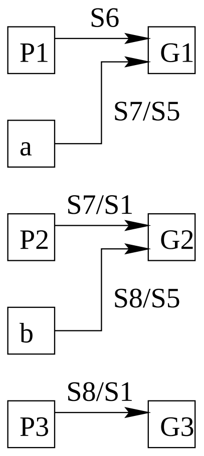
A flow-sensitive pointer analysis algorithm considers the effects of pointer assignments with respect to statement orders. The benefit is that a later assignment may kill earlier definitions of the same pointer, referred to as *strong update.* However, situations that arise in real code often do not benefit from flow-sensitive analysis. For example, most algorithms only allow scalar pointers to be strongly updated [19, 20, 12]. In reality, most dereferences are off heap-based pointers or pointer fields in structures/unions, and most scalar pointers are rarely redefined unless they are used in loops to traverse arrays or linked lists. However, 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 [21].

2.1.2 Context sensitivity

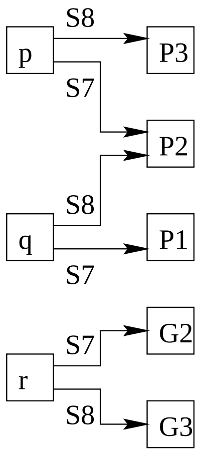
A context-sensitive interprocedural pointer analysis algorithm distinguishes different caller contexts for a common callee so that alias information carried by one caller will not leak into the other. Strictly speaking, maintaining context sensitivity can be further distinguished as maintaining context-sensitive parameter aliases and context-sensitive transfer functions. In the context of interprocedural pointer analysis, transfer functions represent the side-effects of function calls and operate by taking a set of pointers that are accessible via parameters or global variables and returning a set of target locations that are pointed to by these input pointers. Consider the code example in Figure 2.1a.

```
int *P1, *P2, *P3, G1, G2, G3;
int *foo(int **p, int **q, int *r)
{
S1: *p = r;
S2: (**p)++;
S3: G1++;
S4: G2++;
S5: return *q;
}
main()
{
    int *a, *b;
S6: P1 = &G1;
S7: a = foo(&P2, &P1, &G2);
S8: b = foo(&P3, &P2, &G3);
}
```

(a)



(b)



(c)

Figure 2.1 An example about context-sensitivity: (a) code, (b) pointers resolved using context-sensitive transfer functions, (c) context-sensitive aliases among parameters.

By calling function `foo` twice with different values passed as parameters, pointers are initialized differently by instantiated statements contained in `foo`.

When the concrete values bound to parameters are distinguished in the caller's perspective, a context-sensitive transfer function is resulted and it can prevent the *impossible path* problem [22]. Figure 2.1b shows the points-to relations [19] obtained after applying the context-sensitive transfer function of `foo`, where each edge is annotated with the labels of corresponding statements. For example, pointer `P3` in Figure 2.1b will not be resolved to point to `G2` if the transfer function of function `foo` is aware that addresses of `P3` and `G2` are not passed to function `foo` in the same context. On the other hand, Figure 2.1c shows the context-sensitive alias relations among formal parameters of function `foo`, where contexts are distinguished by the call-site statement labels. When the concrete values bound to parameters are distinguished in the callee's perspective, not only true aliases among parameters are identified to guarantee the correctness of optimizations applied to the callee function body, but also spurious aliases among parameters can be eliminated to enable more optimizations. For example, the third parameter `r` in function `foo` may point to `G1` if `foo` is invoked from `S7`, and `G2` if `foo` is invoked from `S8`. If these alias relations are not detected, the compiler may overlap the execution of statements `S2`, `S3`, and `S4`, violating the semantic order of the program since these three statements have WAW dependences. However, the first two parameters of function `foo`, `p` and `q`, may point to `P2` but they never do so simultaneously. Therefore the compiler can schedule the load instruction that retrieves data from `*q` before the store instruction that writes value to `*p` in order to hide the load latency.

Context-sensitive pointer analysis shows little or no precision benefits from benchmarks studied in [23]. However, for those where context-sensitivity is beneficial, the proposed algorithm in this dissertation can maintain context-sensitive transfer functions inexpensively. However, context-sensitive parameter aliases are not maintained.

2.1.3 Whole-program or modular analysis

The most straightforward way to maintain the context sensitivity of parameter aliases and function call side-effects is to analyze a whole program following the flow of statements, including statements in invoked functions [22, 19]. For example, the program in Figure 2.1a can be analyzed by following statements `S6` $\rightarrow$ `S7` $\rightarrow$ `S1-5` $\rightarrow$ `S8` $\rightarrow$ `S1-5`.

When the program is large and the call graph is dense, the analysis cost will be too high for practical use. Partial transfer functions, as proposed by Wilson and Lam [20], use memoization to avoid the redundant analysis of an invoked subset of the call-graph. For the example shown in Figure 2.1, the alias relations among parameters for call sites `S7` and `S8` are identical, so the memoized transfer function obtained from `S7` can be reused for `S8` so that `S1` through `S5` need not be analyzed again. However, function bodies along a call-graph path need to be memory resident simultaneously due to the algorithm's flow-sensitive feature, affecting the scalability of the algorithm.

The recent flow- and context-sensitive modular analysis proposed by Chatterjee et al. [12] handles a subset of the C++ language. Their method further reduces the memory requirement of interprocedural pointer analysis because only method bodies belonging to the same strongly connected components (SCC) [24] in a C++ program need to be memory resident simultaneously. For methods belonging to different SCCs, only the summary transfer functions are brought to memory for propagation. Since real programs contain large SCCs, it is desirable to further reduce the memory requirement for analyzing a single SCC.

2.1.4 Abstract memory 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 [22, 19, 20, 12]. 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 [20], or collectively maintained but differentiated by alias contexts [12], are required. Access paths [22, 25], 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 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. In this thesis, the concept of access paths will be further explained in Section 3.1.

2.1.5 Coverage of language features

A compiler needs a full coverage of all language features in order to perform correct and aggressive optimizations. For example, if individual fields of a structure cannot be disambiguated, indirect accesses from field pointers will be considered as ambiguous and stay unoptimized. This is definitely an undesired situation, since structures are handy programming features and are ubiquitous in modern programs. But many other language features like type casts, unions, and function pointers have long been neglected as well. The recent work proposed by Wilson and Lam [20] is the first to cover all C features in performing interprocedural pointer analysis, but its applicability is limited by the memory requirement of whole-program analysis. On the other hand, although many flow-insensitive and context-insensitive algorithms are asymptotically faster [13, 26, 27], handling realistic C features may have a drastic impact on their complexity. For example, without handling structures and unions, Steensgaard's algorithm has almost linear complexity [26]; when structures and unions are considered, the complexity becomes exponential [28].

2.1.6 Evaluation of interprocedural pointer analysis

Most earlier algorithms are evaluated by resource consumptions, pointer resolutions, and statistics of improved optimization opportunities [29, 30, 31, 32]. These numbers can provide a high-level judgment on the effectiveness about an algorithm, but the real value of interprocedural pointer analysis should be judged by how much performance improvement can be gained from its analysis results. The work by Wilson and Lam in [20] uses pointer analysis results to guide loop parallelizations for one SPEC benchmark, whereas the work by Diwan et al. [32] 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 [30] 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 thesis, 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, which will be described in Chapter 7. 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.

2.2 A New Modular Interprocedural Pointer Analysis Algorithm

In this dissertation, a fully functional interprocedural pointer analysis algorithm which can handle widely used C programs is proposed. The 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 [19]. In accommodating realistic C features, the new algorithm controls its memory requirement and maintains its accuracy in the presence of function pointers by starting with an under-estimated call graph which is augmented iteratively along the course of interprocedural pointer analysis [19]. The offset representation for structure/union fields is also incorporated into access paths to seamlessly handle aliases caused by unions and type casts [20, 33]. In addition, it is demonstrated that the number for recursive heap objects can be bound and the location of acyclic heap objects can be easily disambiguated using access paths.

The proposed algorithm consists of two major stages: the intraprocedural stage and the interprocedural stage. In 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, a set of pointer definitions involving pointers accessible across function boundaries, and a set of pointer assignments involving formal parameters and global variables. The third set is considered as the *summary transfer function* and represented by points-to relations. In the interprocedural stage, bottom-up propagation of summary transfer functions along the call graph is performed. In the presence of function pointers, it is also necessary 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-discovered indirect callee may define function pointers used elsewhere in the program, the bottom-up and top-down propagations need 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 the proposed algorithm in this dissertation is similar to *Relevant Context Inference* (*RCI*), proposed by Chatterjee et al. [12], since both algorithms use modular analysis to reduce the resource requirements. However, several significant differences distinguish the two pieces of work, as highlighted in Figure 2.2.

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

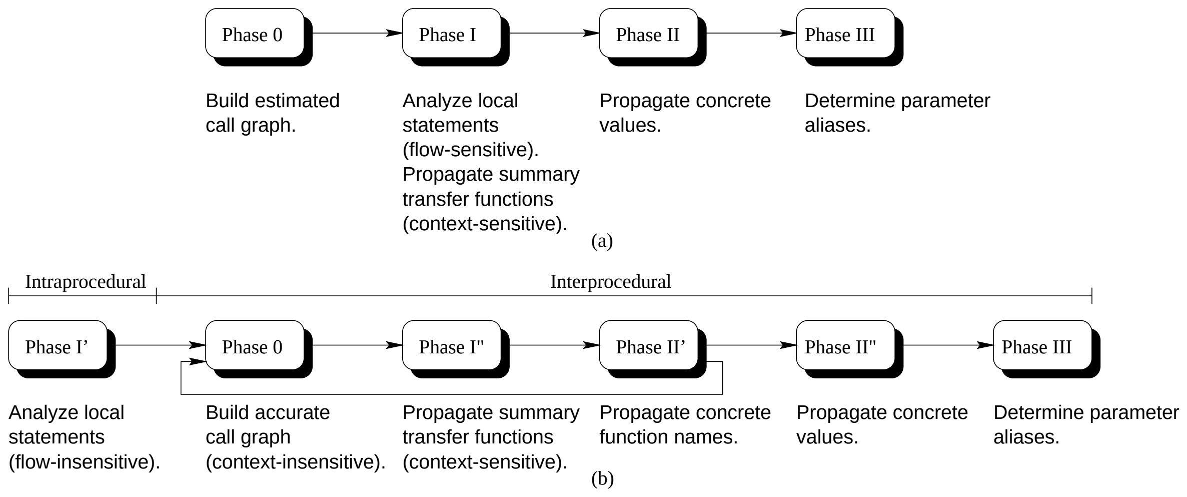

Phase 0
Phase I
Phase II
Phase III
Build estimated call graph.
Analyze local statements (flow-sensitive). Propagate summary transfer functions (context-sensitive).
Propagate concrete values.
Determine parameter aliases.
(a)
Intraprocedural
Interprocedural
Phase I'
Phase 0
Phase I''
Phase II'
Phase II''
Phase III
Analyze local statements (flow-insensitive).
Build accurate call graph (context-insensitive).
Propagate summary transfer functions (context-sensitive).
Propagate concrete function names.
Propagate concrete values.
Determine parameter aliases.
(b)

Figure 2.2 Modular interprocedural pointer analysis: (a) the *RCI* algorithm for a subset of C++, (b) the proposed new modular analysis algorithm for complete C.

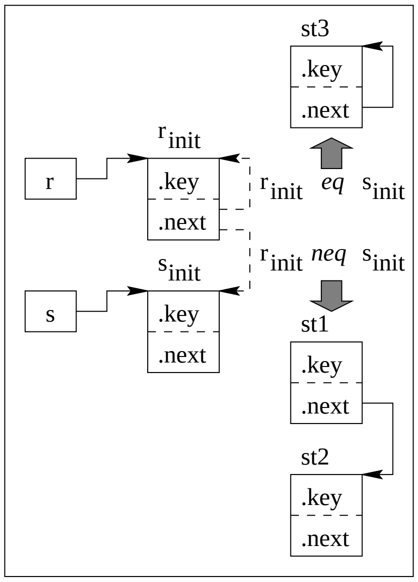
First, *RCI* covers a subset of C++ constructs where many general C features including explicit address operator (&), function pointers, structure members, and general pointer assignments are excluded. The proposed algorithm covers complete C features and can generate safe analysis results to guide code optimizations. Secondly, *RCI* uses an over-estimated call graph. For an indirect call-site, all functions with names taken and compatible signatures are considered as possible callees. In *132.ijpeg*, a SPECcint95 benchmark, there are 188 functions whose names are 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 will waste time in propagating summary transfer functions and produce spurious results. Lastly, *RCI* uses *alias contexts* to lazily enumerate potential aliases among parameters in order to distinguish different versions of transfer functions for different calling contexts. In the proposed method, access paths enable a context-independent representation of transfer functions so that the memory overhead used to represent multiple versions of summary transfer functions is reduced.

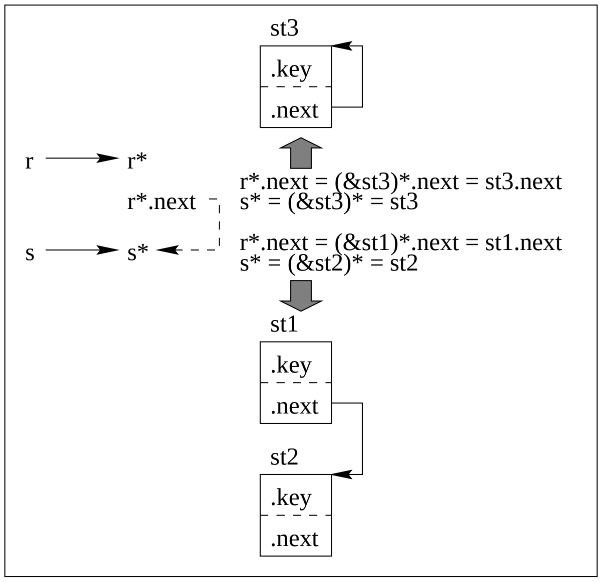
Consider the example shown in Figure 2.3a. When an estimated call graph is used, `*fn2` may invoke both `fn3` and `fn4` since both their names are taken and they have matching signatures. In reality, only `fn3` can be called through `*fn2`, so propagating the summary transfer function of `fn4` would be spurious and unnecessary. Figure 2.3b shows how summary transfer functions are represented for `fn3` in *RCI*. Nodes 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. 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 2.3c, by using postfix access paths [25], 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*`).

```
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);
}
```

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

(a)





(b) (c)

Figure 2.3 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.

Details of the operations performed in the intraprocedural and interprocedural stages will be explained in the next two chapters.

2.3 Related Work

In addition to previous discussions, more related work will be summarized in this section. Landi et al. [22] 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. [19] 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 the proposed work in this dissertation only generates *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. [34] 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 [35] perform pointer analysis for multithread programs.

Many pointer analysis algorithms use types to perform alias analysis [32, 36, 26, 37]. Deutsch [25] uses postfix symbolic access paths to analyze the alias relations among accesses to recursive structures on well-typed programs. In this dissertation, although the access path notation employed is also postfix, the low-level offsets are adopted 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 [27] has cubic worse-case running time, and the work from Fähndrich et al. [38] shows that Andersen's method can be practical for large programs. Shapiro et. al [31] propose a hybrid method with configurable complexity between Steensgaard's and Anderson's algorithms. Hasti et al. [39] propose the idea of iteratively improving the accuracy of pointer analysis in SSA form [18], but empirical results are not available. Zhang et al. [40] and Stocks et al. [41] balance the complexity by switching between various algorithms. The largest benchmark handled has around 60,000 lines of code. The algorithm proposed in this dissertation 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 shape descriptions for heap objects [42, 43, 44, 45, 46, 25]. The proposed algorithm accurately resolves heap objects of scalar types, but conservatively assumes that all heap objects of recursive data types form cyclic data structures.

CHAPTER 3

INTRAPROCEDURAL POINTER ANALYSIS

Pointer analysis is a data-flow analysis which deals with the flow of pointer values. Given two pointers `p1` and `p2`, they cannot point to the same location unless they are initialized by the same value, so the fundamental job in a pointer analysis algorithm is to analyze pointer assignments to calculate the flow of pointer values. Since the C language allows multi-level pointers, a correct pointer analysis algorithm needs to deal with aliases of pointers as well. Once the address of a pointer is passed to another function, like the example in Figure 2.1, pointer analysis needs to be conducted in the interprocedural scope otherwise some pointers cannot be resolved.

In this chapter, how to handle pointer assignments and determine aliases among pointers will be the main focus of discussions. For pointers that are defined through local statements, they can be fully resolved in the intraprocedural stage. For pointers that are defined through function calls, they are only identified in the intraprocedural stage and will be resolved in the interprocedural stage. These interprocedurally accessible pointers are represented in a context-independent format using access paths, based on which a summary behavior of each function is constructed, including MOD/REF information for interprocedurally accessible locations, callee information, and summary transfer functions. The summarized behavior is the only data structure to be maintained in the interprocedural stage so that significant reduction in analysis time and memory usage can be achieved.

3.1 Access Paths

For a C expression that accesses the memory, it can be as simple as a direct access to a variable, or an indirect access through a sequence of dereference operations with offset adjustments. Since different variables represent different memory locations, disambiguating accesses of the former case is simple. To disambiguate accesses of the latter case, both the sequence of dereference and offset operations and the contents of intermediate pointers are required.

An access path is a string recording the sequence of intermediate dereference and offset operations to meet the first requirement of memory disambiguation. Described in regular expressions, the grammar of an access path is $\mathbf{v}(\mathbf{fd}|\mathbf{d})^*(\mathbf{f}|\epsilon)$. The initial token, $\mathbf{v}$, is simply a variable name. Symbol $\mathbf{f}$ is of the form ”`.so_eo`” which denotes the starting and end byte-level offsets of a field in a structure/union, and symbol $\mathbf{d}$ denotes the dereference operation and is symbolically shown as ”`*`” elsewhere in this dissertation when there is no confusion with the closure symbol in regular expressions. Unless the contents of intermediate pointers in an access path are known, an access path is simply an encoded postfix string and is not bound with any particular locations, but simply represents how the denoted location is accessed.

Definition 1 (Construction of access paths) *AP denotes the function that recursively determines the postfix access path for a C expression that retrieves or deposits data from/to the memory:*

$$AP(v) = \mathbf{v} \tag{3.1}$$

$$AP(*exp) = \begin{cases} AP(exp) & \textit{if exp is a function name} \\ AP(exp)* & \textit{otherwise} \end{cases} \tag{3.2}$$

$$AP(exp[index]) = \begin{cases} AP(exp) & \textit{if exp is of an array type but} \\ & \textit{not a formal parameter} \\ AP(exp)* & \textit{otherwise} \end{cases} \tag{3.3}$$

$$AP(exp\ op\ exp1) = AP(exp) \tag{3.4}$$

$$AP(exp \rightarrow field) = AP(exp)*.\mathbf{so\_eo} \tag{3.5}$$

$$AP(exp.field) = AP(exp)*.\mathbf{so\_eo} \tag{3.6}$$

$$NM(\alpha.\mathbf{so1\_eo1}.\mathbf{so2\_eo2}) = \alpha.\mathbf{so3\_eo3} \tag{3.7}$$

$$\textit{where } \mathbf{so3} = \mathbf{so1} + \mathbf{so2} \textit{ and } \mathbf{eo3} = \mathbf{so1} + \mathbf{eo2} \ \diamondsuit$$

Instead of generating all possible access paths from any variables, access paths are generated lazily from observed C expressions in each function. Definition 1 shows the rules that recursively determine the access path of a C expression. Rule (3.1) is the terminal case which initiates the access path with the corresponding variable name. Rule (3.2) handles the dereference operation by appending a "*" symbol after the access path corresponding to the being dereferenced pointer expression unless the prefix string is a function name.

The C language grammar allows function names to be dereferenced, but at run-time the dereference does not occur, which is reflected in Rule(3.2). Similarly, Rule (3.3) handles the duality between array and pointer accesses. If the array expression truly has an array type and is not a formal parameter, no dereference is necessary since the expression accesses a constant location; otherwise a "`*`" symbol is appended. As shown by Rule (3.4), explicit pointer arithmetics are ignored by the *AP* function, resulting in a coarse but safe access path for linearly accessed locations. However, offsets added to pointers by the field operators, "`.`" and "`->`", are faithfully represented in access paths since they are always constants thus can be accurately determined.

Traditionally, fields in access paths are represented by their symbolic names [22, 34, 25]. Although symbolic names serve the purpose to differentiate individual fields in a structure, they do not provide adequate information regarding the aliases caused by unions and type casts [20, 33]. For example, given a nested structure field access `s1.s2.f1` where `f1` is the first field in structure `s2`, type casts alone without any pointer assignments can create an alternative expression, `((struct * S2) (&s1))->f1`, to access the same `f1` field. This is because the programmer can take advantage of the knowledge that the address of a structure is the same as the address of the first field of the structure. When symbolic names for fields are used, two different access paths result: `s1.s2.f1` and `s1.f1`, respectively. They can be explicitly marked as aliases with extra annotations, but offsets for structure fields provide a better solution. In Rules (3.5) and (3.6), the starting offset (`so`) and end offset (`eo`) of a field relative to the in-most enclosing structure are calculated and used to represent the field. Assuming the size of `s2` is 100 bytes and `f1` occupies 4 bytes in `s2`, the encoded access path of `s1.s2.f1` is `s1.0_99.0_3`, while the encoded access path of `((struct * S2) (&s1))->f1` is is `s1.0_3`. Then Rule (3.7) is defined so that back-to-back field offsets are coalesced into a single field by translating the relative offsets from the in-most enclosing structure to the out-most one. The normalization simply proceeds by adding the starting offset of the enclosing structure to the starting and end offsets of the enclosed field. For example, the normalized access path of `s1.0_99.0_3` is `s1.0_3`, since $0+0 = 0$ and $0+3 = 3$. As a result, two aliased expressions have a common access path based on normalized field offsets. Without explicit pointer assignments, access paths using byte offsets can resolve aliases caused by type casts and pointer arithmetics. The next section discusses how to resolve aliases in the presence of arbitrary pointer assignments.

3.2 Right-Most Access Paths and Pointer Assignments

The points-to relation proposed by Emami et al. [19] is also adopted in this dissertation 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 dissertation, 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 (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.* $\diamondsuit$

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, 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 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 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 3 (Finding the right-most forms of an access path) *Under a set of points-to relations S_{PTR}, the evaluation function, EVAL, recursively parses an access path and returns the entire set of right-most access paths as defined below:*

$$EVAL(v \mid (\&v)*, S_{PTR}) = \{v\} \tag{3.8}$$

$$EVAL(\alpha*, S_{PTR}) = \{\beta \mid \exists\theta \in EVAL(\alpha, S_{PTR}), (\theta, \beta) \in S_{PTR}\} \tag{3.9}$$

$$EVAL(\alpha f, S_{PTR}) = \{\gamma \mid \exists\theta \in EVAL(\alpha, S_{PTR}), \gamma = NM(\theta f)\} \tag{3.10}$$

$$EVAL(\alpha*, S_{PTR}) = \{\theta* \mid \exists\theta \in EVAL(\alpha, S_{PTR}), (\theta, \beta) \notin S_{PTR}\} \tag{3.11}$$

$$EVAL(\alpha*, S_{PTR}) = \{\delta \mid \exists\theta \in EVAL(\alpha, S_{PTR}), \delta_{n\leq k}\lambda = \theta\} \diamondsuit \tag{3.12}$$

Definition 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}.* $\diamondsuit$

$EVAL$ is a closure function which takes an access path and a set of points-to relations and then returns the set of right-most aliases of the input path. Rule (3.8) handles the trivial case where no dereference is encountered, or the dereference is simply canceled by address operator (&). Since the trivial case accesses a definitive memory location, the right-most access path is simply the variable itself. For an access path containing a sequence of dereference and offset operations, the path is processed by parsing the composing operations, or *tokens*, from left to right, reflecting the actual sequence of memory dereferences that would occur at run-time. Rule (3.9) addresses the case where the next token is a dereference operation and the pointer access path has outgoing points-to relations, meaning that the pointer is initialized. Since the tokens are parsed from left to right, the evaluation result of α is already available before $\alpha *$ is evaluated. Then for each access path θ found in $EVAL(\alpha, S_{ptr})$, the targets of θ's points-to relations are the right-most aliases of $\alpha *$. Rule (3.10) handles the offset token by concatenating the offset to each path found in the evaluation result of the prefix path. Back-to-back fields also need to be normalized. This whole process can be considered as following the fan-out tree of a set of points-to relations where the root is the initial variable in the access path.

```
     S st1, st2, st3;
     S *sp1, *sp2, *sp3, *sp4;      LHS     *RHS     points-to relation
                                    ----------------------------------
S1:  sp1 = &st1;                    sp1      st1  => (sp1, st1)
S2:  sp2 = &st2;                    sp2      st2  => (sp2, st2)
S3:  sp1->next = sp2;               sp1*.4_7 sp2* => (st1.4_7, st2)
S4:  sp3 = &st3;                    sp3      st3  => (sp3, st3)
S5:  sp4 = &st3;                    sp4      st3  => (sp4, st3)
S6:  sp3->next = sp4;               sp3*.4_7 sp4* => (st3.4_7, st3)

                  (a)                                  (b)
```

Figure 3.1 Constructing points-to relations from pointer assignments: (a) code, (b) corresponding points-to relations for each assignment.

Rules (3.11) and (3.12) will be ignored for now and the example in Figure 3.1 will be considered first. The code in Figure 3.1a is derived from the inlined version of function `main` in Figure 2.3a. The left part of Figure 3.1b lists the encoded access paths for `lhs` and `*rhs` before *EVAL* is invoked. The right part of Figure 3.1b lists the added points-to relations for each statement based on the right-most access paths. For statements `S1` and `S2`, the encoded access paths from C expressions have no dereference operators so only Rule (3.8) is applied when evaluating the access paths. For statement `S3`, the right-most access path of the LHS expression is `st1.next`, or `st1.4_7` in byte offsets, which can be discovered in three steps:

$$\begin{aligned}
1 : EVAL(sp1, \{(sp1, st1), (sp2, st2)\}) &= \{sp1\} \\
2 : EVAL(sp1*, \{(sp1, st1), (sp2, st2)\}) &= \{st1\} \\
3 : EVAL(sp1*.4\_7, \{(sp1, st1), (sp2, st2)\}) &= \{st1.4\_7\}
\end{aligned}$$

Similarly, the right-most access path of the memory location pointed to by the RHS expression is st2, which can be discovered in two steps:

$$\begin{aligned} 1: EVAL(sp2, \{(sp1, st1), (sp2, st2)\}) &= \{sp2\} \\ 2: EVAL(sp2*, \{(sp1, st1), (sp2, st2)\}) &= \{st2\} \end{aligned}$$

As a result, the points-to relation added for statement S3 is (st1.4_7, st2). The points-to relations added for statements S4, S5, and S6 can be derived in the same manner.

Rules (3.8) through (3.10) in Definition 3 guarantee that all initialized pointers must point to at least one right-most access path after being evaluated. Rules (3.11) generates temporary right-most access paths for indirectly accessed locations from uninitialized pointers. This situation can arise when parameters and global variables are assumed as uninitialized in the intraprocedural stage of the modular interprocedural analysis algorithm and the function body contains expressions which dereference from these pointers. The straightforward way, as denoted by Rule (3.11), is to append a "*" symbol after the right-most access paths of the dereferenced pointer. For indirect accesses using pointers of non-recursive types, there are a finite number of locations that can be reached from the pointer, so the total number of access paths that can be generated from the pointer is finite. However, when uninitialized pointers to recursive data structures are involved, infinite access paths may be produced by Rule (3.11).

(sp1, sp1*)	(sp1, sp1*)	(sp1, sp1*)
(sp1*.4_7, sp1*.4_7*)	(sp1*.4_7, sp1*.4_7*)	(sp1*.4_7, sp1*)
(sp1, sp1*.4_7*)	(sp1, sp1*.4_7*)	
	(sp1*.4_7*.4_7, sp1*.4_7*.4_7*)	
	(sp1, sp1*.4_7*.4_7*)	
(a)	(b)	(c)

Figure 3.2 Handling access paths involving recursive data types: (a) results of the first iteration, (b) results of the second iteration, (c) results controlled by k=1.

Consider the common pointer-chasing statement `sp1 = sp1->next` in a linked-list traversal loop where `sp1` is a formal parameter. It will be shown later that pointer assignments need to be analyzed iteratively otherwise the resolved points-to relations are not complete. When the statement is analyzed for the first time, the resultant points-to relations are shown in Figure 3.2a. Path `sp1*` is created when evaluating the prefix path `sp1*` in the RHS path `sp1*.4_7*` according to Rule (3.11). The field operator `.4_7` is then appended to `sp1*`, then `sp1*.4_7*` is produced since `sp1*.4_7` is also an uninitialized pointer.

In the second iteration, `sp1` will be found to point to `sp1*` and `sp1*.4_7*`. Therefore when evaluating the RHS access path `sp1*.4_7*`, the evaluation result of the prefix path `sp1*` is {`sp1*`, `sp1*.4_7*`}. So when `sp1*.4_7` is evaluated, path `sp1*.4_7*.4_7` is added, then when `sp1*.4_7*` is evaluated, path `sp1*.4_7*.4_7*` is added. When iteration 2 finishes, as shown in Figure 3.2b, a new points-to relation (`sp1`, `sp1*.4_7*.4_7*`) will be found in the points-to relation set. Inductively, the analysis will iterate forever and after the *nth* iteration, (`sp1`, `sp1*.(4_7*)`^{n}) is produced.

To address this problem, a *recursion-sensitive* parameter k is introduced which differentiates the first k objects in a linked list accessed from an uninitialized pointer. As defined in Rule (3.12), if no more than k prefix paths of path θ which corresponds to an uninitialized pointer has the same recursive data type as the intended target access path, a "*" symbol is appended after the pointer path like the case of Rule (3.11). Otherwise, the longest prefix path with the same recursive data type is reused as the right-most access path of the pointer's target. The implication is that after the *kth* instance of recursive objects in a linked list, a cycle is always assumed to exist and all later instances of recursive objects are collectively represented by the *kth* object. This is similar to the *k-limiting* approach used in [22], but only the lengths of access paths involving recursive data types are controlled. Figure 3.2c shows the limited representation of recursive access paths where k is set to 1. When evaluating `sp1*.4_7*`, the types of `sp1*` and `sp1*.4_7*` are both `S`. Since `sp1*` is a prefix path of `sp1*.4_7*`, points-to relation (`sp1*.4_7`, `sp1*`) instead of (`sp1*.4_7`, `sp1*.4_7*`) is generated. Because C is not a strong-typed language, each expression may have more than one type due to type casts, implying each access path may have more than one type as well. However, the total number of types is still bound in a program, and a prefix can subsume a suffix path as long as they have partial overlaps in associated types. Section 3.4 will discuss the upper bound of the number of access paths that can be generated from an uninitialized pointer.

3.3 Algorithm of Intraprocedural Pointer Analysis

Based on the previously illustrated definitions of access paths and path-based points-to relations, there are two major tasks performed in the intraprocedural stage: *code transformation* and *pointer analysis*. Their rationale and pseudo codes are given below with a detailed example.

3.3.1 Code transformation

When a function is analyzed as an isolated compilation unit, its formal parameters are assumed to have unknown values, so indirect accesses from these pointers will be represented by access paths initiating from formal parameters. To determine the actual location denoted by the path in a caller function, it can be determined by replacing the formal parameter with the corresponding actual parameter and following the points-to relations found in the caller as briefly discussed in Figure 2.3c. However, what is not shown in the simple example is the mapping between formal and actual parameter pairs, since formal parameters can be named arbitrarily by the programmer, and actual parameters can be arbitrarily complex C expressions. Instead of grouping each formal-actual parameter pair explicitly, they can be easily identified through systematically designed *interface variables*.

There are four categories of interface variables: *formal interface variables*, *actual interface variables*, *outgoing return variables*, and *incoming return variables*. The templates of these interface variables are explained below.

- f_i_foo: the ith formal parameter of function foo.
- a_i_bar_foo_n: the ith actual parameter passed to function foo from function bar at the the nth call-site.
- o_foo: the outgoing return value of function foo.
- i_bar_foo_n: the incoming return value from callee bar in function foo of the nth call-site.

The initial field of each interface variable is designed to distinguish its category. The other fields in interface variables have their special meanings to guarantee the uniqueness of an interface variable in the same function or across function boundaries:

- i: distinguishes individual parameters in a parameter list.
- foo: distinguishes parameters in different functions.
- bar: distinguishes actual parameters passed to to different callees.
- n: distinguishes multiple call-sites to the same callee in the same function.

Figure 3.3 lists the pseudo code of the placement of interface variables. The idea is that all right-most access paths that would stem from the original formal parameters and incoming function return values are now represented by paths that stem from interface variable. This can be guaranteed by Definition 3 and the algorithm listed in Figure 3.3 since formal interface variables and incoming return variables appear as the right-most RHS expressions in each function. In addition, all paths that are accessible from complex

```
Add_Interface_Variable($foo$)
{
1:  $n = 0$;
2:  FOR (each formal parameter $fp$) {
3:      Let $i$ be the rank of formal parameter $fp$;
4:      Add statement "$fp = \mathrm{f\_}i\_foo$" to $foo$;
5:  }
6:  FOR (each call-site $bar()$) {
7:      $n = n+1$;
8:      FOR (each actual parameter $ap$) {
9:          Let $i$ be the rank of actual parameter $ap$;
10:         Let $bar_{path} = \mathrm{AP}(bar)$;
11:         Add statement "$\mathrm{a\_}i\_bar_{path}\_foo\_n = ap$" to $foo$;
12:     }
13:     IF (statement "$rec = bar()$" $\in foo$) {
14:         Add "$rec = \mathrm{i\_}bar_{path}\_foo\_n$" to $foo$;
15:     }
16: }
17: FOR (each return statement) {
18:     Assume the return expression is $ret$;
19:     Add "$\mathrm{o\_}foo = ret$" to $foo$;
20: }
}
```

Figure 3.3 Algorithm of inserting interface variables.

actual parameters and outgoing return values can be easily grasped from their interface counterparts due to their associated points-to relations created by the interface assignments. Line 0 initializes the call-site counter which is incremented by 1 for every call-site at line 7. Lines 2 through 5 insert interface formal variables and assign them to original formal parameters, while lines 13 to 15 handle callee return values according to similar rules. The field bar_{path} means the access path of the function call expression, which is either simply a function name for a direct call, or the directly encoded access path of the indirect call expression. Lines 8 to 12 insert interface actual parameters which are assigned by original actual parameters, where line 17 to 20 insert interface outgoing return values. From these interface variables, the targets of actual parameters and return values can be easily identified.

3.3.2 Pointer analysis

Pointer assignments in each function are analyzed by their lexical order to calculate the points-to relations. The analysis is conducted iteratively until no new points-to relations are created. Although the analysis is flow-insensitive, meaning that a later assignment will not kill an earlier definition anyway, analyzing the function following the lexical order can reduce the total number of iterations since right-most access paths will emerge earlier.

Figure 3.4 shows the intraprocedural pointer analysis algorithm after interface variables have been inserted. For assignments involving a whole structure, points-to analysis is performed for each individual pointer field, as shown in lines 7 to 12. When the algorithm terminates, all pointer assignments made to indirectly accessed memory locations through formal parameters will result in points-to relations involving access paths with formal interface variables, and memory locations that can be accessed by callee functions can be found by following the points-to relations from actual interface variables. Details about how to process the intraprocedural pointer analysis results, together with additional information derived in the intraprocedural stage, will be discussed in Chapter 3.

```
Intraprocedural_Pointer_Analysis($fn$)
{
1:  $S_{PTR}(fn) = \emptyset$;
2:  **DO** {
3:      **FOR** (each pointer assignment "$lhs = rhs$" $\in fn$) {
4:          Let $APL$ = EVAL(AP($lhs$)), $APR$ = EVAL(AP($*rhs$));
5:          $S_{PTR} = S_{PTR} \cup \{(\alpha, \beta) \mid \alpha \in APL \textit{ and } \beta \in APR\}$;
6:      }
7:      **FOR** (each structure/union assignment "$lhs = rhs$" $\in fn$) {
8:          **FOR** (each pointer field $f$ in the structure/union) {
9:              Let $APL$ = EVAL(AP($lhs.f$)), $APR$ = EVAL(AP($*rhs.f$));
10:             $S_{PTR} = S_{PTR} \cup \{(\alpha, \beta) \mid \alpha \in APL \textit{ and } \beta \in APR\}$;
11:         }
12:     }
13: } **WHILE** (new access paths or points-to relations are added)
}
```

Figure 3.4 Algorithm of the intraprocedural pointer analysis stage.

3.3.3 Example

The example in Figure 2.3 is used here to explain the insertion of interface variables and the results of intraprocedural pointer analysis. Shown in Figure 3.5, statements with labels `i`n are extra *interface statements* to enable interface variables to participate in the generation of right-most access paths and points-to relations. Statements `i4` and `i5` are worth mentioning here since the actual parameters are passed to an indirect callee. Before the indirect call-site is resolved, the callee name is simply the encoded access path of the call expression, which is `fn2*` in this case. With these interface variables in place, Figure 3.6 shows the corresponding intraprocedural pointer analysis results of the code example. The corresponding source statement labels are put along with each

```
main()
{
     void (*fn2)(S*, S*);
     S st1, st2, st3;

i1:  a_1_fn1_main_1 = &fn2;
i2:  a_2_fn1_main_1 = fn3;
i3:  a_3_fn1_main_1 = fn4;

S1:  fn1(&fn2, fn3, fn4);

i4:  a_1_fn2*_main_2 = &st1;
i5:  a_2_fn2*_main_2 = &st2;

S2:  (*fn2)(&st1, &st2);

i6:  a_1_fn2*_main_3 = &st3;
i7:  a_2_fn2*_main_3 = &st3;

S3:  (*fn2)(&st3, &st3);
}
```

```
fn1((**fa)(S*, S*),
    (*fb)(S*, S*), (*fc)(S*, S*))
{
     (**temp)(S*, S*);

i8:  fa = f_1_fn1;
i9:  fb = f_2_fn1;
i10: fc = f_3_fn1;

S4:  temp = fa;
S5:  *temp = fb;
}
fn3(S *r, S *s)
{
i11: r = f_1_fn3;
i12: s = f_2_fn3;

S6:  r->next = s;
}
fn4(S *r, S *s)
{
i13: r = f_1_fn4;
i14: s = f_2_fn4;

S7:  s->next = r;
}
```

Figure 3.5 Code example with interface variables.

points-to relation for reference. For example, the side-effect of statement S6 of function fn3 is represented by points-to relation (f_1_fn3*.4_7, f_2_fn3*), which can be clearly interpreted as the second word field of the indirectly accessed structure object from the first parameter will point to whatever location pointed to by the second parameter.

```
## main ##
i1:  (a_1_fn1_main_1, fn2)
i2:  (a_2_fn1_main_1, fn3)
i3:  (a_3_fn1_main_1, fn4)
i4:  (a_1_fn2*_main_2, st1)
i5:  (a_2_fn2*_main_2, st2)
i6:  (a_1_fn2*_main_3, st3)
i7:  (a_2_fn2*_main_3, st3)
```

```
## fn1 ##
i8:  (fa, f_1_fn1*)
i9:  (fb, f_2_fn1*)
i10: (fc, f_3_fn1*)
S4:  (temp, f_1_fn1*)
S5:  (f_1_fn1*, f_2_fn1*)

## fn3 ##
i11: (r, f_1_fn3*)
i12: (s, f_2_fn3*)
S6:  (f_1_fn3*.4_7, f_2_fn3*)

## fn4 ##
i13: (r, f_1_fn4*)
i14: (s, f_2_fn4*)
S7:  (f_2_fn4*.4_7, f_1_fn4*)
```

Figure 3.6 Results of intraprocedural pointer analysis.

3.4 Complexity Analysis

In this section the complexity of the intraprocedural pointer analysis is discussed. The complexity is measured based on a set of parameters. V_f is the sum of variables and call-sites in a single function, including interface variables. T is the number of data types declared in the program, where k is the recursive-sensitive parameter which controls 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 including interface assignments, and M is the number expressions that result in memory accesses in a function.

First it is demonstrated that the number of access paths stemmed from a variable is bound by $O(f^{kT})$. Since a prefix access path can be casted into a structure type, any prefix path can be widened by field tokens, but the length of each access 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 $O(V_f f^{kT})$. The space complexity to store points-to relations in the intraprocedural phase for a single function is therefore $O(V_f^2 f^{2kT})$. The algorithm in Figure 3.4 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 to find its right-most forms using Definition 1 takes $O(V_f^l f^{lkT})$ steps. Since the 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 $O(S(M+S)V_f^l f^{lkT})$.

CHAPTER 4

INTERPROCEDURAL POINTER ANALYSIS

When the iterative intraprocedural pointer analysis finishes, pointers that are defined by local assignments can be resolved. However, some information is still missing, including the concrete values passed to formal parameters and global variables, and the contents of pointers that are modified by pointer assignments in invoked functions. These missing parts will be analyzed in the interprocedural stage.

The tasks performed in the interprocedural phase are carefully staged to reduce the memory and analysis time requirements. First, a summary behavior for each function is extracted. A summary behavior is a subset of function-level activities that can interact with activities in other functions. For example, memory accesses to local variables whose addresses are never taken need not to be analyzed in the interprocedural stage since the scope of their lifetime is strictly limited within the function. The summary behavior extraction is actually conducted at the end of the intraprocedural stage, but it is more appropriate to be discussed in the context of interprocedural pointer analysis. Then, the core of the interprocedural pointer analysis is entered, including three iterative phases followed by two acyclic phases. Operations conducted in the iterative phases involve the construction of the call graph, and the propagation of summary transfer functions and concrete function-type parameter values along the call graph. Then operations conducted in the acyclic phases are much simpler since the major job is to determine the aliases among parameters. Details about the summary behavior extraction will be presented first, followed by the explanation of individual phases in the interprocedural pointer analysis stage.

4.1 Summary Behavior Extraction

A function in a C program often starts with a list of parameters, followed by a set of local variable declarations, and a set of statements that perform computations. Among these many computation activities, only the following types of information need to be maintained in the interprocedural stage:

1. **Caller-allocated locations.** Memory locations that are allocated by the caller can be accessed by the callee if their addresses are passed through formal parameters or global variables. Identifying these locations is critical to guarantee the correctness of load/store optimizations performed for the callee function body. If a certain combination of concrete values passed from one calling context causes two accesses to be aliases, unless function cloning is performed, the alias relation should be respected by the optimizer for all calling contexts. Whether a C expression will access the memory or not can be determined by the rules listed in Table 4.1. Among many C expressions, only five forms of non-array type expressions can access the memory: direct accesses through variable names, field accesses through structure/union names/addresses, indirect accesses through pointers, and indirect

accesses through base addresses and indexes. However, a memory expression's parent expression may decide whether to bypass the memory access or not since sometimes the memory content is of no interests to the computation. For example, expression `i` access the content of variable `i`, but expression `&i` does not access the memory since all it needs is the address of `i`, which is irrelevant with `i`'s content. Similarly, expression `(sizeof i)` does not access the memory either since the result of the expression is determined by its type. In fact, sometimes it is the grand parent expression to determine the existence of memory accesses. For example, expression `((int) i)` accesses the content of `i`, but `&((int) i)` does not, even though type cast `(int)` is the immediate parent expression in both cases. It is because some C operators only have syntax significance instead semantic significance, and only semantic-significant parent operators determine whether the memory needs to be accessed or not. In C, type casts and parentheses only have syntax significance, so they are not considered as significant parents in determining memory accesses. As shown in the top row of Table 4.1, parent operators like `&`, `.`, and `sizeof` do not care about the content of the child expression, so the implied memory access is not performed. On the other hand, semantic-significant parent operators like `->`, `*`, `[ ]`, and other unary/binary operators need the content of the child expression for computation or memory dereference, so the implied memory access in the child expression is performed [1].

[1]The $\emptyset$ operator means that the child expression is the top-level expression.

Table 4.1 Rules of identifying expressions causing memory accesses.

		Semantic-Significant Parent Operators	
		Content Irrelevant	Content Relevant
		`&`, `.`, `sizeof`	`∅`, `->`, `*`, `[ ]`, `other unary/binary operators`
Non-Array Type	`var` `.` `->` `*` `[ ]`	NO	YES

Before register allocation, each identified memory access will have a corresponding load or store instruction in the low-level code, and whether such a memory instruction accesses locations that are also accessible by the caller or not can be easily determined by checking the right-most access paths found by the *EVAL* function for the directly encoded access path: if path elements starting from formal interface parameters or global variables are found in the evaluation result, they are caller-allocated locations. Each such access path then will be annotated with the MOD/REF attributes inherited from the C expression. These attributes will be analyzed by the optimizer when memory instructions are scheduled across jump-subroutine-call (`jsr`) instructions. More optimizations details will be given in Chapter 5.

2. **Summary transfer function.** Since expressions in the callee function may access caller-allocated locations through pointer-type parameters or global variables, the caller function needs to be aware of any new modifications made to these locations

by assignments in the immediate or descendant callees. A summary transfer function collectively represents the side-effects of these assignments, and in the context of interprocedural pointer analysis, a summary transfer function is represented by a set of points-to relations.

Assignments to caller-allocated pointers by local assignments in the callee will be explicitly represented by points-to relations whose pointer and target paths both originate from formal interface parameters or global variables. It is because the *EVAL* function is defined to find the source variable of a chain of pointer definitions and truthfully append a sequence of dereference and offset tokens after the source variable. In addition, the invoked function may allocate heap objects for the caller to use. Since these heap-objects may be of pointer types and may be initialized by statements in the callee, they should also be considered as parts of the summary transfer function of the callee. For these objects to be used by the caller, their addresses must be assigned to formal parameters or global variables. However, since formal parameters and global variables are assigned with heap-objects' addresses, these pointers are not uninitialized any more, meaning accesses to these heap objects will not be normalized to right-most access paths stemming from interface formal parameters and global variables. Later in this chapter the procedures of how to identify callee-allocated heap objects and how to name those will be explained. Once these heap objects are identified, their associated points-to relations are also included in the summary transfer functions.

3. **List of invoked functions.** Points-to relations extracted by the above two aspects only represent the side-effects of local statements but not the side-effects of further invoked function calls. These single-level summary transfer functions will be propagated along the call graph in later phases of the interprocedural pointer analysis stage so that the targets of pointers initialized across multiple-level function calls will be contained in the immediate callee functions' summary transfer function. To facilitate the call-graph construction algorithm which will be described later, each function's summary behavior includes a list of callee function names. In the presence of indirect calls, the directly encoded access path of the indirect call-site is temporarily considered as the callee name. If the $EVAL$ result of an indirect call-site path contains right-most access paths starting from formal interface parameters or global variables, they are also kept in the summary behavior since they contain information about how to resolve this function pointer across function boundaries.

4. **Assignments involving uninitialized pointers.** When the $EVAL$ function is invoked, intermediate pointers in the input access path are processed to identify their targets. When an uninitialized pointer is reached, all remaining dereference and offset tokens in the input path will be transferred and appended after the access path denoting the uninitialized pointer. Therefore the temporary right-most access paths from uninitialized pointers still contain enough information indicating how many more levels of dereferences and offset adjustments to apply once the pointer is resolved. In the interprocedural pointer analysis stage, uninitialized pointers

may be resolved through pointers passed down from callers or after applying the transfer functions of the callees. So the problems involved here are two folds: how to determine pointers that could be defined interprocedurally and how to re-run the *EVAL* function to generate the up-to-date right-most access paths, and therefore to generate the up-to-date points-to relations.

To address the first problem, an access path involves pointers that could potentially receive new definitions in the interprocedural stage if the access path initiates from a formal interface parameter or a global variable, or the path is accessible through a depth-first-search (DFS) from an actual interface parameter or a global variable. To address the second problem, any existing access path involving a prefix path which qualifies as a pointer that could be potentially defined interprocedurally is processed by the *EVAL* function again and the evaluation result will include all new right-most access paths. These new right-most access paths will inherit the same MOD/REF and type information from the input access path. Then if a points-to relation's pointer path or target path involves a prefix path which qualifies as a pointer that could be potentially defined interprocedurally by the aforementioned rule, both the pointer and target paths in the original points-to relation are evaluated. The cross-product points-to relations added between the evaluation results of the pointer path and the target path will accommodate all new points-to relations.

As an example, Figure 4.1 shows the summary behavior of functions listed in Figure 3.5. The MOD/REF sections exclude direct accesses to interface variables since they

```
## main ##
REF:
  fn2
  fn3
  fn4
  fn2*

POINTS_TO:
  (a_1_fn1_main_1, fn2)
  (a_2_fn1_main_1, fn3)
  (a_3_fn1_main_1, fn4)
  (a_1_fn2*_main_2, st1)
  (a_2_fn2*_main_2, st2)
  (a_1_fn2*_main_3, st3)
  (a_2_fn2*_main_3, st3)

CALL_LIST:
  fn1, id = 1
  fn2*, id = 2
  fn2*, id = 3
```

```
## fn1 ##
MOD:
  f_1_fn1*

POINTS_TO:
  (f_1_fn1*, f_2_fn1*)

## fn3 ##
MOD:
  f_1_fn3*.4_7

POINTS_TO:
  (f_1_fn3*.4_7, f_2_fn3*)

## fn4 ##
MOD:
  f_2_fn4*.4_7

POINTS_TO:
  (f_2_fn4*.4_7, f_1_fn4*)
```

Figure 4.1 Example of function summary behavior.

are inserted only for analysis purpose and real code will not be generated for them. Similarly, direct accesses to local variables whose addresses are never passed to other functions need not be included in the summary behavior either. Some points-to relations shown in Figure 3.6 are not included in the summary behavior if the pointer path represents a local variable or a formal parameter which cannot be accessed interprocedurally. For example, points-to relation (`temp`, `f_1_fn1*`) of function `fn1` is not included in its summary behavior since `temp`'s address is not taken. However, points-to relation (`f_1_fn1*`, `f_2_fn1*`) is included since both the pointer and target paths represent caller-allocated locations. It is only the summary behavior instead of the whole function body to be maintained in the interprocedural stage. This greatly improves the memory requirement of the algorithm, and it will be shown later how context-sensitivity for transfer functions are maintained.

4.2 Algorithm of Interprocedural Pointer Analysis

The interprocedural stage of the proposed pointer analysis algorithm has three iterative phases followed by two acyclic phases as shown in Figure 2.2b. The fundamental tasks performed in these phases are similar to *RCI* [12], so the same major phase numbers are used. On the other hand, significant differences in operation details also exist, so minor annotations are added to the major phase numbers.

4.2.1 Phase 0: call graph construction

Starting from the callee list in the summary behavior of function `main`, the call graph can be iteratively constructed performing a DFS. If the program has no indirect function calls, the complete call graph can be constructed in the first invocation of phase 0 analysis. Otherwise, as opposed to approaches which over-estimate the call graph based on function signatures, functions invoked through indirect call-sites are temporarily excluded from the call graph. Through studies over larger programs like the SPEC benchmarks, many indirectly invoked functions share the same function signatures but are called from different call-sites, therefore the estimated call graph will be too large which both affects the accuracy and lengthens the analysis time of the interprocedural stage.

In the proposed mechanism, each unresolved indirect call-site is represented by the encoded access path of the indirect call expression. Through the points-to relations discovered from the side-effects of callees or from the concrete function names passed via formal parameters, an indirect call-site can be resolved by evaluating the temporary right-most access paths of the corresponding call-site. For example, statement `S2` in Figure 2.3 is an indirect call-site. Since function pointer `fn2` is initialized through function calls, analysis conducted in the intraprocedural stage cannot resolve it. Therefore, in the summary behavior of `main`, the call-site is temporarily represented as `fn2*`, which is the corresponding access path of C expression `*fn2`. After points-relation `(fn2, fn3)` is propagated to `main`, which will happen in phase I", evaluating access path `fn2*` in the phase 0 analysis of the next iteration will reveal `fn3` as the callee, since EVAL(`fn2*`, {(`fn2`, `fn3`), ...}) = `{fn3}`. Although not shown in this simple example, an indirect call-site may be resolved to have multiple possible callees. Since the proposed interprocedural pointer analysis algorithm is flow-insensitive in terms of the side-effects of local assignments and the transfer functions of callees, and is context-insensitive in terms of parameter aliases, these multiple callees will not be further differentiated.

The constructed call graph, which could be incomplete in the middle of the iterative process, is partitioned into strongly connected components and viewed as a directed acyclic graph (*SCC-DAG*) [24]. That is, the SCC node containing function `main` is considered as the root of the SCC-DAG, and functions in a recursive chain are grouped as a single SCC node. Then these SCCs are sorted by a bottom-up and a top-down topological order with respect to the root node, the summary behavior of function `main`.

As will be explained in the next two phases, abiding by these orders can shorten the number of iterations of the analysis.

4.2.2 Phase I": propagation of summary functions and callee-allocated heap objects

The problems to be dealt with in the phase I" analysis include what to propagate, where to propagate, and when to propagate. As briefly mentioned before, caller-allocated locations and newly allocated heap objects need to be propagated. The propagation is simply conducted by replacing the formal parameter variable in an access path with corresponding actual parameters in the calling contexts, resulting a context-sensitive representation of summary transfer functions. And the propagation is conducted following a bottom-up topological order of the SCC-DAG since the caller's transfer function should include summary transfer functions of all invoked callees, including callees invoked through more than one level of function calls. Given h as the maximum height of the SCC-DAG, propagating summary transfer functions in a top-down order may require h iterations of the analysis, while only one iteration is required in a bottom-up order. After a function receives propagated points-to relations from the summary transfer functions of all callees, existing points-to relations involving unknown variables are evaluated to augment the summary behavior with new right-most access paths and points-to relations. Then the information contained in the summary behavior is ready to be propagated to further callers.

Let $(r\alpha,\ s\beta)$ be a points-to relation in function `fn`'s summary behavior where r, s are formal interface parameters or global variables, and α, β are two suffix access paths. If $s\beta$ is not symbolically represented as $r\alpha*$, meaning that $r\alpha$ is not an uninitialized pointer, this points-to relation obviously should be considered as part of the summary transfer function of `fn` since it reflects a pointer assignment which can affect a caller-accessible pointer. Since a pointer field in a structure object may point to another field in the same structure object, r and s may be the same variable.

Definition 5 (Propagation of points-to relations) *Let $(r\alpha,\ s\beta)$ be a points-to relation in function fn's summary transfer function where r, s are formal interface 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.* $\diamondsuit$

As shown in Definition 5, the first step of propagating such a points-to relation from the callee to the caller is to identify what locations are denoted by the pointer access path and the target access path in the caller, respectively. Access path $r\alpha$ simply states that from variable r, the final memory location is accessed via a sequence of dereference and offset adjustment operations denoted by α. Since the dereference and offset adjustment operations are context-independent, they are applicable to all calling contexts. As long as the formal parameter in the access path is replaced by the corresponding actual parameter and the transformed access path is evaluated in the caller's context of points-to relations, the denoted locations by the access path can be discovered in a context-sensitive manner. If the caller passes the address of a local variable to the callee, the local variable's name will appear as a right-most access path in the evaluation result. If the caller passes a pointer value received from formal parameters or global variables further down to the callee, the evaluation result will convey the dereference and offset tokens to the source formal parameters or global variables, meaning the caller's summary transfer function is augmented to accommodate the callee's summary transfer function 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.

For example, to propagate points-to relation (`f_1_fn1*`, `f_2_fn1*`) from `fn1` to `main` in Figure 2.3, `f_1_fn1` is first replaced by `a_1_fn1_main_1`. Then the evaluation result of EVAL(`a_1_fn1_main_1*`, $S_{PTR}(main)$) = {fn2} can be found, since points-to relation (`a_1_fn1_main_1`, `fn2`) is added for function `main` due to interface statement `i1`. Similarly, `f_2_fn1` is replaced with `a_2_fn1_main_1` and the evaluation result of EVAL(`a_2_fn1_main_1*`, $S_{PTR}(main)$) = `{fn3}` can be obtained, since points-to relation (`a_2_fn1_main_1`, `fn3`) is added for function `main` due to interface statement `i2`. So the points-to relation propagated to `main` from `fn1` is (`fn2`, `fn3`).

The next interesting question is how dynamically allocated objects are handled by access paths and their existences are propagated across function boundaries. Instead of generating pseudo variable names using synthesized call-site paths, heap objects are named by access paths which are differentiated by interface variables and suffix dereference and offset tokens. If the heap objects are allocated through calling `malloc` directly via local

```
fn5()
{
     int *p1, *p2, *p3;

i1:  a_1_my_malloc_fn5_1 = &p1;
i2:  a_2_my_malloc_fn5_1 = &p2;
i3:  a_3_my_malloc_fn5_1 = &p3;
S1:  my_malloc(&p1, &p2, &p3);
S2:  *p1 = 1;
S3:  *p2 = 2;
S4:  *p3 = 3;
}
```

```
my_malloc(int **q1, int **q2, int **q3)
{
i4: q1 = f_1_my_malloc;
i5: q2 = f_2_my_malloc;
i6: q3 = f_3_my_malloc;
i7: *q1 = i_malloc_my_malloc_1;
i8: *q2 = i_malloc_my_malloc_2;
S5: *q1 = (int *) malloc(sizeof(int));
S6: *q2 = (int *) malloc(sizeof(int));
S7: *q3 = *q2;
}
```

(a)

```
## fn5 ##
MOD:
    p1*
    p2*
    p3*

REF:
    p1
    p2
    p3

POINTS_TO:
  (a_1_my_malloc_fn5_1, p1)
  (a_2_my_malloc_fn5_1, p2)
  (a_3_my_malloc_fn5_1, p3)
  (p1, p1*)
  (p2, p2*)
  (p3, p3*)

CALL_LIST:
    my_malloc, id = 1
```

```
## my_malloc ##
MOD:
    f_1_my_malloc*
    f_2_my_malloc*
    f_3_my_malloc*

REF:
    f_2_my_malloc*

POINTS_TO:
  (f_1_my_malloc*, i_malloc_my_malloc_1*)
  (f_2_my_malloc*, i_malloc_my_malloc_2*)
  (f_3_my_malloc*, i_malloc_my_malloc_2*)

CALL_LIST:
    malloc, id = 1
    malloc, id = 2
```

(b)

```
(p2, H)       or      (p2, p3*)
(p3, H)               (p3, p2*)
```

(c)

Figure 4.2 Handling heap objects: (a) code example, (b) function summary behavior, (c) representing dependence among accesses to heap objects.

statements, these objects are named in the form of `i_malloc_`*foo*_*n*`*`, assuming *foo* is the name of the function containing these calls to `malloc`. Since *n* is a unique number assigned to each call-site, multiple heap objects allocated in the same function via different call-sites can be effectively distinguished. If the heap objects are allocated through calling wrapping functions, these objects are aggressively distinguished by different access paths extended from different variables or different suffix access paths appended after the same variable. That is, all heap objects allocated through calling wrapping functions are assumed to be independent unless they are proven to be dependent.

Consider the example shown in Figure 4.2. In function `fn5`, there are three integer pointers `p1`, `p2`, and `p3`, and through calling `my_malloc`, two instances of heap-based integer objects are allocated. In function `my_malloc`'s summary behavior, these two objects are uniquely named as `i_malloc_my_malloc_1*` and `i_malloc_my_malloc_2*`, respectively. Before propagating these heap objects from `my_malloc` to `fn5`, `p1`, `p2`, and `p3` are assumed to point to disjoint locations as denoted by `p1*`, `p2*`, and `p3*` in Figure 4.2b. 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 right-most access path like `H`, a special form of artificial variables for heap objects, in Figure 4.2c, or adding at least one of the two points-to relations from `p2` to `p3*` or from `p3` to `p2*`. The advantage of the latter option is that it requires no special representation for heap objects, therefore it is chosen in this dissertation and the algorithm used to detect the dependence is presented in Figure 4.3.

```
Determine_Extended_Access_Path($fn$)
{
1:  $S_{EAP}(fn) = \emptyset$;
2:  FOR (each access path $\alpha$ found in fn's summary behavior) {
3:      $S_{EAP}(fn) = S_{EAP}(fn) \cup \{\alpha\}$ if $\alpha$ initiates from an interface variable
            or global variable;
4:      EAP($\alpha$) = $\alpha$;
5:  }
6:  DO {
7:      Let $\gamma$ be the first element in $S_{EAP}(fn)$ and remove $\gamma$ from $S_{EAP}(fn)$;
8:      IF ($\exists\ \beta$ such that $(\gamma, \beta) \in S_{PTR}(fn)$ and EAP($\beta$) is not defined) {
9:          EAP($\beta$) = EAP($\gamma$)$*$;
10:         $S_{EAP}(fn) = S_{EAP}(fn) \cup \{\beta\}$;
11:     }
12:     IF ($\exists\ \beta.so\_eo$ and EAP($\beta.so\_eo$) is not defined) {
13:         EAP($\beta.so\_eo$) = EAP($\beta$)$.so\_eo$;
14:         $S_{EAP}(fn) = S_{EAP}(fn) \cup \{\beta.so\_eo\}$;
15:     }
16: } WHILE ($S_{EAP}(fn) \neq \emptyset$)
}
```

Figure 4.3 Algorithm of constructing extended access paths.

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 a DFS following the points-to relations initiated from formal interface parameters and global variables. An *extended access path*, or *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 variable. Set S_{EAP} is the working list containing access paths whose EAPs have been determined, where lines 2 to 4 fill S_{EAP} with access paths starting from interface variables and global variables. The EAPs of these access paths are simply the access paths' name. For example, the EAP of path `f_1_my_malloc*` is `f_1_my_malloc*` itself.

Lines 6 through 16 process elements in the working list and determine the EAPs based on the relative access paths. Given a points-to relation (γ, β), lines 8 to 11 determine the EAP of β by appending a "`*`" after the the EAP of γ. The reason is obvious since the location denoted by β can be reached from the location denoted by γ through one level of dereference. Due to the nature of aliases, an access path may have multiple ways to be accessed, therefore an access path may have more than one potential EAP. The proposed algorithm defines that the first EAP found for an access path is chosen as the persistent name to be viewed by the caller. For example, `i_malloc_my_malloc_1*` has a unique EAP as `f_1_my_mallloc**`, but `i_malloc_my_malloc_2*` may be assigned as `f_2_my_mallloc**` or `f_3_my_mallloc**`, depending on whether the DFS is performed from `f_2_my_mallloc` or `f_3_my_mallloc` first. In the particular case where the DFS is performed from `f_2_my_mallloc` before from `f_3_my_mallloc`, the EAP of `i_malloc_my_malloc_2*` is `f_2_my_mallloc**`. After the EAPs of aggregates are determined, lines 12 to 15 determine the EAPs of enclosed fields.

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. If the pointer path in a points-to relation is EAP-less, the target path is definitely EAP-less since it will never be put into the working list. The revised version of Definition 5, which propagates transfer functions and detects dependence among accesses to heap objects, is given below.

Definition 6 (Propagation of 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(γ) $\neq \emptyset$. Assuming EAP(γ) $= r\alpha$ and EAP(λ) $= s\beta$, r and s must be formal interface interface 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 conditions are identical to those in Definition 5.* $\diamondsuit$

With EAPs and Definition 6, the points-to relation from path `f_3_my_malloc*` to path `i_malloc_my_malloc_2*` is included in `my_malloc`'s transfer function. The transfer function is represented as (`f_3_my_malloc*`, `f_2_my_malloc**`). After propagating that from `my_malloc` to `fn5`, the points-to relation from `p3` to `p2*` will be added. Otherwise if the DFS is performed for `q3` before `q2`, points-to relation (`p2`, `p3*`) will be added. It does not matter which one is actually added, as long as EVAL(`p2*`, S_{PTR}(fn5)) $\cap$ EVAL(`p3*`, $S_{PTR}(fn5)) \neq \emptyset$ is true since it asserts the dependence of two memory accesses, which will in turn assert the correct execution order of statements `S2`, `S3`, and `S4`.

In addition to summary transfer functions, the summarized MOD/REF memory accesses are also propagated to caller functions. The purpose is to augment the caller's summary behavior so that it also summarizes the footprints of memory accesses of all invoked single level or multi-level callees. The pseudo code of how to propagate the

MOD/REF access paths is omitted here since it is also based on EAPs and the translation rules shown in Definition 6.

4.2.3 Phase II': propagation of function names

In a C function, a function pointer may receive values through local assignments, callees' side-effects, or through values passed via formal parameters. For the first case, it is resolved in the intraprocedural pointer analysis stage; for the second case, it is resolved in the phase I" analysis of the interprocedural stage when the summary transfer functions of the callees are propagated. The analysis conducted in phase II' of the interprocedural stage is to resolve function pointers initialized by the third case. If a function pointer is resolved in the intraprocedural stage, a native function name should appear as a right-most access path of the directly encoded access path of the call-site. If a function pointer is resolved by accommodating the side-effects of the callees, re-evaluating the access path of the indirect call-site will find the propagated function name. If a function pointer is resolved through formal parameters, a right-most access path associated with the call-site access path and initiated from a formal interface variable should be found in the REF section of the callee function. To search for the potential function names passed down from callers, the right-most access path is first transformed with the actual-formal parameter replacement then evaluated in the caller's scope. If the evaluation result contains concrete function names, meaning the function pointer is resolved, they are propagated down to the callee's summary behavior. If parameter-based access paths instead of concrete function names are found, the REF section of the caller function's summary behavior will be augmented and it will be the caller's responsibility to search for concrete function names from further callers, even though the indirect call-site is not contained in the caller function.

Figure 4.4a shows an example where the indirect call-site in function `f8` uses the value passed through a parameter which is initialized in function `f6`. The function body of `foo` is omitted here since it is irrelevant. The summary behavior of each function is shown in Figure 4.4b. Access path `f_1_fn8*` shown in square braces is a right-most access path associated with the call-site path `fn2*` in `fn8`'s summary behavior, and the right-most access path is found by conducting EVAL(`fn2*`, {(`fn2`, `f_1_fn8*`)}).

Figure 4.4c shows the propagated access paths and points-to relations for each function after invoking the phase I" analysis for the first time. Function `f6`'s summary behavior indicates that the dereferenced location from the first formal parameter is modified, so the MOD section of `main` is added with path `fn`, which is the result of EVAL(`a_1_fn6_main*`, {(`a_1_fn6_main`, `fn`)}). Similarly, points-to relation (`fn`, `foo`) is propagated into function `main` from `f6`. Now `fn*` is not a right-most access path since `fn` has outgoing points-to relations. After re-evaluating points-to relation (`a_1_fn7_main`, `fn*`), a new points-to relation (`a_1_fn7_main`, `foo`) is also added. Along with the other SCC chain in the call graph, path `f_1_fn7*` is propagated to the REF section of `fn7` from `fn8`, and in turn it is propagated to the REF section of `main` as `fn*`. Since the MOD/REF sections in a summary behavior are augmented to include caller-allocated locations accessed not only by local expressions but also by invoked callees, the top-down propagation of concrete function names are conducted in a lazy manner: each function

```
main()                 fn6((**fnp)())     fn7((*fn)())    fn8((*fn)())
{                      {                  {               {
     void (*fn)();     S3:  *fnp = foo;   S4: fn8(fn);    S5: (*fn)();
S1:  fn6(&fn);         }                  }               }
S2:  fn7(fn);
}
```

(a)

```
## main ##
REF:
  fn

POINTS_TO:
  (a_1_fn6_main, fn)
  (a_1_fn7_main, fn*)

CALL_LIST:
    fn6, id=1
    fn7, id=2
```

```
## fn6 ##
MOD:
    f_1_fn6*

POINTS_TO:
  (f_1_fn6*, foo)
```

```
## fn7 ##
POINTS_TO:
  (a_1_fn8_fn7, f_1_fn7*)

CALL_LIST:
  fn8, id=1

## fn8 ##
REF:
  f_1_fn8*

CALL_LIST:
  fn2*, id=1 [f_1_fn8*]
```

(b)

```
## main ##
MOD:
  fn

REF:
  fn

POINTS_TO:
  (fn, foo)
  (a_1_fn7_main, fn)
```

```
## fn7 ##
REF:
  f_1_fn7*
```

(c)

```
## fn7 ##
POINTS_TO:
  (f_1_fn7, foo)
  (a_1_fn8_fn7, foo)

## fn8 ##
POINTS_TO:
  (f_1_fn8, foo)
```

(d)

Figure 4.4 Pass function names across functions: (a) code example, (b) summary behavior, (c) information obtained after the first invocation of phase I", (d) information obtained after the first invocation of phase II'.

only requests concrete function names for access paths found in the REF section from immediate callers, and each function never communicates with functions other than immediate callers or callees.

Definition 7 (Concrete value retrieval) *Let $r\alpha$ be an access path in function fn's REF section and $r\alpha*$ is also found in function fn's MOD or REF sections, where r is a formal interface parameter or a global variable, and α is a suffix access path. The targets pointed by concrete values passed via $r\alpha$ from caller fm are: $\{\theta \mid \exists\delta \in EVAL(a\alpha, S_{PTR}(fm))$ and $\exists$ $(\delta, \theta) \in S_{PTR}(fm)\}$ where $a = r$ if r is a global variable. Otherwise a is the corresponding actual parameter in fm.* $\diamondsuit$

Definition 7 shows how to retrieve concrete values from callers. Since a structure may contain multiple scalar fields, where some fields may be pointers to other structures, a single structure-pointer parameter may convey multiple concrete values to the callee. So the actual-formal parameters binding is performed not only for simple variables, but also for access paths starting from formal interface parameters and global variables. Since there are many fields that can but are not used to convey values from a particular caller to a particular callee, concrete values are retrieved lazily: only dereferenced pointers found in the immediate or deeper callees are bound. So in this example, function `fn7` searches its MOD/REF sections and finds path `f_1_fn7*`, which stands for an indirect access through a function pointer. According to Definition 7, the corresponding access path of `f_1_fn7*` in function `main` is `a_1_fn7_main*`. The result of EVEL(`a_1_fn7_main*`, $\{S_{PTR}(main)\}$) is {`foo`}, so a new points-to relation is propagated down to f7 as (f_1_fn7, foo), shown in Figure 4.4d. Now f_1_fn7* is not a right-most access path, and after re-evaluating points-to relation (a_1_fn8_fn7, f_1_fn7*), a new points-to relation, (a_1_fn8_fn7, foo), can be added. When function fn8 requests the content of the function-pointer parameter f_1_fn8, the concrete function name foo can be discovered and the indirect call-site in f8 can be resolved.

4.2.4 Phase II”: propagation of all concrete values

Phase II” analysis is conducted after the fixed point of phases (0-I”-II’)* is reached. At this moment the complete call graph should have been constructed. Then along a top-down topological order of the SCC traversal, access paths representing deferences of formal parameters and global variables in the MOD/REF sections are bound with their concrete values passed from the callers. Unlike the phase II’ analysis which only retrieve function-type concrete values, all types of concrete values are retrieved in this phase, and the retrieval is also conducted lazily for dereferenced pointers only.

4.2.5 Phase III: identification of parameter aliases

Among many concrete values passed down from callers, only a small portion of them are necessary. For example, passing the address of a local variable as an actual parameter is insignificant unless the same address is passed through two different caller-accessible pointers, and both pointers are dereferenced by the callee. So in the phase III analysis, if the evaluation result of an access path found in the MOD/REF sections never has common right-most access paths with the evaluation results of other access paths, its

```
Interprocedural_Pointer_Analysis(prog)
{
1:  DO {
2:      Resolve function pointers for each indirect call-site;
3:      Use DFS to compose SCC-DAG for reachable functions ∈ prog;
4:      FOR (each SCC ∈ prog in bottom-up order) {
5:          Determine EAPs for access paths of each function ∈ the SCC;
6:          Iteratively propagate points-to relations within the SCC if the SCC
                has more than one function;
7:          Reanalyze EAPs for each function if new points-to relations are received;
8:          Propagate the summary transfer function of the SCC to its caller SCCs;
9:      }
10:     FOR (each SCC ∈ prog in top-down order) {
11:         Iteratively propagate function names within the SCC if the SCC has
                more than one function;
12:         Propagate function names from the SCC to its callee SCCs;
13:     }
14: } WHILE (call graph is changed in the previous iteration)
15: FOR (each SCC ∈ prog in top-down order) {
16:     Iteratively propagate concrete values within the SCC if the SCC has
            more than one function;
17:     Retrieve concrete values from caller SCCs;
18: }
19: Determine aliases among parameters;
}
```

Figure 4.5 Pseudo code of the interprocedural pointer analysis stage.

bound values are excluded from the summary behavior since the memory access is always independent with other parameter dereferences across all calling contexts. The trimmed summary behavior will be merged into each function to guide code optimizations, where details of the merge process will be covered in Chapter 5. The pseudo code listed in Figure 4.5 summarizes various phases of analyses conducted in the interprocedural stage.

4.3 Issues about Library Functions

A lot of commonly invoked library functions have side-effects that can be used as alternative ways to initialize pointers. For example, given two pointers `p` and `q`, a library call in the form of `memcpy(&p, &q, 4)` achieves the same effect as the pointer assignment `p = q`. To accommodate their side-effects appropriately in the interprocedural pointer analysis stage, each library function with side-effects are written with template statements. Although these template statements cannot replace the original functionality of library calls, but the equivalent summary behavior can be derived by analyzing these template statements using the algorithm shown in Figure 3.4. For example, the template version of `memcpy` is written as:

```
memcpy(void *p, *q, int n)
{
    *((char **) p) = *((char **) q);
}
```

By analyzing the template statement, the summary transfer function of `memcpy` will include points-to relation (`f_1_mem*`, `f_2_mem**`), which will be processed in the interprocedural stage to expose the effect of the hidden pointer assignment in `memcpy`.

Currently there are 186 library functions modeled by template statements in the IMPACT compiler. They cover all the library functions invoked by SPECcint92 and SPECcint95, MediaBench [47], and many Unix utility benchmarks. For library functions without pointer assignments, as long as they indirectly access locations via formal parameters, they are also represented by template expressions to obtain their memory access footprints that will be referenced by optimization routines to be discussed in Chapter 5. For example, the template version of function `atoi`, which converts a string into an integer, is:

```
atoi(const char *str)
{
    char i;

    i = str[0];
}
```

Because of the pseudo expression that uses `str[0]`, access path `f_1_atoi*` will be posted in the REF section of `atoi`'s summary behavior. So when a code region containing a call to `atoi` is optimized, a later store to the dereferenced location of the actual parameter passed to `atoi` will not be scheduled before the function call to avoid the WAR hazard.

4.4 Complexity Analysis

In this section the complexity of the interprocedural stage algorithm is examined. In addition to the set of intraprocedural parameters introduced in Section 3.4, more parameters are needed to bound the interprocedural complexity. F is the number of user and library functions contained in the program. E is the number of call sites in the program. The space complexity to store all summary behavior of invoked functions simultaneously in the memory is $O(FV_f^2 f^{2kT})$, where in the proposed algorithm no more than two need to be memory-resident simultaneously. For time complexity, transferring an access paths along the call graph is bound by $O(V_f^l f^{lkT})$, and performing a 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 $O(F^2)$. Performing a 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.

CHAPTER 5

CODE OPTIMIZATION USING MEMORY DISAMBIGUATION INFORMATION

In this chapter, the IMPACT compiler optimization framework for memory instructions is presented. The memory disambiguation results generated by the proposed modular interprocedural pointer analysis are used to guide three aggressive compile-time memory access optimizations: *scheduling*, *redundant load/store elimination*, and *loop-invariant access migration*. The potential of these optimizations has been studied by earlier researchers [30, 48], but advanced static memory disambiguation was not available, causing a significant number of indirect memory accesses not optimized.

In the IMPACT compiler, interprocedural pointer analysis is conducted in the frontend, the *Pcode* level as shown in Figure 5.1. The proposed analysis conducted in the function-level phase is implemented in the main *Pcode* module, where the analysis conducted in the interprocedural phase is performed by a stand-alone module named *PIP*, which stands for **P***code* **I***nter***P***rocedural* analysis. A complete round of interprocedural pointer analysis involves the first invocation of *Pcode* module which analyzes each C function and summarizes its behavior. Then the *PIP* module is invoked to propagate the summary behavior along the call graph. Finally, the *Pcode* module is invoked for the

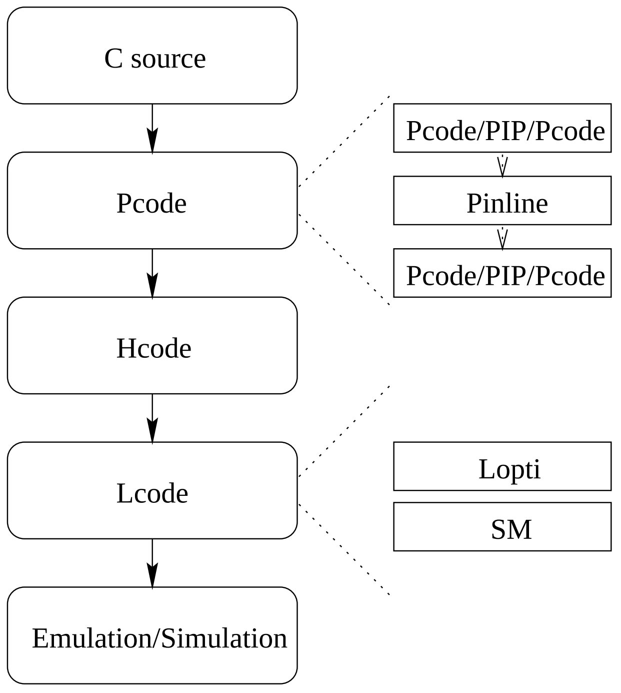


Figure 5.1 Relevant IMPACT modules for static memory disambiguation.

second time to merge the interprocedural analysis results regarding aliases among formal parameters and function side-effects back to each function.

In the overall compilation course of the IMPACT compiler, the *Pcode/PIP/Pcode*-trio are invoked before the middle-level IR, *Hcode*, is reached. Optionally, the trio can be invoked before function inlining to take advantage of function pointers resolved to a small number of callee functions [56]. *Hcode* is the old frontend IR in IMPACT and is in the process of being phased out. *Lcode* is the low-level IR where classic and ILP optimizations are conducted. To convey the memory disambiguation information between the high-level and low-level IRs, a binary relation called *synchronization arcs* or *sync arcs* proposed in [49] are used. In the *Lcode* IR, *load*, *store*, and *subroutine call* (*jsr*) are the three types of instructions than can access the memory. Each jsr is considered as a macro memory instruction which is a collection of load/store instructions accommodating memory accesses not only contained in the immediate callee but also descendant callees. This information is derivable since the interprocedural pointer analysis algorithm can build the complete call graph, where the MOD/REF sections in the summary behavior of a function are fully propagated to callers in the phase I" analysis. With fully resolved pointers, the compiler can classify the relation between two memory instruction as *dependent*, *independent*, or *ambiguous*. An ambiguous relation is often called as *possibility dependent*. To schedule a later load above an earlier store, they need to be independent with each other, and all intermediate store instructions also need to be independent. However, to enable redundant load/store elimination and loop-invariant access migration, it requires stronger relations as two memory instructions need to be dependent. The absence of sync arcs between a pair of memory instructions indicates that they are independent, while the presence of sync arcs between a pair of memory instructions indicates they are dependent or ambiguous, requiring their executions to be synchronized by the semantic order. Since the conducted interprocedural pointer analysis is comprehensive, meaning it can handle all C constructs, it can safely assert that two memory instructions are independent. However, since the pointer analysis is flow-insensitive, it can only produce possibly dependent sync arcs. This is a trade-off between complexity and accuracy, since a flow- and context-sensitive interprocedural pointer analysis algorithm has not been shown in the literature to scale well to handle large programs. Through simple local analysis to be shown later, many possible dependent sync arcs can be converted into definitely dependent sync arcs, enabling memory traffic reduction optimizations. Load/store elimination and loop-invariant access migration routines are implemented in the *Lopti* module, where *SM* is the scheduling manager which can reference the sync arcs to determine whether it is safe or not to percolate loads and stores above other memory instructions.

The remaining chapter is organized as follows. First, an example will be used to point out the benefits provided by advanced static memory disambiguation. Secondly, how sync arcs are annotated to the low-level IR from the interprocedural pointer analysis conducted in the high-level IR will be presented. Finally, algorithms about how to generate definite sync arcs, and how to use sync arcs to guide memory access optimizations will be proposed.

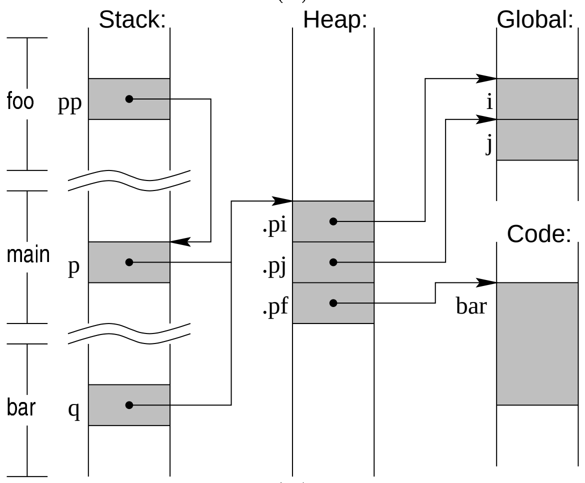
5.1 A Motivating Example

The example in Figure 5.2 is designed to illustrate the problem faced by an optimizing compiler and how well the compiler can optimize the code when all uncertainties about memory dependences are cleared. In the code segment shown in Figure 5.2a, there are two global integer variables `i` and `j`, and one dynamically allocated instance of structure `S`. An instance of `S` contains three pointers: two integer pointers `pi` and `pj`, and one function pointer `pf`. Statements `S1` through `S4` in function `foo` initialize `pi` to point to `i`, `pj` to point to `j`, and `pf` to point to function `bar`. The corresponding points-to relations between pointer and target memory cells contained in this example are shown

```
typedef struct S {
    int *pi;
    int *pj;
    int (*pf)();
} S;

int i, j;

foo(s **pp)
{
S1:  *pp = malloc(sizeof(S));
S2:  (*pp)->pi = &i;
S3:  (*pp)->pj = &j;
S4:  (*pp)->pf = bar;
}
```

```
bar(s *q)
{
S5:  return *q->pi;
}

main()
{
     S *p;

S6:  foo(&p);
S7:  while (i < 10) {
S8:      i = (*p->pf)(p) + 1;
S9:      *p->pj = *p->pj + 2;
S10: }
}
```

(a)



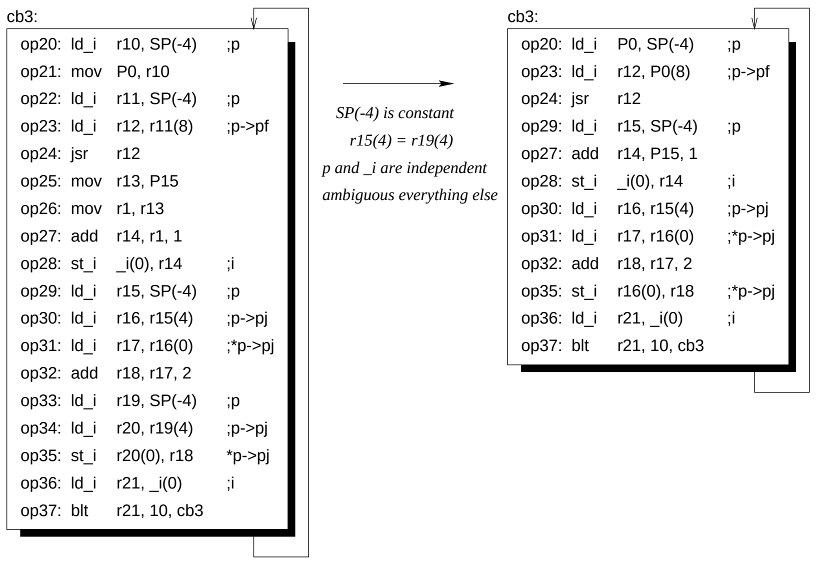
(b)

Figure 5.2 Code example: (a) source program, (b) points-to relations for pointers contained in the program.

in Figure 5.2b. Consider the body of the `while`-loop contained in function `main`, where an unoptimized version of the code is shown on the left portion of Figure 5.3. With limited memory disambiguation information acquired intraprocedurally, the compiler can only eliminate a few redundant accesses to local variable `p`. Since the address of `p` is passed as a parameter to `foo`, any function call in the loop should be considered as containing potential killing stores to `p`. Therefore `op29` cannot be eliminated even though it is dependent with `op20`, leaving a lot of redundant memory accesses in the loop as shown on the right portion of Figure 5.3. However, it is safe to schedule `op29` above `op28`, since it is trivial for the compiler to determine that they access independent locations and there are no intervening ambiguous memory instructions. But for all other memory instructions, their relations are either ambiguous or there are ambiguous memory instructions in between. As a result, the optimized loop body still contains six loads, two stores, and one jsr.

5.2 Sync Arcs Generation

In this section, a systematic description of how to annotate the low-level memory instructions with sync arcs is presented. The first step is to run intraprocedural pointer analysis for each function, where Figure 5.4 shows the summary behavior of each function. Then after interprocedural pointer analysis, the indirect call-site in function `main` can be resolved, since propagating points-to relation (`f_1_foo**.8_11`, `bar`) from `foo` to `main` results in (`p*.8_11`, `bar`). Also, propagating the summary transfer function of `foo` to



```
cb3:
op20:  ld_i   r10, SP(-4)     ;p
op21:  mov    P0, r10
op22:  ld_i   r11, SP(-4)     ;p
op23:  ld_i   r12, r11(8)     ;p->pf
op24:  jsr    r12
op25:  mov    r13, P15
op26:  mov    r1, r13
op27:  add    r14, r1, 1
op28:  st_i   _i(0), r14      ;i
op29:  ld_i   r15, SP(-4)     ;p
op30:  ld_i   r16, r15(4)     ;p->pj
op31:  ld_i   r17, r16(0)     ;*p->pj
op32:  add    r18, r17, 2
op33:  ld_i   r19, SP(-4)     ;p
op34:  ld_i   r20, r19(4)     ;p->pj
op35:  st_i   r20(0), r18     *p->pj
op36:  ld_i   r21, _i(0)      ;i
op37:  blt    r21, 10, cb3
```

```
cb3:
op20:  ld_i   P0, SP(-4)      ;p
op23:  ld_i   r12, P0(8)      ;p->pf
op24:  jsr    r12
op29:  ld_i   r15, SP(-4)     ;p
op27:  add    r14, P15, 1
op28:  st_i   _i(0), r14      ;i
op30:  ld_i   r16, r15(4)     ;p->pj
op31:  ld_i   r17, r16(0)     ;*p->pj
op32:  add    r18, r17, 2
op35:  st_i   r16(0), r18     ;*p->pj
op36:  ld_i   r21, _i(0)      ;i
op37:  blt    r21, 10, cb3
```

Figure 5.3 Moderately optimized code due to the lack of memory disambiguation information.

`main` can resolve pointers `p`, `p->pi`, and `p->pj`. The complete summary behavior after interprocedural pointer analysis for function `main` is shown in Figure 5.5. Since functions `foo` and `bar` are leaf functions in the call graph, and there are no parameter aliases to maintain, their summary behaviors remain the same as the intraprocedural summary.

To determine the dependence among low-level memory instructions, intraprocedural pointer analysis shown in Figure 3.4 is conducted again when each function is being translated from high-level IR to low-level IR. But instead of extracting information to form the summary behavior for each function, the augmented summary behavior produced by interprocedural pointer analysis of the function itself and callee functions is merged back. Merging the interprocedural summary behavior of the function being compiled resolves

```
## main ##
MOD:
  i
  p*.4_7*

REF:
  p
  p*.4_7
  p*.4_7*
  p*.8_11
  p*.8_11*

POINTS_TO:
  (a_1_foo_main_1, p)
  (a_1_p*.8_11*_main_1, p*)

CALL_LIST:
  foo, id = 1
  p*.8_11*, id = 2
```

```
## foo ##
MOD:
  f_1_foo*
  f_1_foo**.0_3
  f_1_foo**.4_7
  f_1_foo**.8_11

REF:
  f_1_foo*

POINTS_TO:
  (f_1_foo*, i_malloc_foo_1*)
  (f_1_foo**.0_3, i)
  (f_1_foo**.4_7, j)
  (f_1_foo**.8_11, bar)

## bar ##
REF:
  f_1_bar*.0_3
  f_1_bar*.0_3*
```

Figure 5.4 Summary behavior of code in Figure 5.2.

the aliases among formal parameters, pointers initialized by callees, and the potential direct callees for each indirect call-site. Then merging the summary behavior of each callee resolves the interaction between local loads/stores and jsr instructions. Although the interprocedural summary behavior of the current function already includes the memory locations accessed by all callees due to the phase I" analysis of the interprocedural stage, acquiring the memory footprint from each individual callee is necessary here, since the MOD/REF information propagated from all callees in phase I" are accumulated to-

```
## main ##                          POINTS_TO:
MOD:                                  (a_1_foo_main_1, p)
  i                                   (a_1_p*.8_11*_main_1, p*)
  p*.4_7*                             (p*.0_3, i)
  p*.0_3                              (p*.4_7, j)
  p*.4_7                              (p*.8_11, bar)
  p*.8_11

REF:                                CALL_LIST:
  p                                   foo, id = 1
  p*.4_7                              p*.8_11*, id = 2 [bar]
  p*.4_7*
  p*.8_11
  p*.8_11*
  p*.0_3
  p*.0_3*
```

Figure 5.5 Complete summary behavior of function `main` in Figure 5.2.

gether, where in the code-generation phase the memory footprint of each jsr instruction needs to be separated.

Figure 5.6 lists the pseudo code that generates sync arcs between potentially dependent memory and jsr instructions. Line 1 performs intraprocedural pointer analysis to resolve pointers that are defined locally and setup the interface variables with callers and callees. Line 2 merges points-to relations that are propagated down from callers and propagated up from callees to the points-to set of `fn`, where Line 3 merges the list of potential callees of each indirect call site `c_exp`. For example, function `main` receives points-to relations for `p*.0_3`, `p*.4_7`, and `p*.8_11`. In addition, the indirect call-site also receives the name of `bar` as a potential callee. The next step is to determine the memory locations that are accessed by a function call. As shown by lines 4 through 11,

```
Generate_Sync_Arcs($fn$)
{
1:  Intraprocedural_Pointer_Analysis($fn$);
2:  Merge points-to relations from $fn$'s interprocedural summary behavior;
3:  Merge resolved callee names for each indirect call-site;
4:  FOR (each call-site $c\_exp$) {
5:      $S_{MOD}(c\_exp) = \emptyset$;
6:      $S_{REF}(c\_exp) = \emptyset$;
7:      FOR (each invoked callee $fm$ of $c\_exp$) {
8:          $S_{MOD}(c\_exp) = S_{MOD}(c\_exp) \cup \text{EVAL}(S_{MOD}(fm), S_{PTR}(fn))$;
9:          $S_{REF}(c\_exp) = S_{REF}(c\_exp) \cup \text{EVAL}(S_{REF}(fm), S_{PTR}(fn))$;
10:     }
11: }
12: FOR (each expression $m\_exp$ found by rules in Table 4.1)
13:     $S_{MEM}(m\_exp) = \text{EVAL}(\text{AP}(m\_exp), S_{PTR}(fn))$;
14: $S_{SYNC\_ARCS} = \emptyset$;
15: FOR (each $m\_exp1$) {
16:     FOR (each $m\_exp2$ reachable from $m\_exp1$)
17:         IF $(S_{MEM}(m\_exp1) \cap S_{MEM}(m\_exp2) \neq \emptyset)$
18:             $S_{SYNC\_ARCS} = S_{SYNC\_ARCS} \cup (m\_exp1, m\_exp2, \text{P})$;
19:     FOR (each $c\_exp$ reachable from $m\_exp1$) {
20:         IF (($m\_exp1$ is a load) and $(S_{MEM}(m\_exp1) \cap S_{MOD}(c\_exp) \neq \emptyset)$)
21:             $S_{SYNC\_ARCS} = S_{SYNC\_ARCS} \cup (m\_exp1, c\_exp, \text{P})$;
22:         IF (($m\_exp1$ is a store) and
                $(S_{MEM}(m\_exp1) \cap (S_{MOD}(c\_exp) \cup S_{REF}(c\_exp)) \neq \emptyset)$)
23:             $S_{SYNC\_ARCS} = S_{SYNC\_ARCS} \cup (m\_exp1, c\_exp, \text{P})$;
24:     }
25: }
26: FOR (each $c\_exp$) {
27:     FOR (each store $m\_exp$ reachable from $c\_exp$)
28:         IF $(S_{MEM}(m\_exp) \cap (S_{MOD}(c\_exp) \cup S_{REF}(c\_exp)) \neq \emptyset)$
29:             $S_{SYNC\_ARCS} = S_{SYNC\_ARCS} \cup (c\_exp, m\_exp, \text{P})$;
30:     }
31: }
}
```

Figure 5.6 Pseudo code of sync arcs generation.

each call-site `c_exp` has a MOD set, S_{MOD}, and a REF set, S_{REF}, where the access paths in each callee fm's MOD/REF sections are translated and evaluated in the context of fn's points-to relations to find the corresponding right-most access paths. The evaluation results are accumulated in each call-site's MOD and REF sets. Then lines 12 and 13 apply the rules listed in Table 4.1 to identify expressions that cause memory accesses, and their evaluation results are recorded in set S_{MEM}. Then the sync arcs set S_{SYNC_ARCS} is generated by lines 14 through 31. Each pair of memory/memory and memory/jsr instructions are checked by comparing their EVAL results. For memory/memory instructions, RAR, RAW, WAR, and WAW relations are respected by generating sync arc ($exp1$, $exp2$, P), which means that expression $exp1$ should be executed before $exp2$, and P means the dependence relation is possible. In other words, the relation between the memory locations accessed by $exp1$ and $exp2$ is ambiguous. For memory/jsr instructions, only RAW, WAR, and WAW relations are respected, since redundant load instructions are not eliminated across function boundaries. Sync arc ($exp1$, $exp2$, P) generated in the high-level IR will be represented as ($op1$, $op2$, P) in the low-level IR, where $op1$ and $op2$ are the corresponding assembly instructions of $exp1$ and $exp2$, respectively.

Function `main` of Figure 5.3a is used as an example here to demonstrate how memory accesses are disambiguated. The merged points-to relations can be found in Figure 5.5. Expression `*p->pf` from statement `S8` is an indirect function call, and it is resolved to invoke function `bar` as shown in Figure 5.5. So the MOD/REF sections of `bar` will be included to identify the memory locations accessed by call-site `S8`. In this particular example, `bar` only reads data from two memory locations: `q->pi` and *q->pi, or f_1_bar*.0_3 and f_1_bar*.0_3* in terms of access paths using interface variables. With points-to relations (a_1_p*.8_11*_main_2, p*) and (p*.0_3, i) found in $S_{PTR}(main)$, their corresponding right-most access paths in main's name space are p*.0_3 and i, respectively, since EVAL(a_1_p*.8_11*_main_2*.0_3, $S_{PTR}(main)$) results in {p*.0_3} and EVAL(a_1_p*.8_11*_main_2*.0_3*, $S_{PTR}(main)$) results in {i}. Both access paths will be included in set S_{REF}(*p->pf) of the indirect call-site. As for expressions causing memory accesses, if a memory expression appears as the top-level LHS expression of an assignment, it will be translated into a store instruction in the assembly code, while all other memory expressions are translated into loads in the assembly code. Figure 5.7a shows the memory expressions of main classified by whether they are translated into loads or stores. The associated right-most access paths of each memory expression is shown in Figure 5.7b. With the EVAL results, the dependence between pair-wise memory and jsr instructions can be determined. Although the underlying pointer analysis is flow-insensitive, the control-flow information can still be used to determine whether or not the dependence information between a pair of memory instructions needs to be posted. For example, if two dependent memory instructions are on the opposite paths of a branch and the branch is not in a loop, since these two instructions cannot reach each other, their dependence can be ignored. In this example, in addition to accesses from expressions with the same access paths are considered as potentially dependent, the indirect jsr will be found as potentially dependent with the store to i. As mentioned before, the RAR dependence between a jsr and a load instruction is not posted, since redundant load elimination is not conducted across function boundaries,

```
LOAD:
  S7: i (op 36)
  S8: p (op 20)
  S8: p (op 22)
  S8: p->pf (op 23)
  S9: p (op 29)
  S9: p->pj (op 30)
  S9: p->pj* (op 31)
  S9: p (op 33)
  S9: p->pj (op 34)

STORE:
  S8: i (op 28)
  S9: p->pj* (op 35)

JSR:
  S8: *p->pf (op 24)
```

(a)

$S_{MEM}(\texttt{i}) = \{\texttt{i}\}$
$S_{MEM}(\texttt{p}) = \{\texttt{p}\}$
$S_{MEM}(\texttt{p->pf}) = \{\texttt{p*.8\_11}\}$
$S_{MEM}(\texttt{p->pj}) = \{\texttt{p*.4\_7}\}$
$S_{MEM}(\texttt{p->pj*}) = \{\texttt{j}\}$
$S_{MOD}(\texttt{p->pf*}) = \{\}$
$S_{REF}(\texttt{p->pf*}) = \{\texttt{p*.0\_3}, \texttt{i}\}$

(b)

```
SYNC_ARCS:
  (op20, op22, P)
  (op22, op20, P)
  (op20, op29, P)
  (op29, op20, P)
  (op20, op33, P)
  (op33, op20, P)
  (op22, op29, P)
  (op29, op22, P)
  (op22, op33, P)
  (op33, op22, P)
  (op29, op33, P)
  (op33, op29, P)
  (op24, op28, P)
  (op28, op24, P)
  (op28, op36, P)
  (op36, op28, P)
  (op30, op34, P)
  (op34, op30, P)
  (op31, op35, P)
  (op35, op31, P)
```

(c)

Figure 5.7 Generating sync arcs for functon main: (a) loads/stores/jsrs in the function, (b) accessed locations for each memory expression and function call, (c) sync arcs.

RAR dependence is equivalent as independent for instruction scheduling. The generated sync arcs for function `main` are listed in Figure 5.7c. Since these memory instructions are in a loop, the generated sync arcs are all symmetric.

5.3 Optimizations Using Sync Arcs

After generating sync arcs based on the interprocedural pointer analysis results, the compiler can perform very aggressive instruction-level optimizations to schedule instructions and to eliminate redundant memory accesses. In addition, the high-level code optimizations can also benefit from fully resolved pointers as well. For example, if an indirect call-site is resolved to invoke only one callee, the indirect call-site can be converted into a direct callee to save the overhead involved with indirect function calls. If the converted call-site is executed very often, the callee body can be inlined to completely remove the function call overhead and function barrier for code optimizations. This would not be possible if function pointers are not resolved. If a function pointer is resolved to have a small number of callees, the indirect call-site can be converted into a sequence of direct calls guarded by comparisons of the function pointer contents. The benefit of such transformation is to trade the indirect call overhead with branch penalties. If the pointer is resolved to have multiple callees, no strictly beneficial transformations are available, but the side-effects of the indirect call-site can be bound much more accurately since the number of potentially reachable callees is reduced from all functions with same signatures to a subset of functions whose names are propagated to the function pointers.

In this section, algorithms that use sync arcs to guide memory access optimizations will be proposed. Three optimization techniques will be discussed in detail, including load/store/jsr scheduling, redundant memory access elimination, and loop-invariant access migration. The code example will be based on the inlined version of function `main`

```
Bool Load_Store_Jsr_Scheduling($fn$, $op1$, $op2$)
{
1:  IF ((($op1$, $op2$, P|D) $\in S_{SYNC\_ARCS}(fn)$) and
            (($op1$ is not a load) or ($op2$ is not a load)))
2:      RETURN false;
3:  FOR (each intermediate instruction $op3$ between $op1$ and $op2$) {
4:      IF ($op2$ is register-dependent on $op3$)
5:          RETURN false;
6:      IF ($op2$ is a store or jsr and ($op3$, $op2$, P|D) $\in S_{SYNC\_ARCS}(fn)$)
7:          RETURN false;
8:      IF ($op3$ is a store or jsr and ($op3$, $op2$, P|D) $\in S_{SYNC\_ARCS}(fn)$)
9:          RETURN false;
10: }
11: RETURN true;
}
```

Figure 5.8 Algorithm that determines the safety of reordering load/store/jsr instructions.

from Figure 5.2, where the indirect call-site at `S8` is replaced by the function body of `bar`.

5.3.1 Instruction scheduling

In the IMPACT compiler, instruction scheduling is performed in the scope of a *control block*, or *cb*. A control block may be a simple basic block, a superblock which is formed by coalescing several basic blocks along a frequently executed path using the superblock formation algorithm [50], or a hyperblock formed for predicated code [51]. A cb still has a single entrance at the top, but may have multiple side exists. Control speculation is allowed so that load instructions can percolate above earlier branch instructions in the control block. The basic idea of instruction scheduling is to reorder the execution of two instructions if the earlier one is stalled while the later one is free to go, or the earlier is not on the critical path but the later one is. The precondition for scheduling is that the later instruction is not dependent on all bypassed instructions. For instructions other than loads, stores, and jsr, the inter-instruction dependence can be easily determined by the register ids of source and destination operands. But for instructions that involve memory accesses, dependence is not determined by register ids but by their contents. With interprocedural pointer analysis performed in the high-level IR in IMPACT, dependence caused by memory instructions are explicitly represented by sync arcs, and the algorithm listed in Figure 5.8 uses sync arcs to determine whether it is safe to reorder a pair of load/store/jsr instructions.

In this algorithm, lines 1 and 2 check if the later instruction, $op2$, is ambiguous with (denoted by P in the sync arc) or dependent on (denoted by D in the sync arc) the earlier instruction, $op1$. If a sync arc exists, unless both instructions are loads, they cannot be reordered since either RAW, WAR, or WAW dependence will be violated. Lines 3 through 10 check if intermediate instructions prevent $op2$ from being executed earlier. Lines 4 and 5 simply check if $op2$'s source operands are data-dependent on an arbitrary intermediate instruction $op3$. For example, if $op2$ is a load instruction whose base register is produced by $op3$, $op2$ cannot be scheduled above $op3$, effectively preventing $op2$ from being scheduled above $op1$. If $op2$ is a store or jsr instruction, lines 6 and 7 check if any intermediate instruction has a RAW or WAW dependence with $op2$. If $op2$ is a load, lines 8 and 9 bypass RAR hazards but respect WAR hazards. If all the checks are passed, $op2$ can be scheduled before $op1$.

```
Generate_Definite_Sync_Arcs($fn$, $op1$, $op2$)
{
1:  IF (($op1$, $op2$, P) $\notin S_{SYNC\_ARCS}(fn)$)
2:      RETURN;
3:  IF (($SRC\_OP1(op1) == SRC\_OP1(op2)$ and
            $SRC\_OP2(op1) == SRC\_OP2(op2)$) or
        ($SRC\_OP1(op1) == SRC\_OP2(op2)$ and
            $SRC\_OP2(op1) == SRC\_OP1(op2)$)) {
4:      FOR (each intermediate instruction $op3$ between $op1$ and $op2$) {
5:          IF ($DEST\_OP(op3) == SRC\_OP1(op3)$ or
                $DEST\_OP(op3) == SRC\_OP2(op3)$)
6:              RETURN;
7:      }
8:      $S_{SYNC\_ARCS}(fn) = S_{SYNC\_ARCS}(fn)$ - {($op1$, $op2$, P)} $\cup$ {($op1$, $op2$, D)};
9:  }
}
```

Figure 5.9 Converts possible sync arcs to definite sync arcs.

Unlike conventional schedulers which treat function calls as implicit scheduling barriers, the proposed algorithm in Figure 5.8 allows instructions to percolate across function calls in the same control block as long as the associated sync arcs do not post RAW, WAW, or WAR hazards.

5.3.2 Redundant load/store elimination

While instruction scheduling mostly relies on the independent classification for memory instructions, eliminating a redundant load or store instruction requires a pair of memory instructions with definite memory dependence. Since the underlying interprocedural pointer analysis algorithms produce may-alias and must-not-alias relations instead of must-alias relations, additional analysis is required to resurrect the definite-alias re-

```
Bool Redundant_Load_Store_Elimination($fn$, $op1$, $op2$)
{
1:  IF (($op1$ does not dominate $op2$) or
        (($op1$, $op2$, D) $\notin S_{SYNC\_ARCS}(fn)$))
2:      RETURN false;
3:  IF (($op2$ is a store but $op1$ is a load)
4:      RETURN false;
5:  FOR (each intermediate instruction $op3$ between $op1$ and $op2$) {
6:      IF ($op3$ is a store or jsr and ($op3$, $op2$, P|D) $\in S_{SYNC\_ARCS}(fn)$)
7:          RETURN false;
8:  }
9:  RETURN true;
}
```

Figure 5.10 Algorithm of redundant load/store elimination.

lation. The proposed analysis is straightforward: if two memory instructions have the same source registers and literal operands, and the source registers are not modified by intermediate instructions, these two instructions are dependent. To increase the successful rate of pattern matching, copy propagation is aggressively conducted for register operands. The algorithm to generate definite sync arcs is listed in Figure 5.9. In the Lcode IR of IMPACT, the addressing mode of a load or store could be register/register or register/literal, where $SRC\_OPn(op)$ means the nth source operand of instruction *op*, and $DEST\_OP(op)$ means the destination operand of *op*.

With definite sync arcs, Figure 5.10 lists the pseudo code which eliminates redundant load/store instructions in the same control block. The predicate function returns true if it is safe to eliminate *op*2 and use the destination operand of *op*1 if it is a load or the third source operand of *op*1 if it is a store. The preconditions are that *op*1 dominates *op*2, and *op*1 and *op*2 are definitely dependent. If the control block is a hyperblock, meaning instructions may be guarded by predicates, *op*1 may not dominate *op*2 unless *op*1's predicates are a superset of *op*2's predicates. Otherwise if the control block is a basic block or superblock, instructions with the earlier order in the code layout dominate later instructions. Lines 3 and 4 check the applicability of store/store elimination, since an earlier load cannot eliminate a later store. Then unless an intermediate store or jsr is not independent with *op*2, which is checked by lines 5 through 8, it is safe to eliminate *op*2.

5.3.3 Loop-invariant access migration

Loop-invariant access migration is a technique that promotes a loop-invariant location into a register so that the dynamic count of memory instructions can be reduced, since the register only needs to be filled once before the loop is entered, and the same memory location only needs to be updated once at each exit block of the loop. Notice that here loop-invariance means the effective address of a memory instruction but not its content. Loop-invariant effective address can be easily asserted if the source addressing operands of a memory instruction are loop-invariant. However, it is not trivial to identify the the location which can be safely promoted if the loop contains indirect memory accesses and jsrs. If the location is read-only in the loop, it is still safe to identify a subset of loads that access the location, albeit some redundant loads are still present in the loop. If the invariant location is ever updated in the loop, failing to identify the whole group of instructions that access the invariant location may cause incorrect results, since migrated

```
Partition_Loop_Invariant_Locations($fn$, $loop$)
{
1:  $S_{INV}(loop) = \emptyset$;
2:  $n = 1$;
3:  FOR (each loop-invariant load or store instruction $op \in loop$) {
4:      $G_n = \{op\}$;
5:      $S_{INV}(loop) = S_{INV}(loop) \cup G_n$;
6:      $n++$;
7:  }
8:  DO {
9:      FOR (each pair of $G_i$ and $G_j \in loop$) {
10:         IF (($opi$, $opj$, D) $\in S_{SYNC\_ARCS}(fn)$ where $opi \in G_i$ and $opj \in G_j$) {
11:             $G_i = G_i \cup G_j$;
12:             $S_{INV}(loop) = S_{INV}(loop)$ - $G_j$;
13:         }
14:     }
15: } WHILE ($S_{INV}(loop)$ changes)
}
```

Figure 5.11 Algorithm of partitioning loop-invariant memory instructions.

loads/stores are using a register's content which is not updated by the remaining store instructions in the loop. Therefore, like redundant load/store elimination, loop-invariant access migration also needs definite sync arcs.

The first step in performing loop-invariant access migration is to partition the loop-invariant memory instructions into several groups so that instructions in the same group are definitely dependent with each other while different groups access disjoint locations. The algorithm in Figure 5.11 forms disjoint groups using sync arcs. Lines 1 to 7 form an initial set of n groups of loop-invariant locations where n is the number of loop-invariant memory instructions in the loop and each group only has one instruction. Then an iterative process in lines 8 through 15 merges two groups of loop-invariant memory instructions if instructions from one group are definitely dependent with instructions in the other group. Since the dependent attributes produced by pattern matching is transitive, checking a single pair of instructions can determine whether two groups represent the same location or not.

The groups finalized by the algorithm in Figure 5.11 only guarantee that all intra-group instructions access the same loop-invariant location. Additional analysis is needed to determine whether a group can be safely migrated or not. As shown in Figure 5.12, lines 2 and 3 of the algorithm first separate the load and store instructions in each group. Then depending on whether the group of instructions are write-only, read-only, or read/write, and how the group interacts with other memory and jsr instructions in the loop, different migration decisions are made as shown below:

- **Write-only group** (lines 4 to 5). A group is write-only if its *LOAD* set created at line 2 is empty, meaning all instructions in this group are store instructions. Since store instructions are usually not on the critical path of a computation chain and dedicating a register may increase the register pressure, a write-only group is not migrated.

- **Read-only group** (lines 6 to 15). If a group is read-only, meaning that there are no store instructions in the group, whether the group can be promoted or not depends on how many loads can be removed and how many refill loads need to be added. If none of the store or jsr instructions in the loop can ever modify the same location of the read-only group, the content of the accessed location is also

```
Loop_Invariant_Access_Migration($fn$, $loop$)
{
1:  FOR (each $G_i \in S_{INV}(loop)$) {
2:      $LOAD(G_i) = \{opl \mid opl \in G_i$ and $opl$ is a load$\}$;
3:      $STORE(G_i) = \{ops \mid ops \in G_i$ and $ops$ is a store$\}$;
4:      IF ($LOAD(G_i) == \emptyset$)
5:          CONTINUE;
6:      IF ($STORE(G_i) == \emptyset$) {
7:          $refill\_load = 0$;
8:          FOR (each store or jsr $op \in loop - G_i$)
9:              IF (($op$, $opl$, P) $\in S_{SYNC\_ARCS}(fn)$)
10:                 $refill\_load++$;
11:         IF ($refill\_load == 0$)
12:             Migrate($fn$, $loop$, $G_i$);
13:         ELSE IF ($refill\_load < |LOAD(G_i)|$)
14:             Migrate_With_Register_Refill($fn$, $loop$, $G_i$);
15:     }
16:     ELSE {
17:         $refill\_load = 0$;
18:         $needs\_spilling = 0$;
19:         IF (($\exists$ store or jsr $op \in loop - G_i$) and (($op$, $opl$, P) $\in S_{SYNC\_ARCS}(fn)$))
20:             $needs\_spilling = 1$;
21:         FOR (each store or jsr $op \in loop - G_i$)
22:             IF (($op$, $opl$, P) $\in S_{SYNC\_ARCS}(fn)$)
23:                 $refill\_load++$;
24:         IF ($refill\_load == 0$)
25:             IF ($needs\_spilling \neq 0$)
26:                 Migrate_With_Register_Spill($fn$, $loop$, $G_i$);
27:             ELSE
28:                 Migrate($fn$, $loop$, $G_i$);
29:         ELSE IF ($n < |LOAD(G_i)|$)
30:             IF ($needs\_spilling \neq 0$)
31:                 Migrate_With_Register_Spill_and_Refill($fn$, $loop$, $G_i$);
32:             ELSE
33:                 Migrate_With_Register_Refill($fn$, $loop$, $G_i$);
34:     }
35: }
}
```

Figure 5.12 Algorithm of determining the safety of loop-invariant migration.

loop-invariant. So loading the content into a register once prior to the beginning of the loop can provide up-to-date value for the entire course of the loop. However, if there are potentially or definitely dependent store or jsr instructions in the loop, the promoted register may hold obsolete values. One way to fix the consistency problem is to reload the dedicated register from the promoted location after every conflicting store and jsr instruction, should the number of conflicting store/jsr instructions is less than the number of load instructions in the group otherwise it is not beneficial. This check is done by line 13.

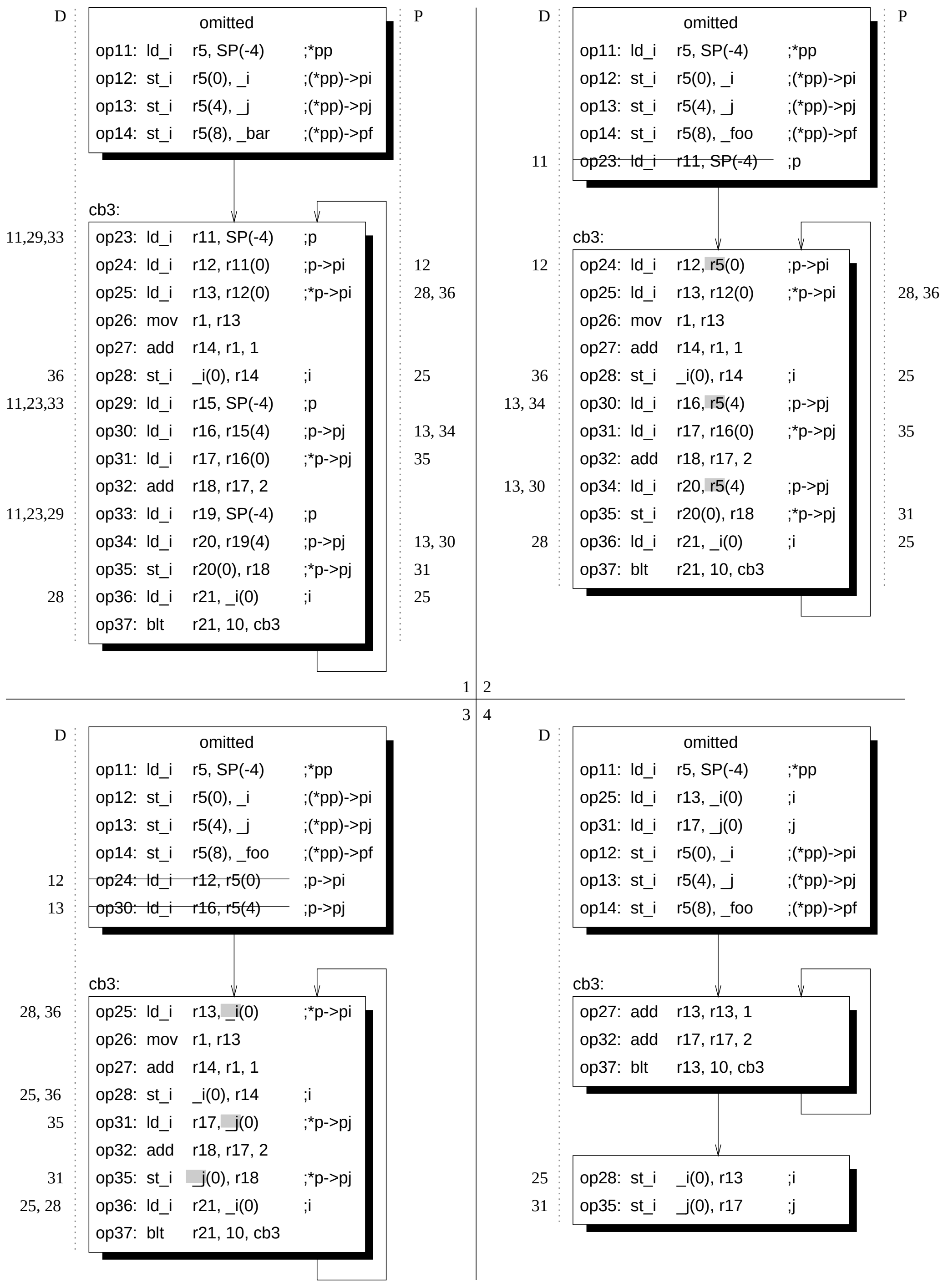
- **Read/write group** (lines 16 to 34). If a group is read/write, more complicated conditions need to be checked before instructions in the group can be migrated, since there are more chances that the memory content may be inconsistent with the promoted register. If there are potentially conflicting load instructions in the loop, the promoted register needs to be spilled before each conflicting loads to guarantee that the load retrieves the up-to-date value. If there are potentially conflicting store instructions in the loop, the dedicated register is spilled right before the store and refilled right after the store to get the up-to-date content, if it is ever modified. Similarly, if there is a conflicting jsr, the dedicated reciter will also be spilled then loaded around the jsr. So the cost-effective factor is the number of conflicting loads and jsrs in the loop: if it exceeds the number of load instructions in the loop, it is not beneficial to migrate this group.

In Figure 5.12, there are three types of load/store migrations. They all perform the same style of code transformation by migrating all the loads in the group to the loop preheader, and by migrating all stores to every loop exit block. Since all of them are definitely dependent memory instructions, only one load and one store will be left after performing redundant load/store elimination. Then all instructions that consume the value loaded by the migrated loads will use the dedicated register which holds the content of the promoted memory location. In addition to the basic operations, Migrate_With_Register_Refill reloads the dedicated register after all conflicting stores and jsrs, where Migrate_With_Register_Refill will added reloads as well as spilling stores around all conflicting stores and jsrs. Since these transformations are straightforward, their pseudo codes are omitted.

5.3.4 A Working Example

Here the loop in function `main` of Figure 5.2 will be revisited as a working example to show the optimization process. Redundant load/store elimination, copy propagation, and loop-invariant access migration are conducted iteratively until a fixed point is reached. As shown in Figure 5.13, the loop can be optimized in four stages:

1. The assembly code shown in stage 1 is generated by inlining both functions `foo` and `bar` into `main`. Instructions `op12` to `op14` are absorbed from `foo`, where `op29` to `op31` are included from `bar`. Inlining the indirect call-site is possible since the function pointer `p->pf` is resolved to a single callee. Shown on the left of an instruction is the list of instruction ids that have earlier semantic order and definite



```
1
D                                                              P
              omitted
          op11:  ld_i   r5, SP(-4)      ;*pp
          op12:  st_i   r5(0), _i       ;(*pp)->pi
          op13:  st_i   r5(4), _j       ;(*pp)->pj
          op14:  st_i   r5(8), _bar     ;(*pp)->pf

          cb3:
11,29,33  op23:  ld_i   r11, SP(-4)     ;p
          op24:  ld_i   r12, r11(0)     ;p->pi              12
          op25:  ld_i   r13, r12(0)     ;*p->pi             28, 36
          op26:  mov    r1, r13
          op27:  add    r14, r1, 1
36        op28:  st_i   _i(0), r14      ;i                  25
11,23,33  op29:  ld_i   r15, SP(-4)     ;p
          op30:  ld_i   r16, r15(4)     ;p->pj              13, 34
          op31:  ld_i   r17, r16(0)     ;*p->pj             35
          op32:  add    r18, r17, 2
11,23,29  op33:  ld_i   r19, SP(-4)     ;p
          op34:  ld_i   r20, r19(4)     ;p->pj              13, 30
          op35:  st_i   r20(0), r18     ;*p->pj             31
28        op36:  ld_i   r21, _i(0)      ;i                  25
          op37:  blt    r21, 10, cb3
```

```
2
D                                                              P
              omitted
          op11:  ld_i   r5, SP(-4)      ;*pp
          op12:  st_i   r5(0), _i       ;(*pp)->pi
          op13:  st_i   r5(4), _j       ;(*pp)->pj
          op14:  st_i   r5(8), _foo     ;(*pp)->pf
11        ~~op23:  ld_i   r11, SP(-4)~~     ;p

          cb3:
12        op24:  ld_i   r12, r5(0)      ;p->pi
          op25:  ld_i   r13, r12(0)     ;*p->pi             28, 36
          op26:  mov    r1, r13
          op27:  add    r14, r1, 1
36        op28:  st_i   _i(0), r14      ;i                  25
13, 34    op30:  ld_i   r16, r5(4)      ;p->pj
          op31:  ld_i   r17, r16(0)     ;*p->pj             35
          op32:  add    r18, r17, 2
13, 30    op34:  ld_i   r20, r5(4)      ;p->pj
          op35:  st_i   r20(0), r18     ;*p->pj             31
28        op36:  ld_i   r21, _i(0)      ;i                  25
          op37:  blt    r21, 10, cb3
```

```
3
D
              omitted
          op11:  ld_i   r5, SP(-4)      ;*pp
          op12:  st_i   r5(0), _i       ;(*pp)->pi
          op13:  st_i   r5(4), _j       ;(*pp)->pj
          op14:  st_i   r5(8), _foo     ;(*pp)->pf
12        ~~op24:  ld_i   r12, r5(0)~~      ;p->pi
13        ~~op30:  ld_i   r16, r5(4)~~      ;p->pj

          cb3:
28, 36    op25:  ld_i   r13, _i(0)      ;*p->pi
          op26:  mov    r1, r13
          op27:  add    r14, r1, 1
25, 36    op28:  st_i   _i(0), r14      ;i
35        op31:  ld_i   r17, _j(0)      ;*p->pj
          op32:  add    r18, r17, 2
31        op35:  st_i   _j(0), r18      ;*p->pj
25, 28    op36:  ld_i   r21, _i(0)      ;i
          op37:  blt    r21, 10, cb3
```

```
4
D
              omitted
          op11:  ld_i   r5, SP(-4)      ;*pp
          op25:  ld_i   r13, _i(0)      ;i
          op31:  ld_i   r17, _j(0)      ;j
          op12:  st_i   r5(0), _i       ;(*pp)->pi
          op13:  st_i   r5(4), _j       ;(*pp)->pj
          op14:  st_i   r5(8), _foo     ;(*pp)->pf

          cb3:
          op27:  add    r13, r13, 1
          op32:  add    r17, r17, 2
          op37:  blt    r13, 10, cb3

25        op28:  st_i   _i(0), r13      ;i
31        op35:  st_i   _j(0), r17      ;j
```

Figure 5.13 Aggressive code optimization with memory disambiguation information.

sync arcs with the instruction, while instruction ids involved in possible sync arcs are shown on the right. In stage 1, the only loop-invariant group contains `op23`, `op29`, and `op33`. Since none of them have conflicting stores and jsrs in the loop, they are simply migrated to the loop preheader and no spill/refill instructions are needed.

2. The original register dedicated for holding the content of `p` is `r11` in stage 2. But since `op23` and `op11` are definitely dependent, and no intermediate instructions conflict with `p`, `op23` can be eliminated. As result, all data-dependent instructions in the loop that use `r11`, `r15`, and `r19` in stage 1 now all use `r5` in stage 2. Consequently, the relations between `op13`, `30`, and `op34` now become definitely dependent. They also form a loop-invariant group, since `r5` is not defined in the loop. The other loop-invariant group contains a single instruction as `op24`. Since no conflicting stores and jsrs are contained in the group with these two groups, they are also migrated to the loop preheader.

3. The original registers dedicated for holding the contents of `p->pi` and `p->pj` are `r12` and `r16`, respectively. Again, they can be eliminated because of `op12` and `op13`, after propagating the labels of `_i` and `_j` to source operands of memory instructions in the loop. Then all the sync arcs in the loop become definite sync arcs. Consequently, two more loop-invariant groups are identified, containing `op25`, `op28`, `op36`, and `op31`, `op35`, respectively. Since both groups are read/write groups, loads are migrated to the loop preheader and stores are migrated to the exit blocks.

4. Instructions `op25` and `op31` load `i` and `j` before the loop is entered. Since there are no sync arcs between these two loads and other stores in the preheader, both loads can be scheduled prior to the stores to hide their latency. Control block `cb3` now only contains 3 arithmetic instructions and can be executed much more efficiently then the code shown in Figure 5.3.

These optimization algorithms are independent with the interprocedural pointer analysis conducted in the frontend as long as sync arcs are generated in the assembly code, meaning these optimizations can directly benefit from future improvements made on static memory disambiguation. The quantitative contribution of these memory access optimizations will be presented in Chapter 7.

CHAPTER 6

DEEP ANALYSIS ON FUNCTION POINTERS

A fundamental requirement for performing interprocedural optimizations is a complete call graph, which represents the dynamic invocation of functions for a program. For a programming language which only allows functions to be invoked statically, the construction of a complete call graph can be done in a single traversal of the program starting from the entry function, and the resultant call graph is precise. However, if the programming language supports function pointers, which defer the actual callee determination until run-time, the construction of a complete call graph requires extra compile-time analysis. The least precise but still valid call graph can be constructed by assuming that an indirect call-site reaches all functions in the program, including library functions. A straightforward improvement is to rule out functions whose names are neither passed as parameters nor assigned to other variables. However, for programs containing a significant number of indirect call-sites with disjoint callees, the resulting call graph can be quite inaccurate, which can hurt potential optimization opportunities. Although a call graph is required for the interprocedural data-flow analysis, the call graph construction itself requires an incremental interprocedural data-flow analysis, since function pointers may be defined interprocedurally. The final call graph is resolved iteratively where existing function pointers receive new function names propagated from already discovered functions in the partial call graph, and newly added functions may add more function pointers [52, 53, 54]. Murphy et al. [55] evaluate nine call graph extractor tools using three software systems (*mapmaker, mosaic, and gcc*). All of the tools generate false negative call graphs: calls that can in fact take place in some execution of the program, but which are omitted from the call graph. Although this is acceptable to some software engineering purposes, a compiler needs a false positive call graph in order to perform correct optimizations.

Although algorithms for call graph construction have been widely studied in the literature, some characteristics found in the SPEC benchmarks cannot be handled by earlier algorithms. The major reason comes from the underlying programming language, C, used for these benchmarks. C allows functions to be invoked through function pointers, where function pointers may be accessed through non-function pointers via multi-level pointers and structure/union fields, and function pointers may be manipulated as non-function pointers due to type casts. Therefore earlier techniques geared towards *Fortran* will not suffice [53, 54]. It is difficult, if not impossible, to resolve function pointers separately from other general pointers. In fact, for programs written in C, the call graph construction problem has evolved into an interprocedural pointer analysis problem; once all pointers are resolved, all function pointers are also resolved, and the call graph can be generated. Stated in another way, an interprocedural pointer analysis is not complete if it cannot resolve all function pointers. Due to either simplified assumptions or high complexity in the algorithm, recent work in interprocedural pointer analysis has not demonstrated the ability to construct the complete call graph for significant programs like SPEC [22, 34, 19, 20, 26, 31].

In this chapter, a comprehensive study of function pointers using SPEC benchmarks is reported. SPEC benchmarks are selected for this study because of their size, wide acceptance, and realistic function pointer manipulation. Benchmarks included in this study are *008.espresso*, *023.eqntott*, *072.sc*, *085.cc1*, *124.m88ksim*, *126.gcc*, *130.li*, *132.ijpeg*, *134.perl*, and *147.vortex*, which have complicated code constructs that are not handled by conventional call graph extractor.

6.1 Case Studies of Function Pointers

In this section, code examples extracted from the SPEC benchmarks are presented. These examples can provide insight to the manipulation of function pointers in realistic programs. The usual motivation for programmers to use function pointers is to write concise code: using a single indirect call-site to invoke multiple similar callees instead of multiple individual static call-sites. However, pointers and type casts in C make the problem much harder than in Fortran. In general, function pointers can be classified into four categories with difficulty levels ranked from the easiest to the hardest as follows:

1. **Simple variables**. Function pointers declared as local or global variables whose addresses are never taken. That is, the definition and usage of such pointers can be accurately identified without the need to consider aliases.

2. **Function parameters**. Function pointers declared as parameters like those used in Fortran.

3. **Statically initialized global function pointers**. Function pointers declared and initialized in the global data section. Function pointers may be fields in global structures/unions.

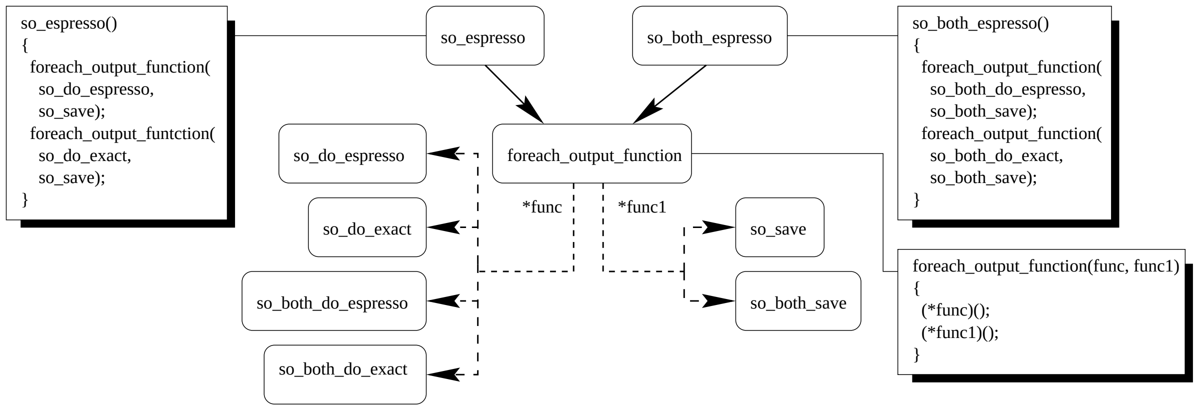
4. **Heap objects.** Function pointers allocated dynamically from the heap.

Function pointers used as simple variables can be resolved by traditional data-flow analysis, where function pointers passed as parameters can be handled by the *BoundTo* analysis as proposed by Hall and Kennedy [54]. Function pointers initialized statically can be handled properly as long as various forms of structure/union initialization can be processed. However, heap-based function pointers cannot be resolved without using a general interprocedural pointer analysis algorithm. For each category of function pointers, at least one code example is provided to better illustrate the difficulties involved and the potential solutions. For clarity and space reasons, only directly related code segments are extracted.

6.1.1 Function pointers as simple variables

Function pointers declared as simple variables without aliases are easy to resolve using traditional data-flow analysis. For example, in function `emit_case_nodes` of *085.cc1*, the following code segment can be found:

```
gen_bgt_pat = unsignedp ? gen_bgtu : gen_bgt;
```



```
so_espresso()
{
  foreach_output_function(
    so_do_espresso,
    so_save);
  foreach_output_funtction(
    so_do_exact,
    so_save);
}
```

```
so_both_espresso()
{
  foreach_output_function(
    so_both_do_espresso,
    so_both_save);
  foreach_output_function(
    so_both_do_exact,
    so_both_save);
}
```

```
foreach_output_function(func, func1)
{
  (*func)();
  (*func1)();
}
```

Figure 6.1 Partial call graph of *008.espresso*.

```
(*gen_bgt_pat)();
```

It is trivial to find functions `gen_betu` and `gen_bgt` as the potential callees of the indirect call-site. In the proposed interprocedural pointer analysis algorithm, this type of function pointers can be easily resolved by the points-to relations added in the intraprocedural stage. Another case is found in *126.gcc*, where global function pointer `decl_printable_name` is assigned with `decl_name` in function `main` and used in many other functions. Although the function pointer has no aliases across function boundaries, interprocedural pointer analysis is still required in this case to construct the complete call graph since global variables are involved. Surprisingly, these simple cases only happen in these two largest benchmarks of SPEC.

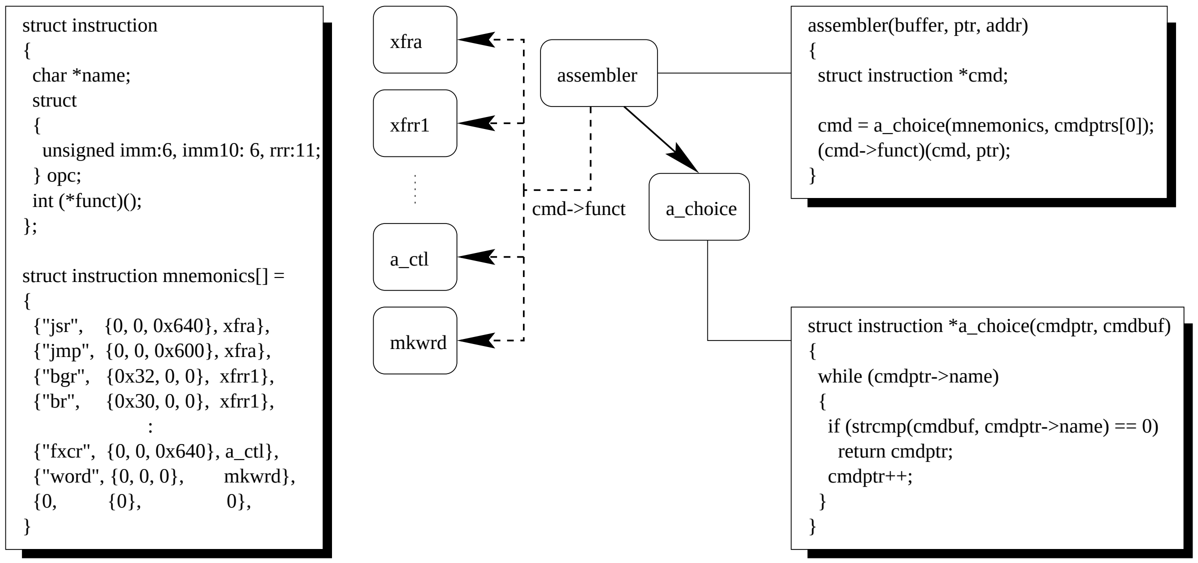
6.1.2 Function pointers as parameters

Function pointers passed as parameters can also be handled with widely studied techniques. For example, in benchmark *008.espresso*, function `foreach_output_function` can be called form four call-sites in two callers as shown in Figure 6.1. Throughout this chapter, solid edges in the call graph are direct call-sites, while dashed edges are indirect call-sites with the call-site expression annotated. The functions invoked by the two indirect call-sites in function `foreach_output_function` depend on the actual binding of parameters `func` and `func1`, which can be resolved using the analysis conducted in phase II' in the interprocedural stage, where the evaluated access path starts from an formal interface parameter followed by a single dereference operator. If the interprocedural analysis is context-sensitive in parameter aliases [19], it can further assert that some components of the call graph are mutually exclusive. For example, `foreach_output_function` never calls `so_do_espresso` and `so_both_save` in the same invocation.

6.1.3 Function pointers as statically initialized global arrays

An alternative way to achieve call-site reuse is to initialize a global table containing function pointers which can be retrieved later through indices. It is a common technique used by language-processing programs like *085.cc1*, *124.m88ksim*, *126.gcc*, and *130.li*.

In Figure 6.2 which is extracted from benchmark *124.m88ksim*, the `instruction` structure is defined as the data structure to store information regarding each assembly instruction's mnemonic name, opcode, and the native function's address. Global variable



```
struct instruction
{
  char *name;
  struct
  {
    unsigned imm:6, imm10: 6, rrr:11;
  } opc;
  int (*funct)();
};

struct instruction mnemonics[] =
{
  {"jsr",   {0, 0, 0x640}, xfra},
  {"jmp",   {0, 0, 0x600}, xfra},
  {"bgr",   {0x32, 0, 0},  xfrr1},
  {"br",    {0x30, 0, 0},  xfrr1},
                    :
  {"fxcr",  {0, 0, 0x640}, a_ctl},
  {"word",  {0, 0, 0},     mkwrd},
  {0,       {0},           0},
}
```

```
assembler(buffer, ptr, addr)
{
  struct instruction *cmd;

  cmd = a_choice(mnemonics, cmdptrs[0]);
  (cmd->funct)(cmd, ptr);
}
```

```
struct instruction *a_choice(cmdptr, cmdbuf)
{
  while (cmdptr->name)
  {
    if (strcmp(cmdbuf, cmdptr->name) == 0)
      return cmdptr;
    cmdptr++;
  }
}
```

Figure 6.2 Partial call graph of *124.m88ksim*.

`mnemonics` is declared as an `instruction` array, and is initialized in the data section of file *asmcmdstr.c*. Function `assembler` has an indirect call-site using the statically initialized `mnemonics` table. But instead of directly indexing the table, an indexing function `a_choice` is used. To handle this case right, the interprocedural pointer analysis algorithm has to handle global variable initializations including structure fields. When function return values are propagated back to the caller in the phase I" analysis of the interprocedural stage, variable `cmd` in `assembler` can be identified as holding the address of array `mnemonics`, indicating `cmd->funct` and `mnemonics[x].funct` to be aliases [1]. Thus the function names stored in the `mnemonics` array are resolved to be the possible callees of `cmd->funct`.

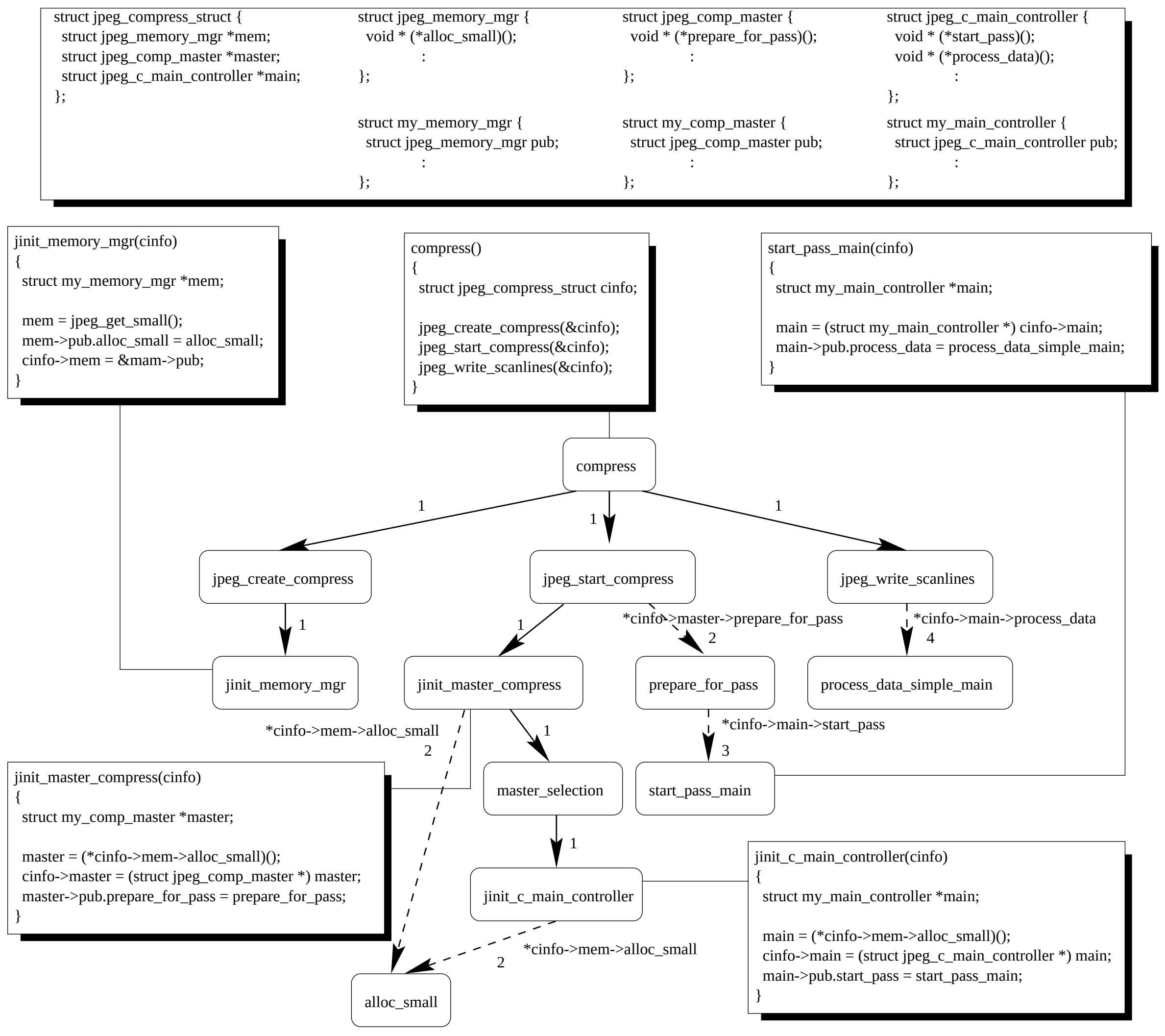
[1]The whole array is treated as a single object.

6.1.4 Function pointers as anonymous objects

Benchmark *132.ijpeg* uses function pointers to maintain the program's portability. As an image-processing program, many function pointers are initialized at the program entry point to use a platform-specific set of intrinsic functions. When studying the benchmark, several important characteristics are interesting as discussed below.

- **Extensive use of heap-based function pointers**. Once a heap-object is allocated by the callee and attached to the formal parameter, as discussed in Chapter 4, in addition to propagating values from the caller to the callee, values will need to be propagated from the callee to the caller as well. In function `compress`, `cinfo` is a local variable of type `jpeg_compress_struct` which is listed in the top portion of Figure 6.3. The address of variable `cinfo` is passed as a formal parameter named as `cinfo` also to all subsequent callees [2]. In function `jinit_memory_mgr`, it allocates a structure object of type `my_memory_mgr`, which is a superset of structure `jpeg_memory_mgr`. Then it initializes expression `cinfo->memory->alloc_small`, which is a function pointer for dynamic memory allocation, as `alloc_small`. When function `jinit_memory_mgr` returns, the memory location accessed via expression `cinfo->mem->alloc_small` is still alive, so its content needs to be propagated back to the caller, function `jpeg_create_compress`. Eventually, the phase I" analysis will report the points-to relation of (`cinfo.mem->alloc_small`, `alloc_small`) to function `compress`.

[2]The type of cinfo in compress is "struct jpeg_compress_struct" but is "struct jpeg_compress_struct *" in all other functions.



```
struct jpeg_compress_struct {
  struct jpeg_memory_mgr *mem;
  struct jpeg_comp_master *master;
  struct jpeg_c_main_controller *main;
};
```

```
struct jpeg_memory_mgr {
  void * (*alloc_small)();
           :
};

struct my_memory_mgr {
  struct jpeg_memory_mgr pub;
           :
};
```

```
struct jpeg_comp_master {
  void * (*prepare_for_pass)();
           :
};

struct my_comp_master {
  struct jpeg_comp_master pub;
           :
};
```

```
struct jpeg_c_main_controller {
  void * (*start_pass)();
  void * (*process_data)();
           :
};

struct my_main_controller {
  struct jpeg_c_main_controller pub;
           :
};
```

```
jinit_memory_mgr(cinfo)
{
  struct my_memory_mgr *mem;

  mem = jpeg_get_small();
  mem->pub.alloc_small = alloc_small;
  cinfo->mem = &mam->pub;
}
```

```
compress()
{
  struct jpeg_compress_struct cinfo;

  jpeg_create_compress(&cinfo);
  jpeg_start_compress(&cinfo);
  jpeg_write_scanlines(&cinfo);
}
```

```
start_pass_main(cinfo)
{
  struct my_main_controller *main;

  main = (struct my_main_controller *) cinfo->main;
  main->pub.process_data = process_data_simple_main;
}
```

```
jinit_master_compress(cinfo)
{
  struct my_comp_master *master;

  master = (*cinfo->mem->alloc_small)();
  cinfo->master = (struct jpeg_comp_master *) master;
  master->pub.prepare_for_pass = prepare_for_pass;
}
```

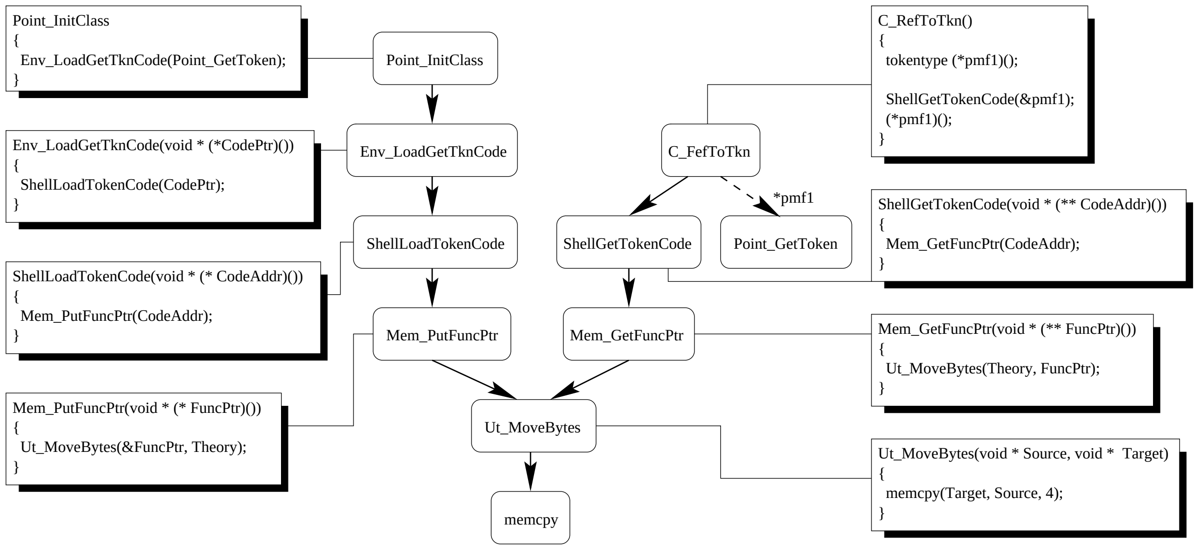
```
jinit_c_main_controller(cinfo)
{
  struct my_main_controller *main;

  main = (*cinfo->mem->alloc_small)();
  cinfo->main = (struct jpeg_c_main_controller *) main;
  main->pub.start_pass = start_pass_main;
}
```

Figure 6.3 Partial call graph of *132.ijpeg*.

- **Function pointers indirectly accessed from parameters**. Passing a pointer parameter allows not only the parameter itself but also all locations accessible from the pointer to be visible to the callee. Following top-down traversal of the SCC nodes conducted in phase II', the function pointer `cinfo->mem->alloc_small` in `jinit_master_compress` and `jinit_c_main_controller` can be found to be bound to `alloc_small`, thus the indirect call-sites can be resolved. Unlike function pointers passed as parameters, resolving function pointers indirectly accessed from parameters involves the evaluation of access paths starting from formal interface parameters followed by at least two dereference operators.

- **Function pointers initialized with type casts**. As explained in [33] and Chapter 3, structure pointers can be used interchangeably if their target structures share a common initial sequence of fields. In the *132.ijpeg* example listed in Figure 6.3, pointer `mem->pub.alloc_small` is an alias of `cinfo->mem->alloc_small`, since the structures pointed by `mem` and `cinfo->mem` share the same initial sequence of fields. With normalized field representation in byte offsets, the proposed postfix access paths can resolve these function pointers in the presence of type casts.

- **Iterative nature of call-graph construction**. The number associated with each indirect call-site indicates the order of the callee being added to the call graph. Function `start_pass_main` initializes the function pointer used in function `jpeg_write_scanlines`. But it cannot be discovered until the indirect call-site in `prepare_for_pass` is resolved, which is initialized in `jinit_c_main_controller`. And `prepare_for_pass` is not connected to the call graph until the initialization in `jinit_master_compress` is discovered. That is, the construction of the partial call graph requires at least four iterations. And whether the optimal number of iterations can be achieved depends on if the phase I" analysis is performed in the



```
Point_InitClass
{
  Env_LoadGetTknCode(Point_GetToken);
}
```

```
Env_LoadGetTknCode(void * (*CodePtr)())
{
  ShellLoadTokenCode(CodePtr);
}
```

```
ShellLoadTokenCode(void * (* CodeAddr)())
{
  Mem_PutFuncPtr(CodeAddr);
}
```

```
Mem_PutFuncPtr(void * (* FuncPtr)())
{
  Ut_MoveBytes(&FuncPtr, Theory);
}
```

```
C_RefToTkn()
{
  tokentype (*pmf1)();

  ShellGetTokenCode(&pmf1);
  (*pmf1)();
}
```

```
ShellGetTokenCode(void * (** CodeAddr)())
{
  Mem_GetFuncPtr(CodeAddr);
}
```

```
Mem_GetFuncPtr(void * (** FuncPtr)())
{
  Ut_MoveBytes(Theory, FuncPtr);
}
```

```
Ut_MoveBytes(void * Source, void *  Target)
{
  memcpy(Target, Source, 4);
}
```

Figure 6.4 Partial call graph of *147.vortex*.

bottom-up traversal of the call graph and the phase II' analysis is performed in the top-down traversal of the call graph.

Benchmark *147.vortex* basically shares the same features as *132.ijpeg*. In addition, instead of using explicit assignments to propagate function addresses, *147.vortex* is a real example that uses library call `memcpy` as an alternative approach for pointer assignments. In Figure 6.4, function `Point_InitClass` [3] passes the address of function `Point_GetToken` as a parameter all the way down to function `Ut_MoveBytes`, which invokes `memcpy` to store `Point_GetToken` into an anonymous object retrievable from global variable `Theory`. In function `C_RefToTkn` which contains an indirect call-site, function pointer `pmf1` is initialized by passing its address to function `ShellGetTokenCode`, which

[3]These function names are extracted after macro expansions.

Table 6.1 Statistics of function pointers in SPEC benchmarks.

Benchmark	Characteristics					
	Lines	Functions	Dead Func.	Dir. Call	Ind. Call	& Function
008.espresso	14838	361	46	2674	15	12
023.eqntott	12053	62	2	358	11	5
072.sc	8639	179	8	1459	2	20
085.cc1	90857	1452	51	8332	67	588
124.m88ksim	19092	252	13	1496	3	57
126.gcc	205583	2019	187	19731	132	229
130.li	7597	357	1	1267	4	190
132.ijpeg	29290	477	16	1016	641	188
134.perl	26874	276	13	4367	3	3
147.vortex	67205	923	295	8521	15	44

eventually invokes `memcpy` to retrieve the address of function `Point_GetToken`. Since the transfer functions of library routines are also analyzed in the interprocedural pointer analysis stage, the case represented by *147.vortex* can be handled properly.

6.2 Call-Graph Resolution of SPEC Benchmarks

Table 6.1 lists the size information and the usage of function pointers of the studied SPEC benchmarks. For each benchmark, *Lines*, *Functions*, and *Dead Func.* show the number of lines, functions, and unreachable functions in each program. The *Dir. Call*, *Ind. Call*, and *& Function* columns show the number of direct call-sites, indirect call-sites, and functions whose addresses are taken. Unreachable functions can be found in large programs developed by multiple programmers over time, or found in programs after extensive code transformations like function inlining performed by the compiler.

Table 6.2 Resolution of function pointers.

Bench-	Simple			Parameter					Global					Heap				
mark	0	1	2	0	1	2	3	≥ 4	0	1	2	3	≥ 4	0	1	2	3	≥ 4
008.espresso	0	0	0	0	2	3	0	10	0	0	0	0	0	0	0	0	0	0
023.eqntott	0	0	0	0	2	0	9	0	0	0	0	0	0	0	0	0	0	0
072.sc	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
085.cc1	0	0	15	0	0	0	0	19	0	0	0	0	32	0	3	0	0	0
124.m88ksim	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0
126.gcc	4	11	16	0	3	0	0	14	1	1	0	0	73	0	9	0	0	0
130.li	0	0	0	0	0	0	1	0	0	0	0	0	3	0	0	0	0	0
132.ijpeg	0	0	0	0	0	0	0	0	0	0	0	0	0	4	381	16	11	15
134.perl	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0
147.vortex	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1	2	0	2

Unreachable functions can be determined after all function pointers are resolved: for functions that are not connected to the call graph starting from `main` or not registered through signal handling routines, they can be safely removed from the program. This optimization can improve both the compilation time and binary size. The resolution of a worst-case call-graph extractor is bound by the product of *Ind. Call* and *& Function*, which will be very large for benchmarks like *132.ijpeg* and *085.cc1*. The resolution of the call graph using the proposed interprocedural pointer analysis algorithm is shown in Table 6.2.

In Table 6.2, number n at cell *(a, b)* represents that in benchmark a, n instances of function pointers are resolved to b callees. Data presented in this table are obtained before function inlining is performed. The ideal case is to resolve a function pointer into one callee, since the indirect call-site can be converted into a direct one, so that the overhead associated with indirect function calls can be eliminated. If the compiler

Table 6.3 Uninitialized indirect call sites of SPEC benchmarks found by interprocedural pointer analysis.

Benchmark	Uninitialized function pointers (file/function/call-site)
126.gcc	function.c/push_function_context/*save_machine_status function.c/pop_function_context/*save_machine_status sched.c/actual_hazard_this_instance/function_units[unit].blockage_function
132.ijpeg	jcapi.c/jpeg_write_scanlines/*cinfo→progress→progress_monitor jdapi.c/jpeg_read_scanlines/*cinfo→progress→progress_monitor jcapi.c/jpeg_finish_compress/*cinfo→progress→progress_monitor jdapi.c/jpeg_start_decompress/*cinfo→progress→progress_monitor
134.perl	stab.c/stab_str/*uf→uf_val stab.c/stabset/*uf→uf_set doarg.c/do_subr/*sub→usersub
147.vortex	trans01.c/C_CreateObject/*pmf1

supports inlining, the callee can even be inlined. As the table shows, benchmark *132.ijpeg* would benefit most from the converted calls: 381 out of 427 indirect call-sites can be transformed into direct ones [4]. But on the other hand, there are 4 indirect call-sites in *130.li*, with 190 functions whose addresses are taken. The resolved pointers indicate that 3 indirect call-sites can invoke 187 possible callees, while the other indirect call-site can invoke 3 possible callees. Even in this case, a more accurate estimate of the indirect call-site's side-effects can be obtained since only a subset of the functions in the whole program can be reached.

Table 6.2 also indicates that heap-based indirect call-sites tend to have a small number of resolved callees, where parameter and statically initialized global call-sites have more callees. The situation for parameter call-sites can be improved by performing function

[4]There are 214 function pointers found in dead functions and excluded.

cloning or inlining, at the cost of increased code size. Table 6.2 also shows that some benchmarks have unresolved function pointers. To make sure that they are caused by flaws of the interprocedural pointer analysis algorithm, these function pointers are verified by tracing the program with the system debugger. It is found that these pointers are indeed uninitialized and not exercised with multiple profile inputs. Detailed descriptions for these pointers can be found in Table 6.3.

CHAPTER 7

EXPERIMENTAL RESULTS

In this dissertation, a wide range of experiments were conducted to evaluate various aspects of the presented static memory disambiguation framework. First, the resource requirements of the modular interprocedural pointer analysis algorithm are reported, including the analysis time and memory usage. To demonstrate the feasibility of the proposed algorithm, the SPEC suites are analyzed, which are much larger than the common benchmarks analyzed by previous work. In addition, a personal computer equipped with an Intel Pentium II processor running at 450 MHz with 256MB of RAM instead of a high-end workstation is adopted to be the experimental platform. 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. Secondly, the accuracy of the modular interprocedural pointer analysis algorithm is studied. Since flow-sensitivity is traded for lower complexity based on the argument that pointers to acyclic data structures are rarely redefined, quantitative statistics are provided to back up the argument. Finally, a set of quantitative studies of performance improvements enabled by static memory disambiguation are presented, incorporating the optimization techniques discussed in Chapter 5. Performance data are obtained on an emulated processor with parameter-controlled issue width, numbers of functional units, and memory

Table 7.1 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

system configurations. Programs represented in the *Lcode* IR can be emulated by reverse translated C code, whose execution is used to gather the profile information and guide the simulation process.

The following sections begin with quantitative results followed by some real examples showing the strength and weakness of the proposed modular interprocedural pointer analysis.

7.1 Analysis Costs

In this section, the analysis costs for the SPECcint92 and SPECcint95 benchmarks are reported. As listed in Table 7.1, some general characteristics of each benchmark are shown first, 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. As discussed in Chapter 6, removing dead functions can speedup the compilation process and improve the run-time performance. In addition to inherited dead functions in the program, after function inlining with up to 60% of code increase, more dead functions are found in many benchmarks [1]. 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

[1]The number of dead functions shown in Table 6.1 reflects the original code prior to inlining.

Table 7.2 Resource requirements of conducting interprocedural pointer analysis.

Benchmark	Intraprocedural stage	Interprocedural stage		Total time
	Time (Sec)	Time (Sec)	Mem. (MB)	(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

SCCs. The proposed modular analysis technique in Chapter 4 can reduce the memory usage in analyzing a single SCC since only the summary behavior of each function needs to be maintained.

Table 7.2 shows the resource consumption statistics collected from the PC-based experimental environment. 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 7.2.

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 7.2, 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 sections, it will be shown that the new algorithm is also accurate and can provide significant performance improvements.

7.2 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

Table 7.3 Resolutions of interprocedural pointer analysis.

Benchmark	Targets per pointer					Pointer per target				
	1	2	3	≥ 4	Avg.	1	2	3	≥ 4	Avg.
008.espresso	3071	436	40	24	1.16	1341	329	253	282	1.76
023.eqntott	310	115	0	7	1.31	165	30	31	37	1.77
026.compress	34	1	1	0	1.08	21	6	2	2	1.52
072.sc	662	46	13	6	1.12	291	88	36	40	1.62
085.cc1	9231	1659	384	189	1.26	2527	617	607	980	2.01
099.go	312	0	0	0	1.00	159	24	5	21	1.46
124.m88ksim	622	49	3	15	1.15	562	88	35	16	1.29
126.gcc	16147	3679	705	451	1.31	4481	1092	795	1852	2.00
129.compress	37	2	0	0	1.05	36	9	0	0	1.20
130.li	1576	243	50	4	1.19	588	239	58	126	1.73
132.ijpeg	3897	766	135	23	1.23	3166	829	201	144	1.38
134.perl	2873	691	340	435	1.62	1205	477	206	543	2.04
147.vortex	6768	121	41	5	1.03	4848	671	214	347	1.35

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 7.3 shows the statistics of the target-per-pointer metric and the pointer-per-target metric observed on right-most 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 the target-per-pointer columns reflect the numbers of outgoing points-to relations carried by pointer-type access paths, where numbers in the pointer-per-target 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 comparison between flow-sensitive and insensitive pointer analysis algorithms is to conduct both analyses on the same programs. Via the statistics gathered regarding resource requirements, pointer resolutions and performance improvements, the pros and cons of both methods can be obtained. 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 dissertation.

Figure 7.1 classifies pointer definitions found in each program into 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

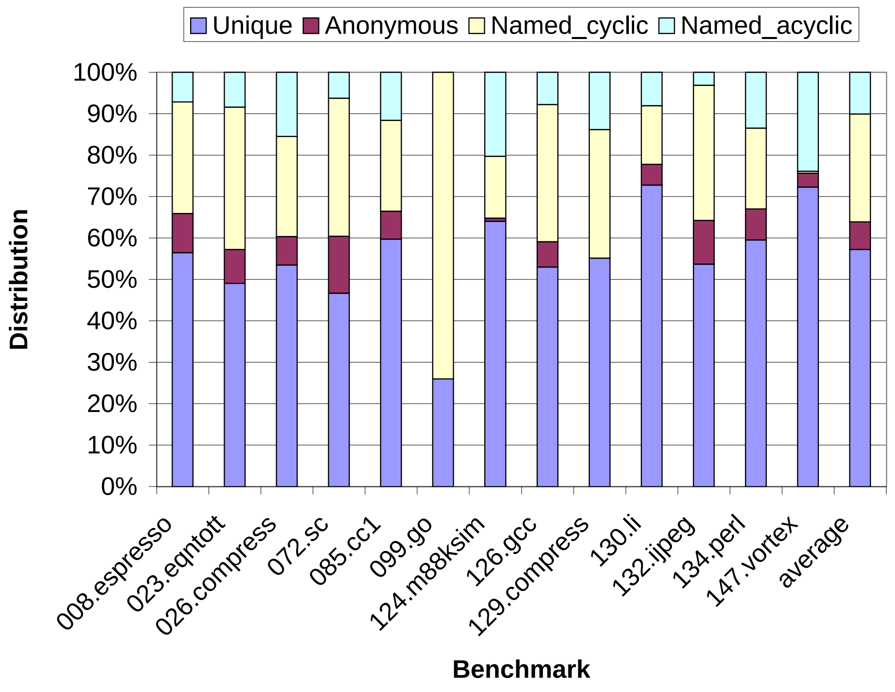


Figure 7.1 Distribution of pointer definitions.

unique category as well. As shown in Figure 7.1, 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 re-

Table 7.4 Processor configurations.

	Issue	Mem	IALU	FALU	Branch	Mem. Latency (L1/L2/main)
1	4	2	2	2	2	2/4/100
2	6	3	3	3	2	2/4/100
3	8	4	4	4	2	2/4/100

solved 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.

7.3 Performance Improvements

In order to understand the potential benefits of memory access optimizations for future processors, the optimized programs are evaluated on a set of simulated processors to obtain the performance data. As shown in Table 7.4, these simulated processors can fetch, decode, and issue from 4 to 8 instructions per cycle in order. The numbers of ALUs, memory ports, and floating point ALUs are proportional to the issue width of the processor, where the numbers of branch units, integer and floating point registers are fixed at 2 and 64, respectively. 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 [57].

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 [2]. 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 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 as described in Chapter 5. Side-effects of all functions can be accurately modeled in this version. Except for the differences mentioned

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

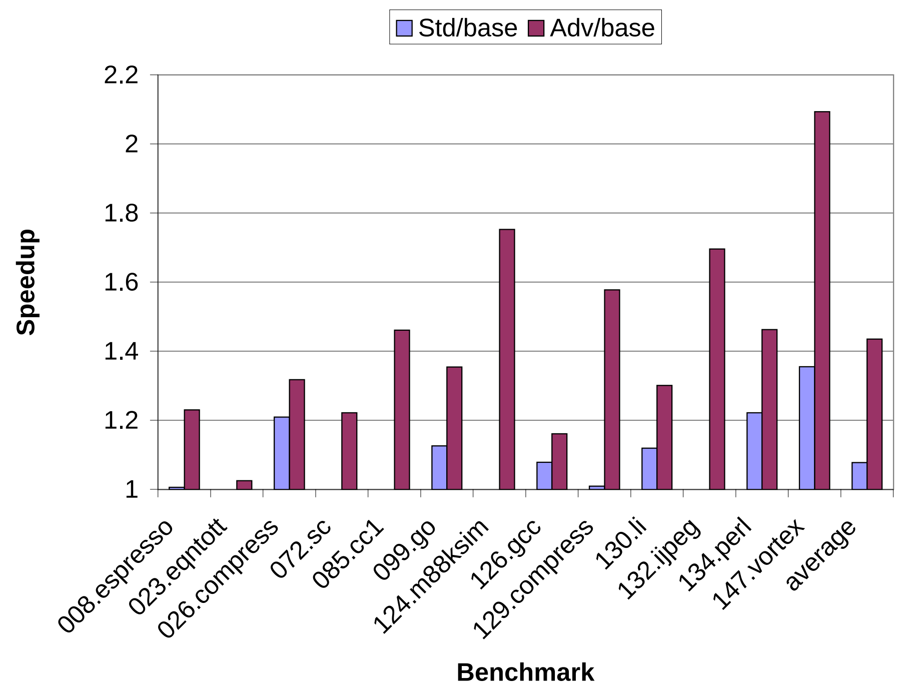


Figure 7.2 Performance improvements enabled by memory disambiguation on an 8-issue processor.

above, all three versions are compiled through inlining up to 60% of code increase [56], classical optimizations [58], and superblock optimization [59]. The training input sets are used for all levels of experiments in this dissertation.

Figure 7.2 plots the speedups in execution time of the *standard* and *advanced* versions over the base version observed on the 8-issue processor configuration as shown in configuration 3 of Table 7.4. 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 the advanced memory

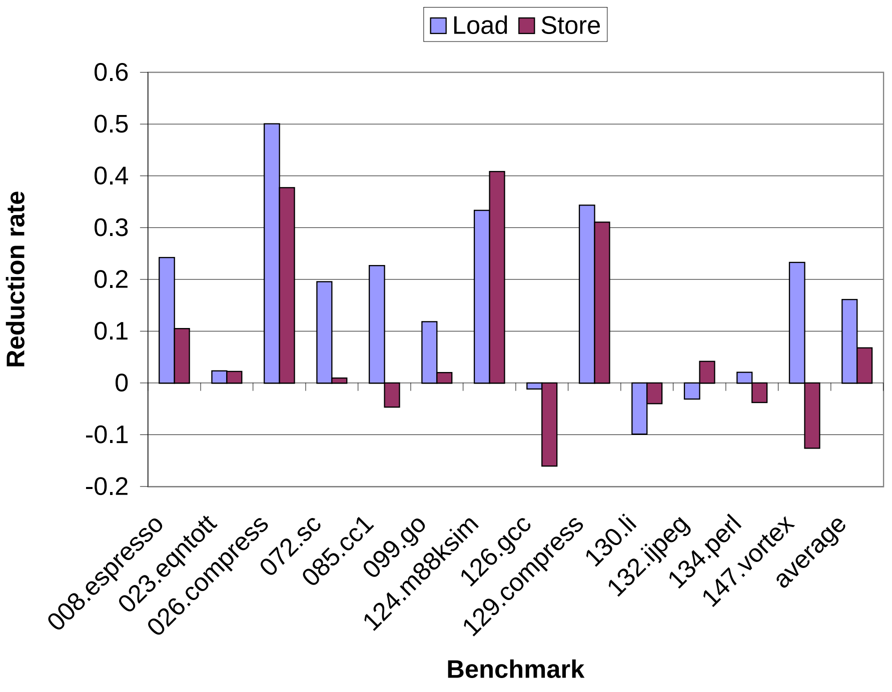


Figure 7.3 Reduction rates of load and store instructions.

disambiguation techniques discussed in this dissertation, 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 memory disambiguation, the run-time behaviors of the standard and advanced versions of code are further studied. As discussed in Chapter 5, promoting values from the memory to the register is the most effective technique for the compiler to optimize memory accesses, and load/store scheduling can be used to effectively hide the latency of remaining loads. However, it is hard to derive their exact contributions in the final performance improvement since their effects may overlap in some cases. For example, the latency of a redundant but remaining load may be fully hidden by the scheduler. Therefore, this dissertation approximates each optimization's contribution by: (i) showing the amount of reduction of dynamic loads and stores, and (ii) showing the performance improvement achieved without aggressive load/store scheduling. The left bar of each benchmark in Figure 7.3 shows the reduction of dynamic loads between the standard and advanced versions. As these numbers show, using simple copy propagation to convert ambiguous relations to dependent relations can eliminate as much as 40% of dynamic loads for *026.compress*, 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 7.3 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

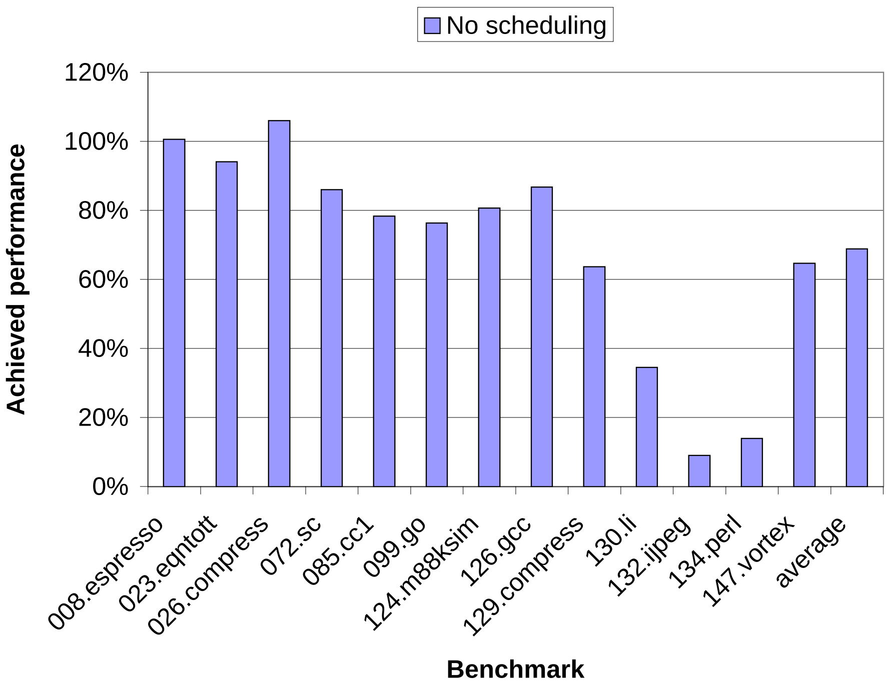
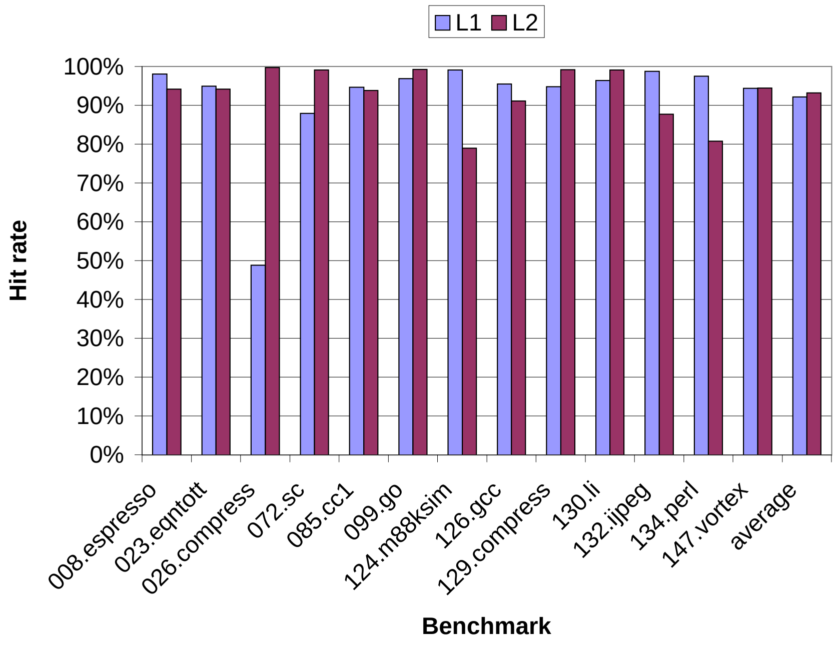


Figure 7.4 Ratio of the performance improvements enabled by scheduling only versus full memory access optimizations.

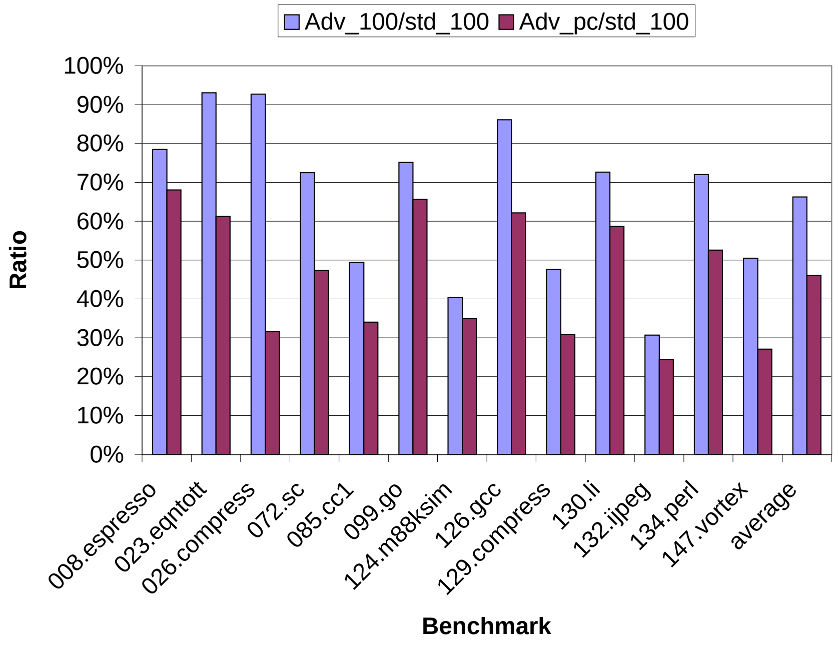
be discarded. The second reason is the increased register pressure due to more memory contents are held in registers, which may increase both the numbers of load and store instructions because of register spilling and refilling. As shown in Figure 7.2, significant performance improvements are still obtained even though there are extra memory instructions.

Figure 7.4 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 the use of sync arcs in guiding instruction scheduling. Comparing Figure 7.3 and 7.4, 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.

In modern processors, the L1 cache latency is usually more than one cycle and the main memory latency can easily approach 100 cycles. The effectiveness of redundant memory access elimination and loop-invariant memory access migration is less sensitive to the memory latency since the memory instructions are eliminated. In fact, the longer the memory latency is, the better the performance improvement. However, instruction scheduling is more sensitive to the memory latency since scheduling is usually performed under the assumption of a uniformed memory latency, which is the L1 cache hit latency. If both L1 and L2 caches suffer misses, given the memory latency of 100 cycles, it is hard for the compiler to schedule the load instruction far enough to tolerate the memory latency even in the presence of accurate memory disambiguation information. Figure 7.5 studies the relation between the impact of static memory disambiguation and extreme memory latencies. First, Figure 7.5a shows the L1 and L2 cache hit ratios for the fully disambiguated and optimized programs on the 8-issue processor. Figure 7.5b shows the



(a)



(b)

Figure 7.5 The memory latency factor: (a) L1 and L2 hit ratios, (b) cycles blocked due to pending loads.

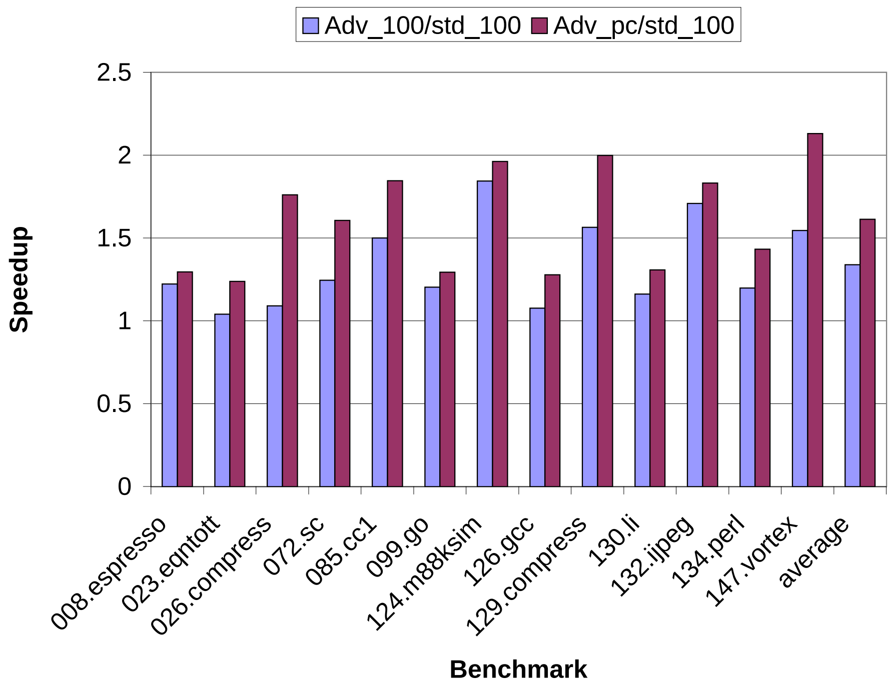


Figure 7.6 Performance improvements on a 8-issue processor with perfect data caches.

relative percentage of cycles blocked due to a pending load instruction. The 100% scale reflects the load-stalled cycles in the *standard* version, the left bar is the normalized stall ratio of the *advanced* version given the memory latency to be 100 cycle, and the right bar reflects the normalized stall ratio under a perfect L1 data cache. The left bar in Figure 7.6 shows the relative performance of the 100-cycle main memory latency configuration over the standard version, while the right bar shows the relative performance of the perfect cache configuration over the same base performance. There are three major observations from Figure 7.5 and Figure 7.6:

1. The L1 and L2 cache hit ratios are pretty high, meaning the main memory latency is rarely experienced by load instructions.

2. Around 32% of load-stalled cycles are eliminated by fully disambiguated memory accesses, where a perfect cache can eliminate 20% more stalled cycles.

3. A perfect cache provides around 30% more speedups, which can be considered as the upper boundary condition to interpolate the performance improvements enabled by static memory disambiguation for memory latencies between 0 and 100 cycles. Benchmark *026.compress* benefits the most from perfect caches due to its low L1 cache hit rate. Benchmark *147.vortex* is the second largest beneficiary due to most of its cache misses happen on the critical path of during its execution.

As shown in Figure 7.5b, static memory disambiguation can reduce the wasted cycles due to pending load instructions. Therefore it is interesting to see how they are translated to better utilization of increased machine resources. For each benchmark, the three processor configurations as summarized in Table 7.4 are studied. For each processor configuration, the *standard* version and the *advanced* version of programs are simulated to evaluate the performance variance. In Figure 7.7, each benchmark has 5 bars where each bar shows the relative speedup to the 4-issue processor without interprocedural pointer analysis. As the trend shows, without memory disambiguation, the compiler cannot effectively utilize the machine resources. As a result, the trend of the speedups is flat for the *standard* versions. On the other hand, the performance of programs optimized with interprocedural procedural pointer analysis keep climbing higher as the machine has

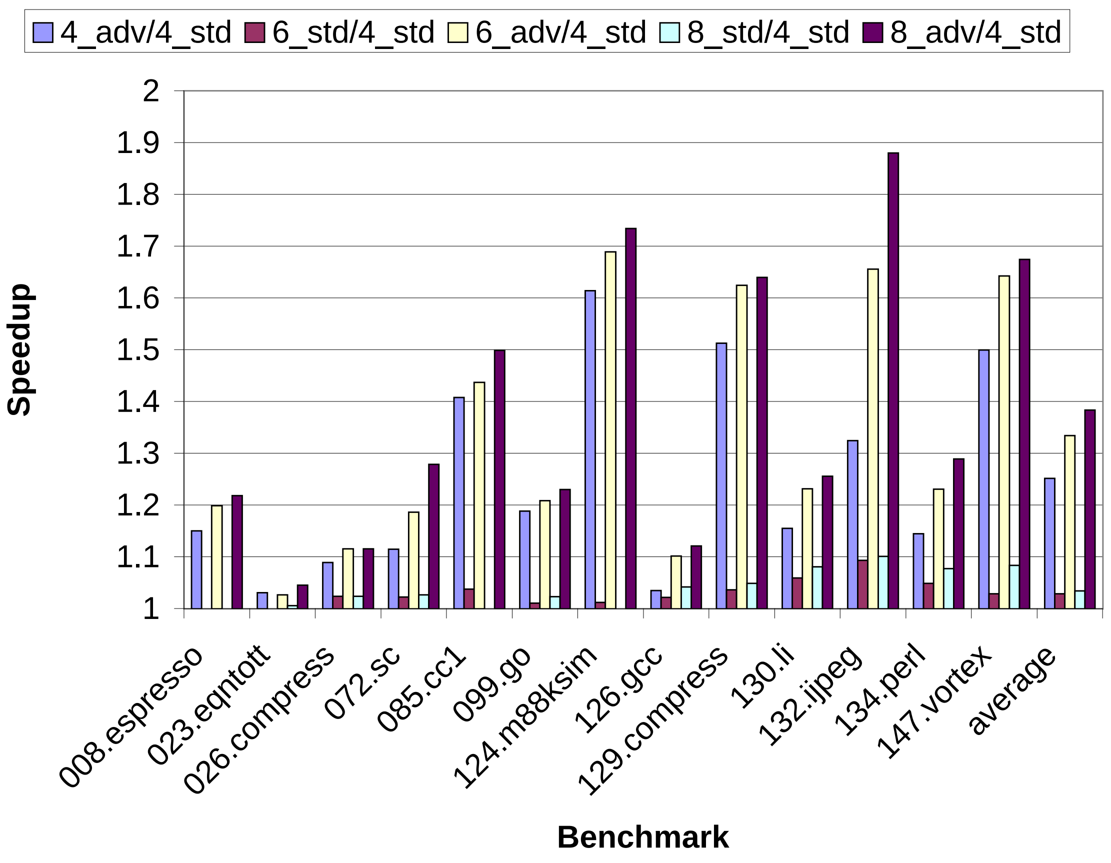


Figure 7.7 Effectiveness of static memory disambiguation with increased machine resources.

more resources. Benchmark *132.ijpeg* shows the ideal case where the performance speedup is improved from 1.32 to 1.67 to 1.88 when the processor issues 4, 6, or 8 instructions per cycle. Figure 7.7 demonstrates that the proposed memory disambiguation framework is able to provide scalable performance improvements.

It is also interesting to see how well the proposed static memory disambiguation does compared to a perfect pointer analysis algorithm, together with the ideal usage of the dy-

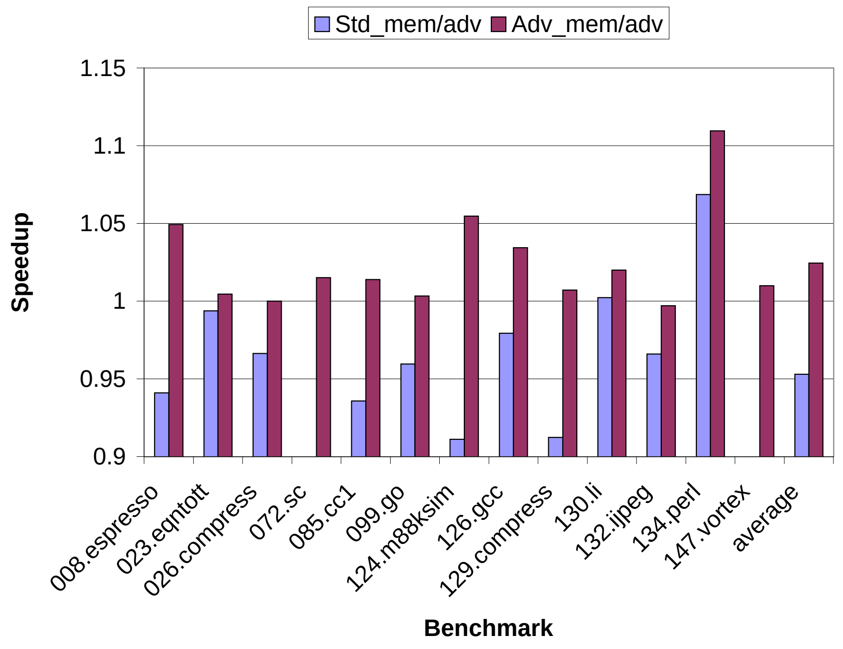


Figure 7.8 The effectiveness of static memory disambiuation versus perfect data speculation.

namic memory disambiguation mechanism incorporated in the IA-64 architecture [9, 10]. Therefore memory address profiling [15] is conducted to generate perfect sync arcs which will allow more optimization and scheduling opportunities which could have been disabled by spurious pointer analysis results. That is, as long as the input to a benchmark is fixed, a new version of binary called the *profiled* version which is optimized under the guidance of profile-generated sync arcs can be generated. When memory address profiling is conducted on the *advanced* version of the program, the performance number reflects the additional improvement that can be obtained from instruction scheduling given perfect memory disambiguation information. On the other hand, when memory address profiling is conducted on the *standard* version of the program, the performance number reflects the performance improvement that can be obtained from solely dynamic memory disambiguation. With statistics collected from the 8-issue processor configuration, for each benchmark in Figure 7.8, the left bar shows the relative performance between the *advanced* version and the profile-guided *standard* version, while the right bar shows the relative performance between the profile-guided *advanced* version and the *advanced* version. The average speedup for the left bar is 0.96, where it is as low as 0.9 for *072.sc* and as high as 1.07 for *134.perl*. On the other hand, the average speedup of the right bar is 1.03, with the maximum of 1.12 from *134.perl*. Two main observations from Figure 7.8 are:

1. Generally speaking, the proposed interprocedural procedural pointer analysis and compile-time memory optimizations outperform perfect run-time data speculation due to the benefit of redundant load/store elimination and loop-invariant access migration. If data speculation is not guided by perfect knowledge like memory access profiling, the benefit of data speculation is expected to be much lower due to mis-speculation penalties.

2. The classification of ambiguous/independent relations between memory accesses provided by the proposed interprocedural pointer analysis algorithm is accurate, since only 3% more performance can be exploited by the scheduler when spurious dependent relations are removed.

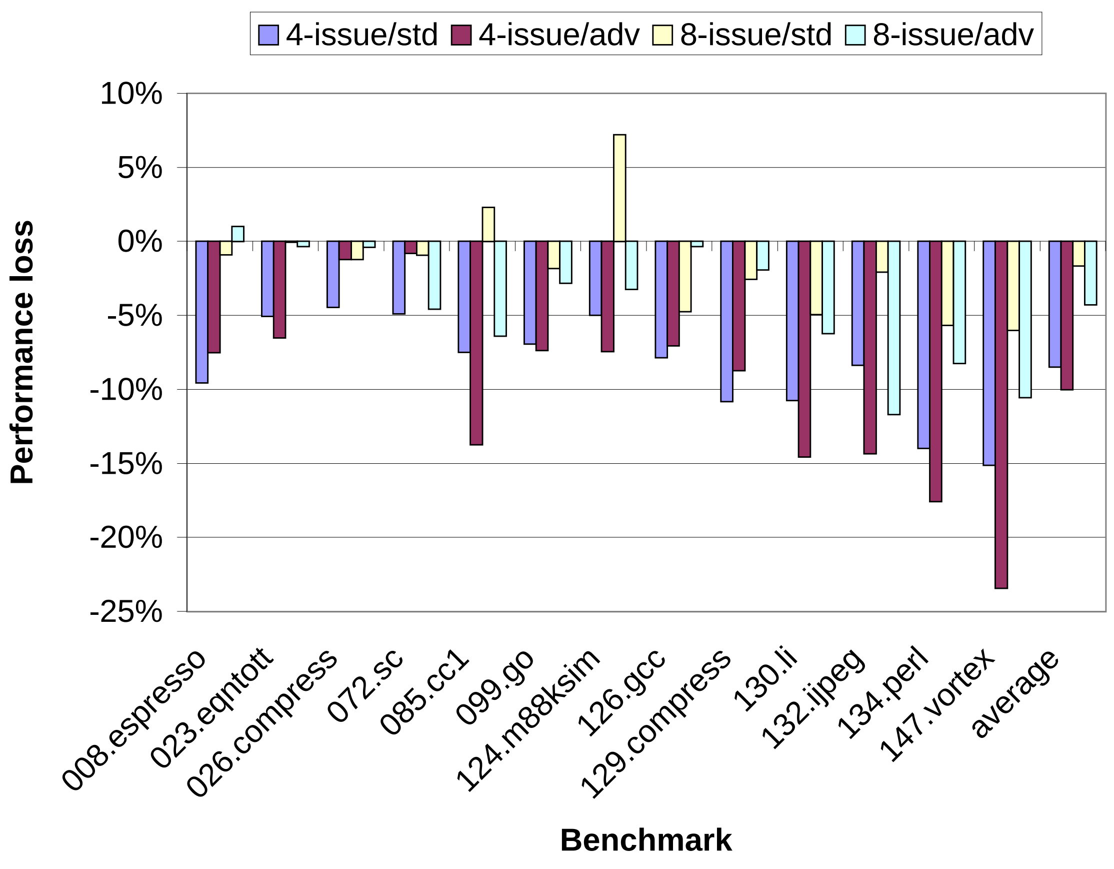


Figure 7.9 Performance losses due to reduced machine resources.

Benchmark *134.perl* indicates that there are more than 10% performance losses caused by spurious memory disambiguation information. Together with examples extracted from other benchmarks, the strength and weakness of the proposed compile-time memory disambiguation technique will be discussed in the next section.

Due to various design restrictions like the circuitry complexity and power consumption, it may not be possible to afford abundant memory ports to fully harvest the results of static memory disambiguation. Therefore another experiment is conducted to understand the performance loss due to reduced memory resources. As shown in Figure 7.9, each bar shows the performance loss of the specified processor configuration with the specified sophistication of memory disambiguation when the number of memory port is cut by 50% compared to the original specification listed in Table 7.4. Without advanced memory disambiguation, most loads and stores are executed in a serialized manner, therefore reducing the number of memory ports by half has less performance impact than the case with interprocedural pointer analysis, where more loads and stores could be executed in parallel. Because only one memory port is employed by the 4-issue processor configuration, it suffers more performance loss than the 8-issue processor case. The average negative effect on the 8-issue case with interprocedural pointer analysis is 4.5%, which is very small compared to the enabled performance gain. Benchmarks *008.espresso*, *085.cc1*, and *124.m88ksim* actually observe some performance improvements due to streamlined memory resources. The reason is fewer speculative loads are executed when the scheduler sees more memory port contentions, resulting fewer L1 data cache misses. From the observation of Figure 7.9, the majority of benefits provided by the proposed static memory disambiguation framework can be achieved even when the memory resources are reduced.

7.4 Case Studies

This section will proceed by case studies showing the significant benefits of register promotion and scheduling, followed by programming features that harden accurate static memory disambiguation. The goal is to provide a more solid understanding of the strength and weakness of the modular interprocedural pointer analysis algorithm using real code examples.

7.4.1 Register promotion

Figure 7.10 lists the source code of function `alignd` from benchmark *124.m88ksim*. The function is used to align 64-bit floating point numbers. Because the indirect memory accesses in the loop involve pointer-type parameters, the compiler is not able to determine their independence without interprocedural pointer analysis and therefore leaves the loop body conservatively unoptimized. With the modular interprocedural pointer analysis proposed in this thesis, the compiler is able to determine that `*s`, `*bmantlo`, `*bmanthi`, and `*amanthi` all stand for independent and loop-invariant locations, so they can be completely migrated out of the loop. As a result, the loop body which originally contains 7 loads and 4 stores now only contain arithmetic instructions after performing loop-invariant access migration. The profiling weight indicates that the loop executes for more than 1 million times. After this optimization, the length of the critical path drops from 18 cycles to 3 cycles on an 8-issue processor, eliminating more than 15 million cycles in total during the execution of the program.

7.4.2 Scheduling

Figure 7.11 lists the source code of function `rgb_ycc_convert` from benchmark *132.ijpeg*. Similar to the example in *124.m88ksim*, the `for`-loop in `rgb_ycc_convert`

```
void alignd(int aexp,ULONG *amanthi,ULONG *amantlo,int bexp,
ULONG *bmanthi,ULONG *bmantlo,int *resexp,ULONG *s)
{
   int expdiff;
   expdiff = aexp - bexp;
   *amanthi <<= 2;
   *amanthi |= *amantlo>>30;
   *amantlo <<= 2;
   *bmanthi <<= 2;
   *bmanthi |= *bmantlo>>30;
   *bmantlo <<= 2;
   if (expdiff >= 0) {
      for (*s = 0 ; expdiff > 0 ; expdiff--) {
         *s |= *bmantlo & 1;
         *bmantlo >>= 1;
         *bmantlo |= *bmanthi<<31;
         *bmanthi >>= 1;
      }
      *resexp = aexp;
   }
   else {
      expdiff = -expdiff;
      for (*s = 0 ; expdiff > 0 ; expdiff--) {
         *s |= *amantlo & 1;
         *amantlo >>= 1;
         *amantlo |= *amanthi<<31;
         *amanthi >>= 1;
      }
      *resexp = bexp;
   }
}
```

Figure 7.10 Code example extracted from *124.m88ksim*.

```
static  void
rgb_ycc_convert (j_compress_ptr cinfo,
                 JSAMPARRAY input_buf, JSAMPIMAGE output_buf,
                 JDIMENSION output_row, int num_rows)
{
  my_cconvert_ptr cconvert = (my_cconvert_ptr) cinfo->cconvert;
  register int r, g, b;
  register INT32 * ctab = cconvert->rgb_ycc_tab;
  register JSAMPROW inptr;
  register JSAMPROW outptr0, outptr1, outptr2;
  register JDIMENSION col;
  JDIMENSION num_cols = cinfo->image_width;

  while (--num_rows >= 0) {
    inptr = *input_buf++;
    outptr0 = output_buf[0][output_row];
    outptr1 = output_buf[1][output_row];
    outptr2 = output_buf[2][output_row];
    output_row++;
    for (col = 0; col < num_cols; col++) {
      r = ((int) ( inptr[0 ] )) ;
      g = ((int) ( inptr[1 ] )) ;
      b = ((int) ( inptr[2 ] )) ;
      inptr += 3 ;
      outptr0[col] = (JSAMPLE)
        ((ctab[r+0]+ctab[g+(1*(255 +1))]+ctab[b+(2*(255+1))])>>16);
      outptr1[col] = (JSAMPLE)
        ((ctab[r+(3*(255+1))]+ctab[g+(4*(255+1))]+
          ctab[b+(5*(255+1))])>>16);
      outptr2[col] = (JSAMPLE)
        ((ctab[r+(5*(255+1))]+ctab[g+(6*(255+1))]+
          ctab[b+(7*(255+1))])>>16);
    }
  }
}
```

Figure 7.11 Code example extracted from *132.ijpeg*.

contains indirect accesses involving pointer-type parameters. For example, in the `for`-loop the first access expression, `inptr[0]`, dereferences the second pointer parameter, `input_buf`, by two levels. Also in the `for`-loop, the last store expression, `outptr2[col]`, modifies the memory location accessed through three levels of indirection via the third pointer parameter, `output_buf`. The critical path of the `for`-loop takes 23 cycles. After unrolling the loop by 4 times, due to the ambiguous relation between the first load of the next iteration and the last store of the previous iteration, these unrolled loop bodies cannot be mixed at all, resulting a critical path of 92 cycles. With interprocedural pointer analysis, the compiler can prove that all the indirect memory accesses are independent, enabling the scheduler to place load instructions before store store instructions to hide the load latency. Even though the total number of memory instructions contained in the loop body remains the same, the scheduler is able to reduce the length of the critical path down to 30 cycles. The profiling weight indicates that the loop is executed for 1 million times, meaning a total reduction of 60 million cycles for the total execution time by the scheduler.

7.4.3 Spurious memory disambiguation

Unlike the previous two cases, Figure 7.12 shows a negative example which exposes some weakness of the proposed interprocedural pointer analysis algorithm. In Figure 7.12, function `do_splice` contains a `for`-loop inside which two pointers `src` and `dst` point to the same array but with an offset of `diff`. Since the conducted pointer analysis treats the whole array as a single object, `*src` and `*dst` are considered as ambiguous memory

```
do_splice(ary,gimme,arglast)
register ARRAY *ary;
int gimme;
int *arglast;
{
    register STR **st = stack->ary_array;
    register int sp = arglast[1];
    int max = arglast[2] + 1;
    register STR **src;
    register STR **dst;

                 :
    newlen = max - sp;
    diff = newlen - length;
                 :
    src = &ary->ary_array[offset-1];
    dst = src - diff;
    for (i = offset; i > 0; i--)
        *dst-- = *src--;
                 :
}
```

Figure 7.12 Code example extracted from *134.perl*.

accesses, preventing both redundant access elimination and scheduling. Unless the offset `diff` is replaced by a literal, this situation cannot be helped by a flow-sensitive algorithm since the content of `diff` may be zero. With memory access profiling, it is detected that `diff` is always nonzero, allowing the load of `*src` to be scheduled before the store of `*dst` when the loop is unrolled. Therefore the length of the critical path is reduced from 12 cycles to 7 cycles. The profiling weight shows that the loop is executed for 9 million times, therefore perfect memory disambiguation can further reduce 45 million cycles. With well-defined compile-time heuristics, it is possible to use run-time data speculation to seek optimization opportunities on cases like this code example. However, it is beyond the scope of this dissertation.

CHAPTER 8

CONCLUSIONS

8.1 Summary and Contribution

The need for advanced compile-time memory disambiguation techniques is more and more imminent as the advancement in the processor keeps outpacing the improvement in the memory subsystem. This thesis has presented a practical interprocedural pointer analysis algorithm which uses the modular feature to tolerate the resource requirements. Unlike previously proposed algorithms which either handle only a subset of language features or only small programs, this dissertation presents a comprehensive method to handle all C constructs and significant programs like the SPECcint92 and SPECcint95 benchmark suites.

This dissertation also presents three effective optimization techniques to utilize memory disambiguation information including scheduling, redundant memory access elimination, and loop-invariant access migration. The validated results through simulation not only demonstrate the correctness of the conducted analysis and optimization, but also show that significant performance improvements can be obtained. The case studies of the usage of function pointers not only provide concrete examples to help understand pointer manipulations, but also provide valuable real examples for future algorithms to verify.

The case studies about the strength and weakness of static memory disambiguation also provide concrete examples to understand the capability of the proposed flow-insensitive and context-sensitive pointer analysis algorithm.

8.2 Future Work

The experiment results demonstrate that the proposed static memory disambiguation framework is very feasible and can provide accurate results and significant performance improvements on the SPEC-class programs. Besides the encouraging results, there are several open issues for future research.

With the upcoming IA-64 architecture which supports hardware-assisted data speculation, it provides a viable run-time alternative for memory disambiguation. From the studies conducted in this dissertation, the perfect use of data speculation can provide performance improvements close to the level provided by the full scale interprocedural pointer analysis. With the perfect use of data speculation on top of static memory disambiguation, additional performance benefits may be obtained. Therefore an interesting question is whether both techniques can be used constructively together. Memory access profiling better fits as a tool to understand the upper-bound of the effectiveness of the proposed static memory disambiguation framework than as guidances for data speculation due to its expense to perform. So heuristics that can judge the confidence of static dependence classifications are valuable because if they are accurate, both static and dynamic memory disambiguation techniques can be integrated together to explore the ultimate performance.

Although the size and intricacy of the SPEC benchmarks post an important milestone towards the practical use of static memory disambiguation, their sizes are still a fraction of those of many real world applications. There may very well be many cases that are handled too conservatively by the proposed modular interprocedural pointer analysis algorithm. The proposed redundant load/store elimination, loop-invariant access migration, and scheduling techniques can be applied to any static memory disambiguation framework as long as the *sync arcs* representation is also adopted. Together with the demonstrated low resource consumptions, the proposed memory disambiguation framework has a great potential to be ported into a product compiler. Therefore it would be extremely valuable to learn if the modular feature can keep scaling well to large applications.

REFERENCES

[1] T. M. Austin and G. S. Sohi, "Zero-cycle loads: Microarchitecture support for reducing load latency," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, December 1995, pp. 82–92.

[2] C. Chen and A. Wu, "Microarchitecture support for improving the performance of load target prediction," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997, pp. 228–234.

[3] R. J. Eickemeyer and S. Vassiliadis, "A load-instruction unit for pipelined processors," *IBM Journal of Research and Development*, vol. 27, pp. 547–564, July 1993.

[4] M. Golden and T. N. Mudge, "Hardware support for hiding cache latency," University of Michigan, tech. rep., February 1993.

[5] J. Gonzalez and A. Gonzalez, "Speculative execution via address prediction and data prefetching," in *Proceedings of the 1997 International Conference on Supercomputing*, July 1997, pp. 196–203.

[6] B. Cheng, D. A. Connors, and W. W. Hwu, "Compiler-directed early load-address generation," in *Proceedings of the 31st Annual International Symposium on Microarchitecture*, December 1998.

[7] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," in *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 226–237.

[8] G. S. Tyson and T. M. Austin, "Improving the accuracy and performance of memory communication through renaming," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997, pp. 218–227.

[9] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu, "Dynamic memory disambiguation using the memory conflict buffer," in *Proceedings of 6th International Conference on Architectual Support for Programming Languages and Operating Systems*, October 1994, pp. 183–193.

[10] Intel Corporation, Santa Clara, CA, *IA-64 Application Developer's Architecture Guide*, May 1999.

[11] "Spec newsletter." http://www.spec.org.

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

[13] 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., Springer-Verlag, 1995, pp. 234–250. Proceedings from the 7th Workshop on Languages and Compilers for Parallel Computing.

[14] S. Horwitz, "Precise flow-insensitive may-alias analysis is NP-hard," *ACM Transactions on Programming Languages and Systems*, vol. 19, January 1997.

[15] D. A. Connors, "Memory profiling for directing data speculative optimizations and scheduling," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1997.

[16] 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*, June 1988, pp. 241–249.

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

[18] 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.

[19] 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*, June 1994, pp. 242–256.

[20] 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*, June 1995, pp. 1–12.

[21] 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.

[22] 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*, June 1992, pp. 235–248.

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

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

[25] 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*, June 1994, pp. 230–241.

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

[27] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, DIKU, University of Copenhagen, May 1994.

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

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

[30] 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*, June 1997, pp. 308–319.

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

[32] 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*, June 1998, pp. 106–117.

[33] 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*, May 1999, pp. 91–103.

[34] 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*, January 1993, pp. 232–245.

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

[36] 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.

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

[38] 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*, June 1998, pp. 85–96.

[39] 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*, June 1998, pp. 97–105.

[40] 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.

[41] 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*, March 1998, pp. 21–31.

[42] 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.

[43] 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*, July 1988, pp. 21–34.

[44] 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.

[45] 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*, June 1990, pp. 296–310.

[46] 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*, June 1992, pp. 249–260.

[47] 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*, December 1997, pp. 330–335.

[48] 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*, May 1999, pp. 64–76.

[49] D. M. Gallagher, "Memory disambiguation to facilitate instruction-level parallelism compilation," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[50] 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 structure for VLIW and superscalar compilation," Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, tech. rep., February 1992.

[51] S. A. Mahlke, "Exploiting instruction level parallelism in the presence of conditional branches," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[52] B. G. Ryder, "Constructing the call graph of a program," *IEEE Transactions on Software Engineering, SE-5, 3*, pp. 216–226, May 1979.

[53] A. Lakhotia, "Constructing call multigraphs using dependence graphs," in *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*, January 1993.

[54] M. W. Hall and K. Kennedy, "Efficient call graph analysis," *ACM Letters on Programming Languages and Systems*, vol. 1, pp. 227–242, September 1992.

[55] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan, "An empirical study of static call graph extractors," *ACM Transactions on Software Engineering and Methodolody*, vol. 7, pp. 158–191, April 1998.

[56] B. Cheng, "A profile-driven automatic inliner for the impact compiler," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1997.

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

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

[59] 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.

VITA

Ben-Chung Cheng was born in Taipei, Taiwan, in 1969. He attended Taipei Municipal Chien-Kuo Senior High School. He received his B.S. degree in Computer Science and Information Engineering from National Taiwan University in 1992. After committing two years to military service, he came to the U.S. in 1994 to pursue the Ph.D. degree in Computer Science at the University of Illinois at Urbana-Champaign. He joined the IMPACT research group directed by Professor Wen-mei Hwu in the spring semester of 1995. His doctoral research focuses on effective and efficient interprocedural pointer analysis for C programs. He spent the summer of 1997 at Intel Corporation in Santa Clara, California, working on static memory disambiguation issues of the IA-64 research compiler. After completing his Ph.D. work in 2000, he joined Sun Microsystem in Sunnyvale, California, as a software engineer.