An Empirical Study of Function Pointers Using SPEC Benchmarks

Ben-Chung Cheng* Wen-mei W. Hwu†

*Department of Computer Science

†Department of Electrical and Computer Engineering and

The Coordinated Science Laboratory

University of Illinois at Urbana-Champaign

May 10, 1999

Abstract

Since the *C* language imposes little restriction on the use of function pointers, the task of call graph construction for a *C* program is far more difficult than what the algorithms designed for *Fortran* can handle. From the experience of implementing a call graph extractor in the IMPACT compiler, we found the call graph construction problem has evolved into an interprocedural pointer analysis problem. A complete and precise call graph can be constructed from fully resolved function pointers. In this paper, we report an empirical study of function pointers in the complete SPECint92 and SPECint95 benchmarks. We evaluate the resolution of function pointers and the potential program transformations enabled by a complete call graph. We also examine several real examples of function pointer manipulation found in these benchmarks. They can be considered as critical issues in the design of a complete interprocedural pointer analysis algorithm.

1 Introduction

With the rapid advancement in modern processor designs, exploiting parallelism within the body of a single function cannot fully utilize the computation resources. Also, the presence of function calls in a stream of instructions not only invokes run-time overhead, but also hampers compile-time data-flow analysis, constraining code optimizations like register allocation and global variable migration [1]. One way to break function boundaries is to inline the callee function to the caller function so that the transfer of control and the barrier of compilation are eliminated. Another way is to perform interprocedural data-flow analysis, so that the potential side-effects of a function call can be safely modeled when making aggressive code optimization decisions. Other optimization methods to overcome the performance hazards incurred by function calls can be found in [2, 3, 4, 5, 6].

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 variables, 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 variables may be defined interprocedurally. The final call graph is resolved iteratively where existing function variables receive new function names propagated from already exploited functions in the partial call graph, and newly added functions may add more function variables [7, 8, 9].

Although algorithms for call graph construction have been widely studied in the literature, we find some characteristics of the SPEC benchmarks [10] 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 struct/union fields, and function pointers may be manipulated as non-function pointers due to type casting. Therefore earlier techniques geared towards *Fortran* will not suffice [8, 9]. 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 benchmarks like SPEC [11, 12, 13, 14, 15, 16].

In this paper, we provide a comprehensive study of function pointers using SPEC benchmarks. SPEC benchmarks were selected for this study because of their size, wide acceptance, and realistic function pointer manipulation. Benchmarks included in this paper are *li*, *sc*, *eqntott*, *espresso*, *m88ksim*, *perl*, *ijpeg*, *vortex*, *cc1* and *gcc*, whose details can be found in [10]. This paper does not present a new algorithm for interprocedural pointer analysis. Instead,

we examine several problematic code constructs we experienced while implementing a call graph extractor integrated in the interprocedural pointer analysis framework in the IMPACT compiler [17, 18]. From the point of view of interprocedural pointer analysis, treating function pointers should be no different than general pointers, and the ways function pointers are manipulated are completely applicable to general pointers. From the point of view of call graph construction, an incomplete call graph indicates incompleteness in the interprocedural pointer analysis. We believe these large benchmarks can provide more valuable insight to both interprocedural pointer analysis and call graph construction than synthesized small examples.

The rest of this paper is organized as follows: Section 2 reviews background information. Section 3 shows the preliminary experimental results of call graph construction in the IMPACT compiler, including the resolution of function pointers and potential code transformations enabled by a complete call graph. Section 4 presents a comprehensive case study of four major types of function pointer manipulation found in the SPEC benchmarks. Finally, conclusions are given in Section 5.

2 Background

Many algorithms have been designed to construct the complete call graph for languages supporting limited forms of function or procedural variables [7, 8, 9, 19, 20, 21]. The basic idea is to propagate the contents of parameters along the existing call graph. For example, in the *BoundTo* analysis proposed by Hall and Kennedy [9], if a procedure variable passed as a parameter is bound to a new procedure, the new procedure will be included in the call graph.

Spillman [22] and Weihl [23] enable call graph construction in the presence of aliases.

The accuracy of such analysis is further improved by several flow- and context-sensitive interprocedural pointer analysis algorithms [11, 12, 13, 14]. However, a complete solution scalable to the class of SPEC programs has not been found in the literature.

Murphy et al. [24] 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.

3 Call Graph Construction in the IMPACT Compiler

3.1 Interprocedural pointer analysis

We constructed the complete call graphs of all SPECint92 and SPECint95 benchmarks with function pointers using the IMPACT compiler. The interprocedural pointer analysis algorithm in IMPACT performs a decomposed flow-insensitive analysis with limited context sensitivity. Instead of virtually inlining all acyclic function calls to identify the possible targets of pointers, the interprocedural analysis is decomposed into the intraprocedural phase and the interprocedural phase for lower complexity. In the intraprocedural phase, pointers which can be accessed by other functions are identified. In the interprocedural phase, these interprocedurally accessible pointers exchange information with pointers in other functions along the call graph. Newly resolved targets of function pointers will add more functions into the call graph, causing more paths for information exchange. The analysis is conducted iteratively until no more targets are resolved for any pointers. To be complete, the analysis incorporate techniques to handle type casting and dynamically allocated objects. Due to space restrictions, more details can be found in [17].

Benchmark	Characteristics					Interprocedural analysis	
	Lines	Functions	Dir. Call	Ind. Call	& Function	Time (Sec)	Memory (MB)
li	7597	357	1267	4	190	148.66	19.35
sc	8639	179	1459	2	20	4.07	4.30
eqntott	12053	62	358	11	5	1.09	2.50
espresso	14838	361	2674	15	12	29.59	10.62
m88ksim	19092	252	1496	3	57	3.63	5.05
perl	26874	276	4367	3	3	416.30	30.17
ijpeg	29290	477	1016	641	188	186.21	30.23
vortex	67205	923	8521	15	44	138.79	24.92
cc1	90857	1452	8332	67	588	298.97	45.22
gcc	205583	2019	19731	132	229	1201.66	99.42

Table 1: Benchmark characteristics.

3.2 Results

Table 1 lists some characteristics of the SPEC benchmarks we tested. For each benchmark, we measured its size in terms of function and line counts. They would directly affect the applicability of some interprocedural analysis algorithms due to high complexity. We also calculated the number of direct and indirect call sites found in each benchmark, together with the number of functions whose names are assigned to function pointers. The product of the latter two can provide a rough measurement of the difference between a conservatively estimated call graph and a precisely calculated call graph. The analysis time and memory requirement were gathered from an HP/9000/780 workstation running at 180 MHz with 256 MB of physical memory. We found that the complete call graphs for these SPEC benchmarks can be generated within a reasonable amount of time given a complete and efficient interprocedural pointer analysis algorithm. Although benchmark like *gcc* takes around 20 minutes to finish, we are not aware of any faster algorithms with comparable capabilities.

Table 2 lists the resolution of function pointers excluding those found in dead functions. We classify function pointers into four categories with difficulty levels we ranked the easiest to the hardest as follows:

1. Simple variables: Function pointers declared as local or global variables whose ad-

	Simple			Parameter					Global					Heap					Dead functions	
Benchmark	0	1	2	0	1	2	3	≥ 4	0	1	2	3	≥ 4	0	1	2	3	≥ 4	Total	Dead
li	0	0	0	0	0	0	1	0	0	0	0	0	3	0	0	0	0	0	357	1
sc	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	179	8
eqntott	0	0	0	0	2	0	9	0	0	0	0	0	0	0	0	0	0	0	62	2
espresso	0	0	0	0	2	3	0	10	0	0	0	0	0	0	0	0	0	0	361	46
m88ksim	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	252	13
perl	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	276	13
ijpeg	0	0	0	0	0	0	0	0	0	0	0	0	0	4	381	16	11	15	477	179
vortex	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1	2	0	2	923	295
cc1	0	0	15	0	0	0	0	19	0	0	0	0	32	0	3	0	0	0	1452	51
gcc	4	11	16	0	3	0	0	14	1	1	0	0	73	0	9	0	0	0	2019	187

Table 2: Resolution of function pointers.

dresses 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 structs/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 [9]. Function pointers initialized statically can be handled properly as long as various forms of struct/union initialization can be processed. However, heap-based function pointers cannot be resolved without using a general interprocedural pointer analysis algorithm.

In Table 2, number n at cell *(a, b)* other than the dead-function columns represents that in benchmark a, n function pointers are resolved to b callees. The ideal case is to resolve a function pointer into one callee, since the indirect call site can be converted into a direct

Benchmark	Uninitialized function pointers (file/function/call-site)
perl	stab.c/stab_str/*uf→uf_val stab.c/stabset/*uf→uf_set doarg.c/do_subr/*sub→usersub
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
vortex	trans01.c/C_CreateObject/*pmf1
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

Table 3: Uninitialized indirect call sites of SPEC benchmarks found by interprocedural pointer analysis.

one, so that the overhead associated with indirect function calls can be eliminated. If the compiler supports inlining, the callee can even be inlined. As the table shows, benchmark *ijpeg* would benefit most from the converted calls: 381 out of 427 indirect call sites can be transformed into direct ones [1]. If the resolved callee number is small, the indirect callee can be converted into a series of `if-then-else` statements or a *switch* statement predicated by the function pointer's content. This transformation trades the indirect call overhead with branch penalties. If code expansion is of less concern, function inlining is also applicable. When the indirect call site is resolved to a large number of callees, the indirect-to-direct transformation will not justify the overhead. For example, there are 4 indirect call sites in *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 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.

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

The situation for parameter call sites can be improved by performing function cloning or inlining, at the cost of increased code size. Table 2 also shows that some benchmarks have unresolved call sites. We verified these function pointers by tracing the program and using the system debugger. We found these pointers are indeed uninitialized and not exercised with multiple profile inputs. Detailed locations for these pointers can be found in Table 3. These benchmarks originally may contain more uninitialized function pointers which were falsely resolved due to spurious aliases. However, it is difficult to identify them but the resultant call graph is still false positive.

Removing dead functions can speedup the compilation process and reduce the resultant code size. Dead functions can be inherent in the program or resulted because of function inlining. Once all the function pointers are resolved, the call graph starting from `main` can be constructed. The unconnected functions are dead functions if they are not used as the signal handling routines. These signal handling routines can be determined by applying the *BoundTo* analysis for the second parameter of library functions `signal` and `sigset`. The rightmost portion in Table 2 shows the number of dead functions found in each benchmark, where the total number of functions is also listed for easy comparison. It indicates that some SPEC benchmarks contain a significant portion of dead functions.

It is hard to verify if the constructed call graph is false positive. Therefore we perform aggressive code transformations to hopefully expose some errors. We converted all indirect call sites with a single callee into direct ones. We also inserted guarding code around indirect call sites which will terminate the execution of the program if an unexpected callee is encountered. All dead functions were also deleted from the original benchmarks when we built the binary for testing. We used several inputs to test each benchmark, and we found they all produced correct results.

4 Code Examples of SPEC Benchmarks

In this section we present some code examples extracted from the SPEC benchmarks which can provide insight to the manipulation of function pointers. The usual motivation for programmers to use function pointers is to write concise code: using a single indirect call site to invoke n similar callees instead of n individual static call sites. However, pointers and type casting in *C* make the problem much harder than in *Fortran*. For each category of function pointers discussed in Section 3, we provide one or two examples to better illustrate the difficulties involved and potential solutions. For clarity and space reasons, we only extract directly related code segments.

4.1 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 *cc1*, the following code segment can be found:

```
gen_bgt_pat = unsignedp ? gen_bgtu : gen_bgt;
(*gen_bgt_pat)();
```

It is trivial to find functions `gen_betu` and `gen_bgt` as the potential callees of the indirect call site. Another case is found in *gcc*, where global function pointer `decl_printable_name` is assigned with `decl_name` in function `main` and used in many other functions. Surprisingly, these simple cases only happen in these two largest benchmarks of SPEC.

4.2 Function pointers as parameters

Function pointers passed as parameters can also be handled with widely studied techniques. For example, in benchmark *espresso*, function `foreach_output_function` can be called form

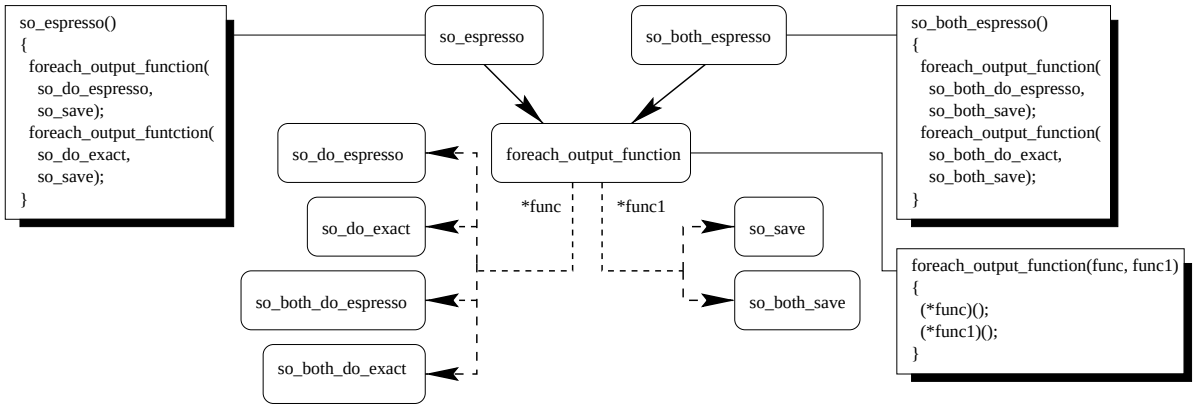

Figure 1: Partial call graph of *espresso*.

four call sites in two callers as shown in Figure 1. Throughout this paper, 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 *BoundTo* analysis [9]. If the interprocedural analysis is context-sensitive [13], 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.

4.3 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 *li*, *m88ksim*, *cc1*, and *gcc*.

In Figure 2 which is extracted from *m88ksim*, the `instruction` struct is defined as the data structure to store information regarding each assembly instruction's mnemonic

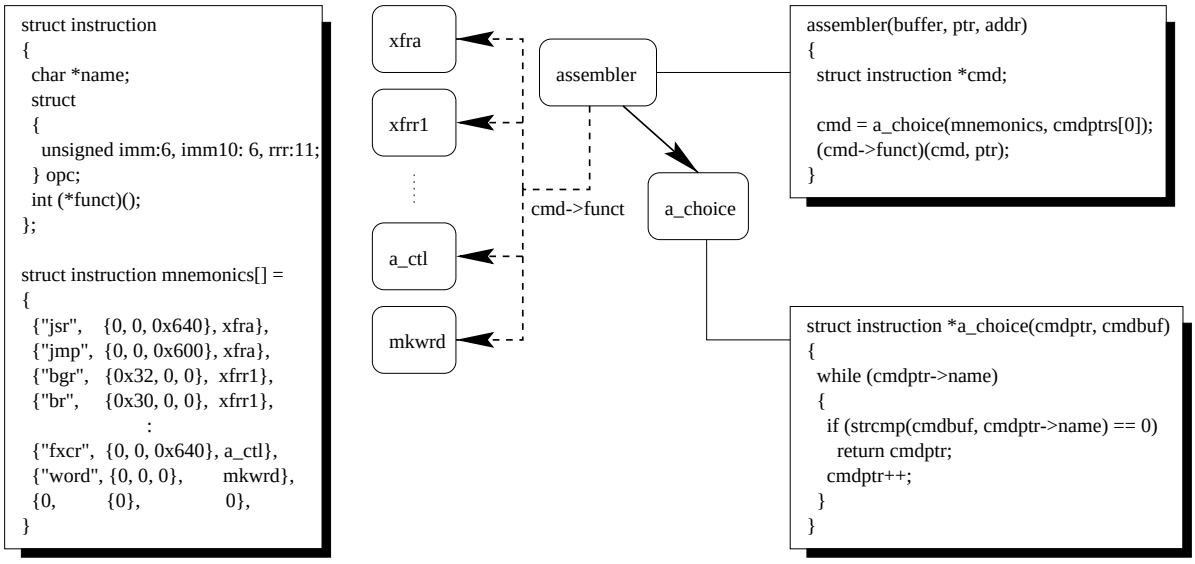

Figure 2: Partial call graph of *m88ksim*.

name, opcode, and the native function's address. Variable `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 struct fields. Also, the original *BoundTo* analysis has to be extended to analyze the binding for function return values as well. Therefore variable `cmd` in `assembler` can be identified as holding the address of array `mnemonics`, indicating `cmd->funct` and `mnemonics[x].funct` to be aliases [2]. Thus the function names stored in the `mnemonics` array are resolved to be the possible callees of `cmd->funct` [3].

[2]Assume the whole array is treated as a single object.

[3]The dereference operator before a function pointer is optional at the indirect call site.

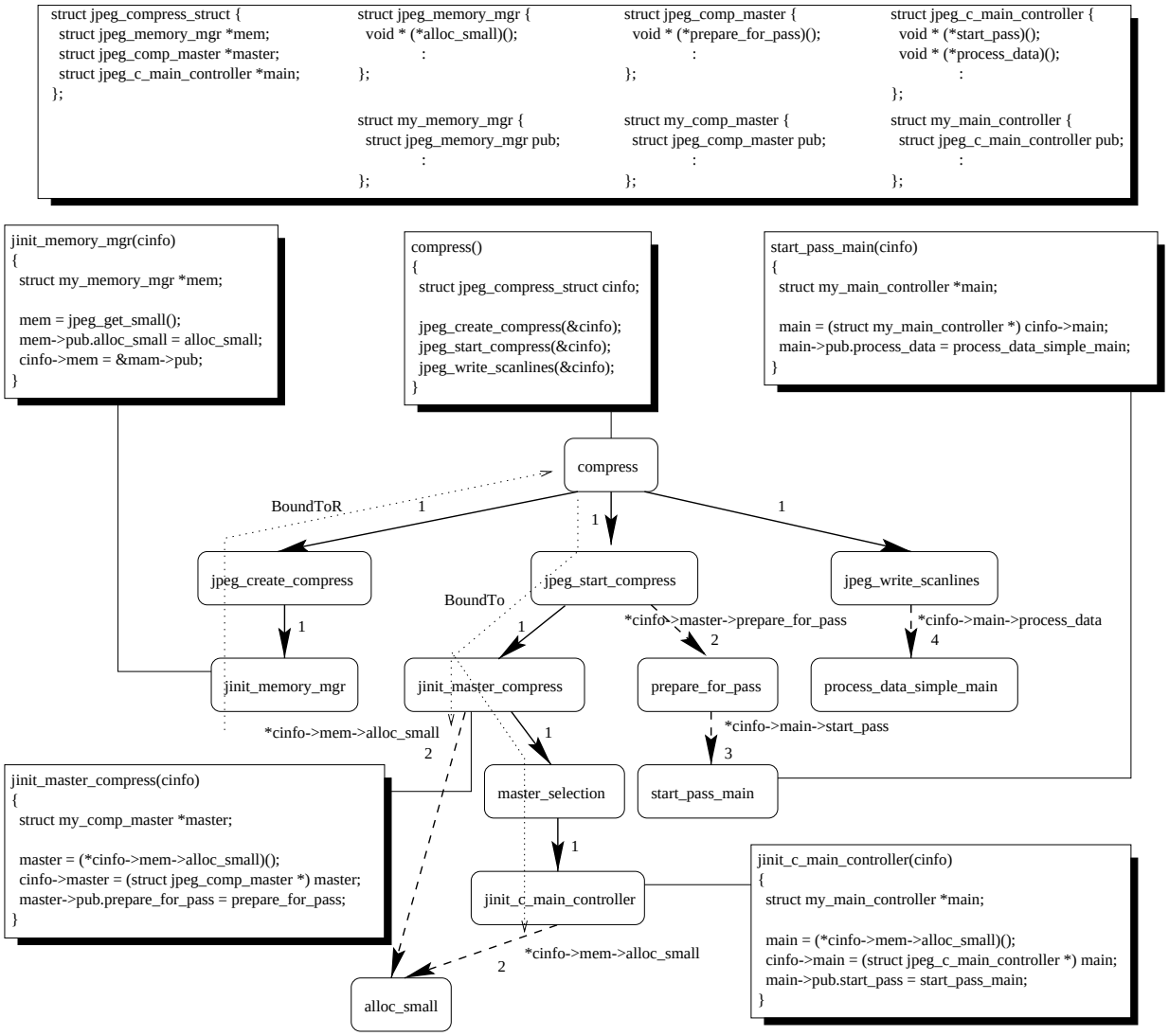

Figure 3: Partial call graph of *ijpeg*.

4.4 Function pointers as anonymous objects

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

The first characteristic is dynamically allocated function pointers. Once a heap-object is allocated by the callee and attached to the formal parameter, the original *BoundTo* analysis needs to be extended as a two-way analysis. That is, not only values from the caller will be bound to the callee, values will need to be bound from the callee to the caller as well. Assume the extension is called a *BoundToR* analysis and consider the call graph path marked with *BoundToR* in Figure 3. In function `compress`, `cinfo` is a local variable of type `jpeg_compress_struct` as listed in the top portion of Figure 3. The address of variable `cinfo` is passed as a formal parameter named as `cinfo` also to all subsequent callees [4].

In function `jinit_memory_mgr`, it allocates a struct object of type `my_memory_mgr`, which is a superset of struct `jpeg_memory_mgr`. Then it initializes `cinfo->memory->alloc_small`, which is a function pointer for dynamic memory allocation, as `alloc_small`. When function `jinit_memory_mgr` returns, the location accessed via `cinfo->mem->alloc_small` is still alive, so its content needs to be bound back to the caller, function `jpeg_create_compress`. Eventually, the *BoundToR* analysis will report the binding of (`cinfo.mem->alloc_small` = `alloc_small`) to function `compress`. If the heap objects is reachable from the callee's return value, it needs to be handled by the *BoundToR* analysis as well.

The second characteristic will also result in extensions related to the *BoundTo* analysis: not only the parameters and return values themselves need to be bound, but also their reachable locations. This is because 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 the *BoundTo* paths in Figure 3, 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.

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

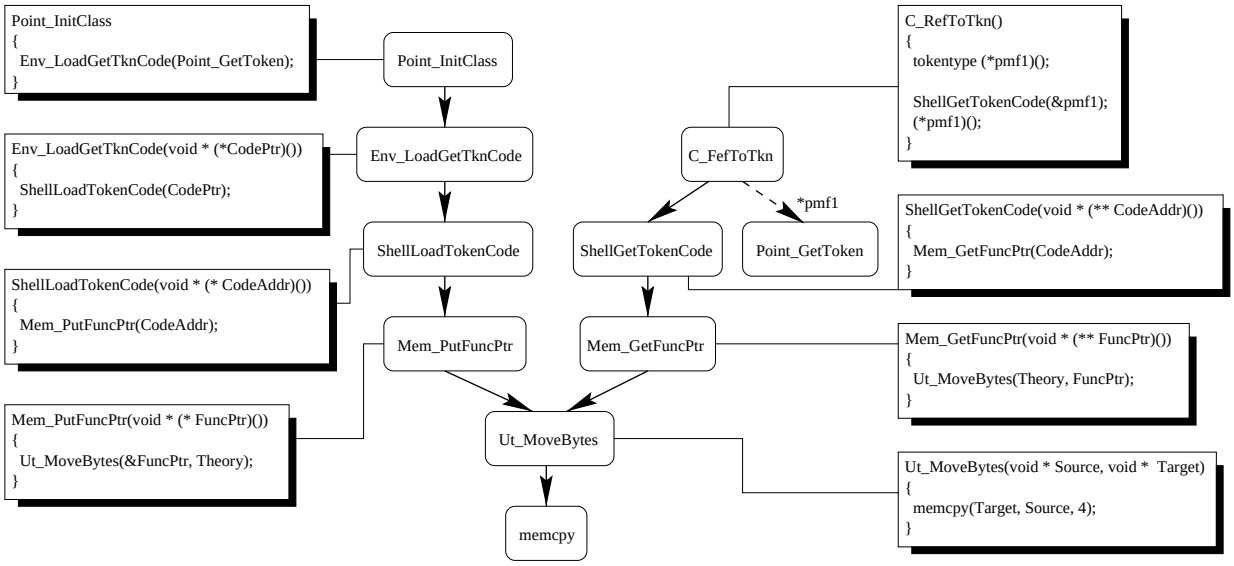

Figure 4: Partial call graph of *vortex*.

The third feature is that the interprocedural pointer analysis needs to be aware of type casting. As explained in [25], struct pointers can be used interchangeably if their target structs share a common initial sequence of fields. In the *ijpeg* example listed in Figure 3, pointer `mem->pub.alloc_small` is an alias of `cinfo->mem->alloc_small`, since the structs pointed by `mem` and `cinfo->mem` share the same initial sequence of fields. Unless *BoundTo* and *BoundToR* handle type casting, pointers of these types cannot be resolved.

One final feature we want to discuss in the *ijpeg* benchmark is that it exemplifies the iterative nature of interprocedural pointer analysis. The number associated with each indirect call edge indicates the order of the callee being added to the call graph. Function `start_pass_main` initializes the function pointer used in `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 con-

struction of the partial call graph requires at least four iterations. And whether the optimal number of iterations can be achieved depends on if the *BoundTo* analysis is performed in the top-down traversal and the *BoundToR* analysis is performed in the bottom-up traversal of the currently resolved call graph.

Benchmark *vortex* basically shares the same features as *ijpeg*. In addition, instead of using explicit assignments to propagate function addresses, *vortex* uses library call `memcpy` as an alternative approach. For example, calling "`memcpy(dest, src, 4)`" is equivalent to performing "`*(char **)dest = *(char **)src`". In Figure 4, function `Point_InitClass` [5] 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 eventually invokes `memcpy` to retrieve the address of function `Point_GetToken`. Once parameters of library functions are included in the *BoundTo* and *BoundToR* analysis, the case represented by *vortex* can be handled properly.

5 Conclusions

We presented a comprehensive case study for SPEC benchmarks to illustrate why the call graph construction problem has evolved into the interprocedural pointer analysis problem. Using detailed analysis of several real examples extracted from large benchmarks, we demonstrated that in order to design a false positive call graph extractor integrated within an interprocedural pointer analysis, the algorithm needs to handle all aspects of *C* including extended parameter binding analysis from caller to callee and vice versa, statically initialized

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

global tables, dynamically allocated objects, type casting, and library function side-effects. We also demonstrated that it is possible to construct a complete call graph for the SPEC benchmarks using an efficient interprocedural pointer analysis algorithm. The statistics of resolved indirect call sites indicated that substantial code transformations can be enabled by the resolved call graph for some SPEC benchmarks, including indirect-direct call-site conversion, function inlining, and dead code removal.

This paper's results suggest several areas for future research. One possible research area is the study of real programs written in object-oriented languages like *C++* or *Java*. In such cases the call graph construction is even more difficult because of object inheritance and function overloading. Another possible research area is the study of how to formally validate the correctness of a statically constructed call graph.

References

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

[2] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Communications of the ACM*, vol. 19, pp. 137–147, March 1976.

[3] J. Banning, "An efficient way to find the side effects of procedure calls and the aliases of variables," in *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, pp. 29–41, January 1979.

[4] T. Way and L. L. Pollock, "Using path spectra to direct function cloning," in *Proceedings of the 1998 Workshop on Profile and Feedback-directed Compilation*, October 1998.

[5] R. Bodik, R. Gupta, and M. L. Soffa, "Interprocedural conditional branch elimination," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pp. 146–158, June 1997.

[6] J. P. Hoeflinger, *Interprocedural Parallelization Using Memory Classification Analysis*. PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1998.

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

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

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

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

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

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

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

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

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

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

[17] B. Cheng and W. W. Hwu, "A practical interprocedural pointer analysis framework," Tech. Rep. CRHC-99-01, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, April 1999.

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

[19] K. Walter, "Recursion analysis for compiler optimization," *Communications of the ACM*, vol. 19, pp. 511–516, September 1976.

[20] D. Callahan, "The program summary graph and flow-sensitive interprocedural data flow analysis," in *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 47–55, June 1988.

[21] O. Shivers, *Control-flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, Pittsburg, PA, 1991.

[22] T. C. Spillman, "Exposing side-effects in a pl/i optimizing compiler," in *Proceedings of IFIP Conference*, pp. 56–60, 1971.

[23] W. E. Weihl, "Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables," in *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, pp. 83–94, January 1980.

[24] 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.

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