

# A Name Based Heuristic for Identifying Potential Indirect Callees in the Absence of Profile Information

Archana Ravindar and Dibyendu Das

HP India Software Operations  
{archana,dibyendu.das}@hp.com

**Abstract.** Pointer disambiguation has a vital role to play in assisting the compiler to get a much more clearer view about the program. In addition to better program comprehension, it can help one get better performance. In this work, we consider the problem of resolving function pointers to their potential callees. Typically profile information is used to come up with a reasonably accurate set of potential callees for an indirect call site. But in this paper, we propose a static name based heuristic that can be used in the absence of profile information. The name based heuristic works with the properties of names of function pointers and functions. We exploit the name property to help us narrow down the possible choices of callees at an indirect call site. We first evaluate a simple static heuristic based on signature match and observe that there is scope for further improvement. Applying the name heuristic, we get upto 2.5% improvement in the SpecFp 2000 benchmark 177.mesa.

**Key words:** Compiler, function pointers, indirect call graph, potential callee, static analysis.

## 1 Introduction

Understanding how a program is structured helps effectively utilizing the compiler optimizer. Unless the compiler has a complete picture of the program structure it cannot exploit the opportunities for optimization fully. The presence of pointers makes inferring the program structure and behavior difficult. In this paper we focus on the problem of dealing with a specific class of pointers - the *indirect function pointers* in a program. An indirect call is where the address of the call target has to be loaded from memory. Indirect calls make it difficult for the compiler to understand the control flow of the program since the contents of the function pointer can be ascertained only at run time.

Apart from the compiler, several program comprehension tools require information about the program. A typical example of a program representation is the *call graph* that represents the caller-callee relationship among the different procedures in a program. When there are function pointers involved, it becomes hard for the compiler to come up with an estimate of the potential callees that

could be invoked at an indirect call site or an I-call site. As a result the call graph in such cases is generally obfuscated.

The presence of indirect function pointers also makes it difficult to get optimized performance out of modern computer architectures that tend to rely heavily on speculation of instructions and data to keep their pipelines full and busy [3]. Speculating conditional calls is relatively easy since they typically have two possible targets; hence their prediction mechanisms are likely to be less complex. However indirect calls can potentially have a large number of possible targets. As a result, speculating such calls can be a challenging task and often requires sophisticated hardware call prediction mechanisms to achieve good success rates. Hence it becomes important to deal with indirect function pointers in such a way that it has a minimal impact on the program understanding and performance.

```
{  
  ...  
  (*fX());  
  ...  
}
```

**Fig. 1.** An Indirect call site.

Our objective is to identify the potential callees for as many I-call sites in a program as possible. Once this is carried out we can replace all such I-calls by direct calls. Consider an indirect call site `fX` as shown in Fig 1. Assuming the set of potential callees for `fX` contains `func1`, the I-call can be replaced as shown in Fig 2. Similarly if the set of potential callees for `fX` contains two choices `func1` and `func2`, the I-call can be replaced as shown in Fig 3. This process is termed as *indirect call promotion*. Promoting such indirect calls can help the compiler inline them and achieve better performance. Generally profiling is used to get us the set of most probable callees for an I-call site. The challenge lies in getting this information in the absence of profile information.

```
if(*fX == func1)  
  func1();  
else  
  (*fX());
```

**Fig. 2.** Promotion of an Indirect Call Site using one callee.

In order to ensure that I-call promotion does not give rise to additional overhead, we need to have potential callee sets of small size for every I-call site. We term such sets as *minimal* sets. In our experiments we have restricted the size to less than or equal to two. Program profiling can help to a large extent

```

if(*fX == func1)
    func1();
else if(*fX == func2)
    func2();
else
    (*fX)();

```

**Fig. 3.** Promotion of an Indirect Call Site using two callees.

in giving us a reasonably accurate set of potential callees, that is minimal, but the program has to be run once to give us profile information. Moreover, people are interested in getting out of box performance without having to spend their time and effort on profiling. Also, profiling may not always represent the runtime paths exercised.

In this paper we attempt to solve the problem of identifying potential callees at an indirect call site at compile time in the absence of any profile information. We believe that for many benchmarks, the names of function pointers and the functions carry significant information and can be exploited to help identify potential callees at compile time. Based on this we propose a name based heuristic that is reasonably successful in narrowing down the possible choices of potential callees for an I-call site.

For our experiments we choose the HP PA C Compiler (B.11.11.16). We first evaluate a simple static heuristic that identifies potential callees based on signature match between the function pointer and the possible callees. We observe that although this simple heuristic is successful to a small extent in identifying potential callees, there is scope for improving the performance further. We find that implementing the name heuristic improves the performance by 2.5% in case of SpecFp 2000 benchmark, *177.mesa* and by 1.8% in case of SpecInt 2000 benchmark, *255.vortex*. Also we observe that applying the name heuristic does not degrade the performance of any benchmark.

The paper is organized as follows. Section 2 describes related work. The basic idea of the name based heuristic is described in Section 3. Section 4 discusses the results achieved followed by conclusions and future work in Section 5.

## 2 Related Work

There have been different approaches to resolve indirect function calls in the past both in hardware and software. As architectures become more complex and try to issue and execute more instructions in a given frame of time, the need for speculating right instructions increases correspondingly. The success of such an architecture relies heavily on predicting the control flow of programs and hence on predicting indirect jumps accurately. [4] discusses the design of a target cache in hardware that aims to increase the success rate of predicting indirect jumps. Our approach to resolve function pointers is basically a software based approach.

[3] evaluates a static compilation strategy wherein the type signatures of methods are used to determine potential callees at an I-call site and a dynamic strategy which uses profile based information to determine the possible callees at a given I-call site. In this work, we try to guess the callee information in the absence of any profile data.

Other approaches view the problem of identifying potential callees at an indirect call site as a specific instance of the *points-to* problem. Points-to analysis involves computing for every pointer in a program, the set of variables that it could possibly point to. Points-to could either be a flow sensitive one or a flow insensitive one. Flow sensitive analysis takes into account the order in which the statements are executed whereas flow insensitive analysis assumes that the statements could be executed in any order. Similarly context insensitive points-to analysis ignore the flow of control between program points and do not distinguish between calling contexts of procedures whereas context sensitive analysis keeps track of different calling contexts. [5] introduced a context sensitive, flow insensitive points-to analysis. [14] describes the impact of function pointers on the call graph and is based on the algorithm described in [5]. [12] describes a fast linear algorithm for points-to that is flow and context insensitive. [6] presents a context sensitive points-to algorithm. Similarly there are several other ways of performing points-to, each differing in their accuracy and complexity of results [7], [8], [10], [13].

The abstraction of pointer variables in the program for points-to analysis also has a bearing with respect to the scalability of the algorithm. [11] tries to improve the accuracy of the points-to for context sensitive algorithms by performing variable substitution and hence reduce the input size of the problem. [9] carries out an Andersen style points-to analysis by taking a database centric approach. Although the system is successful in performing points-to very quickly as compared to conventional points-to approaches and is much more scalable, it is difficult to implement and maintain. Points-to analysis is associated with many issues like complexity, precision and scalability. Context sensitive and flow sensitive algorithms are precise but they are usually complex making it difficult to implement and they are also not very scalable. At the other extreme, flow insensitive context insensitive analysis is very simple and inexpensive with respect to time, space and implementation effort but the resultant callee set is likely to be imprecise to be of practical use. The name matching heuristic on the other hand, is very simple to implement and has no scalability issues. It is reasonably accurate and can be applied in conjunction with points-to analysis for better results. However we should note that the name matching heuristic is applicable to resolving I-calls only.

### 3 Name Matching

In this section we describe the name matching algorithm that exploits the properties of names of functions and function pointers. Our objective is to get a

minimal set of potential callees for a given I-call site. Each such I-call site can then be promoted as described in Figures 2 and 3.

It is observed that many a times there is a close relationship between the name of a function pointer and the name of the function that it is pointing to. This does not happen merely by accident but is mostly intentional. Moreover, this kind of feature is desirable since it gives rise to better program readability and understandability, making software development and maintenance easier. If we look at some of the SpecInt 2000 and SpecFp 2000 benchmarks, we observe that certain key words are common between the function pointer name and the function name.

Example 1:	Example 2:
(176.gcc):	(177.mesa):
interim_eh_hook = interim_eh;	table->AlphaFunc = gl_save_AlphaFunc;

Making use of this property, we define a static heuristic that helps narrow down the choice of possible callees for a given I-call site. For an indirect call site  $fA$ , we say  $fB$  is a probable callee at  $fA$ , if both  $fB$  and  $fA$  have certain vital words common between them. Depending on how sophisticated the design is, the decision of selecting vital words will vary.

To begin with, we could get the names of the function pointer and the function into some standard form for comparison. Names could contain words punctuated with *underscores* in between. If names follow the *hungarian* notation, they would contain letters in *caps* in between. All names are converted into a standard form in which the underscores are removed and all letters are converted to lower case for easy comparison. For instance `decl_name` is converted to `declname`. Similarly `ReadIndexPixels` gets converted to `readindexpixels`.

We define a function `MatchesName(x,y)` for every function pointer  $x$  and potential callee  $y$ . If the name  $y$  matches the name  $x$  as described below, `MatchesName(x,y)` returns true, else it returns false.

**Algorithm** The following algorithm describes how two closely related names are matched using the rules mentioned above.

```
bool MatchesName(string x, string y)
{
    string n_x=Standardize(x);
    string n_y=Standardize(y);
    if(Substr(n_x,n_y) == true || Substr(n_y,n_x) == true)
        return true;
    else return false;
}

string Standardize(string x)
{
    // returns x with all underscores removed and with all
    // its letters converted to lower case
}
```

```

bool Substr(string x, string y)
{
    // return true if x is a substring of y, false otherwise
}

```

Applying this algorithm to the above examples we get,

<p>Example 1: (176.gcc): interim_eh_hook = interim_eh; n_x=interimehhook; n_y=interimeh;  MatchesName(interim_eh_hook, interim_eh) and MatchesName(AlphaFunc,gl_save_AlphaFunc) evaluate to true;</p>	<p>Example 2: (177.mesa): table-&gt;AlphaFunc = gl_save_AlphaFunc; n_x=alphafunc; n_y=glsavealphafunc;</p>
---	--

The scheme described above is a simple approach. As we shall see, this heuristic is quite successful in narrowing down the possible choices of callees. As the name matching algorithm becomes more sophisticated it would be able to recognize more complicated patterns occurring in the functions. The name property has the potential to be used as an effective heuristic when we do not have profile based information about the program.

### 3.1 Compiler Infrastructure

We incorporate the name matching heuristic in the *High Level Optimizer (HLO)* phase of the HP PA C Compiler. Several important optimizations like inlining, cloning and loop transformations apart from scalar optimizations like constant propagation, dead code elimination are carried out by the *HLO* that gets enabled when +O3 or +O4 is on. The HLO performs indirect call promotion before it passes through the inlining and cloning optimizer phases to provide new opportunities to the inliner and cloner. The existing compiler supports computation of potential callees and indirect call promotion only under profiling [2]. The profile information is used to position the most frequently executed call closer to the if condition to give better performance.

**Signature Match** When the compiler is run under the non profile mode it has no information about the list of callees that each function pointer could point to. The current PA compiler does not compute sophisticated points-to information that can be used to provide us with the list of potential callees. So we need to devise ways of guessing the potential callees at compilation time with whatever information is available to us.

To begin with, we consider the function signatures as a possible filter to give us the list of potential callees for every I-call site. For each I-call site, we compare the signature of the function pointer with the signature of the list of *address exposed functions*. Address exposed functions are functions whose addresses are taken at some point in the program. Among the set of address exposed functions, few of them will match the signature of the function pointer.

All such functions figure in the potential callee set for the I-call site. All those call sites having minimal potential callee sets are promoted. The other call sites that have potential callee sets of cardinality greater than two are usually not worth pursuing.

The success of the promotion scheme depends on the number of I-call sites that get promoted. It is observed that very few I-call sites tend to have minimal potential callee sets when only signatures are matched and as a result only a few of the I-calls get promoted (see Table 1). There is hence a need for additional heuristics to increase the number of I-call promotions.

### 3.2 Implementing Name Matching

Name matching acts as an additional filter to signature match and increases the number of I-call sites that can be promoted. We incorporate the name matching algorithm at the point where the list of address exposed callees for an I-call site is considered. Once a potential callee passes the signature match test, it undergoes a possible name match test. If it matches the name of the function pointer according to the algorithm described in Section 3, the callee is recorded. After all the possible callees are examined, the set of potential callees got by merely matching signatures, is examined. If the set is minimal, promotion is carried out. If signature match fails to get us a minimal set then the set of potential callees got by name matching is examined. If the set thus got is minimal, the I-call is promoted as described in Figures 2 and 3.

Any static heuristic depends on the following factors to have a significant impact on the performance. Firstly, it should succeed in collecting minimal potential callee sets for most of the I-call sites. Secondly, these I-call sites should be hot. On promoting and inlining such hot I-call sites the program is likely to execute faster.

**Algorithm** This subsection describes the name matching algorithm discussed in the previous section in the form of pseudocode

```

for each callsite  $\in$  list of I-Call sites {
  for each callee  $\in$  list of address exposed functions {
    if (Sigmatch(callee, callsite) == true) {
      smatches++;
      if(scallee1 == NULL) scallee1 = callee;
      else if(scallee2 == NULL) scallee2 = callee;
    }
    if(NameMatch(callee, callsite) == true) {
      nmatches++;
      if(ncallee1 == NULL) ncallee1 = callee;
      else if(ncallee2 == NULL) ncallee2 = callee;
    }
  } /* for each addr exposed function */
  if(smatches < 3) Promote(callsite, scallee1, scallee2);
  else if(nmatches < 3) Promote(callsite, ncallee1, ncallee2);
} /* for each I-call site */

```

## 4 Results

In this section we evaluate the static heuristics based on signature match and name match algorithms. The baseline we use is the HP PA C Compiler (B.11.11.16) which does not perform any indirect call promotion in the absence of profile information. We use the HP 9000 rp4400 Server with a 1GHz PA-8800 dual processor and 16382 MB of main memory for our study. We choose a set of SpecInt 2000 and SpecFp 2000 benchmarks as shown in Table 1.<sup>1</sup>

### 4.1 Number of I-Call Sites Matching Signature

**Table 1.** Number of I-Call Sites Matching Signature

Benchmark	# I-call sites	Size of Potential Callee Set			
		Minimal Set (=1)	Minimal Set (=2)	(<=5)	(>5)
164.zip	2	2	0	0	0
176.gcc	137	0	2	3	122
253.perl	59	12	0	0	47
255.vortex	15	0	1	0	9
177.mesa	670	4	6	6	138

Table 1 describes the number of I-call sites that get their signatures matched with exactly one callee, with two callees, with less than or equal to five callees and with more than five callees. We observe that *164.zip* gets all its I-calls matched using signatures alone and that the potential callee set is minimal. But *176.gcc* has a very few number of I-call sites that are associated with minimal potential callee sets. This can be attributed to the way *176.gcc* is written. In *176.gcc*, majority of the function pointers are declared using an array indexed by opcode kind. In such cases the number of matches got by signature match alone tend to be very high. The same can be said about *253.perl*.

### 4.2 Number of I-Call Sites Matching Name

Similarly Table 2 describes the number of I-call sites that get their names matched with exactly one callee, with two callees, with less than or equal to five callees and with more than five callees. As we saw in the earlier subsection, *176.gcc* is dominated by function pointers declared in the form of an array and such cases are hard to be caught using name matching. But we do find 10

<sup>1</sup> Other benchmarks do not contain indirect call sites and hence were not considered for the study

call sites that get their names matched using the name matching algorithm. We observe that the name heuristic works well with *177.mesa* where it succeeds in identifying 531 call sites (79% of the total number of I-call sites ) having minimal callee sets.

**Table 2.** Number of I-Call Sites Matching Name

Benchmark #	I-call sites	Size of Potential Callee Set			
		Minimal Set (=1)	Minimal Set (=2)	(<=5)	(>5)
164.gzip	2	0	0	0	0
176.gcc	137	10	0	0	0
253.perl	59	0	0	0	0
255.vortex	15	0	4	0	4
177.mesa	670	164	367	77	31

### 4.3 Pattern Matches

Table 3 describes some of the kind of names that are matched using the algorithm described in Section 3. On observing *177.mesa* we find that the majority of call sites that exhibit the pattern xxx either call functions `gl_xxx` or `gl_save_xxx`. The name matching heuristic exploits this property and succeeds in improving the performance of the program.

**Table 3.** Kind of Pattern Matches.

Benchmark	Function Pointer	Callee
176.gcc	<code>interim_eh_hook</code> <code>print_error_function</code>	<code>interim_eh</code> <code>default_print_error_function</code>
255.vortex	<code>Point_x</code> <code>Point_Theta</code>	<code>CartesianPoint_x</code> <code>CartesianPoint_Theta</code>
177.mesa	<code>SetBuffer</code> <code>ReadColorSpan</code>  <code>ReadIndexSpan</code> <code>WriteIndexSpan</code> <code>WriteColorPixels</code>  <code>QuadFunc</code> <code>xxxx</code>	<code>set_buffer</code> <code>read_color_span</code> <code>read_color_span3</code>  <code>read_index_span</code> <code>write_index_span</code> <code>write_color_pixels</code> <code>write_color_pixels3</code>  <code>quad</code> <code>gl_xxxx</code> <code>gl_save_xxxx</code> (185 such occurrences)

#### 4.4 Matches Got from Profiling

Table 4 describes the number of matches that were found using the profiling approach. Promotion of indirect calls is carried out through profiling by the PA high level optimizer. The basic idea is that the indirect callee candidate generally has an unexplained surplus of calls and the surplus is greater than or equal to the calls from the call site. The algorithm makes use of profile information to select for every I-call, the best two possible candidates having the maximum surplus calls as potential callees and promotes the I-call. The number of the first best and the second best candidates that are computed in this way are depicted in Table 4.

Comparing the number of matches got from profiling with the number of matches got from name matching (Table 2), it is clear that most of the time, profiling helps identify more number of potential callees, which is expected. That is because profile information is a run time entity and is much more accurate in identifying the possible calls the indirect call site might be making. However note that the number of matches got in *177.mesa* by name matching are significantly higher than those caught by profiling. That could be because many of these paths might not have been hot enough to be considered by the profiling approach. Since our approach is a static one it considers every possible alternative. Moreover *177.mesa* is dominated with indirect call sites whose names very closely relate to the possible callees as described in Table 3 and this is exploited well by our name matching algorithm

**Table 4.** Matches got from Profiling.

Benchmark	# I-call sites	First Best Match	Second Best Match
164.gzip	2	2	0
176.gcc	137	47	46
253.perl	59	31	22
255.vortex	15	7	6
177.mesa	670	29	18

#### 4.5 Execution Time

Table 5 describes the execution times achieved by running signature match and name match. Since the number of I-call sites in *164.gzip* are very few, promoting them has little impact on the overall execution time. *176.gcc* and *255.perl* show slight improvement in the performance primarily because only a small proportion of the I-call sites get promoted by either signature match or name match algorithms. However we observe that *177.mesa* and *255.vortex* respond quite well to the name based heuristic. The execution time for *255.vortex* decreases by 3.8 seconds and *177.mesa* by 6 seconds giving a performance improvement by 1.8% and 2.5% respectively.

Table 5. Execution Time.

Benchmark	Baseline (seconds)	Signature Match (seconds)	% Improvement	Name Match (seconds)	%Improvement
164.gzip	230.5	229.8	0.3	229.6	0.4
176.gcc	122.3	121.7	0.5	121	1.1
253.perl	259.56	257.7	0.7	257.7	0.7
255.vortex	212.9	212.9	0	209.1	1.8
177.mesa	229.1	228.5	0.3	223.5	2.5

## 5 Conclusions and Future Work

Resolving function pointers and promoting them is a very beneficial process. It opens up new opportunities for the compiler to do additional optimization. It can help one achieve better performance out of modern architectures that carry out call target prediction. There have been several approaches in hardware and software to solve this problem. In this paper, we describe a simple heuristic based on the properties of the names of function pointers and functions. Although this heuristic was primarily developed for a compiler that does not profile indirect call targets, it can be used in the presence of profile information to further sharpen the accuracy of the results. We observe that applying the name matching heuristic fetches performance improvement of 1.8% in case of *255.vortex* and 2.5% in case of *177.mesa*. Also it doesn't degrade the performance of any benchmark.

The name matching heuristic is very simple to implement and can be applied to all function pointers irrespective of how they are initialized. Application of points-to analysis to full fledged programs suffer from certain drawbacks. If the implementation is reasonably time and space efficient, the resultant callee set becomes highly imprecise and hence likely to be too large to be used for further analysis. On the other hand a more stricter points-to analysis which can provide a much smaller probable callee set may be too expensive to implement and generally does not scale with larger programs.

The heuristic that we propose involves just a name comparison and is hence much more economical to implement and does not have scalability issues. Depending on how programs are written, the resultant callee set produced by the name comparison is likely to be a competitive one. Thus the name heuristic forms an attractive option in cases where one needs a reasonably precise set and at the same time does not want to spend too much time on carrying out a full points-to analysis. As part of future work we intend to study the impact of larger minimal sets (of cardinality greater than 2). Specifically we intend to study the impact of using the parallel compare instruction supported by the titanium architecture[15], using which one can execute multiple conditional statements in a shorter amount of time. Such a feature could be valuable in cases where the size of the minimal set ranges from 2 to 10. We also intend to look towards

incorporating the same for C++ and carry out the study for the SPEC 2006 benchmarks.

## References

1. <http://www.spec.org>
2. Andrew Ayers, Robert Gottlieb and Richard Schooler, Aggressive Inlining. In PLDI'97, p134-145.
3. Brad Calder and Dirk Grunwald, Reducing Indirect Function Call Overhead In C++ Programs, In POPL'94, p.397-408.
4. Wen-mei W. Hwu and Pohua P. Chang, Inline Function Expansion for Compiling C Programs, In PLDI'89, p.246-257.
5. L.O. Andersen, Program Analysis and Specialization for the C Programming Language. PhD Thesis, DIKU, University of Copenhagen, May 1994.
6. M. Emami, R. Ghiya and L. Hendren. Context sensitive interprocedural points-to analysis in the presence of function pointers. In PLDI'94, p.242-256.
7. M. Fahndrich, J. Rehof and M. Das. Scalable context sensitive flow analysis using instantiation constraints. In PLDI'00, p.253-263.
8. J. Foster, M. Fahndrich and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. Proceedings of the 7th International Symposium on Static Analysis, p.175-198, June 29-July 01, 2000.
9. Nevin Heintze and Olivier Tardieu, Ultra-fast aliasing analysis using CLA: a million lines of C code in a second, In PLDI'2001, p.254-263.
10. S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. SIGPLAN Notices, 24(7):p.28-40, 1989.
11. A. Rountev and S. Chandra. Offline variable substitution for scaling points-to analysis. In PLDI'00, p47-56.
12. B. Steensgard. Points-to analysis in almost linear time. In POPL'96, p.32-41.
13. M. Shapiro and S. Horwitz. Fast and accurate flow insensitive points-to analysis. In POPL'97, p.1-14.
14. G. Antoniol, F. Calzolari and P. Tonella, Impact of Function Pointers on the Call Graph, In European Conference on Software Maintenance and Reengineering. 1999.
15. James S. Evans and Gregory L. Trimper, Itanium Architecture for Programmers: Understanding 64-Bit Processors and EPIC Principles, Prentice Hall.