Importance of Heap Specialization in Pointer Analysis

Erik M. Nystrom, Hong-Seok Kim and Wen-mei W. Hwu

Center for Reliable and High Performance Computing
University of Illinois, Urbana-Champaign

{nystrom, hskim, hwu}@crhc.uiuc.edu

ABSTRACT

Specialization of heap objects is critical for pointer analysis to effectively analyze complex memory activity. This paper discusses heap specialization with respect to call chains. Due to the sheer number of distinct call chains, exhaustive specialization can be cumbersome. On the other hand, insufficient specialization can miss valuable opportunities to prevent spurious data flow, which results in not only reduced accuracy but also increased overhead.

In determining whether further specialization will be fruitful, an object's escape information can be exploited. From empirical study, we found that restriction based on escape information is often, but not always, sufficient at prohibiting the explosive nature of specialization.

For in-depth case study, four representative benchmarks are selected. For each benchmark, we vary the degree of heap specialization and examine its impact on analysis results and time. To provide better visibility into the impact, we present the points-to set and pointed-to-by set sizes in the form of histograms.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: program analysis; D.3.3 [**Language Constructs and Features**]: procedures, functions, and subroutines

General Terms

algorithms, languages

Keywords

pointer analysis, context sensitivity, heap specialization

1. INTRODUCTION

Many software-engineering tools rely on pointer analysis to resolve complex memory activities. In these tools, the accuracy of the results is one factor governing the number of false positives and negatives observed. On the other hand, the efficiency of obtaining the results is also important if tools are meant to be used on a daily basis.

In pointer analysis, abstraction of heap objects greatly affects overall quality (both accuracy and efficiency). Many analyses in the literature [1, 2, 5, 6, 13] assign a unique global variable per allocation site. This approach, called a *per-site scheme*, is less likely to impact scalability. On the other hand, it has a potential to introduce spurious data flow through under-specialized objects. This phenomenon is often observed when analyzing programs with custom wrappers around allocation routines. In general, the per-site scheme can easily become ineffective in forming the appropriate view of heap usage.

In many context-sensitive pointer analyses [3, 8, 9, 10, 12], heap specialization is performed by cloning heap objects along paths in call graphs, called *call chains*. Even though imperfect, this simple method is found to be useful in eliminating many sources of spurious data flow. On the other hand, uncontrolled heap specialization can quickly overload the analysis process impacting scalability. Therefore, while heap specialization is necessary, a controlling method is needed that can provide a finer resolution of heap usage while leaving overall scalability unaffected.

Our analysis framework is aggressive in preventing unnecessary specialization. The techniques currently employed by our framework are non-lossy in the sense that they never result in accuracy degradation. In many programs, these techniques are sufficient in controlling the amount of specialization. However, in certain programs, we feel that further (potentially lossy) restrictions are necessary to make analysis time more predictable.

For in-depth case study, four representative SPEC benchmarks were selected. For each of the benchmarks, to provide better visibility into the impact of heap specialization, we present the points-to set (PT) and pointed-to-by set (PB) sizes in the form of histograms. The following is a summary of our study.

1. In many cases, the per-site scheme leaves substantial room for accuracy improvement. Sometimes, we noticed that its analysis time is longer than that of heap-specializing alternatives.

2. Often, there exists a clear threshold beyond which additional heap specialization yields little benefit. However, generalization of this observation remains as future work.

3. When performed superfluously, especially beyond the

PASTE'04, June 7–8, 2004, Washington, DC, USA.

threshold, heap specialization begins to significantly affect scalability.

The longer-term goal of this research is to understand heap usage better and eventually develop methods to apply heap specialization only when necessary.

2. ANALYSIS FRAMEWORK

In C programs, indirect calls can be made using function pointers. In the presence of indirect calls, a cyclic dependency exists between call-graph construction and pointer analysis. Our framework breaks this cyclic dependency using an iterative approach as in [3, 8]. It starts with a call graph consisting of only direct calls. Based on this incomplete call graph, pointer information is constructed. Then, using this as feedback, the call graph is updated and the process iterates until there are no more changes.

Each iteration consists of three phases: recursion merging, computation of pointer information, and call-graph updating. Recursion merging renders the call graph acyclic, from which pointer information is computed in two phases. The bottom-up phase propagates procedure summaries from callees to callers, followed by the removal of side-effect causing assignments at the level of callees. Then the top-down phase computes the actual pointer information using a single run of a context-insensitive analysis. This results in a fully context-sensitive result.

Intraprocedural data flow is modeled as a field-sensitive extension of Andersen's pointer analysis [1] while array indices are ignored. Field sensitivity is modeled in a fashion similar to [11] with four differences: 1) physical offsets are used instead of field indices; 2) offsets are discovered online rather than derived offline from type information; 3) to guarantee termination in the presence of cyclic offsets, the maximum offset of a heap object is bounded by the size of the largest type;[1] 4) Assignments are not bound to a pointer size but, instead, vary by the size of the C construct. When combined with online detection of field offsets, it makes the analysis more tolerant to type abuse in C programs.

3. HEAP SPECIALIZATION

Heap specialization is performed during the bottom-up phase of the two-phase computation where call chains are walked backwards from callees to callers. It begins by assigning a unique heap object for each allocation routine. As the procedure summary of a callee is inserted into a caller's call site, a fresh (specialized) heap object is introduced for each heap object in the summary. Therefore, in our framework, the per-call-site scheme is equivalent to applying heap specialization only up to the call sites of the allocation routines.

Example 1 Figure 1 is a small example derived from actual code in 132.ijpeg, one of the benchmarks used for the case study in §5. Note that we have simplified the code (such as shortening variable names) for clarity. We focus on the following facts.

[1]A similar issue and solution are observed in [11], except the maximum numerical index was limited to the largest number of fields in any type.

```
typedef union {
    funcptr *alloc;
    funcptr *prepare;
} FUNCS;

typedef struct {
    FUNCS *mem;
    FUNCS *master;
} CINFO;

void main() {
    CINFO cinfo;
    memory_mgr(&cinfo);
    master_decompress(&cinfo);
1:  (*cinfo->master->prepare)(&cinfo);
}

void memory_mgr (CINFO *cinfo) {
2:  cinfo->mem = alloc(sizeof FUNCS); /* H1 */
3:  cinfo->mem->alloc = alloc;
}

void master_decompress (CINFO *cinfo) {
4:  cinfo->master = (*cinfo->mem->alloc)(size); /* H2 */
5:  cinfo->master->prepare = prepare;
}

void* alloc(int size) {
    return malloc(size);
}
```

Figure 1: Example 1 derived from 132.ijpeg.

1. Procedure `alloc` is a custom wrapper around the standard allocation routine `malloc`. Effectively, this code fragment has only a single allocation site.
2. In dynamic execution, two heap objects are allocated: one created in `memory_mgr` (H_1) and the other created `master_decompress` (H_2).
3. Both objects H_1 and H_2 are associated with the type `FUNCS`. Since `FUNCS` is defined as a `union`, two field names `alloc` and `prepare` refer to the same offset.
4. In `memory_mgr`, field `alloc` of object H_1 acquires the pointer to procedure `alloc`. On the other hand, in `master_decompress`, field `prepare` of object H_2 acquires the pointer to procedure `prepare`.
5. In call site `*cinfo->mem->alloc` in `master_decompress`, since `cinfo->mem` points to object H_1, only the procedure `alloc` can be called indirectly.
6. In call site `*cinfo->master->prepare` in `main`, since `cinfo->master` points to object H_2, only the procedure `prepare` can be called indirectly.

By Fact 1, the per-call-site scheme will create only a single heap object (`H`) that represents both H_1 and H_2, resulting in

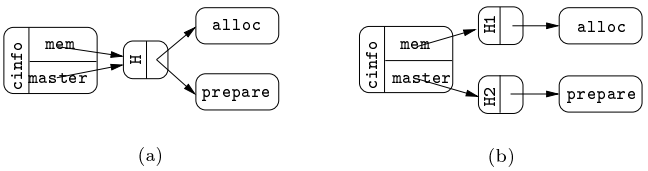

Figure 2: Storage graph of Example 1 before (a) and after (b) heap specialization.

```
void main() {
    if( connection_machine ) { unlap1(-1); }
    if( doglobal ) { unlap2(-2); }
}

void unlap(int flag)
{
  int *left_queue = malloc3();
  int *right_queue = malloc4();
  int *center_queue = malloc5();

  for( i = 1 ; i <= cell_count ; i++ ) {
     left_queue[] = ...
     right_queue[] = ...
     center_queue[] = ...
  }
  ...
}
```

Figure 3: Example 2 derived from 300.twolf. All the objects in this example have purely local effects, thus, specialization is not necessary.

object H acquiring both `alloc` and `prepare` as in Figure 2(a). Therefore, the two call sites at Lines 1 and 4 invoke both `alloc` and `prepare`, leading to further spurious data flow. On the other hand, specialization of heap objects beyond allocation sites can prevent spurious data flow by creating two distinct heap objects thereby resolving that H_1 can point to only `alloc` while H_2 only to `prepare`. This leads to a more accurate call graph where Line 1 can only call `prepare` and Line 4 only `alloc`.

As seen in Example 1, heap specialization can increase accuracy by disallowing spurious data flow through under-specialized heap objects. However, its blind application can easily inflate analysis time. The following subsection describes the techniques used by our framework in determining whether further specialization is necessary.

3.1 Overhead reduction

The restriction scheme currently employed by our framework is based on the *escape* information of heap objects as in [10, 12]. Intuitively speaking, if the visibility of a heap object spans beyond that of a procedure, we say that the heap object *escapes* the procedure. For instance, both heap objects H_1 and H_2 in Example 1 escape procedure `alloc`. For a more complete list of references and in-depth discussion of escape analysis, we refer readers to [4].

Example 2 The code fragment in Figure 3 shows a simple example derived from 300.twolf where the per-site scheme is sufficient.

In this example, arrays `left_queue`, `right_queue`, and `center_queue` are created only for local use, thus, do not escape `unlap`. Therefore, further specialization of the arrays (differentiating those in $\texttt{unlap}_1$ and $\texttt{unlap}_2$) will unnecessarily incur overhead since each specialized version will be identical.

Example 3 Procedure `dummy` in Figure 4 creates an object of `int` type, assigns `m` to it, then returns its contents.

In this example, when the per-site scheme is used, the heap object created in `dummy` ends up acquiring both `1` and `2`. On the other hand, by specializing this object further, one

```
main() {
    int a = dummy (1);
    int b = dummy (2);
}

dummy (int m) {
    int *obj = malloc(sizeof int); *obj = m; return *obj;
}
```

Figure 4: Example 3 demonstrating the effect of compaction for non-escaping objects.

```
typedef struct node { char n_type; struct node *n_next; } NODE;

NODE *newseg;

void main(void) {
   NODE *n1 = newnode1(1);
   NODE *n2 = newnode2(2);
   classify (n1->n_type);
}

NODE *newnode(int type) {
    NODE *nnode;
    if (newseg == NIL) { findmem(); }
    nnode = newseg;
    newseg = nnode->n_next;
    nnode->n_type = type;
    return (nnode);
}

void findmem() { newseg = calloc(1,ALLOCSIZE); ... }
```

Figure 5: Example 4 derived from 130.li. In this example, since objects escape only through global variables, specialization does not aid accuracy.

can find that the object created in call site $\texttt{dummy}_1$ acquires only `1` while the other object created in the other call site $\texttt{dummy}_2$ acquires only `2`. Consequently, in `main`, local variable `a` acquires only `1`, while variable `b` acquires only `2`.

Even in this case, our framework can exploit the fact that the object does not escape `dummy` and avoid the specialization overheads using procedure summaries as follows.

1. From the perspective of caller `main`, only the net effect of callee `dummy` is important. Therefore, instead of specializing the entire callee, only its *summary* needs to be specialized. Since the object does not escape `dummy`, all its data flow can be represented in a compact form, *i.e.* `return m`. In this form, the same overall effect is achieved without specializing the heap object.

2. From the perspective of `dummy`, when considering call-chain unspecific queries (what the object acquires during the entire run of the code fragment), the object can acquire either `1` or `2`. Specialization cannot change this fact. On the other hand, call-chain specific queries (what the object acquires from a subset of call sites, *e.g.* $\texttt{dummy}_1$), can be answered efficiently in a demand-driven fashion by back-tracing data flow.

While the exposure of a heap object to a caller is often an opportunity to improve accuracy through specialization (as in Example 1), when the object escapes only through a global variable, surprisingly, there is no benefit. This observation turns out to be useful in dealing with programs that use global allocation pools, such as 130.li.

Example 4 The code fragment in Figure 5 shows a simple example derived from 130.li where specialization, again, does not result in improved accuracy.

In this situation, the program uses a custom global allocation pool. The heap objects assigned into `n1` and `n2` are stored and fetched from a global allocation pool rooted at `newseg`. Specialization for $\texttt{newnode}_1$ and $\texttt{newnode}_2$ will not aid accuracy in a flow-insensitive analysis because global variable `newseg` will acquire both heap objects. Therefore, `n1` and `n2` end up pointing to both heap objects.

3.2 Limitation

From empirical studies, we found that, for a number of benchmarks (including 008.espresso, 134.perl, 176.gcc, and 253.perlbmk), the above-mentioned techniques are ineffective at preventing the explosive nature of heap specialization. Unlike local variables, a heap object's lifetime is not inherently bounded by a single procedure and many heap objects tend to repeatedly escape into higher-level procedures. When combined with a dense call graph, such repeated escape can produce a large number of heap objects, sometimes, with little impact on accuracy.

For instance, in 008.espresso, a common high-level object is passed down through a number of call layers. Within each procedure in the call layers, heap objects are allocated and their pointers assigned into the fields of the common high-level object. Heap specialization causes situations where new objects are created, only to be assigned into the same field of the same object. The high density of the call graph results in a large number of specialization opportunities but their eventual interaction with a common high-level object blurs their values.

4. MEASUREMENT STRATEGY

The primary goal of this work is to investigate the impact of heap object specialization. To this end, the experiments vary the number of heap generations allowed for a small number of applications. However, some metric must be used to compare the effect of differing amounts of specialization. If points-to sets (PTs) or pointed-to-by sets (PBs) of a heap object remain very similar even after specialization, the expense of introducing more objects may outweigh the benefit. Therefore, to predict the impact of specialization, it is important to quantify how distinguishable heap objects are among themselves.

Simple metrics, like total PT size or even average PT size per node, are inconclusive in the presence of varying amounts of heap specialization. This is because differing amounts of specialization can cause a substantial difference in the number of locations. Clearly if, in one instance there is only a single heap object, while another creates 1000 heap objects, comparing absolute PT size across all objects would be inappropriate. A similar observation with regard to choice of metrics is made by Hind [7].

To help delineate when heap specialization is worth the cost, we present data in the form of histograms. Each histogram groups nodes by PT (or PB) sizes into buckets and connects buckets from the same analysis configuration by a line. Compared to a single, aggregate number a histogram provides greater visibility into the results. The area under the line is roughly proportional to the total number of heap objects after specialization. The breadth of the histogram shows how much variation there is in heap object resolution.

Figure 6 shows initial heap specialization results for four benchmarks. Parts (a-b), (d-e), (g-h), and (j-k) are the histogram lines, one line per heap specialization configuration. As an example, the black line in Figure 6(a) shows that roughly 4 heap objects have a PB size of 1, 20 heap objects a PB size of 5, 450 heap objects a PB size of 10, 140 heap objects a PB size of 50, and so on. Note that parts (c), (f), (i), and (l) are the analysis times for the particular heap specialization settings.

One obvious property of the histograms is their shape. Before heap specialization, the histogram consists of one or two points. Once specialization takes place, a large range in PT and PB size becomes apparent, sometimes ranging from singleton sets to sets of almost ten thousand elements. From the perspective of this work, the relative values and distributions are far more important than the absolute measurements.

The trends between histogram lines give a rough estimation of both benefits and wastes from specializing heap objects. A shift to the left depicts an increase in precision. A shift upward denotes an overall increase in nodes. If "too much" specialization occurs, it is possible for the trend to show a shift to the right. This is not really a loss of precision, but depicts wasted attempts at specialization that result in densely connected clumps of heap objects. Figure 6(a) shows a clear trend up and to the left as more specialization takes place. This means heap specialization increased the number of objects but also benefited the precision of many objects. A large part of Figure 6(h) shows a shift up and, in some cases, to the right. This represents a wasteful increase in heap objects for little or no benefit. Section 5 provides an extensive discussion of the results. While not a perfect metric, we feel these histograms are a good start for presenting trends.

5. CASE STUDY

The end goal of our research is to investigate (a) quantification of the effects of specialization on accuracy. (b) how much specialization is actually necessary to have an effective view of heap usage; (c) the cost of such specialization in terms of analysis time.

To this end, having the restriction methods based on escape information in § 3.1 as a baseline, we applied an additional simple controlling mechanism that limits the number of heap generations. For example, by imposing a generation limit of 2 (GL-2), the heap object assigned for each allocation routine can be specialized at most 2 times. From this perspective, per-site scheme site is equivalent to generation limit of 1 (GL-1). and exhaustive specialization to generation limit of infinity (GL-IN). This method is simple but allows straightforward interpretation and comparison among different degrees of specialization.

The SPEC benchmarks can be roughly partitioned into following categories: (1) those that lack any real heap activity (*e.g.* 099.go and 129.compress), (2) those with heap behavior consisting almost entirely of global allocation pools (*e.g.* 130.li and 254.gap), (3) those with non-pool heap behavior where specialization is bounded by the techniques in § 3.1 (*e.g.* 132.ijpeg and 300.twolf), and (4) heap behavior where those techniques are ineffective at controlling specialization (*e.g.* 008.espresso and 253.perlbmk). For a more in-depth study, four representative benchmarks were

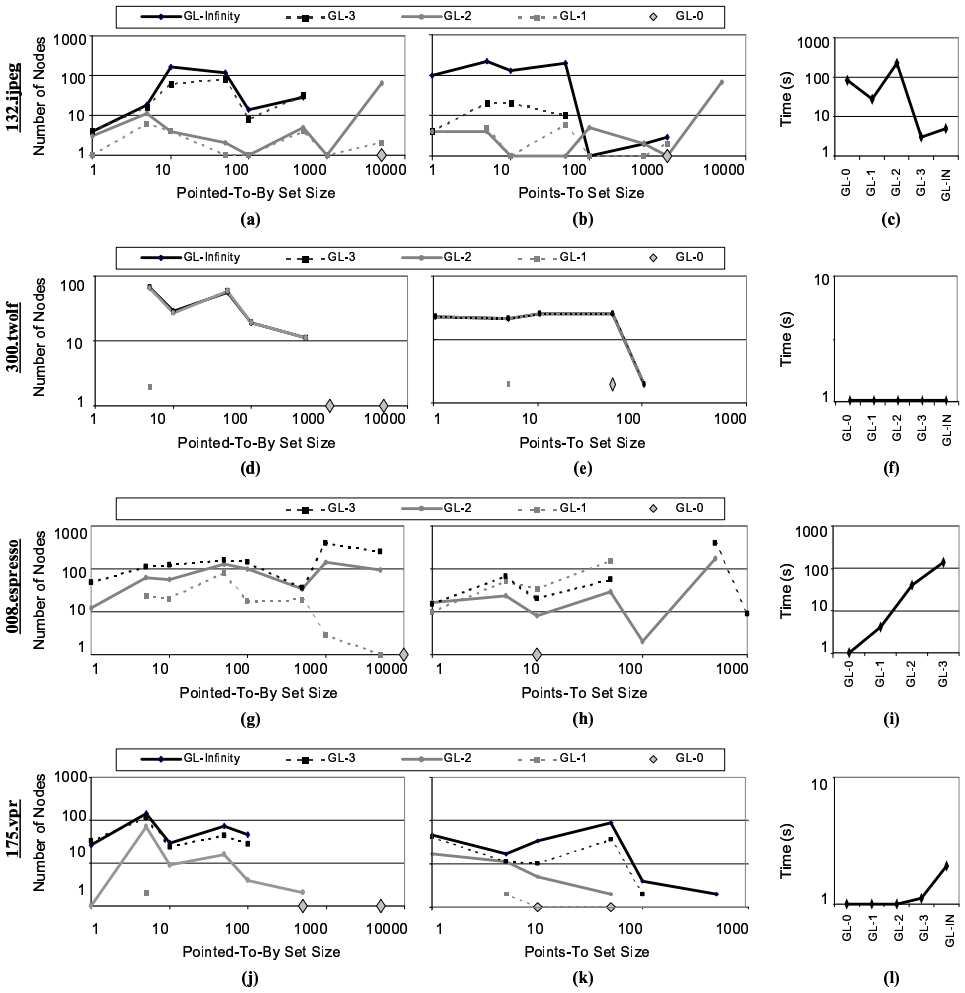

Figure 6: Histograms showing the effects of varying levels of heap specialization on points-to set (PT) size, pointed-to-by set (PB) size, and analysis time on 132.ijpeg, 300.twolf, 008.espresso, and 175.vpr. Note that GL-0 is equivalent to one heap location per allocation routine and is therefore only one or two points.

selected: 132.ijpeg (25k LOC), 300.twolf (20k LOC), and 175.vpr (17k LOC) from category (3) and 008.espresso (13k LOC) from category (4).

5.1 132.ijpeg

The GL-1 histogram for 132.ijpeg shows a dramatic range of PT and PB sizes. For example, the one object at GL-0 had a PB size of about 10000. However, at GL-1, there are many objects with PB sizes less than 100. On the other hand, some objects were created at 10000 that did not benefit, purely increasing overall problem size. However, the total benefit outweighed the total cost, resulting in a decrease in overall analysis time.

GL-2 does little to help precision and greatly increases the number of heap locations. Looking at PB size again, at GL-0 there was one object with a PB size of 10000, at GL-1 a handful of objects of that size, but at GL-2 there are almost 100 such objects. The overhead from this many highly connected objects results in a ten fold increase in analysis time, relative to GL-1, and as shown in Figure 6(c), 3 times as much as the GL-0 run.

Previous increases in the amount of heap specialization performed resulted in more heap objects and much greater analysis times. However, GL-3 shows an interesting change. At GL-3, the precision suddenly increases with the largest PB now under 1000 and PT under 100, with a large proportion of the sets well under 100 and 10 respectively. This increase in precision coincides with a *100 fold decrease* in analysis time. While initial increases in specialization did little to benefit accuracy, and drastically impacted analysis time, at GL-3, a threshold was reached where key objects were finally distinguished. In 132.ijpeg, the bulk of these objects are those created by `jinit_XXX` routines which include important function pointers that configure various indirect calls. We found that the failure to distinguish them results in a very inaccurate call graph. Due to the indirection to `malloc` provided by `cinfo->mem->alloc` as well as multiple allocation wrappers, a few layers of specialization are necessary to realize the large gains.

Finally, for unrestricted heap specialization (GL-IN), there is little further precision benefit over GL-3. The number of heap locations continues to increase, though only to a small degree, resulting in a corresponding increase in analysis time.

132.ijpeg shows that heap specialization can substantially improve pointer analysis results and greatly reduce analysis time. It also shows that specialization well beyond per-allocation site (GL-1) is necessary to realize this benefit. In fact, the situation gets worse until a threshold is reached. Finally, this example shows that too much specialization (*i.e.* completely unrestricted) can unnecessarily increase the problem size and analysis time without improving the results.

5.2 300.twolf

From the perspective of heap specialization, 300.twolf and 175.vpr are fairly straightforward. 300.twolf shows little gain until GL-2, because almost all calls to `malloc` and `calloc` are made through `safe_malloc` and `safe_calloc` wrappers. Once two generations are allowed, all but two specialization possibilities are handled. This is reflected by the formation of a broad distribution and the dramatic shift to the upper left when going from GL-1 to GL-2.

5.3 175.vpr

175.vpr follows a similar pattern. Due to use of the custom wrappers `my_malloc` and and `my_calloc`, GL-1 has little effect. Further specializations help differentiating heap locations with GL-3 obtaining almost all of the benefits. However, looking at the sharp increase in PT size in Figure 6(k), it is also evident that overspecialization has begun. GL-IN does not provide any obvious gains but, looking at both Figure 6(k) and the times in Figure 6(l), it noticeably increased the overhead.

Overall, both 300.twolf and 175.vpr show the need for more than GL-1. For 175.vpr, better control is needed to avoid the overheads of blind specialization.

5.4 008.espresso

Unlike the previous two benchmarks, heap specialization for 008.espresso can cause the analysis process to grind to a halt. GL-1 creates some benefit over no specialization but also shows a significant number of heap objects and a 5 fold increase in analysis time. Any further specialization only exacerbates the problem. At GL-2 and GL-3 the number of heap locations grow substantially without obvious reductions in PT and PB sizes. At each increase, the analysis time swells. In fact, an attempt at GL-Inf did not complete for us. Figure 6(i) shows the continued, and exponential increase in analysis time as heap specialization is increased. The portions of the call graph leading to the allocation routines is fairly dense in 008.espresso and the number of locations increases sharply from 1, to 168, 633, and then 1328 locations. While similar in heap object quantity to 132.ijpeg the specialization does little to help accuracy and, instead, only adds overhead.

6. FUTURE WORK

This work has shown that there are merits to specializing beyond allocation sites. It has also shown that, unless used selectively, heap specialization can incur wasteful overhead, even if specialization occurs only when a heap object is visible through parameters. For instance, the analysis of 008.espresso does not terminate when specialization is performed exhaustively. There is a need for a comprehensive mechanism to control specialization. From our perspective, the mechanism may involve one or more of the following: (1) analytical approximation and prediction of the cost-benefit of specialization (2) derivation of benefit from call graph shape information (for example, do not specialize at a fork in the call graph, but only at merges) (3) leveraging type information (specialize at type changes). (4) late unification of specialized nodes.

7. REFERENCES

[1] Lars O. Andersen. *Program Analysis and Specialization for the C Programming Language.* Ph.D thesis, DIKU, Unversity of Copenhagen, 1994.

[2] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 1999.

[3] Ben-Chung Cheng and Wen-mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[4] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for java using escape analysis. *ACM Transactions on Programming Languages and Systems*, 2003.

[5] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[6] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Proceedings of the Static Analysis Symposium*, 2000.

[7] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2001.

[8] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1999.

[9] Chris Lattner and Vikram Adve. Data Structure Analysis: A Fast and Scalable Context-Sensitive Heap Analysis. Technical Report, CS Dept., University of Illinois. 2003.

[10] Donglin Liang and Mary Jean Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *SAS*, 2001.

[11] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis for C. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2004.

[12] Erik Ruf. Effective synchronization removal for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000.

[13] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 1996.