Evaluating a Demand Driven Technique for Call Graph Construction*

Gagan Agrawal[1], Jinqian Li[2], and Qi Su[2]

[1] Department of Computer and Information Sciences, Ohio State University
Columbus, OH 43210
`agrawal@cis.ohio-state.edu`

[2] Department of Computer and Information Sciences, University of Delaware
Newark DE 19716
{`li,su`}`@eecis.udel.edu`

Abstract. With the increasing importance of just-in-time or dynamic compilation and the use of program analysis as part of software development environments, there is a need for techniques for demand driven construction of a call graph. We have developed a technique for demand driven call graph construction which handles dynamic calls due to polymorphism in object-oriented languages. Our demand driven technique has the same accuracy as the corresponding exhaustive technique. The reduction in the graph construction time depends upon the ratio of the cardinality of the set of *influencing nodes* and the total number of nodes in the entire program.

This paper presents a detailed experimental evaluation of the benefits of the demand driven technique over the exhaustive one. We consider a number of scenarios, including resolving a single call site, resolving all call sites in a method, resolving all call sites within all methods in a class, and computing reaching definitions of all actual parameters inside a method. We compare the analysis time, the number of methods analyzed, and the number of nodes in the working set for the demand driven and exhaustive analyses.

We use SPECJVM programs as benchmarks for our experiments. Our experiments show for the larger SPECJVM programs, `javac`, `mpegaudio`, and `jack`, demand driven analysis on the average takes nearly an order of magnitude less time than exhaustive analysis.

1 Introduction

A call graph is a static representation of dynamic invocation relationships between procedures (or functions or methods) in a program. A node in this directed graph represents a procedure and an edge $(p \rightarrow q)$ exists if the procedure p can invoke the procedure q. In program analysis or compiler optimizations for object-oriented programs, call graph construction becomes a critical step for at

* This research was supported by NSF CAREER award ACI-9733520 and NSF grant CCR-9808522.

least two reasons. First, because the average size of a method is typically quite small, very limited information is available without performing interprocedural analysis. Second, because of the frequent use of virtual functions, accuracy and efficiency of the call graph construction technique is crucial for the results of interprocedural analysis. Therefore, call graph construction or dynamic call site resolution has been a focus of attention lately in the object-oriented compilation community [3,4,8,9,11,13,14,15,19,20,21,24].

We believe that with an increasing popularity of just-in-time or dynamic compilation and with an increasing use of program analysis in software development environments, there is a need for *demand driven call graph analysis* techniques. In a dynamic or just-in-time compilation environment, aggressive compiler analysis and optimizations are applied to selected portions of the code, and not to other less frequently executed or never executed portions of the code. Therefore, the set of procedures called needs to be computed for a small set of call sites, and not for all the call sites in the entire program. Similarly, when program analysis is applied in a software development environment, demand driven call graph analysis may be preferable to exhaustive analysis. For example, while constructing static program slices [23], the information on the set of procedures called is required only for the call sites included in the slice and depends upon the slicing criterion used. Similarly, during program analysis for regression testing [16], only a part of the code needs to be analyzed, and therefore, demand driven call graph analysis can be significantly quicker than an exhaustive approach.

We have developed a technique for performing demand driven call graph analysis [1,2]. The technique has two major theoretical properties. The worst-case complexity of our analysis is the same as the well known 0-CFA exhaustive analysis technique [18], except that our input is the cardinality of the set of influencing nodes, rather than the total number of nodes in the program representation. Thus, the advantage of our demand driven technique depends upon the ratio of the size of set of influencing nodes and the total number of nodes. Second, we have shown that the type information computed by our technique for all the nodes in the set of influencing nodes is as accurate as the 0-CFA exhaustive analysis technique. This paper presents an implementation and detailed experimental evaluation of our demand driven call graph construction technique. The implementation has been carried out using the SABLE infrastructure developed at McGill University [22].

Initial work on call graph construction exclusively focused on exhaustive analysis, i.e., analysis of a complete program. Many recent efforts have focused on analysis when entire program may not available, or cannot be analyzed because of memory constraints [6,17,19]. These efforts focus on obtaining most precision with the amount of available information. In comparison, our goal is to reduce the cost of analysis when demand-driven analysis can be performed, but not compromise the accuracy of analysis. We are not aware of any previous work on performing and evaluating demand-driven call graph analysis for the purpose of efficiency, even when the full program is available. Our work is also related to previous work on demand driven data flow analysis [10,12]. Their work assumes

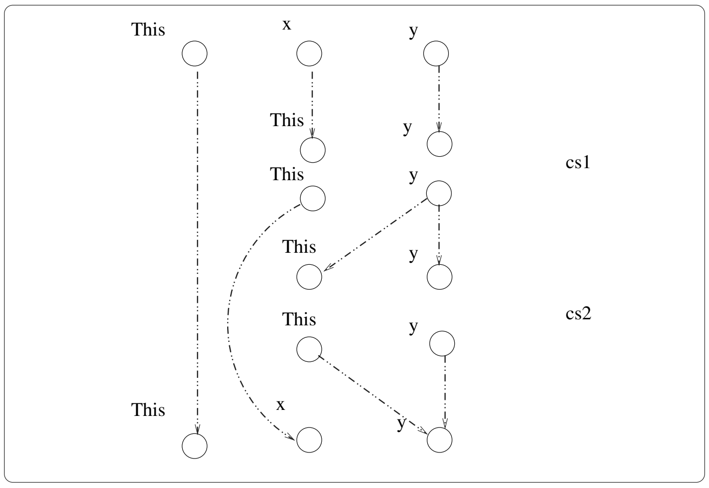

Fig. 1. Procedure `A::P`'s portion of PSG

that a call graph is already available and does not, therefore, apply to the demand driven call graph construction problem.

The rest of the paper is organized as follows. The demand driven call graph construction technique is reviewed in Section 2. Our experimental design is presented in Section 3 and experimental results are presented in Section 4. We conclude in Section 5.

2 Demand Driven Call Graph Construction

In this section, we review our demand driven call graph construction technique. More details of the technique are available from our previous papers [1,2].

We use the interprocedural representation Program Summary Graph (PSG), initially proposed by Callahan [5], for presenting our demand driven call graph analysis technique. Procedure `A::P`'s portion of PSG is shown in Figure 1. We also construct a relatively inaccurate initial call graph by performing relatively inexpensive Class Hierarchy Analysis (CHA) [7].

In presenting our technique, we use the following definitions.

$pred(v)$: The set of predecessors of the node v in the PSG. This set is initially defined during the construction of PSG and is not modified as the type information becomes more precise.

$proc(v)$: This relation is only defined if the node v is an entry node or an exit node. It denotes the name of the procedure to which this node belongs.

TYPES(v): The set of types associated with a node v in the PSG during any stage in the analysis. This set is initially constructed using Class Hierarchy Analysis, and is later refined through data-flow propagation.

THIS_NODE(v): This is the node corresponding to the THIS pointer at the procedure entry (if v is an entry node), procedure exit (if v is an exit node), procedure call (if v is a call node) or call return (if v is a return node).

THIS_TYPE(v): If the vertex v is a call node or a return node, THIS_TYPE(v) returns the types currently associated with the call node for the THIS pointer at this call site. This relation is not defined if v is an entry or exit node.

PROCS(S): Let S be the set of types associated with a call node for a THIS pointer. Then, PROCS(S) is the set of procedures that can actually be invoked at this call site. This function is computed using Class Hierarchy Analysis (CHA).

We now describe how we compute the set of nodes in the PSG for the entire program that influence the set of procedures invoked at the given call site c_i. The PSG for the entire program is never constructed. However, for ease in presenting the definition of the set of influencing nodes, we assume that the PSG components of all procedures in the entire program are connected based upon the initial sound call graph.

Let v be the call node for the THIS pointer at the call site c_i. Given the hypothetical complete PSG, the set of influencing nodes (which we denote by S) is the minimal set of nodes such that: 1) $v \in S$, 2) $(x \in S) \wedge (y \in pred(x)) \rightarrow y \in S$, and 3) $x \in S \rightarrow \text{THIS\_NODE}(x) \in S$

Starting from the node v, we include the predecessors of any node already in the set, until we reach internal nodes that do not have any predecessors. For any node included in the set, we also include the corresponding node for the THIS pointer (denoted by THIS_NODE) in the set.

The next step in the algorithm is to perform iterative analysis over the set of nodes in the Partial Program Summary Graph (PPSG) to compute the set of types associated with a given initial node. This problem can be modeled as computing the data-flow set TYPES with each node in the PPSG and refining it iteratively. The initial values of TYPES(v) are computed through class hierarchy analysis that we described earlier in this section. If a formal or actual parameter is declared to be a reference to class `cname`, then the actual runtime type of that parameter can be any of the subclasses (including itself) of `cname`.

The refinement stage can be described by a single equation, which is shown in Figure 2. Consider a node v in PPSG. Depending upon the type of v, three cases are possible in performing the update: 1) v is a call or exit node, 2) v is an entry node, and 3) v is a return node.

In Case 1, the predecessors of the node v are the internal nodes, the entry nodes for the same procedure, or the return nodes at one of the call sites within this procedure. The important observation is that such a set of predecessors does not change as the type information is made more precise. So, the set TYPES(v) is updated by taking union over the sets of TYPES(v) over the predecessors of the node v.

We next consider case 2, i.e., when the node v is an entry node. *proc*(v) is the procedure to which the node v belongs. The predecessors of such a node are call nodes at all call sites at which the function *proc*(v) can possibly be called, as per the initial call graph assumed by performing class hierarchy analysis.

$$\text{TYPES}(v) = \begin{cases} \text{TYPES}(v) \bigcap (\bigcup_{p \in pred(v)} \text{TYPES}(p)) \\ \qquad \text{if v is call or exit node} \\ \text{TYPES}(v) \bigcap (\bigcup_{(p \in pred(v)) \wedge (proc(v) \in \text{PROCS}(\text{THIS\_TYPE}(p)))} \text{TYPES}(p)) \\ \qquad \text{if v is an entry node} \\ \text{TYPES}(v) \bigcap (\bigcup_{(p \in pred(v)) \wedge (proc(p) \in \text{PROCS}(\text{THIS\_TYPE}(v)))} \text{TYPES}(p)) \\ \qquad \text{if v is a return node} \end{cases}$$

Fig. 2. Data-flow equation for propagating type information

Such a set of possible call sites for *proc*(v) gets restricted as interprocedural type propagation is performed. Let p be a call node that is a predecessor of v. We want to use the set TYPES(p) in updating TYPES(v) only if the call site corresponding to p invokes *proc*(v). We determine this by checking the condition *proc*(v) $\in$ PROCS(THIS_TYPE(p)). The function THIS_TYPE(p) determines the types currently associated with the THIS pointer at the call site corresponding to p and the function PROCS determines the set of procedures that can be called at this call site based upon this type information.

Case 3 is very similar to the case 2. If the node v is a return node, the predecessor node p to v is an exit node. We want to use the set TYPES(p) in updating TYPES(v) only if the call site corresponding to v can invoke the function *proc*(p). We determine this by checking the condition *proc*(p) $\in$ PROCS(THIS_TYPE(v)). The function THIS_TYPE(v) determines the types currently associated with the THIS pointer at the call site corresponding to v and the function PROCS determines the set of procedures that can be called at this call site based upon this type information.

Theoretical Results: The technique has two major theoretical properties [2]. The worst-case complexity of our analysis is the same as the well known 0-CFA exhaustive analysis technique [18], except that our input is the cardinality of the set of influencing nodes, rather than the total number of nodes in the program representation. Thus, the advantage of our demand driven technique depends upon the ratio of the size of set of influencing nodes and the total number of nodes. Second, we have shown that the type information computed by our technique for all the nodes in the set of influencing nodes is as accurate as the 0-CFA exhaustive analysis technique.

3 Experiment Design

We have implemented our demand driven technique using the SABLE infrastructure developed at McGill University [22]. In this section, we describe the design of the experiments conducted, including benchmarks used, scenarios used for evaluating demand driven call graph constructions, and metrics used for comparison.

Benchmark Programs: We have primarily used programs from the most commonly used benchmark set for Java programs, SPECJVM. The 10 SPECJVM programs are `check`, `compress`, `jess`, `raytrace`, `db`, `javac`, `mpegaudio`, `mtrt`,

Benchmark	no. of classes	no. of methods	no. of PSG nodes
check	20	96	3954
compress	15	35	601
jess	8	41	1126
raytrace	28	130	6518
db	6	34	1452
javac	180	1004	48147
mpegaudio	58	270	6205
mtrt	4	6	51
jack	61	261	14080
checkit	6	8	495

Fig. 3. Description of benchmarks

`jack`, and `checkit`. The total number of classes, methods, and PSG nodes for each of these benchmarks is listed in Figure 3. The number of classes ranges from 4 to 180, the number of methods ranges from 6 to 1004, and the number of PSG nodes ranges from 51 to 48147.

Scenarios for Experiments: In Section 2, our technique was presented under the assumption that the call graph edges need to be computed for a single call site. In practice, demand driven analysis may be invoked under more complex scenarios. For example, one may be interested in knowing the reaching definitions for a set of variables in a method. Performing this analysis may require knowing the methods invoked at a set of call sites in the program. Thus, demand driven call graph analysis may be performed to determine the call graph edges at the call sites within this set. Alternatively, there may be interest in fully analyzing a single method or a class, and selectively analyzing codes from other methods or classes to have more precise information within the method or class.

We have conducted experiments to evaluate demand driven call graph construction under the following scenarios:

– *Experiment A:* Resolving a single call site in the program. We have only considered the call sites that can potentially invoke multiple methods after Class Hierarchy Analysis (CHA) is applied. This is the simplest case for the demand driven technique, and should require analyzing only a small set of procedures and PSG nodes in the program.
– *Experiment B:* Computing reaching definitions of all actual parameters at all call sites within a method. Computing interprocedural reaching definitions will typically require knowing calling relationship at a set of call sites. This scenario depicts a situation in which demand driven call graph construction is invoked while computing certain data-flow information on a demand basis.
– *Experiment C:* Resolving all call sites within a method. This is more complicated than the experiment A above, and represents a more realistic case when interprocedural optimizations are applied at a portion of the program.

– *Experiment D:* Resolving all call sites within all methods within a class. This scenario represents analyzing a single class, but performing selective analysis on portions of code from other classes to improve the accuracy of analysis within the class.

Metrics Used: We now describe the metrics used for reporting the benefits of demand driven call graph construction over exhaustive call graph analysis. Performing demand driven analysis will require fewer PSG nodes to be analyzed, fewer procedures to be analyzed, and should require lesser time. We individually report these three factors. Specifically, the three metrics used are:

– *Time Ratio:* This is the ratio of the time required for demand driven analysis, as compared to exhaustive analysis. This metric evaluates the benefits of using demand driven analysis, but is dependent on our implementation.
– *Node Ratio:* This is the ratio of the number of nodes in PPSG to the total number of nodes in PSG of the entire program. This metric is an implementation independent indicative of the benefits of the analysis.
– *Procedure Ratio:* This is the ratio of the number of methods analyzed during demand driven analysis, as compared to the total number of methods in the entire program. Since each method's portion of the full program representation used in our analysis is constructed only if that method needs to be analyzed, and is always constructed in entirety if the methods needs to be analyzed; this metric demonstrates the space-efficiency of demand driven call graph construction.

4 Experimental Results

We now present the results from our experiments. Our experiments were conducted on a Sun 250 MHz Ultra-Sparc processor with 512 MB of main memory. We first present results from exhaustive analysis. Then, we present results from demand driven analysis for scenarios A, B, C, and D.
Exhaustive Analysis: To provide a comparison against demand driven analysis, we first include the results from exhaustive 0-CFA call graph construction on our set of benchmarks.

The results from exhaustive analysis are presented in Figure 4. The time required for Class Hierarchy Analysis (CHA), time required for the iterative call graph refinement, and the number of call sites that are not-monomorphic after applying CHA are shown here. Call sites that can potentially invoke multiple methods after CHA has been applied are the ones that can benefit from more aggressive iterative analysis.

The time required in CHA phase in our implementation is dominated by setting up of data-structures, and turns out to be almost the same for all benchmarks. The time required for the iterative refinement phase varies a lot between benchmarks, and is roughly proportional to the size of the benchmark.

Two important observations from the Figure 4 are as follows. First, only 4 of the 10 programs have call sites that are polymorphic after the results of

Benchmark	CHA time (sec.)	Iter. Analysis (sec.)	Polymorphic Call Sites After CHA
check	72.3	27.7	0
compress	84.5	13.3	0
jess	96.5	59.4	0
raytrace	82.1	60.9	39
db	72.8	12.0	0
javac	85.6	2613	577
mpegaudio	73.4	462	35
mtrt	80.2	3.5	0
jack	74.1	250.7	77
checkit	73.6	5.3	0

Fig. 4. Results from exhaustive analysis

CHA are known. These 4 programs are `raytrace`, `javac`, `mpegaudio`, and `jack`. These are also the 4 largest programs among the programs in this benchmark set, comprising 28 to 180 classes and 130 to 1004 methods. For the smaller programs, CHA is as accurate as any analysis for constructing the call graph. The second observation is that for 7 of 10 programs, the total time required for exhaustive call graph construction is dominated by the CHA phase. For the three remaining programs, `javac`, `mpegaudio`, and `jack`, the time required for iterative analysis is 30 times, 6 times, and nearly 4 times the time required for CHA analysis, respectively.

Therefore, for the smaller programs in the benchmark set, CHA analysis is sufficient, and they do not benefit from more aggressive analysis. The dominant cost of analysis is CHA, which remains the same during demand driven call graph construction. So, these programs cannot benefit from demand driven analysis.

On the other hand, the time required for analysis is dominated by the iterative phase in the larger programs. A large number of call sites are polymorphic after applying CHA, and are therefore likely to benefit from iterative analysis. Since the iterative analysis is applied on a much small number of nodes in the demand driven technique, these programs are likely to benefit from the proposed demand driven analysis. This is analyzed in details in the remaining part of this section.

Experiment A: In the first set of experiments, we perform demand driven analysis to resolve a single call site in the program. We only consider call sites that are known to potentially invoke multiple procedures after CHA has been applied.

As we described in the previous subsection, only `raytrace`, `javac`, `mpegaudio`, and `jack` contain such *polymorphic* call sites. Therefore, the results are only presented from these call sites.

The averages for time ratio, node ratio, and procedure ratio for these 4 programs is shown in Figure 5. The analysis time compared in this table is the time

Benchmark	No. of Cases	Analysis Time		PPSG Nodes		Procedures	
		Avg. (sec.)	Ratio	Avg. No.	Ratio	Avg. No.	Ratio
raytrace	39	3.78	6.2%	96.6	1.5%	13.7	10.5%
javac	577	341.2	13.1%	9831	20.4%	747	74.5%
mpegaudio	35	15.6	3.3%	186.3	3.0%	31.9	11.8%
jack	77	11.8	4.7%	422.3	2.9%	46.1	17.6%

Fig. 5. Results from experiment A

for iterative analysis only. For both demand driven and exhaustive versions, additional time is spent in performing CHA.

The average of the ratio of the number of nodes that need to be analyzed during demand driven analysis is extremely low for `raytrace`, `mpegaudio`, and `jack`, ranging between 1.5% and 3.0%. This results in an average iterative analysis time ratio of less than 7%. Even the number of procedures that need to be analyzed is less than 20% for these three programs. The results for `javac` are significantly different, but still demonstrate gains from the use of demand driven analysis. The average node ratio is 20.4%, resulting in an average time ratio of 13.1%. However, the average procedure ratio is nearly 75%. This means that for most of the cases, a very large fraction of procedures need to be involved in demand driven analysis. Use of demand driven analysis does not result in significant space savings for `javac`.

After including the time for CHA, the average time ratio are 60%, 16%, 17%, and 26% for `raytrace`, `javac`, `mpegaudio`, and `jack`, respectively. The gains from demand driven analysis for `raytrace` are limited, because the time required for CHA exceeds the exhaustive iterative analysis time. `javac`, which had the highest ratio before CHA time was included, has the lowest ratio after including CHA because the time required for exhaustive iterative analysis is more than 30 times the time required for CHA.

Demand driven analysis gives clear benefits in the case of `javac`, `mpegaudio`, and `jack`, because the time required for the iterative phase dominates the time required for CHA. To further study the results from these three benchmarks, we present a series of *cumulative frequency* graphs. For the experiment A, cumulative frequency graphs for the benchmarks `javac`, `mpegaudio`, and `jack` are presented in Figures 6, 7, and 9, respectively. A point (x, y) in such a graph means that the fraction x of the cases in the experiments had a ratio of less than or equal to y.

The results from `javac` follow an interesting trend. 56 of the 577 cases require analysis of 120 or fewer procedures, or nearly 12% of all procedures. The same set of cases requires analyzing 257 or fewer nodes, or less than 1% of all nodes. The time taken for these cases is also less than 2% of the time for exhaustive analysis. However, the ratios are very different for the remaining cases. The next 413 cases require analysis of the same set of 837 procedures, or 83% of all procedures. The remaining cases require between 838 and 876 procedures to be analyzed. The analysis time is between 15% and 20% of the exhaustive analysis time, and the number of nodes involved for these cases is nearly 25% of the total number of nodes.

The results from `mpegaudio` are as follows. 11 of the 35 cases require analysis of between 73 and 98 procedures, or between 27% and 36% of all procedures. The same 11 cases require analysis of between 8% and 10% of nodes, and between 2% and 4% of time. The other 24 cases require analysis of less than 12% of all procedures, and less than 1.5% of nodes and time.

For `jack`, 61 of 77 cases require analysis of 59 or 57 procedures, or nearly 20% of all procedures. The same set of cases require between 4% and 6% of time, and 2% and 4% of all nodes. The other 16 cases involve analyzing less than 5% of all procedures, less than 1% of time, and less than 0.5% of all nodes.

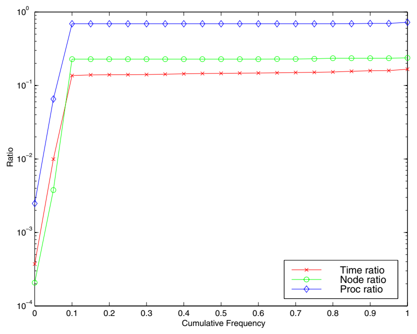

Fig. 6. Experiment A: Cumulative frequency of time, node, and procedure ratio for `javac`

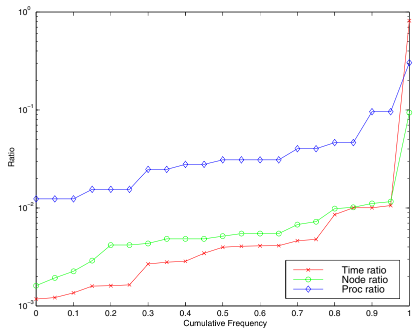

Fig. 7. Experiment A: Cumulative frequency of time, node, and procedure ratio for `mpegaudio`

Experiment B: In the second set of experiments, we evaluated the performance of demand driven call graph construction when it is initiated from demand driven data flow analysis. The particular data flow problem we consider is the computation of reaching definitions for all actual parameters in a procedure. We report results from this experiment only on `raytrace`, `mpegaudio`, and `jack`. The 6 smaller programs in SPECJVM benchmark set do not contain any polymorphic call sites. Even after many attempts, we could not complete this experiment for `javac`, which is the largest program in this benchmark set. We believe that it was because of very large memory requirements when reaching definition and call graph construction analyses are combined.

The average time, node, and procedure ratios for the three benchmarks are presented in Figure 8. As compared to the experiment A, we are reporting results from a significantly larger number of cases, because this analysis was performed on all procedures. At the same time, for many cases in experiment B resolution of several polymorphic call sites may be required. The three ratios for `mpegaudio` are lower for the experiment B, as compared to the ones obtained from experi-

Benchmark	No. of Cases	Analysis Time		PPSG Nodes		Procedures	
		Avg. (sec.)	Ratio	Avg. No.	Ratio	Avg. No.	Ratio
raytrace	129	4.36	7.2%	354.3	5.4%	28.7	22.0%
mpegaudio	270	5.48	1.2%	133.5	2.2%	26.8	9.9%
jack	261	15.44	6.2%	524.9	3.7%	94.8	36.6 %

Fig. 8. Results from experiment B

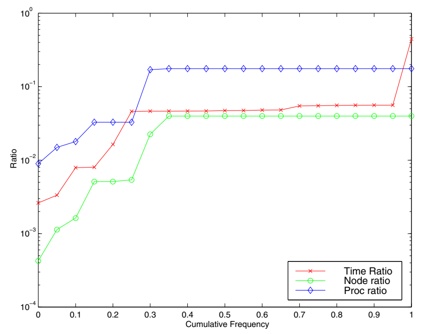

Fig. 9. Experiment A: Cumulative frequency of time, node, and procedure ratio for `jack`

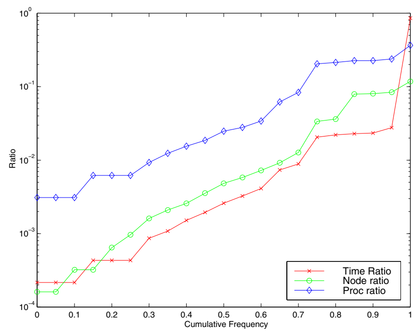

Fig. 10. Experiment B: Cumulative frequency of time, node, and procedure ratio for `mpegaudio`

ment A. For `raytrace` and `jack`, the reverse is true; the three ratios are higher for the experiment B.

The ratio for iterative analysis time are 7.2%, 1.2%, and 6.2% for `raytrace`, `mpegaudio`, and `jack`, respectively. After including the time for CHA, the ratios of the time required are 60%, 14%, and 27%, respectively.

We studied the results in more details for `mpegaudio` and `jack`. The cumulative frequency plots for these two benchmarks are presented in Figures 10 and 11, respectively.

The results from `mpegaudio` are as follows. 192 of 270 cases require analysis of 33 or fewer procedures, or less than 12% of all procedures. The same set of cases require analysis of less than 2% of all nodes, and take less than 1% of time for exhaustive analysis. For the remaining cases, the number of procedures to be analyzed is distributed fairly uniformly between 66 and 118.

For `jack`, the trends are very different. 126 of 261 cases require analysis of 162 or 161 procedures, or nearly 62% of all procedures. The same set of cases require analysis of nearly 800 nodes, or 6% of all nodes. The time required for this set of cases is nearly 9% of the time for exhaustive iterative analysis. The portions of the program that need to be analyzed for this set of cases (48% of all cases) is almost the same. This has the following implications. If demand driven analysis is performed for one of these cases, and then needs to be performed for another case in the same set, very limited additional effort will be required.

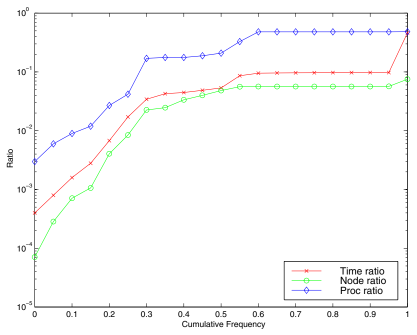

Fig. 11. Experiment B: Cumulative frequency of time, node, and procedure ratio for `jack`

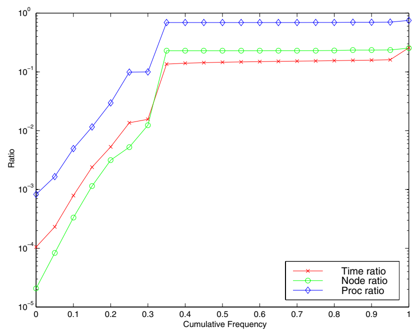

Fig. 12. Experiment C: Cumulative frequency of time, node, and procedure ratio for `javac`

Benchmark	No. of Cases	Analysis Time		PPSG Nodes		Procedures	
		Avg. (sec.)	Ratio	Avg. No.	Ratio	Avg. No.	Ratio
raytrace	130	4.51	7.4%	358.9	5.5%	29.1	22.4%
javac	1004	271.1	10.3%	7634.5	15.8 %	587	58.5%
mpegaudio	270	5.37	1.2%	133.5	2.1 %	26.8	9.9%
jack	261	14.9	5.9%	524.9	3.7%	94.8	36.3%

Fig. 13. Results from experiment C

Experiment C: Our next set of experiments evaluated the performance of demand driven call graph construction when all call sites in a procedures had to be resolved. We present data only from `raytrace`, `javac`, `mpegaudio`, and `jack`, because they contain polymorphic call sites. For these programs, we include results from analysis of all methods, even if they do not contain any polymorphic call site.

The averages of time, node, and procedure ratios are presented in Figure 13. The averages are very close to the results for experiment B. We believe that this because all call sites in a method had to be resolved for experiment C, and all cites that can potentially invoke a method had to be resolved for experiment B.

The three ratios for `javac` are lower for experiment C, as compared to the experiment A. This is because the averages are taken over much larger number of cases in the experiment C. Many of the procedures do not require analysis of any polymorphic call site, and contribute to a lower overall average.

The cumulative frequency plots for `javac`, `mpegaudio`, and `jack` are presented in Figures 12, 14, and 15, respectively.

Results from `javac` for experiment C are similar to the results from experiment A, with one important difference. A larger fraction of cases can be analyzed with a small fraction of procedures and nodes. 316 of 1004 cases require between 1 and 125 procedures, or up to 12% of all procedures. The remaining 688 cases require between 837 and 907 procedures, nearly 25% of all nodes, and nearly 15% of exhaustive analysis time.

Results from `mpegaudio` for experiment C are very similar to the results from experiment B. 192 of 270 cases (the same number as in experiment B) require analysis of at most 33 procedures, while the remaining cases need analysis of between 66 and 118 procedures. The same trend (closeness between results from experiments B and C) continues for `jack`.

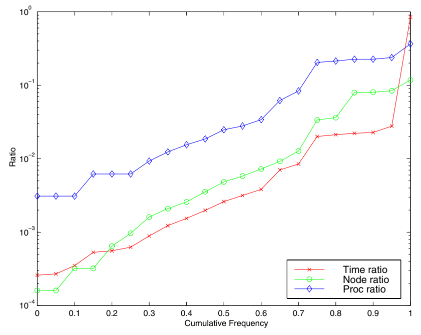

Fig. 14. Experiment C: Cumulative frequency of time, node, and procedure ratio for `mpegaudio`

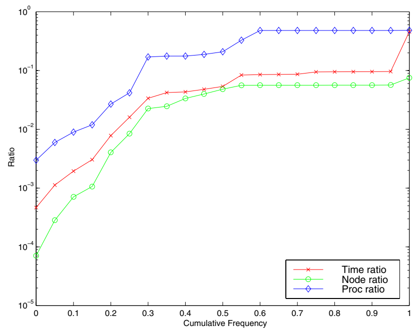

Fig. 15. Experiment C: Cumulative frequency of time, node, and procedure ratio for `jack`

Experiment D: Our final set of experiments evaluates demand driven analysis when all call sites in all procedures of a class are to be resolved. Figure 16 presents average time ratio, node ratio, and procedure ratio for `raytrace`, `javac`, `mpegaudio`, and `jack`. Even though each invocation of demand driven analysis may involve resolving several call sites, the ratio are quite small. For `raytrace`, `mpegaudio`, and `jack`, the averages of time ratios and node ratios are still less than 10%. The averages for `javac` are a bit higher, consistent with the previous experiments. The average time ratio and node ratio are 13.1% and 20.6%, respectively. Space savings are not significant with `javac`, but quite impressive for the other three benchmarks. After including the time required for CHA, the average time ratio is 61% for `raytrace`, 16% for `javac`, 16% for `mpegaudio`, and 25% for `jack`.

In comparison with the results from experiment C, the averages of ratios from experiment D are all higher for `raytrace`, `javac`, and `mpegaudio`, as one would normally expect. The surprising results are from `jack`, where all three ratios are lower in experiment D. The explanation for this is as follows. The results from experiment D are averaged over a smaller number of cases, specifically, 61 instead of 261 for `jack`. It turns out that the procedures that require the most time, number of nodes, and number of procedures to be analyzed belong to a small set of classes. Therefore, they contribute much more significantly to the

Benchmark	No. of Cases	Analysis Time		PPSG Nodes		Procedures	
		Avg. (sec.)	Ratio	Avg. No.	Ratio	Avg. No.	Ratio
raytrace	28	5.32	8.7%	598.3	9.2%	41.5	31.9%
javac	180	343.6	13.1%	9940	20.6%	741.3	73.8%
mpegaudio	58	14.1	3.1%	280.5	4.5%	47.6	17.6 %
jack	61	7.49	4.7%	291.3	2.1%	27.6	10.5%

Fig. 16. Results from experiment D

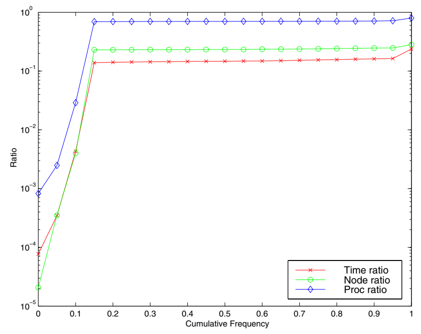

Fig. 17. Experiment D: Cumulative frequency of time, node, and procedure ratio for `javac`

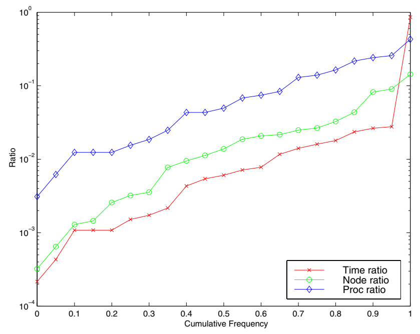

Fig. 18. Experiment D: Cumulative frequency of time, node, and procedure ratio for `mpegaudio`

average ratios in the results from the experiment C, than in the results from experiment D.

Details of the results from `javac`, `mpegaudio`, and `jack` are presented in Figures 17, 18, and 19, respectively. Again, the results from `javac` are very different from the results on the other two benchmarks. In `javac`, 20 of the 180 classes can be resolved by analyzing a small fraction of procedures. Specifically, these cases require analysis of between 1 and 63 procedures, i.e., less than 7% of all procedures in the program. However, the other 160 cases require analysis of between 837 and 963 procedures in the program. Each of the cases from this set requires analyzing nearly 25% of all the nodes in the program, and between 15% and 20% of the time for exhaustive analysis. However, the sets of influencing nodes that need to analyzed for these cases are almost identical. Our theoretical result, therefore, implies that after one of these cases has been analyzed, the time required for other cases will be very small.

For `mpegaudio`, the number of procedures that need to be analyzed for the 58 cases ranges from 1 to 139, or from less than 1% to nearly 50%. The distribution is fairly uniform. The time required for demand driven analysis for these cases also has a fairly uniform distribution, between 0.1 second to 22.5 second, or between 0.02% to 5% of the time required for exhaustive analysis. Similarly, the

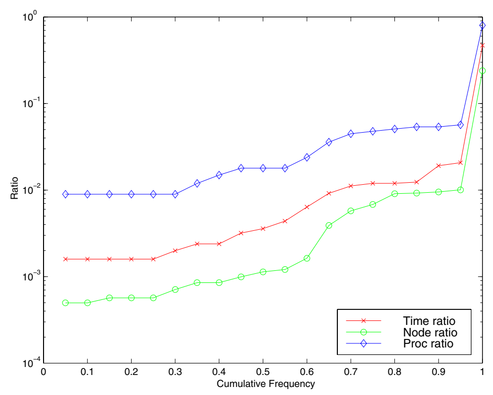

Fig. 19. Experiment D: Cumulative frequency of time, node, and procedure ratio for `jack`

number of nodes ranges from 2 to 880, or from 0.03% to 13%. The results from `jack` are similar.

5 Conclusions

We have presented evaluation of an algorithm for resolving call sites in an object oriented program on a demand driven fashion. The summary of our results using SPECJVM benchmarks is as follows:

- The time required for Class Hierarchy Analysis (CHA), which is a prerequisite for both exhaustive and demand driven iterative analysis, dominates the exhaustive call graph construction time for 7 of the 10 SPECJVM programs. However, CHA itself is sufficient for constructing an accurate call graph for 6 of these 7 programs. The time required for exhaustive iterative analysis clearly dominates CHA time for the three largest SPECJVM programs, `javac`, `mpegaudio`, and `jack`.
- For resolving a single call site, demand driven iterative analysis averages at nearly 10% of the time required for exhaustive iterative analysis. The number of nodes that need to be analyzed averages at nearly 3% for `mpegaudio` and `jack`, but around 20% for `javac`. The number of procedures that need to be analyzed is less than 20% for `mpegaudio` and `jack`, but nearly 75% for `javac`.
- The averages for the number of nodes and procedures analyzed and the time taken surprisingly stays low when all call sites within a class or a method are analyzed instead of a single call site. This is because the program portions that need to be analyzed for resolving different call sites within a method or a class are highly correlated.

References

1. Gagan Agrawal. Simultaneous demand-driven data-flow and call graph analysis. In *Proceedings of International Conference on Software Maintainance (ICSM)*, September 1999. 30, 31
2. Gagan Agrawal. Demand-drive call graph construction. In *Proceedings of the Compiler Construction (CC) Conference*, March 2000. 30, 31, 33
3. David Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '96)*, pages 324–341, October 1996. 30
4. Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 397–408, Portland, Oregon, January 1994. 30
5. D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, Atlanta, GA, June 1988. 31
6. R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant Context Inference. In *Proceedings of the Conference on Principles of Programming Languages (POPL)*, pages 133–146, January 1999. 30
7. Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 93–102, La Jolla, California, 18–21 June 1995. *SIGPLAN Notices* 30(6), June 1995. 31
8. Greg DeFouw, David Grove, and Craig Chambers. Fast interprocedural class analysis. In *Proceedings of the POPL'98 Conference*, 1998. 30
9. A. Diwan, K. S. McKinley, and J. E. B. Moss. Using Types to Analyze and Optimize Object-Oriented Programs. *ACM Transactions on Programming Languages and Systems*, 23(1):30–72, January 2001. 30
10. E. Duesterwald, R. Gupta, and M. L. Soffa. A Practical Framework for Demand-Driven Interprocedural Data Flow Analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, November 1997. 30
11. David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, 1997. 30
12. S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *In SIGSOFT '95: Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 104–115, 1995. 30
13. Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 367–378, San Francisco, California, January 1995. 30
14. Hemant Pande and Barbara Ryder. Data-flow-based virtual function resolution. In *Proceedings of the Third International Static Analysis Symposium*, 1996. 30
15. M. Porat, M. Biberstein, L. Koved, and M. Mendelson. Automatic detection of immutable fields in Java. In *Proceedings of CASCON*, 2000. 30
16. Gregg Rothermel and M. J. Harrold. Analyzing regression test selection. *IEEE Transactions on Software Engineering*, 1996. 30
17. Atanas Routnev, Barbara G. Ryder, and William Landi. Data-Flow Analysis of Program Fragments. In *Proceedings of the Conference on Foundations of Software Engineering (FSE)*, pages 235–253, September 1999. 30

18. O. Shivers. The semantics of Scheme control-flow analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, volume 26, pages 190–198, New Haven, CN, June 1991. 30, 33
19. V. C. Sreedhar, M. Burke, and J. D. Choi. A Framework for Interprocedural Optimization in the Presence of Dynamic Class Loading. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2000. 30
20. Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallee-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Fifteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '2000)*, pages 264–280. ACM Press, October 2000. 30
21. Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Fifteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '2000)*, pages 281–293. ACM Press, October 2000. 30
22. Raja Vallee-Rai. Soot: A Java ByteCode Optimization Framework. Master's thesis, McGill University, 1999. 30, 33
23. Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, 1984. 30
24. A. Zaks, V. Feldman, and N. Aizikowitz. Sealed calls in java packages. In *Proceedings of Conference on Object Oriented Programming Systems and Languages (OOPSLA)*, pages 83–92. ACM Press, October 2000. 30