Profile-Guided Context-Sensitive Program Analysis *

Saumya Debray
Department of Computer Science
University of Arizona
Tucson, AZ 85721, U.S.A.
debray@cs.arizona.edu

Abstract

Interprocedural analyses can be classified as either context-insensitive, which tend to sacrifice precision to gain efficiency, or context-sensitive, which are more precise but also more expensive. This paper discusses a profile-guided approach to interprocedural analysis that is context-sensitive, and hence more precise, for the "important" call sites for a function, and context-insensitive, and hence more efficient, for the "unimportant" call sites. Experiments indicate that this approach can be significantly more efficient than a traditional context-sensitive analysis without sacrificing much of the pragmatic value of the dataflow information gathered.

1 Introduction

For code analysis and optimization purposes, compilers typically construct a control flow graph for each function in a program [1]. Control flow across function boundaries is often represented using an *interprocedural control flow graph* (e.g., see [15]), which consists of the control flow graphs of all the functions in the program, together with edges representing calls and returns that link the flow graphs of different functions. A function call is represented using a pair of nodes, a *call node* and a *return node*: there is an edge, called the *call edge*, from a call node to the entry node of the callee, with a corresponding edge, called a *return edge*, from the exit node of the callee to the return node.

Interprocedural program analyses can be classified as being either *context-sensitive* or *context-insensitive*, depending on the manner in which dataflow information is propagated from a call site, through a called procedure, and back to the basic block to which execution returns at the end of the call. Traditionally, these have been viewed as fundamentally different approaches to interprocedural analysis. A context-insensitive program analysis does not give any special treatment to call edges or return edges, except possibly for the book-keeping that might be involved in going from the caller's name space to the callee's or vice versa. The dataflow information at the entry to a function is computed as the meet of the information at the various call sites for that function,[1] the dataflow information eventually obtained at the exit node for the function is then propagated to the return node at each of its call sites. As a result, the analysis algorithms remain relatively simple and efficient. The drawback, however, is that of loss in precision, because information can be propagated along unrealizable paths in the interprocedural control flow graph, i.e., paths that can never occur at runtime because of mismatched call/return edges.

The problem of imprecision can be addressed using a context-sensitive analysis. Such analyses propagate information only along realizable execution paths, i.e., they don't propagate information from the call node of one call site, through the body of the callee, and then to the return node of a different call site. Since the results of analysis for one call site are not "polluted" by dataflow information from another call site, this leads to more precise dataflow information, but the analysis algorithms and data structures become more complex and potentially more expensive. For example, comparing context-insensitive and context-sensitive control flow analyses for object-oriented languages, Grove *et al.* report that for many programs, even modest amounts of context-sensitivity lead to huge increases in resources used, going from around 1 minute of analysis time and 1–10

* This work was supported in part by the National Science Foundation under grant CCR-9711166.
[1] The discussion here is couched in terms of a forward analysis; the situation is analogous for backward analyses.

MB of memory to over 24 hours(!) of time and/or 450 MB of memory [12]. Comparing pointer alias analyses for C programs, Ruf reports that the context-sensitive version is 2–3 orders of magnitude slower than the context-insensitive version [17]. These experiences suggest that it would be worthwhile to try and reduce the cost of context-sensitive analyses if this could be done without significantly affecting the pragmatic utility of the information produced by them.

In practice, it turns out that the dynamic distribution of calls for the different call sites of a function typically tends to be skewed towards a small number of frequently executed call sites. For example, in the SPEC-95 benchmark *m88ksim*, the most frequently called function, `uext()`, has 19 call sites, of which only two are frequently executed; these two call sites account for almost 95% of the calls to this function at runtime.[2] In the SPEC-95 benchmark *perl*, the most frequently called function, `eval()`, has 40 call sites: of these, the two most frequently executed call sites account for over 81% of the calls to the function, and if we consider the top four call sites, we get over 98% of the dynamic calls to it. Figure 1 shows, for the eight SPEC-95 integer benchmarks, the fraction of the total number of dynamic calls in a program (plotted along the y-axis) that are accounted for when we consider at most a fixed fraction of each function's call sites (plotted along the x-axis). It can be seen that considering at most half of the call sites of each function allows us to account for almost 75% of all the dynamic calls in the program; if we disregard *li* and *m88ksim*, considering only a third of the call sites of each function suffices to account for close to 80% of all the dynamic calls.[3] What this means is that most call sites aren't executed all that often. For the frequently executed call sites of a function, a context-sensitive analysis gives us the precision that we desire, but at the cost of expending considerable time and space resources analyzing the infrequently executed call sites to a function. A context-insensitive analysis doesn't incur these overheads, but the efficiency gains come at the cost of sacrificing precision for all of the call sites, including the frequently executed ones for which we would like precise dataflow information. Pragmatically, therefore, neither of these two approaches to interprocedural analysis is entirely satisfactory.

This paper describes a profile-guided approach to context-sensitive interprocedural analyses that attempts to address this situation in a way that obtains the precision benefits of context-sensitivity for frequently executed call sites but doesn't waste a lot of resources on infrequently executed ones. Using this approach, "classical" context-sensitive and context-insensitive analyses are no longer seen as two fundamentally distinct approaches to program analysis, but rather as simply the two extreme points of an entire spectrum of different analyses. The intermediate points of this spectrum, which have cost-benefit tradeoffs in between the two extremes, can be interesting and useful from a pragmatic perspective. We illustrate our ideas using an alias analysis for executable code [9], where the alias information is used for instruction scheduling; however, the underlying ideas are quite general and applicable to other context-sensitive interprocedural analyses as well.

2 Profile-Guided Context-Sensitive Analysis

The intuition behind our idea is very simple. Our goal is to incur the time and space overheads of a context-sensitive analysis only for the important call sites of a function. To this end, we use execution profile information to partition the call sites of each function into equivalence classes. The idea is to carry out inter-procedural analysis in such a way that it is context-insensitive *within* each partition, but context-sensitive *across* partitions. In terms of inter-procedural execution paths, this means that the analysis propagates dataflow information along paths such that a call edge from a function f to a function g is matched up with a return edge from g to a function h if and only if the corresponding call sites in f and h are in the same partition. Note that in the special case where, for each function, all of its the call sites are grouped into the same partition, we get a traditional context-insensitive analysis. On the other hand, in the case where every call site for each function is in a different partition by itself, we get a traditional context-sensitive analysis.

[2] Unless otherwise mentioned, execution profiles for SPEC-95 benchmarks refer to their training inputs, since that is what a compiler would use to guide its decisions.

[3] The reason the fraction of dynamic calls don't go to 1.0 until we consider all call sites is that, for a given fraction on the x axis, we consider *at most* that fraction of a function's call sites. Because of this, functions that have exactly one call site aren't considered until $x = 1.0$.

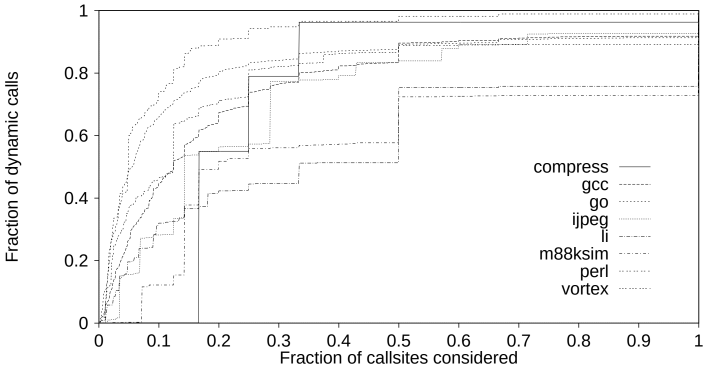

Figure 1: Call site execution frequency distributions: SPEC-95 integer benchmarks

Consider a function f that has six call sites $c_1, \ldots, c_6$. Call sites c_1, c_2 and c_3 are executed a large number of times; the remainder are infrequently executed. Further, suppose that the dataflow information available about the arguments to f at call sites c_2 and c_3 differ only on one argument, which happens to be used only in a very infrequently executed portion of the body of f. The analysis of f could then be carried out as follows:

- Since call sites c_4, c_5, and c_6 are infrequently executed, it doesn't really matter, pragmatically, if the dataflow information obtained for these call sites is precise or not. We could, therefore, simply "merge" the dataflow information about the arguments to f (i.e., compute their meet) at these three call sites, analyze f once using this merged information, and propagate the resulting information at the exit from f to the return blocks for these call sites.
- Since call sites c_2 and c_3 happen to disagree only on an unimportant argument—that is, one that is used only in an infrequently executed portion of f—we could merge the dataflow information at these two call sites, analyze f once using the merged information, and propagate the results back to this pair of call sites. This would lose information about the argument these call sites disagree on, but since this argument is only used in an infrequently executed portion of the body of f, the pragmatic impact of this loss of information is likely to be small.

In this case, we have partitioned the call sites for f into three sets: $\{c_1\}$, $\{c_2, c_3\}$, and $\{c_4, c_5, c_6\}$. Note that, while the partitioning is driven by profile information, it need not be a simple-minded decision made purely by the relative execution frequency of the different call sites for a function. It is possible, and can be useful, to take into account the perceived utility of the dataflow information (whose evaluation may very well use execution profile information as well) in different partitions. This was done, in the example above, for call sites c_2 and c_3, which were placed into the same partition even though they are both frequently executed.

We sketch here the changes necessary to adapt a traditional context sensitive analysis to use our profile-guided approach, which are fairly minor. We make the following assumptions:

1. The analysis is carried out over a domain of values D that is a complete meet-semilattice with meet operator $\wedge$ and greatest element $\top$ (i.e., $x \wedge \top = \top \wedge x = x$ for all $x \in D$).
2. For each procedure p in the program, the analysis maintains a function $\mathsf{Summary}_p$: given (an encoding of) a calling context κ for a procedure p, $\mathsf{Summary}_p(\kappa)$ returns a pair $(\bar{x}, \bar{y})$, where $\bar{x}$ summarizes the dataflow information at the entry to p for any call originating at the given calling context, and $\bar{y}$ describes the corresponding dataflow information at the exit from p.

3. The dataflow information currently available for a procedure p and calling context κ, at the entry to p, is given by the function $\mathsf{EntryInfo}_p(\kappa)$:

$$\mathsf{EntryInfo}_p(\kappa) = \begin{cases} \bar{x} & \text{if } (\bar{x}, \bar{y}) = \mathsf{Summary}_p(\kappa) \text{ is defined} \\ \top & \text{otherwise} \end{cases}$$

4. The analysis of a procedure p for a calling context κ, with dataflow information $\bar{u}$ at the entry to p, is carried out by a procedure $\mathsf{analyze\_proc}(p, \bar{u}, \kappa)$. If the analysis is in a recursion cycle, this procedure computes an approximation for the exit information instead of analyzing the body of the procedure. Otherwise, it maps the dataflow information $\bar{u}$ to the locals of p to obtain an initial set of dataflow facts $\bar{v}$; initializes the analysis of the body of p using the dataflow information $\bar{x} = \bar{v} \wedge \mathsf{EntryInfo}_p(\kappa)$, i.e., using the meet of the previously recorded entry information for that calling context and the current information at the call site; analyzes the body of p starting with this initial dataflow information; and updates $\mathsf{Summary}_p(\kappa)$ to be $(\bar{x}, \bar{y})$, where $\bar{y}$ is the resulting dataflow information at the exit from p.

None of this is very new, and we don't pursue the details further. There are also obvious refinements, such as storing in $\mathsf{Summary}_p$ only the dataflow information that may be affect or be affected by to a caller of p; these are not directly relevant for our purposes and so are not discussed here. A traditional context-sensitive analysis involves initializing dataflow values to $\top$ and then repeatedly traverse its call graph, analyzing each procedure at each of its call sites with the appropriate dataflow information available at that call site, until there is no change anywhere.

In order to adapt this to a profile-guided analysis, we proceed as follows. Suppose that we have the inter-procedural control flow graph for a program P, together with a partitioning on the calling contexts of each procedure in P induced by execution profile information for P (for simplicity of presentation we assume here that this partitioning is fixed prior to analysis; in practice, it may change during analysis, as in the example discussed above, but it isn't difficult to extend the discussion here to accommodate this). The only change, compared to a traditional context-sensitive analysis, is that the encoding for a calling context at a call site is now determined by the partition that calling context is in, so that calling contexts in the same partition have the same encoding.

To see that this simple change achieves what we want, consider the analysis of a procedure p for a particular context κ, such that the dataflow information at the call site under consideration is $\bar{u}$. The procedure $\mathsf{analyze\_proc}$ starts by taking the meet of the dataflow information $\bar{u}$ (appropriately mapped to p's name space) together with the information $\mathsf{EntryInfo}_p(\kappa)$ describing the currently known entry information for that calling context. Since contexts in the same partition have the same context encoding, this effectively computes the meet of $\bar{u}$ with the currently known entry information at each context that is in the same partition as κ. After p has been analyzed, the resulting information at the exit from p is used to update $\mathsf{Summary}_p(\kappa)$: the effect of this is to propagate the resulting dataflow information to all calling contexts in the same partition as κ. In effect, this realizes an analysis that is context-insensitive for calling contexts that are in the same partition, but is context-sensitive across partitions.

3 An Example Analysis: Alias Analysis of Executable Code

The previous section discussed profile-guided analyses in general terms. We next describe our experiences with an implementation of a particular profile-guided analysis. The primary issue that has to be addressed is the mechanism for partitioning the set of call sites in a program given an execution profile. An evaluation of specific criteria we tested is given after a quick overview of our analysis problem.

The ideas described in this paper were motivated by an alias analysis algorithm we have developed for executable code [9]. The basic idea in this algorithm is to reason about arithmetic computations modulo some pre-selected value k: a set of addresses is then represented by an *address descriptor*, which is a pair $\langle I, S \rangle$, where I is an instruction, called the *defining instruction* for (a register described by) that address descriptor, and S a set of residues, modulo k, with respect to the value computed by instruction I. Since k is fixed, so is the set of possible residues modulo k, which means that S can be represented as a bit vector. Additionally, as a matter of practicality, at vertices in the (inter-procedural) flow graph that have multiple predecessors, we use a widening operation to "merge" the information coming in along the incoming control flow edges. The essential idea behind this operation is that if we have two incoming edges at a vertex in the flow graph such that the values for a register r being propagated along these edges is described by address descriptors $\langle I_1, S_1\rangle$ and $\langle I_2, S_2\rangle$ respectively, and $I_1 \neq I_2$, then the information about r is widened to $\perp$, denoting a lack of information. This allows us to associate a single address descriptor with a register at each program point of interest, rather than a set of descriptors, and keeps the memory requirements of the analysis reasonable. Overall, this leads to a reasonably time and space efficient analysis algorithm.

The original formulation of our analysis was context-insensitive [9]. This turns out to lead to an undesirable loss in precision in a number of situations. An example of this can be seen in the function `alignd()` in the SPEC-95 benchmark *m88ksim*. The callers of this function pass it, as arguments in registers `r17` and `r18`, pointers into distinct structures in their stack frames; the function contains a very heavily executed loop containing a basic block that has a series of indirect loads and stores through these pointers, with the following structure:

```
...
store r7, 0(r18)
load r6, 0(r17)
...  use r6, define r6
store r6, 0(r18)
load r23, 0(r17)
...  use r23, define r23
store r23, 0(r17)
load r8, 0(r18)
...
```

We would like to schedule the load instructions in this block to better hide their latency. In the absence of accurate aliasing information about registers `r17` and `r18`, however, we are unable to move the load instructions past the immediately preceding store instructions. This significantly constrains the quality of instruction scheduling possible in this block.

It turns out, unfortunately, that the context-insensitive version of our alias analysis [9] doesn't help us in this situation. The problem is that, since registers `r17` and `r18` are loaded with pointers into the caller's stack frame at each call site to `alignd()`, the defining instruction for each of these registers is different at each call site. Because of this, when we compute the meet of the incoming information at the entry node of the function, the information associated with these registers is widened to $\perp$, as described above: that is, all of the information is lost. On the other hand, abandoning the widening would cause a huge increase in the cost of the analysis.

The obvious solution to this problem would be to use a context-sensitive interprocedural analysis. However, the defining instructions for a register are generally different at different call sites to a function, which means that the aliasing information associated with different call sites will also be different. This implies that very little sharing of information across call sites is possible, and also that the callee will have to be analyzed separately for each such call site. Given that executable programs—especially statically linked programs, where all of the code for the various library routines is available for analysis—tend to be significantly larger than the corresponding source level entities, this indicates that the cost of a traditional context-sensitive analysis is likely to be quite high.

We chose, instead, to use a profile-guided analysis. Before going into details, we specify what it means for a basic block to be considered *hot*. Given a value ϕ in the interval (0,1], we determine the largest execution frequency threshold N such that, by considering only those basic blocks that have execution frequency at least N, we are able to account for at least the fraction ϕ of the total number

Program	Standard	Callsite Freq. Based			Relevance	Conservative
		$\theta = 1.0$	$\theta = 0.8$	$\theta = 0.67$	$\theta = 1.0$	
compress	861 **(100.0%)**	365 **(42.4%)**	312 **(36.2%)**	306 **(35.5%)**	240 **(27.9%)**	340 **(39.5%)**
gcc	25109 **(100.0%)**	9701 **(38.6%)**	4643 **(18.5%)**	4003 **(15.9%)**	9382 **(37.4%)**	3671 **(14.6%)**
go	3062 **(100.0%)**	2023 **(66.1%)**	1010 **(33.0%)**	903 **(29.5%)**	1813 **(59.2%)**	925 **(30.2%)**
ijpeg	2019 **(100.0%)**	1027 **(50.9%)**	862 **(42.7%)**	838 **(41.5%)**	669 **(33.1%)**	858 **(42.5%)**
li	2651 **(100.0%)**	1101 **(41.5%)**	840 **(31.7%)**	808 **(30.5%)**	967 **(36.5%)**	822 **(31.0%)**
m88ksim	2766 **(100.0%)**	884 **(32.0%)**	717 **(25.9%)**	678 **(24.5%)**	728 **(26.3%)**	705 **(25.5%)**
perl	5693 **(100.0%)**	1428 **(25.1%)**	970 **(17.0%)**	897 **(15.8%)**	1262 **(22.2%)**	889 **(15.6%)**
vortex	10107 **(100.0%)**	4505 **(44.6%)**	2344 **(23.2%)**	2125 **(21.0%)**	4328 **(42.8%)**	2046 **(20.2%)**
Geo. Mean	**100.0%**	**41.1%**	**27.3%**	**25.4%**	**34.1%**	**25.6%**

Table 1: Analysis Costs: Total call sites analyzed

of instructions executed by the program (as indicated by its basic block execution profile). Any basic block whose execution count is at least N is then said to be *hot* with respect to the threshold ϕ. For example, given $\phi = 0.95$ (the value we use for our experiments), the hot basic blocks of a program consist of those that allow us to account for at least 95% of the instructions executed at runtime. We use profile information to partition the call sites for each procedure as follows: if a call site is considered to be "important" according to some criterion, it is given its own partition where it is the only member; otherwise, it is put into a partition that contains all "unimportant" call sites. We experimented with three classes of criteria for determining the importance of call sites:

1. *Callsite Frequency-based Cutoffs* : Here the decision as to whether a call site is important or not is made based on its execution frequency relative to the execution frequency of the other call sites to the function. Specifically, we use a cutoff threshold θ, and mark the call sites of each function in descending order of execution frequency until the fraction of the dynamic calls to the function accounted for by the marked call sites is at least θ. In the case where $\theta = 1.0$, all call sites to a function with non-zero execution frequency are considered to be important.

2. *Relevance-based Cutoffs* : Here the frequency-based criteria are augmented by the notion of "relevance." We define a function f as being *relevant* if either (*i*) f contains a hot basic block; or (*ii*) there is a relevant function g that is reachable from f in the call graph of the program. The intuition is that a function is relevant if it is possible for the (forward) dataflow information computed for it to influence the dataflow information at a frequently executed basic block. Thus, in this case a call site is considered important if its execution frequency is high enough relative to the other call sites to the function, and the called function is relevant.

3. *Conservative* : Here a call site is considered important if its contribution to the runtime execution profile of the called function is high enough that, even if all other call sites for the callee are ignored, some basic block in the callee would be hot.

Our experiments were carried out in the context of *alto*, a link-time optimizer we have constructed for the DEC Alpha architecture [14]. Programs were compiled with the DEC C compiler V5.2-036 invoked as `cc -O4`, with linker options to retain relocation information and to produce statically linked executables. Low-level profile information was generated with *pixie*, using the SPEC training inputs. Timings were obtained on a lightly loaded DEC Alpha workstation with a 300 MHz Alpha 21164 processor with 512 Mbytes of main memory, running Digital Unix 4.0.

Table 1 shows the number of call sites considered by a number of different criteria we experimented with.[4] Here, "Standard" refers to a traditional context-sensitive inter-procedural analysis. For each program, the top row gives the actual number of call sites considered, while the numbers below these show the ratio compared to the "Standard" column. It does not come as a great surprise that profile-guided techniques lead to a decrease in the number of call sites considered: what is interesting, perhaps, is the magnitude of the reduction. Simply disregarding only those call sites that are never executed (Callsite frequency based cutoffs, $\theta = 1.0$) leads to close to a 60% reduction, on the average, in the number of call sites considered (and amount of memory used). When relevance information is also taken into account, the average number of of callsites considered drops to just over a third of that for the standard analysis. More stringent criteria further reduce the number of call sites considered, to around a quarter of the number considered by the standard analysis.

Table 2 shows the time taken by the analysis under different selection criteria for call sites. Disregarding call sites that are not executed leads to a reduction in analysis time of over 60% on the average. When relevance is also taken into account, the analysis time drops to just over 33% of the time for the standard analysis, on the average. Callsite frequency based criteria with lower cutoff thresholds give analysis times that are, on the average, about 18%–20% of that of the standard analysis. Even though the number of call sites processed using the conservative criterion is comparable to that for the call site frequency based criterion with $\theta = 0.67$, the additional work involved in identifying important call sites for the former leads to analysis times that are, on the average, about 40% of the standard analysis, i.e., about double that for the call site frequency based scheme with $\theta = 0.67$. These averages are somewhat distorted by the *gcc* benchmark, however: the problem with this program is that it contains many functions with a large number of call sites whose frequency distributions are not as sharply skewed as those of the other programs. Because of this, the relatively simple-minded algorithms we use to preprocess the program and identify important call sites end up being quite expensive overall. As a result, some of the profile-based analyses for this program ended up being more expensive than the standard analysis. If we don't consider *gcc*, for the remaining seven benchmarks we get an average of about 31% for the callsite frequency based scheme with $\theta = 1.0$ and 26% for the relevance-based scheme with $\theta = 1.0$.

Together, these tables indicate that by using profile information to avoid analyzing unimportant call sites, we can achieve significant reductions in the space and time requirements for an analysis. By itself, this result doesn't come as a huge surprise: indeed, by sufficiently reducing the number of distinct call site partitions for the functions in a program, we can approach the efficiency of a context-insensitive analysis. The key question, then, is whether these performance improvements are accompanied by a significant degradation in the quality of dataflow information for the frequently executed portions of the program. This issue is addressed in the next section.

4 Using the Analysis Information: Cloning for Instruction Scheduling

Our goal is to use interprocedural alias analysis to determine whether the information available for a particular call site for a function makes possible code improvements—in this case, better instruction schedules—within the function that are not possible in general. If this turns out to be the case, we clone the function as necessary (provided that the resulting code improvements are judged to be significant enough to warrant the code growth) to allow this to be done. Our approach to cloning is conceptually similar to that of Cooper *et al.* [7, 8], except that we use profile-guided considerations for low level code improvements to guide our cloning decisions, rather than focusing on inter-procedural constant propagation. To limit the amount of code growth, cloning is considered only for important call sites. Further, we require that the alias information available at such a call site C should allow us to identify a pair of memory references I and J (at least one of which must be a store instruction), in some basic block B within the function, as accessing non-overlapping memory locations, such that (i) I and J cannot be identified as being non-overlapping without the

[4] Our analysis maintains information about a fixed number of registers for each call site, so its memory usage is directly proportional to the number of call sites considered. In particular, this means that the ratios of memory usage for these criteria are identical to the ratios shown in Table 1. For this reason they are not shown separately.

Program	Standard	Callsite Freq. Based			Relevance	Conservative
		$\theta = 1.0$	$\theta = 0.8$	$\theta = 0.67$	$\theta = 1.0$	
compress	6.72 **(100.0%)**	3.47 **(51.6%)**	3.03 **(45.2%)**	3.02 **(44.9%)**	2.05 **(30.5%)**	3.70 **(55.1%)**
gcc	633.24 **(100.0%)**	1176.37 **(1.858)**	235.41 **(37.2%)**	176.49 **(27.9%)**	1177.49 **(1.859)**	480.38 **(75.9%)**
go	82.40 **(100.0%)**	57.51 **(69.8%)**	34.03 **(41.3%)**	25.33 **(30.7%)**	54.55 **(66.2%)**	88.00 **(1.068)**
ijpeg	26.85 **(100.0%)**	7.85 **(29.2%)**	7.43 **(27.7%)**	7.37 **(27.4%)**	5.63 **(21.0%)**	8.33 **(31.0%)**
li	18.70 **(100.0%)**	8.47 **(45.3%)**	6.48 **(34.7%)**	5.73 **(30.7%)**	6.52 **(34.8%)**	6.35 **(34.0%)**
m88ksim	45.98 **(100.0%)**	8.87 **(19.3%)**	6.57 **(14.3%)**	6.30 **(13.7%)**	8.18 **(17.8%)**	12.53 **(27.3%)**
perl	661.27 **(100.0%)**	89.40 **(13.5%)**	45.08 **(6.8%)**	41.47 **(6.3%)**	87.73 **(13.3%)**	86.00 **(13.0%)**
vortex	407.42 **(100.0%)**	92.91 **(22.8%)**	22.28 **(5.5%)**	19.35 **(4.7%)**	91.93 **(22.6%)**	177.51 **(43.6%)**
Geo. Mean	**100.0%**	**38.9%**	**20.8%**	**18.4%**	**33.2%**	**40.5%**

Table 2: Analysis Time (secs)

alias information from C, i.e., based on purely local reasoning; (*ii*) it is possible to reorder I and J, i.e., there is no chain of def-use dependences between them that forces one to be executed before the other; and (*iii*) B is a hot basic block. After the interprocedural alias analysis, we examine, for each function, each of its important call sites, and determine the set of memory reference instruction pairs in that function that can be identified as being non-overlapping based on the alias information at that call site, and that satisfy the additional criteria mentioned above. The call sites for each function are then grouped into buckets such that two call sites are in the same bucket if and only if they have the same set of independent pairs of memory reference instructions. A clone is created for a bucket if the combined execution freqency of the relevant call sites is high enough that at least one independent memory reference pair in the resulting clone would fall in a hot basic block.

The extent of cloning is shown in Table 5: each entry in this table is of the form m/n, where m indicates the number of clones created, and n the total number of instructions added by the cloning process. It can be seen that the total amount of cloning is not very large, with only a relatively small number of clones being created. The largest amount of code growth occurs for *go* (17%), *vortex* (12%), and *li* (9.4%); the remaining programs experience code growths of less than 5% each. This indicates that the cloning criteria discussed earlier in the section are not unduly liberal.

Turning to the question of quality of dataflow information obtained using profile-guided analysis, Table 3 shows, for each of our benchmarks, the number of independent instruction pairs identified in functions that were cloned. Since a function is cloned only if we determine that the dataflow information available at a call site enables an optimization that would not otherwise be enabled, this is an indication of the extent to which the context-sensitive analysis is useful. Notice that the analyses that disregard call sites that are not executed and/or not relevant find exactly the same number of independent instruction pairs in hot basic blocks as the standard analysis. Not surprisingly, the more restrictive criteria lead to a drop in the number of independent instruction pairs identified; the drop is most pronounced for the programs *gcc* and *li*.

However, a reduction in the static count of the number of independent instructions found does not, by itself, indicate a problematic lack of precision, since it may happen that the differences are due to instructions in infrequently executed portions of the program. The runtime importance of the pairs of independent memory references found is shown in Table 4, where each pair of independent memory references in a cloned function is weighted by the execution frequency of the basic block in which those references occur. There are two points to be noted here. The first is that this is a measure of opportunities for optimization in the important basic blocks in the program, and thus a

Program	Standard	Callsite Freq. Based			Relevance	Conservative
		$\theta = 1.0$	$\theta = 0.8$	$\theta = 0.67$	$\theta = 1.0$	
compress	0 (–)	0 (–)	0 (–)	0 (–)	0 (–)	0 (–)
gcc	192 **(100.0%)**	192 **(100.0%)**	146 **(76.0%)**	142 **(74.0%)**	192 **(100.0%)**	107 **(55.7%)**
go	197 **(100.0%)**	197 **(100.0%)**	189 **(95.9%)**	161 **(81.7%)**	197 **(100.0%)**	189 **(95.9%)**
ijpeg	0 (–)	0 (–)	0 (–)	0 (–)	0 (–)	0 (–)
li	37 **(100.0%)**	37 **(100.0%)**	33 **(89.2%)**	33 **(89.2%)**	37 **(100.0%)**	3 **(08.1%)**
m88ksim	118 **(100.0%)**	118 **(100.0%)**	118 **(100.0%)**	118 **(100.0%)**	118 **(100.0%)**	111 **(94.1%)**
perl	41 **(100.0%)**	41 **(100.0%)**	41 **(100.0%)**	41 **(100.0%)**	41 **(100.0%)**	39 **(95.1%)**
vortex	216 **(100.0%)**	216 **(100.0%)**	201 **(93.1%)**	201 **(93.1%)**	216 **(100.0%)**	35 **(16.2%)**

Table 3: Efficacy of Analysis: independent instruction pairs identified (static counts)

reasonable measure of the pragmatic quality of the dataflow information obtained, even though the scheduler may decide to not reorder a pair of memory references that are deemed to be independent from the alias information. The second is that while this doesn't account for possible independent references in blocks that are not hot, note that with $\phi = 0.95$, such blocks account for only 5% of the dynamic instruction count of the program, so their significance for code optimization purposes is small. The conclusions from Table 4 are encouraging: analyses that disregard call sites that are not executed and/or not relevant do as well as the standard context-sensitive analysis, and even the more stringent criteria, which eliminate most call sites as unimportant, do quite well: if we ignore *compress* and *ijpeg*, for which the interprocedural analysis uncovered no important memory reference pairs, with $\theta = 0.8$ the dynamic counts for the remaining programs (Table 4) is about 90% of that obtained using the standard analysis, while with $\theta = 0.67$ we get about 84% of that with the standard analysis. Considering that these analysis take about 20% of the time and 25% of the space, on the average, as the standard analysis, the tradeoff they represent does not seem unreasonable.

Overall, we conclude from these results that a relevance-based approach with $\theta = 1.0$, i.e., where we ignore any call site that is that either not executed, or whose analysis cannot influence the dataflow information at any frequently executed basic block, gives the best results. This approach does as well as the classical context-sensitive analysis in terms of pragmatic utility, but is considerably more efficient in both space and time than a classical context-sensitive analysis, requiring only about a third of the space and time used by the classical analysis. Moreover, lower values of the cutoff threshold θ also yield reasonable tradeoffs between efficiency and precision.

We are currently in the process of integrating this into our scheduler. Preliminary results are encouraging: e.g., for the *m88ksim* benchmark, we see a 3% improvement in running time (which goes from 229.6 secs to 222.5 secs) because of scheduling using the improved alias information; however, we have not had the time to carry out extensive performance evaluations. We expect to have more complete empirical results shortly.

5 Related Work

While there is a considerable body of work on context-sensitive interprocedural analyses (e.g., see [10, 13, 18]) and profile-guided program optimization (e.g., see [3, 5, 6, 11, 16]), there does not seem to be a great deal of work on the use of execution profiles to guide program analyses. The only work that we are aware of on this topic, and the work that is most closely related to ours, is that of Ammons and Larus [2], who consider how dataflow analyses may be modified to invest extra effort in analyzing frequently executed paths within a procedure in order to obtain better information along

Program	Standard	Callsite Freq. Based			Relevance	Conservative
		$\theta = 1.0$	$\theta = 0.8$	$\theta = 0.67$	$\theta = 1.0$	
compress	0.00 (–)	0.00 (–)	0.00 (–)	0.00 (–)	0.00 (–)	0.00 (–)
gcc	0.39 **(100.0%)**	0.39 **(100.0%)**	0.32 **(82.2%)**	0.30 **(76.6%)**	0.39 **(100.0%)**	0.25 **(64.9%)**
go	7.48 **(100.0%)**	7.48 **(100.0%)**	7.02 **(93.8%)**	6.10 **(81.7%)**	7.48 **(100.0%)**	7.24 **(96.8%)**
ijpeg	0.00 (–)	0.00 (–)	0.00 (–)	0.00 (–)	0.00 (–)	0.00 (–)
li	1.68 **(100.0%)**	1.61 **(95.5%)**	1.39 **(82.7%)**	1.30 **(77.5%)**	1.61 **(95.5%)**	0.70 **(41.6%)**
m88ksim	32.68 **(100.0%)**	32.68 **(100.0%)**	32.67 **(100.0%)**	32.65 **(99.9%)**	32.68 **(100.0%)**	32.59 **(99.7%)**
perl	0.20 **(100.0%)**	0.20 **(100.0%)**	0.19 **(96.2%)**	0.19 **(95.5%)**	0.20 **(100.0%)**	0.18 **(90.5%)**
vortex	767.50 **(100.0%)**	767.50 **(100.0%)**	672.43 **(87.6%)**	586.89 **(76.5%)**	767.50 **(100.0%)**	473.14 **(61.6%)**

Table 4: Efficacy of Analysis: independent instruction pairs identified (dynamic counts, $\times 10^6$)

those paths, possibly incurring an additional time and space cost. Conceptually, our approach is a dual of theirs in the sense that the key idea in our approach is to invest less effort on unimportant program fragments in order to reduce the time and space costs of the analysis. Also, Ammons and Larus consider intra-procedural analyses, while we focus on inter-procedural analyses. Notice that our approach could nevertheless be used to pursue goals similar to those of Ammons and Larus, in the following way: given a particular space and/or time "budget" for an analysis, one could use either an ordinary analysis that examines the entire program, or a more sophisticated analysis that is more expensive, but also more precise, using our ideas to ignore the unimportant portions of the program. Using the more sophisticated analysis would provide more precise information for the important areas of the program, but the added expense of such an analysis would be mitigated by not applying it to the unimportant portions of the program. Overall, such an approach could allow a compiler to obtain better dataflow information for the important portions of a program while expending a comparable analysis effort and remaining within a particular time or space "budget."

Zhang *et al.* describe how, for pointer alias analysis of C programs, a program can be partitioned such that objects in different partitions can be analyzed independently of each other, possibly using different algorithms [19]. They show that a judicious mix of flow-sensitive and flow-insensitive pointer alias analyses can reduce analysis costs without significantly sacrificing the quality of the information obtained. While their goals are conceptually very similar to ours, the technical details are very different. They also don't use profile information to guide their decisions about whether to use a flow-sensitive or flow-insensitive analysis for a particular partition, but this isn't inherent in their model, which could be modified easily enough to take profile information into account.

Chambers *et al.* have observed that the degree of context-sensitivity of an inter-procedural analysis can be controlled by selective merging of calling contexts [4, 12]. However, they don't consider using profile information to control the processing of different call sites to a function. Moreover, while this doesn't appear to be inherent in their framework, their examples seem to suggest that, once certain parameters controlling the degree of context-sensitivity have been selected, all the functions in the program are analyzed in the same way, regardless of their pragmatic importance.

6 Conclusions

Traditional context-insensitive interprocedural analyses are simple and efficient, but can be imprecise; context-sensitive analyses, which are typically more precise, have the drawback of being more complex and expensive. Pragmatically, it turns out that most functions typically have a skewed execution frequency distribution—that is, most runtime calls are accounted for by a small number

Program	*Original*	*Code Growth due to cloning*					
		Standard	Callsite Freq. Based			Relevance	Conservative
			$\theta = 1.0$	$\theta = 0.8$	$\theta = 0.67$	$\theta = 1.0$	
compress	316/20707	0/0	0/0	0/0	0/0	0/0	0/0
gcc	2465/353002	34/9228	34/9228	30/7992	29/7806	34/9228	22/7139
go	945/83929	16/14235	16/14235	15/14197	18/15788	16/14235	15/14197
ijpeg	788/62639	0/0	0/0	0/0	0/0	0/0	0/0
li	722/40832	6/3836	6/3836	5/3801	5/3801	6/3836	1/83
m88ksim	638/53498	2/1050	2/1050	2/1050	2/1050	2/1050	1/222
perl	722/97079	6/4469	6/4469	6/4469	6/4469	6/4469	5/4406
vortex	1446/155030	17/18802	17/18802	14/15392	14/15392	17/18802	8/7675

Table 5: Code growth due to cloning (Functions/Instructions)

of call sites—which means that neither approach is entirely satisfactory. This paper describes a simple approach whereby profile information can be used to partition the call sites of each function such that the interprocedural analysis is context-sensitive across partitions but context-insensitive within each partition. This allows an analysis to be context-sensitive, and hence precise, for the important call sites of a function, and context-insensitive, and hence efficient, for the unimportant ones. Experiments indicate that the resulting analyses can be significantly cheaper than a traditional context-sensitive analysis without significantly compromising the pragmatic quality of the information obtained.

Acknowledgements

Thanks are due to Scott Nettles for helpful comments on an earlier draft of this paper.

References

[1] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers – Principles, Techniques and Tools*, Addison-Wesley, 1986.

[2] G. Ammons and J. R. Larus, "Improving Data-flow Analysis with Path Profiles", *Proc. SIGPLAN '98 Conference on Programming Language Design and Implementation*, June 1998, pp. 72–84.

[3] R. Bodík, R. Gupta, and M. L. Soffa, "Complete Removal of Redundant Expressions", *Proc. SIGPLAN '98 Conference on Programming Language Design and Implementation*, June 1998, pp. 1–14.

[4] C. Chambers, J. Dean, and D. Grove, "Frameworks for Intra- and Interprocedural Dataflow Analysis", Technical Report 96-11-02, Dept. of Computer Science and Engineering, University of Washington, Seattle, 1996.

[5] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "Profile-guided automatic inline expansion for C programs", *Software Practice and Experience* vol. 22 no. 5, May 1992, pp. 349–369.

[6] R. Cohn and P. G. Lowney, "Hot Cold Optimization of Large Windows/NT Applications", *Proc. MICRO 29*, Dec. 1996.

[7] K. D. Cooper, M. W. Hall, and K. Kennedy, "Procedure Cloning", *Proc. 1992 International Conference on Computer Languages*, pp. 96–105.

[8] K. D. Cooper, M. W. Hall, and K. Kennedy, "A Methodology for Procedure Cloning", *Computer Languages* 19(2), April 1993, pp. 105–118.

[9] S. K. Debray, R. Muth, and M. Weippert, "Alias Analysis of Executable Code", *Proc. 1998 ACM Symposium on Principles of Programming Languages*, pp. 12–24.

[10] M. Emami, R. Ghiya and L. J. Hendren, "Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers", *Proc. SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 1994, pp. 242–256.

[11] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction", *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[12] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call Graph Construction in Object-Oriented Languages", *Proc. 12th. Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA), Oct. 1997, pp. 108–124.

[13] W. Landi and B. G. Ryder, "A Safe Approximate Algorithm for Interprocedural Pointer Aliasing", *Proc. SIGPLAN '92 Conference on Programming Language Design and Implementation*, June 1992, pp. 235–248.

[14] R. Muth, S. K. Debray, S. Watterson, and K. De Bosschere, "`alto` : A Link-Time Optimizer for the DEC Alpha", Draft, Sept. 1998. Available at `www.cs.arizona.edu/people/debray/papers/alto.ps`.

[15] E. W. Myers, "A Precise Inter-Procedural Data Flow Algorithm", *Proc. 8th ACM Symposium on Principles of Programming Languages*, Jan. 1981, pp. 219–230.

[16] K. Pettis and R. C. Hansen, "Profile-Guided Code Positioning", *Proc. SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990, pp. 16–27.

[17] E. Ruf, "Context-Insensitive Alias Analysis Reconsidered", *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995, pp. 13–22.

[18] R. P. Wilson and M. S. Lam, "Efficient Context-Sensitive Pointer Analysis for C Programs", *Proc. SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995, pp. 1–12.

[19] S. Zhang, B. G. Ryder, and W. Landi, "Program Decomposition for Pointer Aliasing: A Step toward Practical Analyses", *Proc. Fourth Symposium on the Foundations of Software Engineering*, Oct. 1996.