A Study and Analysis of Function Inlining

Prasad Deshpande
pmd@cs.wisc.edu

Amit Somani
somani@cs.wisc.edu

Abstract

Function inlining is a widely known technique which has been used to improve program performance. Inlining replaces a function call by the body of the function. In this report, we study the effects of inline function expansion by observing the empirical behavior of a set of C and C++ programs. Our results indicate that for the average C program improvements in timings due to inlining are not much (2-3%). However, C++ programs benefit much more (up to 46%) from inlining due to smaller average function size and a larger call stack depth.

1 Introduction

Almost all programs are typically written as a collection of smaller computing units (procedures or functions) to aid program development. However, function invocation in a program has an associated overhead which might affect the execution time of the program. It also affects compile time optimizations like register allocation, dead code elimination, common subexpression elimination, constant propagation etc. Besides, conservative assumptions need to be made when analyzing a call. Hence, we need some method to alleviate the performance problems due to function calls. Inter-procedural analysis can be used to track down dataflow information across procedure calls to eliminate the need for conservative assumptions thus allowing program optimization to take place. We study the other option, i.e. inlining the subroutines to reduce the call overhead and enable optimizations. The rest of report is organized as follows. Section 1.1 discusses some existing schemes to reduce function call overhead. In Section 1.2, we discuss inline function expansion as another method to make function calls inexpensive. Section 2 describes the benchmark programs used for our study. In Section 3 we describe the tools and the methodology that we use for our measurements. The results are presented in Section 4. In that section we also discuss the factors affecting the execution time behavior of inlined versus non-inlined programs. We summarize and conclude in Section 5.

1.1 Current Solutions

Some processors provide hardware support for minimizing the overhead due to function calls. For example, the SUN SPARC RISC processors provide overlapping register windows to reduce the overhead of passing parameters on a stack. In fact, if the number of parameters is less than the size of “available” output window, one incurs almost no overhead in the function call. (Another notable example is the CRISP processor which attempts to simulate register allocation in hardware). Later in the report, we attempt to evaluate how well the register window scheme performs and whether we can improve over and above “hardware” register window schemes. It is obvious however that one big disadvantage of such schemes is inflexibility (for e.g. the number of overlapping register windows one can fit on a chip is fixed).

There are software schemes too which attempt to address this problem. Most prominently, inter-procedural analysis has been used to reduce the overhead involved in function calls. In particular, [1] shows that inter-procedural register allocation can be used to reduce the register save/restore calls across function call boundaries. By combining profiling and inline expansion, link-time register allocation can lead to a similar performance as hardware register window schemes [9]. However, most of the schemes used in software are complex in nature. Subroutine inlining (also referred to as inline function expansion or procedure integration) provides a simple method to attack this problem. It may also be less costly than inter-procedural analysis in terms of time.

1.2 Function Inlining

Inline function expansion replaces a function call with the body of the function. This has the effect of removing the actual function call/return overhead associated with the function. Also, no conservative assumptions need to be made about the callee as the function body for the callee has been inlined. This should lead to increased global optimization opportunities like common subexpression elimination, constant propagation etc. It can also provide better opportunity for loop optimizations like loop unrolling, loop interchange etc.

However, inlining subprograms does not come for free. It will cause the code size to increase. It will also cause the number of local variables at the call site (at which the callee was inlined) to increase. An increase in the number of local variables may actually cause more register spills (especially for the register window architecture). If the register allocator decides to spill a heavily used variable which was previously in a register, the number of load/stores would increase significantly. Depending on the nuances of a particular compiler, there might be other side-effects which affect the performance of the program [2].

In the rest of the report, we study the effects of programs inlining on program execution behavior and optimizer effectiveness.

2 Benchmark Programs

We have measured 10 programs written in C and C++. The C programs include five out of the six[1] programs from the SPECint92 benchmark suite, and one C program from the SPECfp92. The C++ programs are not from any particular benchmark suite but are in general large programs.

[1]GCC was the only program which was left out.

026.COMPRESS	A file compression program that uses Lempel-Ziv coding.
023.EQNTOTT	A translator from a logic formula to a truth table.
008.ESPRESSO	A logic optimization program that minimizes Boolean functions.
022.LI	A Lisp interpreter that is an adaptation of XLISP.
072.SC	A spreadsheet program.
052.ALVINN	A single precision FP benchmark which trains a neural network called ALVINN using back propagation.

Table 1: Summary of C Benchmark Programs: The first five programs in this list are from the SPECint92 benchmark suite. 052.alvinn is from the SPECfp92 suite.

MERGER	A program to join (merge) two files based on equality of two columns.
SHORE	A heterogeneous persistent object repository.
STRING	A program that computes the length of the longest string in the input
QTREE	An in-memory implementation of Quad-trees.

Table 2: Summary of C++ Benchmark Programs

We sought programs which were both reasonably large in terms of code size, had a good number of function calls and those that represented some typical applications. In this respect, the C programs from SPECint92 easily fitted our requirements. It was significantly more difficult to find a "C++ benchmark" suite which was publicly available. Hence, almost all the applications tested were those which were locally accessible. For all these programs, unless noted otherwise the environment shown in Table 3 was used.

3 Tools and Metrics Used

Our method of analysis had two steps. For most of the programs except the large ones like SHORE we first profiled the non-inlined programs using QPT. QPT gives statistics on different functions including the number of calls to each function and the total time spent in that function. We then selected some functions which are called most often for inlining. These functions were inlined by writing the keyword *inline* in their definitions and compiling with "-O" option of GCC. GCC does inlining in the initial stages (parse stage) of compilation. So, the function definition has to be present in the same file and before the call site for inlining to take place. We now profiled the inlined

Machine Architecture	SPARC4M
Compiler	GCC 2.6.2
Oper. System	SunOS 4.1.3

Table 3: Execution Environment for programs

version of the program. The data presented in the tables has been obtained by profiling the inlined and the non-inlined versions using QPT. Since inlining in this way was not possible for SHORE, we compared two versions of the program, one compiled with the option "-finline-functions" and other with "-fno-inline".

We have used a set of metrics for comparing the programs. The most important among these are the Code size, Compilation time and Execution time. These measure the tradeoffs between non-inlined and inlined programs.

4 Results

In this section we compare the programs using the different metrics. Firstly, we present static information about the programs to be able to identify overall program characteristics. Then, we go on to report and analyze how dynamic program characteristics are affected by subprogram inlining.

For each of the metrics, we report the actual numbers and the ratio of its value for the inlined program to the non-inlined program. For some of the metrics, bar graphs are presented in the Appendix.

4.1 Static Program Characteristics

4.1.1 Instruction Count and Code Size

After inlining there should be an increase in code size, since the body of the function gets inlined at each of the call sites. Increase in code size could lead to more page faults and cache misses. But Davidson et al. argue that these do not have a significant effect on the execution times of the programs ([6], [3]).

Static information about the programs studied is provided in Table 4. From the table we see that there is a moderate increase in the code size for inlined code. This is due to selective inlining of only the most frequently called functions. However, except for SHORE[2], the average increase in code size is small (refer Table 5 for the ratios on the increase in code size).

The static instruction counts in the programs are in proportion to the code size too. There is a small increase in the number of static instructions in the program. This is due to the fact that the function calls get replaced by the body of the function at several call sites.

4.1.2 Basic Blocks, Average Block Size and Compilation Time

The numbers for the basic blocks are quite interesting. Larger basic blocks would offer more opportunity for architecture-specific optimizations, such as instruction scheduling. After inlining, the increase in the size of the basic blocks is at best minimal. In fact, in some cases the size (e.g. 022.LI, 072.SC, SHORE) actually decreases. This can be best explained with the help of an

[2] Code size for shore increased to 4 times the original size. This is because there are many functions (class member functions and others) which get inlined.

Program	Code Size(bytes)		Instn. Count		Total Basic Blocks		BBlock Size	
	NonInlined	Inlined	NonInlined	Inlined	NonInlined	Inlined	NonInlined	Inlined
026.COMPRESS	122880	131072	13284	13891	2960	3068	4.488	4.528
023.EQNTOTT	147456	147456	16150	16347	3606	3660	4.479	4.466
008.ESPRESSO	294912	311296	49283	53286	10000	10987	4.928	4.850
022.LI	221184	229376	30814	31425	6076	6327	5.071	4.967
072.SC	335872	335872	45410	45518	8977	9004	5.058	5.055
052.ALVINN	147456	147456	1891	19110	4090	4116	4.624	4.643
MERGER	450560	458752	18374	18367	4284	4293	4.289	4.278
SHORE	3612672	14090240	256959	381953	39909	71605	6.439	5.334
STRING	466944	466944	24488	24521	5659	5665	4.327	4.329
QTREE	1531904	1761280	32091	46740	6814	9326	4.710	5.012

Table 4: Static Information for Program Execution: The programs were compiled using the version 2.6.2 of the GNU CC compiler. In all cases, optimization was enabled using the "-O" flags. Inlining was effected by profiling the program using QPT and then selectively inlining the functions. To get the non-inlined versions, the programs were compiled with the "-fno-inline" flag.

Program	Code Size	Instn. Count	Total Basic Blocks	Basic Block Size
026.COMPRESS	1.066667	1.045694	1.036486	1.008913
023.EQNTOTT	1.000000	1.012198	1.014975	0.997098
008.ESPRESSO	1.055556	1.081225	1.098700	0.984172
022.LI	1.037037	1.019829	1.041310	0.979491
072.SC	1.000000	1.002378	1.003008	0.999407
052.ALVINN	1.000000	1.010470	1.006357	1.004109
MERGER	1.018182	0.999619	1.002101	0.997435
SHORE	3.900227	1.486436	1.794207	0.828390
STRING	1.000000	1.001348	1.001060	1.000462
QTREE	1.149733	1.456483	1.368653	1.064119

Table 5: Inlined to Non-inlined Ratios for static program characteristics

example. Consider a function $f()$. Let us assume that this function has a conditional statement and a few compute statements. Further assume, that the function is called from m sites. In the non-inlined version of the program, $f()$ will account for 2 basic blocks. However, once it gets inlined it will cause the number of basic blocks to increase by $2 * m$. Thus if a function has very small basic blocks, inlining the function at each call site will cause the average basic block size to reduce.

It may be noted that as the number of basic blocks increases, slower will be the data flow analysis. This is simply due to the fact the number of nodes in the Control Flow Graph increases. We observe that the number of basic block does increase by a reasonable amount. As we see later in Section 4.3, the compilation time for inlined programs does increase significantly. One of the major factors that causes slower compilation is increase in the number of basic blocks. The inlining algorithm will also add to the compilation time. This is especially true in the case when the compilation takes place with a "-finline-functions" flag. In this case, gcc evaluates whether to inline a function or not by running it through a set of heuristics[3].

4.2 Dynamic Program Characteristics

4.2.1 Total Calls and Returns

The Call/Return overhead is one of the important reasons because of which function inlining is a good idea. On a conventional machine, every call will translate into preparing the activation record (AR) of the callee, putting the AR on stack and restoring the state on the return back to the caller. Inlining reduces this overhead at the expense of increase in code size. As discussed previously (in Section 1.1), the register window scheme is used to reduce function call overhead. Note that even in the case of a register window scheme, one would gain as the reduction of the number of calls would indirectly reduce the call stack depth for the program.

From Table 6, we observe that the number of calls and returns reduces after inlining since the call to the function is replaced by the function body. This number measures the decrease in call overhead due to inlining. The actual decrease depends on how many functions are inlined. In most cases the number of calls reduces significantly after inlining major functions. But in some cases it is not possible to inline the functions which are called most frequently. This may in happen if the function is a library function (like printf), or if the function is called via a function pointer (this happened in the benchmark 023.eqntott where most of the calls are through the function pointer cmppt) and also if the function is recursive (as was the case with 072.sc).

4.2.2 Total number of loads and stores

The dynamic count of the load and store instructions are listed in Table 8. We discuss the number of loads and store instructions with respect to the register allocation in both the inlined and the non-inlined versions of the code. From previously published work [6], it is currently believed that, except for some cases, the number of loads/stores in a program should actually decrease

[3]The exact heuristics are listed in the file `integrate.c` of the gcc source.

Program	Total Dynamic Instructions		Total Calls		Total Returns	
	NonInlined	Inlined	NonInlined	Inlined	NonInlined	Inlined
026.COMPRESS	77501763	82137716	251688	796	363	363
023.EQNTOTT	1250136378	1249558827	4363884	4253396	3677361	3583983
008.ESPRESSO	1083961026	1060954142	8822362	5563375	4522628	1904067
022.LI	4955641467	4519411551	154643762	81345044	20771990	2707161
072.SC	679869515	679449401	7864432	7861829	2210161	2210146
052.ALVINN	4968121221	4970065222	4376881	4346881	1882166	1870166
MERGER	647780300	631483941	14321266	11352192	7606789	5442241
SHORE	356306952	345507123	3057027	873273	1210171	250483
STRING	371934660	368801414	14663450	14190081	676625	225806
QTREE	578885393	357768172	36733486	21972714	900421	376032

Table 6: Some dynamic characteristics of program execution

Program	Dynamic Instns	Total Calls	Total Returns
026.COMPRESS	1.059817	0.003163	1.000000
023.EQNTOTT	0.999538	0.974681	0.974607
008.ESPRESSO	0.978775	0.630599	0.421009
022.LI	0.911973	0.526016	0.130327
072.SC	0.999382	0.999669	0.999993
052.ALVINN	1.000391	0.993146	0.993624
MERGER	0.974843	0.792681	0.715445
SHORE	0.969690	0.285661	0.206981
STRING	0.991576	0.967718	0.333724
QTREE	0.618029	0.598166	0.417618

Table 7: Inlined to Non-inlined Ratios for dynamic characteristics

Program	Total Loads		Total Stores		Total NOPs	
	NonInlined	Inlined	NonInlined	Inlined	NonInlined	Inlined
026.COMPRESS	15392602	17585661	6513248	6513249	1000999	998
023.EQNTOTT	205942679	205958883	15135482	15135482	4329834	4205688
008.ESPRESSO	234371721	234362731	51602583	51608116	7394010	4912242
022.LI	1070358792	1059466029	462299098	459870629	75236014	56493079
072.SC	127120525	127120441	30395366	30395368	7913137	7910533
052.ALVINN	1430597537	1430573737	495520308	495520308	1283451	1271251
MERGER	52944840	51584840	16294242	15614242	10289482	7954926
SHORE	53413584	53564745	32396003	32435298	1931674	463132
STRING	89946734	89946739	24001551	24001551	5181412	4505185
QTREE	58973799	58601394	42895122	43196568	21835593	238472

Table 8: Breakdown of instructions

with inlining. This view would hold on a conventional (non register window) architecture because inlining would eliminate the need for setting up the call stack, passing parameters etc. all of which would generate load/store instructions. However, on a register window architecture like the SUN SPARC, since there is lesser overhead in setting up the call stack anyway, this might not be true. And indeed, in our study, the number of load and store instructions does not change[4] much.

COMPRESS is an interesting example that typifies why the register window architecture does so well against inlining. In order to do so, we provide a little background information on COMPRESS. There is one major function which is called quite often: *output*(). It may also be noted that this function has a lot of local variables.

Now, for each level of the call stack (until a register window overflow/underflow), the architecture provides for a set of registers for the local variables. In the case of COMPRESS, since the caller has already used up all the local registers, the newly generated local variables (due to inlining) have to be spilled on to the stack. In fact, the case for the non-inlined version of the program is still better. This is because each call invocation gets a fresh set of registers (again till a register window overflow/underflow occurs). There the overhead incurred in the call is very less because of a minimal amount of set-up that is required.

Hence, on a register window architecture, inlining functions with a large number of local variables is likely to lead to worse performance. This will happen because the number of loads and store instructions will increase due to the register spills.

4.2.3 Number of NOPs

The number of NOPs in a program can be treated as a measure of the instruction scheduling in the programs. Table 10 shows that the number of NOPs has significantly reduced for the inlined

[4]Except for COMPRESS.

Program	Branches		Total Traps	
026.COMPRESS	14440235	14189339	252342	1450
023.EQNTOTT	344533497	344516387	7898117	4205688
008.ESPRESSO	221574632	222759651	8897924	5640101
022.LI	1004828696	949641805	182208867	56493079
072.SC	147068712	147066059	11013493	11010890
052.ALVINN	510335169	510317169	4845576	4815576
MERGER	162534933	161730401	17216205	14247130
SHORE	62234631	61313035	3158782	995641
STRING	81748592	81478084	21269713	20796341
QTREE	61729735	48365304	36834034	1000974

Table 9: Branches and Traps

Program	Total Loads	Total Stores	Total NOPs	Branches	Total Traps
026.COMPRESS	1.142475	1.000000	0.000997	0.982625	0.005746
023.EQNTOTT	1.000079	1.000000	0.971328	0.999950	0.532492
008.ESPRESSO	0.999962	1.000107	0.664354	1.005348	0.633867
022.LI	0.989823	0.994747	0.750878	0.945078	0.310046
072.SC	0.999999	1.000000	0.999671	0.999982	0.999764
052.ALVINN	0.999983	1.000000	0.990494	0.999965	0.993809
MERGER	0.974313	0.958267	0.773112	0.995050	0.827542
SHORE	1.002830	1.001213	0.239757	0.985192	0.315198
STRING	1.000000	1.000000	0.869490	0.996691	0.977744
QTREE	0.993685	1.007028	0.010921	0.783501	0.027175

Table 10: Inlined to Non-inlined ratios of different instructions

version of the code. An interesting observation is that the reduction in the number of NOPs in the inlined programs appears to be directly related to the amount of inlining for each program. Put another way, the number of NOPs depends upon the number of calls in the program.

For some of the programs studied, the average size of the basic block decreases. However, it may be noted that the average basic block size is a static measure of the program representation. The average basic block size might be decreasing just because of an increase in the number of small basic blocks due to inline function expansion at each call site. Thus, despite this, we could still get better code scheduling because the increase in the size of other basic blocks would lead to more architecture specific optimizations.

Due to this dramatic improvement in code scheduling, the execution time should have improved. However, in most cases, the effect due to better instruction scheduling is just offset by the dynamic instruction count. Overall, as seen in Table 6 more instructions are executed in inlined programs than in their non-inlined counterparts.

However, just a decrease in number of NOPs without an accompanying decrease in the number of dynamically executed instructions can be misleading. Let us consider the case of 026.COMPRESS. Recall that this was the only program for which the number of load/stores actually went up by about 10%[5]. In this case, more instructions can be scheduled in the branch delay slots of these extra load/store instructions. As we shall see in the next section, indeed, the decrease in NOPs did not translate into an improvement in execution time for 026.COMPRESS.

4.3 Execution Times

We measured the execution times using the shell builtin *time* command. This gives the system and user times. Adding these times gives the total running time for the program. To account for the variations, we ran each program five times and considered the average time. The program was executed once before the time measurements so as to set up the cache.

Despite having discussed various metrics to study the effectiveness of inlining, execution time remains the bottom line for any program improvement technique. The execution times for the inlined and the non-inlined programs are reported in Table 11.

From the tables it is clear that for most of the benchmark programs (except for 022.LI) there is no significant improvement in the execution times whereas the C++ programs do show some improvement due to inlining. This difference between difference C and C++ program behavior (see [5]) is due to the following reasons:

- The average function size in C++ programs is lesser than that in C programs. As discussed previously, it would be preferable to inline smaller functions with fewer local variables. Hence, we see improvement in execution time in inlined C++ programs versus that in C programs.
- Another important characteristic that distinguishes C++ programs from C programs is the median call stack depth. Again, since most of our study is for the register window architecture,

[5] As noted previously, this was due to inlining-induced register spills

Program	Execution Time (sec)		Compilation Time(sec)	
	NonInlined	Inlined	NonInlined	Inlined
026.COMPRESS	3.792	3.736	6.480	8.670
023.EQNTOTT	28.714	28.562	24.34	24.63
008.ESPRESSO	24.018	23.27	105.23	112.710
022.LI	43.688	36.77	38.250	53.710
072.SC	23.614	23.638	60.090	65.140
052.ALVINN	141.374	141.362	4.680	5.280
MERGER	20.796	20.116	17.260	19.33
SHORE	46.95	37.732		
STRING	23.57	23.244	7.840	9.240
QTREE	15.722	8.382	36.110	94.830

Table 11: Timings for the Programs

Program	Execution Time	Compilation Time
026.COMPRESS	0.985232	1.337963
023.EQNTOTT	0.994706	1.011915
008.ESPRESSO	0.968857	1.071082
022.LI	0.841650	1.404183
072.SC	1.001016	1.084041
052.ALVINN	0.999915	1.128205
MERGER	0.967301	1.119930
SHORE	0.803663	
STRING	0.986169	1.178571
QTREE	0.533138	2.626142

Table 12: Inlined to Non-inlined timing ratios

having a larger call stack depth will result in more register window (overflow/underflow) traps. This will adversely affect program performance. And since inlining helps in reduce the call stack depth, one would benefit more for C++ programs than for C programs.

We now try to analyze the individual benchmark programs.

1. *026.COMPRESS :* This program took almost as much time or sometimes more after inlining. This program has one function *output()* which is called most frequently. On inlining this function the number of calls/returns did reduce significantly. But there was also a significant increase in the number of loads and stores which offset the advantage gained by inlining. We found that this function was quite big and had many local variables. So after inlining the number of local variables in the caller routine exceeded the number of registers causing them to spill thus increasing the number of loads and stores.

2. *023.EQNTOTT :* Most of the function calls here occur through a function pointer *cmppt*, so it was not possible to inline these calls. Hence there is no significant change in the timings.

3. *008.ESPRESSO :* This function shows a moderate improvement of about 3.2% after inlining.

4. *022.LI :* This was the only C function which showed a significant improvement in running time. As more and more functions were inlined we got improvements up to 15%. The reason why inlining worked here was that it had many small functions with few variables which were being called quite often.

5. *072.SC :* This program had recursive functions which couldn't be inlined. So there no improvement. Here again the number of load/stores has increased after inlining.

6. *052.ALVINN :* This program showed almost no improvement. Here none of the inlined functions are called very frequently. So the number of calls does not reduce much. Also the inlined functions have many local variables and are computationally intensive.

7. *MERGER and STRING :* These programs showed a moderate improvement, *merger* by about 3% and *string* by 1.5%. In this case many functions could not be inlined since they were library routines.

8. *SHORE and QTREE :* Both are substantial pieces of C++ code with many class definitions and member functions. A lot of inlining takes place with the finline-functions option. So there is a significant improvement in the timings. *SHORE* showed an improvement of about 20% whereas *Qtree* showed an improvement of about 47%.

Thus most programs show small or no improvements while a few do really well after inlining.

Machine	Execution Time (sec)	
	NonInlined	Inlined
SUN SPARC	3.792	3.736
DEC MIPS	8.9188	9.0924
HP RISC	5.956	5.136

Table 13: Timings on different Machines

Machine	Code Size(bytes)		Instn. Count		Total Basic Blocks		BBlock Size	
	NonInlined	Inlined	NonInlined	Inlined	NonInlined	Inlined	NonInlined	Inlined
SUN SPARC	122880	131072	13284	13891	2960	3068	4.488	4.528
DEC MIPS	74040	74040	6331	6645	2960	3068	4.488	4.528
HP RISC	143648	147744						

Table 14: Static Information on different Machines

4.4 Different Architectures

To find how the architecture affects the program behavior, we studied the effects of inlining of 026.compress program on the SUN SPARC, DEC MIPS and the HP RISCs. Inlining should be more effective on non-register window architectures since the call/return overhead is more. Tables 13, 14, 15 and 16 compare the metrics for these three cases.

Since QPT hasn't been implemented on the HPs we couldn't get the figures for it. We could make a few observations from the tables. As noted earlier 026.COMPRESS did not show any improvement on the SUN SPARCs. On the MIPS too, there wasn't any improvement. But the inlined program did perform better on the HP by about 12%. We had claimed that on the register window architecture of the SUN, the increased number of local variables causes register spills leading to more loads and stores. Both the MIPS and the HPs do not have register windows. Hence more registers should be available and spills should reduce. This does seem to be the case for the MIPS since we can see from the table that the number of loads and stores reduces after inlining. But still the dynamic instruction count for the inlined program is more. There seem to be some side effects from inlining. QPT statistics show that the number of some instructions like *addiu*, *lui* seem to increase significantly. Why this happens is not very clear, but maybe it is due to the compiler not producing the best code.

Machine	Total Dynamic Instructions		Total Calls	
	NonInlined	Inlined	NonInlined	Inlined
SUN SPARC	77501763	82137716	251688	796
DEC MIPS	14689046	14187259	251329	437

Table 15: Some dynamic characteristics for different Machines

Machines	Total Loads		Total Stores		Total NOPs	
	NonInlined	Inlined	NonInlined	Inlined	NonInlined	Inlined
SUN SPARC	15392602	17585661	6513248	6513249	1000999	998
DEC MIPS	16362611	15359075	7516934	6513398	18381849	18381835

Table 16: Breakdown of instructions for different Machines

NonInlined	Inlined	Ratio
26.668	24.196	0.9073

Table 17: Execution Time for PERM

4.5 Some Other Tests

Inlining opens up possibilities for further optimizations like constant propagation, common subexpression elimination, code hoisting etc. To test whether gcc would actually do these, we wrote small programs for each of these cases. In simple cases gcc did perform the optimizations after inlining. But as the test cases were made more complicated, the optimizations were less effective. In many cases there were no additional optimizations. Thus in most real-life programs we don't see much scope for improved optimizations after inlining.

The register window architecture of the SUNs does seem to have reduced the call overhead successfully. But when the call stack depth exceeds the number of windows, a trap is generated. This is a very costly operation and is avoided by inlining. To check the effect on inlining on register-window traps, we wrote PERM[6] which was recursive in nature and the functions are called at different call stack depths.

To measure the register window traps we used *Quantify*. The non-inlined version did generate many traps as the call stack depth was often exceeded. On inlining the number of register traps reduced by almost half. Consequently the execution time also improved by about 9%(refer Table 17).

5 Summary and Conclusions

We have studied the effects of function inlining on a set of C and C++ programs. Throughout the course of the previous section, we have discussed individual metrics as some of the reasons that effect the performance of function inlining. Besides summarizing our findings in this section, we attempt to derive the "big" picture of why inlining is important. Following are some of our key results:

- It has been believed that there is lot of overhead due to function calls. This statement would have been true without the register windows architecture. Our findings indicate that the register windows architecture performs really well in most cases. In these cases inlining does

[6] A program generates all permutations of numbers.

not buy one much because there is very little overhead involved in making those function calls. This was the typical behavior that we observed in the C programs that we studied.

- *Function Size & Call Stack Depth* If the function is large and has many local variables then inlining causes register spills and has a negative effect on the execution time. This could cause the inlined program to perform worse than the non-inlined program as was observed for 026.COMPRESS. So, inlining is recommended for functions which are small in size and have a few local variables.

  Register window architecture works well when the call stack depth is less than the number of windows. More depth would lead to register window traps which are costly.

  Grunwald et al [5] observe that C programs tend to have smaller call stack than C++ programs (median of 9.9 versus 12.1). Also as compared to C programs, C++ programs have order of magnitude more procedure calls. C++ procedures on the average tend to be smaller compared to C procedures. All these factors indicate that inlining should be more profitable for C++ programs. Indeed this is what we observed. C++ programs did show much more significant improvement as against the C programs. Therefore, function inlining for C++ programs is a good idea.

- *Inlining Algorithm in general is more difficult* In most real-life programs there are many hurdles to inlining. We cannot inline recursive functions, functions called through function pointers or library functions. Thus a large portion of the function calls cannot be inlined and there are no significant improvements as was observed in some benchmarks.

- *More Optimization after inlining ? (No Way !)* It was believed that inlining function will lead to more optimization opportunities. While that is true for simple programs, it does not work that well in general. The state of the art compilers are not able to take full advantage of inlining.

- *Increase in Compilation Time* The compilation times increases significantly (about 30% on the average) for the inlined program. Inlining itself takes times and the number of basic blocks also increases leading to higher analysis time.

Thus our results show that for most real-life C programs inlining is not such a good idea. The improvements, if at all are minimal. On an average we can only expect an improvement of about 2-3%. But inlining does seem promising for C++ programs where we obtained improvements from 10 to 46%.

References

[1] Chow F. C., *Minimizing Register Usage Penalty at Procedure Calls*, Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation, June 1988.

[2] Cooper Keith D., Hall Mary W., and Torczon Linda, *Unexpected Side Effects of Inline Substitution: A Case Study*, ACM Letters on Programming Languages and Systems, March 1992, pp. 23-32.

[3] Davidson J. W. and Holler A. M., *A Study of a C Function Inliner*, Software Practice Experience, August 1988, pp. 775-790.

[4] Emer J. and Clark D., *A Characterization of Processor Performance in the VAX-11/780*, Proceedings of the 11th Annual Symposium on Computer Architecture, June 1984.

[5] Grunwald Dirk, et al *Quantifying Behavioral Differences Between C and C++ Programs*, Journal of Programming Languages, Vol 2, Num 4, 1994.

[6] Holler A. M., *A Study of the Effects of Subprogram inlining*, Ph.D. Thesis, March 1991

[7] Hwu W. W. and Chang P. P., *Inline Function Expansion for Inlining C Programs*, In Proc. of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, July 1989, pp. 246-257.

[8] Richardson S. and Ganapathi M., *Interprocedural Analysis versus Procedural Integration*, Information Processing Letters, August 1989, pp. 137-142.

[9] Wall D. W., *Register Windows vs. Register Allocation*, Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation, June 1988.

Appendix

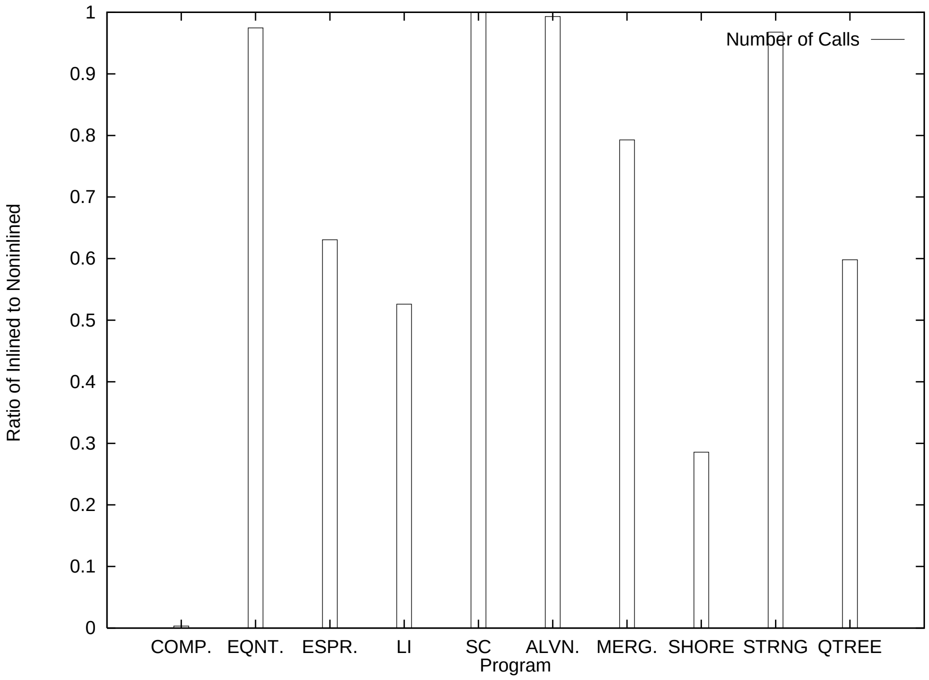

Figure 1: Inlined to Non-inlined Ratios for number of calls

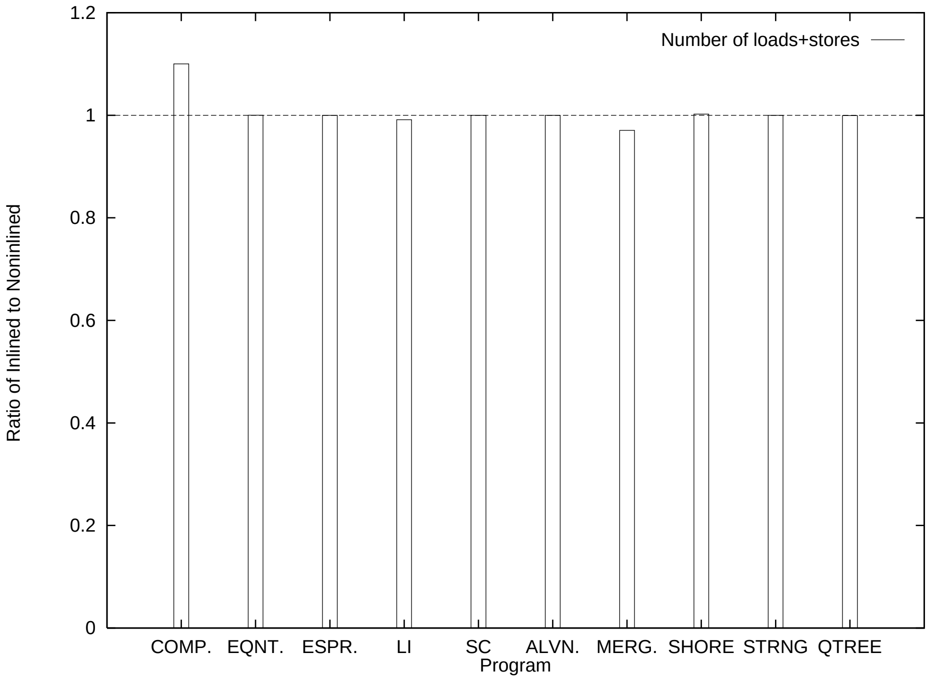

Figure 2: Inlined to Non-inlined Ratios for number of load/stores

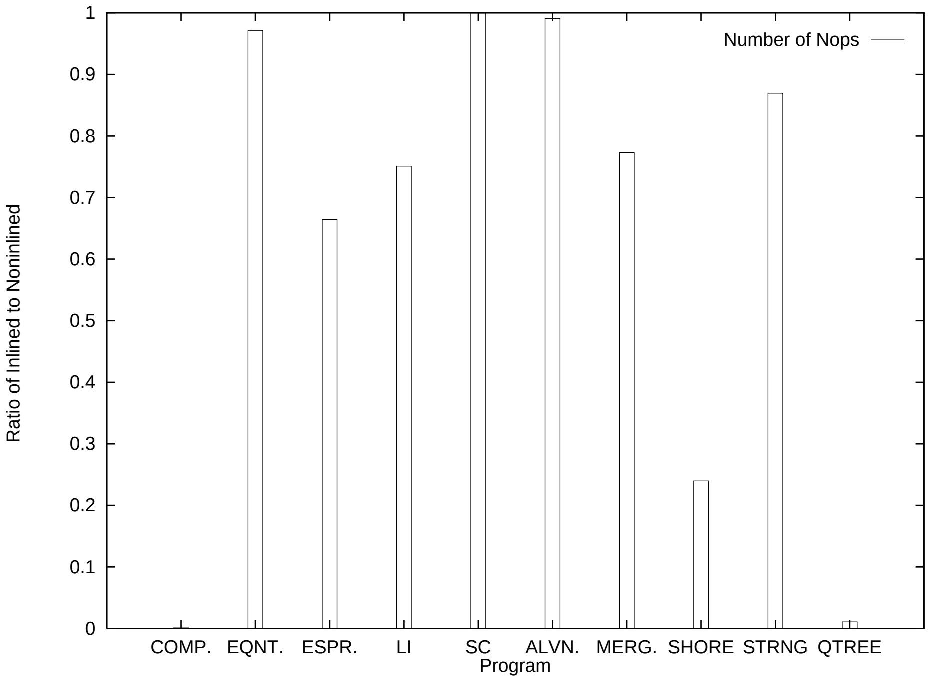

Figure 3: Inlined to Non-inlined Ratios for number of nops