Predicting Program Behavior Using Real or Estimated Profiles

David W. Wall

Digital Equipment Corporation
Western Research Laboratory

Abstract

There is a growing interest in optimizations that depend on or benefit from an execution profile that tells where time is spent. How well does a profile from one run describe the behavior of a different run, and how does this compare with the behavior predicted by static analysis of the program? This paper defines two abstract measures of how well a profile predicts actual behavior. According to these measures, real profiles indeed do better than estimated profiles, usually. A perfect profile from an earlier run with the same data set, however, does better still, sometimes by a factor of two. Unfortunately, using such a profile is unrealistic, and can lead to inflated expectations of a profile-driven optimization.

1. Introduction

Many people have built or speculated on systems that use a run-time profile to guide code optimization. Applications include the selection of variables to promote to registers [7,8], placement of code sequences to improve cache behavior [3,6], and prediction of common control paths for optimizations across basic block boundaries [2,5].

We can evaluate such a technique by timing the program, profiling it, optimizing it based on the profile, timing the optimized version, and finally comparing the two times. Unfortunately, it is common for researchers to use the same data set for the profiling run as for the timing runs. This may give a distorted picture of the efficacy of the technique, because in practice we will optimize based on some profiles, and then run the program many times on data sets that may not match those of the profiling runs. If there is considerable difference in program behavior from one run to another, we might find that a real profile is no better than an estimated profile derived from a static analysis of the program.

Thus two questions arise that are rarely adequately answered. How well does a profile from one run predict the behavior of another? And how well can you do with a statically estimated profile? It is important to answer these questions in general terms as well as specific. A profile from a different run may be very useful for one kind of optimization but nearly useless for another. The optimization may require finding the specific program entities that are most used, or it may require only finding some that are used a lot.

This paper describes a study of how well an estimated profile predicts real behavior, and how well a profile from one run predicts the behavior of a different run.

2. Methodology.

For the purposes of this paper, a *profile* is a mapping from instances of some kind of program entity, like variables or procedures, into numeric weights. The weight may be the number of times the program entity was referenced, or the total cost of all those references, or something else altogether. We assume that a profile is sorted in decreasing order of weight.

We will be concerned with five kinds of profiles. The first is the *basic block* profile, which is the mapping from each basic block to the number of times it is executed. The second is the *procedure entry* profile, which maps each procedure to the number of times it is entered. The third is the *procedure time* profile, which maps each procedure into the number of instructions executed during all of its calls.* The fourth is the *call* profile, which maps each distinct call site to the number of times it is executed. The last is the *global variable* profile, which

* This is "time" only in an abstract sense, as it neglects cache misses and fpu stalls; nevertheless it estimates a procedure's runtime better than a count of its invocations does.

Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.
Toronto, Ontario, Canada, June 26-28, 1991.

maps each global variable to the number of times it is directly referenced.

We might use a global variable profile to decide which globals to promote to registers. A call profile might show us which calls are worth expanding inline, and a procedure profile which procedures are worth optimizing extra hard. Block profiles are useful for cache optimization.

We are interested both in real profiles and in estimated profiles. In either case we construct the profile by dividing the program into basic blocks, deciding how many times each basic block is executed, and combining these counts with static information from the code segment of the program and its loader symbol table.

For a real profile, we use real basic block counts obtained by running the program on a particular set of test data. The *pixie* tool from Mips [4] instruments an executable file with basic block counting; when the instrumented program is run, it produces a table telling how many times each basic block was executed. From this table, in combination with static information from the uninstrumented executable file, we derive the five kinds of profiles.

For an estimated profile, we use estimated basic block counts obtained from a static analysis of the program code. We divide the program into basic blocks, and connect them into procedures and flow graphs based on the branch structure.* We then identify the loops by computing the dominator relation and finding the back edges, edges such that the tail dominates the head. A loop consists of the set of back edges leading to a single dominator, together with the edges that appear on any path from the dominator to the head of one of the back edges [1]. We also build a static call graph by finding all the direct calls in the program. This graph will not include calls through procedure variables, but we will note the presence of such calls so we can identify procedures that not distant from true leaf procedures.

Given this information, we considered four different ways of estimating basic blocks counts. Each of these estimates has one or two parameters. The first is the *loop-only(n)* estimate, in which a block's count is initially 1 and is multiplied by n for each loop that contains it; this ignores the effects of the call graph. The second is the *leaf-loop(n,k)* estimate, in which we compute the distance d of each procedure from the most distant leaf in the call graph, and then multiply the loop-only(n) count by the larger of 1 and $1024 / k^d$ for the procedure containing each basic block. The third is the *call-loop(n)* estimate, in which the loop-only(n) count is multiplied by the static number of direct calls of the block's procedure. The fourth is the *call+1-loop(n)* estimate, which is the loop-only(n) count multiplied by one more than the static number of direct calls of the block's procedure. The call+1-loop estimate is like the call-loop estimate, but procedures that are called only indirectly will not be shut out altogether; unfortunately procedures that are never called are also readmitted.

As a reality check, we also computed *random* profiles in which the items in the profiles are permuted randomly. If a random profile predicts real behavior as well as an estimated profile, we know our estimating technique is not very good.

An optimizer would use a profile by selecting the heaviest entries in it and doing something special to them: promoting them to registers, optimizing them extra hard, or whatever. The question is how well an *optimizing* profile, real or estimated, predicts a *timing* profile assumed to describe the behavior of a production run we are hoping will be fast. For this study we considered two general methods and two specific mentods of evaluating an optimizing profile.

In the first general method, *key matching*, we take the top n entries of the optimizing profile and see how many of them are also in the top n entries of the timing profile. For instance, consider the procedure profiles in Figure 1. If we let $n = 8$, we see that the first 8 members of the optimizing profile include 5 of the first 8 members of the timing profile. Thus the optimizing profile gets a score of 5/8, or 0.625.

306068	full_row
242254	force_lower
190252	malloc
190250	free
126993	set_or
86450	setp_implies
71835	d1_order
60790	set_clear
•	• •

878373	cdist0
245657	d1_order
138058	force_lower
72374	setp_disjoint
48672	cdist01
47029	malloc
47027	free
36491	full_row
•	• •
19131	set_or
15065	set_clear
•	• •
4792	setp_implies
•	• •

Figure 1. Optimizing profile (left) and timing profile.

In the second general method, *weight matching*, we take the top n entries of the optimizing profile and look up their weights in the timing profile, and then compare the total to the total of the top n entries of the timing profile. For example, taking the profile in Figure 1 and again assuming $n = 8$, the total of the optimizing profile's top 8 entries as recorded in the timing profile is

* The Mips code generation is stylized enough that we can recognize indirect jumps that represent case-statements, and can deduce what the possible successor blocks are.

program	*globals*	*procs*	*calls*	*blocks*	*description*
bisim	283	1193	4645	16054	multi-level hardware simulator
bitv	383	1384	4650	17565	timing verifier
udraw	534	4806	17653	46530	drawing editor
egrep	43	72	182	1475	file searcher
sed	54	73	277	2049	stream editor
emacs	487	1024	3568	15098	Gosling's text editor
yacc	66	104	501	2870	parser generator
ccom	170	478	3188	9483	Titan C front end
gcc1	610	1554	8526	40833	gnu C front end
eqntott	55	129	477	2696	truth table generator
espresso	73	434	2813	10505	set operation benchmark

Figure 2. The eleven test programs.

553250, while the total of the timing profile's top 8 entries is 1513681. By this measure, then, the optimizing profile gets a score of 553250/1513681, or 0.365. Note that key matching is symmetric (we get the same score comparing A to B as comparing B to A), but weight matching is asymmetric.

For each kind of profile, we will apply this approach for different values of n, to see how well one real profile predicts others, and how well an estimated profile predicts real ones. For each test program and profile class, we have several different methods of estimating profiles and several different test runs that produce real profiles, so we can get a lot of individual scores of one profile against another. With such a wealth of data, little of which corresponds to any kind of continuum we can graph, our only choice is to present selections and averages across several different dimensions.

Key matching and weight matching are fairly abstract methods, so we also considered two specific ways of scoring an optimizing profile. In each of these we assume a hypothetical optimization that depends on a profile, and compare the improvement in performance from an optimizing profile to the improvement from using the timing profile as its own optimizing profile. One optimization is the promotion of global variables to registers. The other is intensive optimization of the most important procedures.

We should note one important limitation of this approach of this paper. It does not address the stability of a profile over successive versions of the same program undergoing development. One would expect that some kinds of profiles, such as global variable use or procedure invocation, might be relatively stable even when the program is modified. Although a program under development might not be run enough times to merit profile-based optimization, it would still be interesting to know whether it would be feasible. A thorough study of this question may be in order, but is not considered here.

3. Programs and data used

Our test suite consists of eleven programs. Two of them, a text editor and a drawing editor, are interactive. Two are CAD tools used at WRL. Two are different C compiler front ends; one is recursive descent, the other yacc-based. Three of them are SPEC benchmarks. Figure 2 describes the complete test suite.

Wherever possible, we tried to give the programs realistic but quite different input data, in the hopes of maximizing the differences in their behavior. We ran bisim three different ways: completely high-level simulation, high-level functional units with a transistor-level register file, and transistor-level functional units with a high-level register file. Bitv was run to verify a datapath, a register file, and a write buffer. The drawing editor was used to draw schematics and also a home landscape design. Egrep and sed were run with both simple and complicated patterns, and with large and small inputs. Emacs was used to edit source files, English text files, and very long simulation configuration files. Yacc was used with a high-level language grammar, an intermediate language grammar, and a command grammar for a window manager. The two C compilers were run with the same four source files, two written by humans and two generated by the C++ front end. The eqntott and espresso benchmarks from SPEC were run with different inputs provided by SPEC.

4. Results

We used both key matching and weight matching for n = 1, 2, 4, 8, 16, 32, 64, and 128. Given a test program and a value of n, we proceeded as follows. We computed estimated profiles nine different ways, including the random profile. An estimated profile was scored against each real profile for the same test program; we then averaged these scores. Each real profile was scored against each of the *other* real profiles, but not against itself; we then averaged all the scores comparing two real

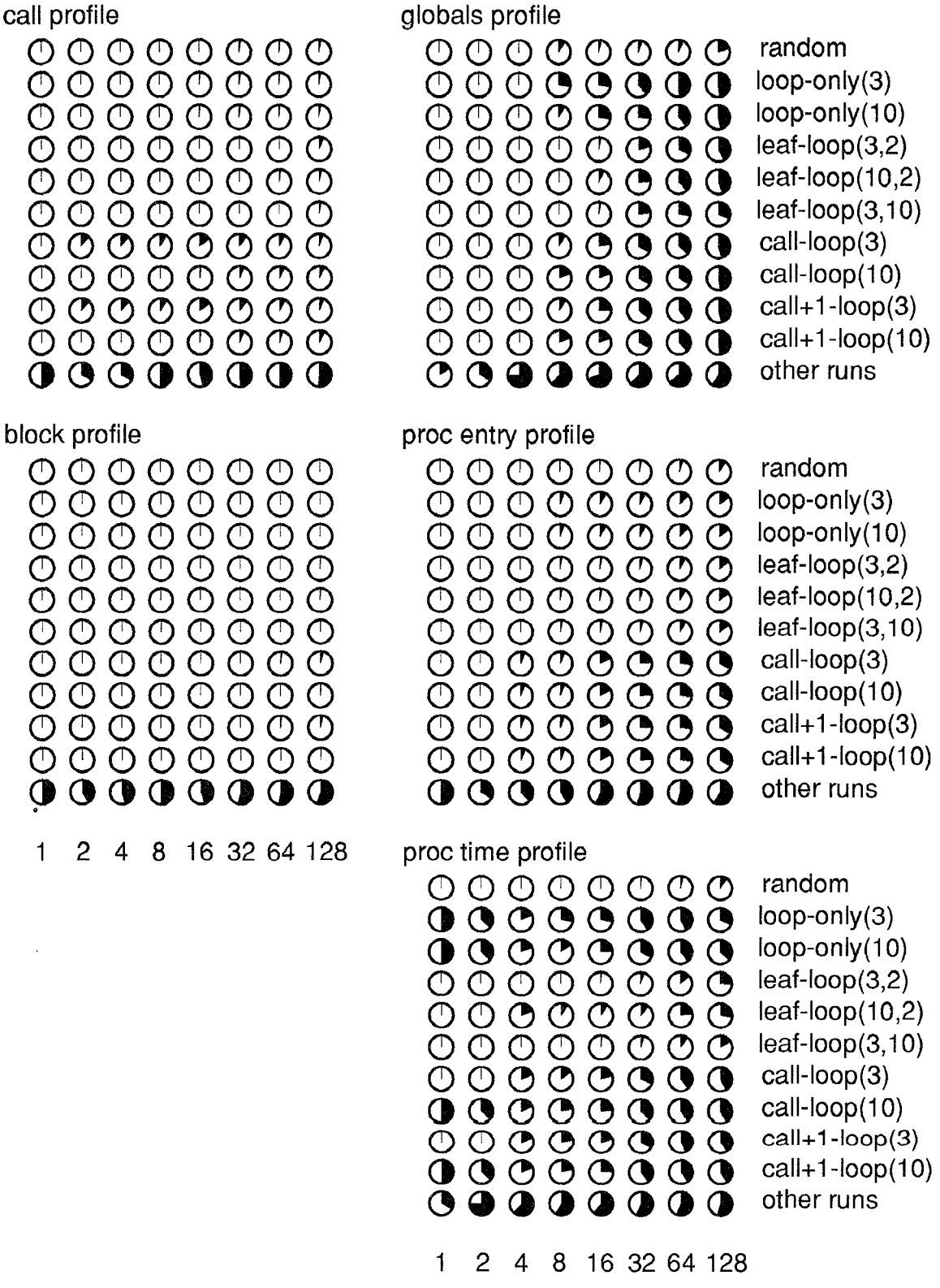

Figure 3. Average key-matching scores for gcc1.

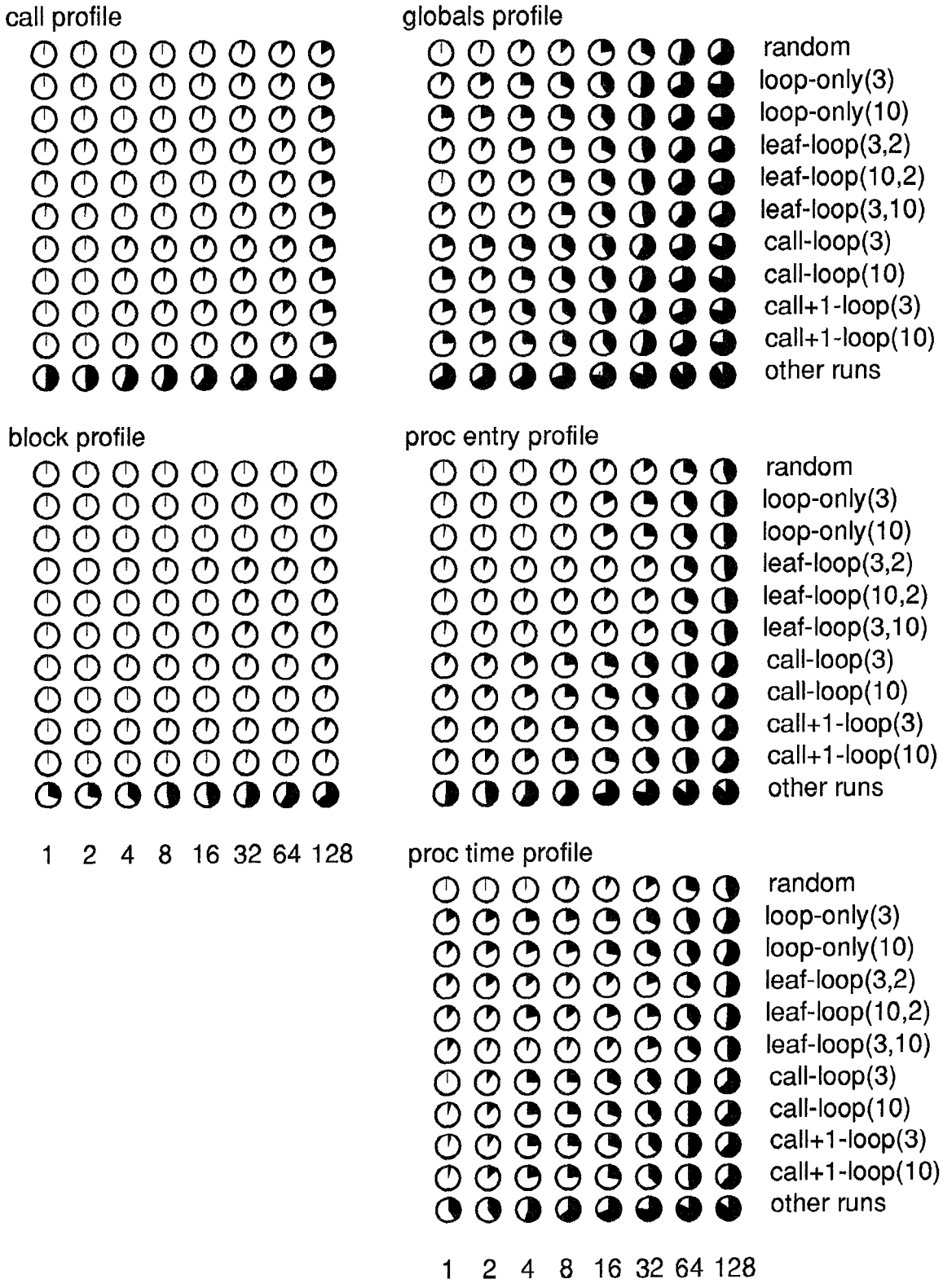

Figure 4. Average key-matching scores.

profiles. For each test program and each of the two matching techniques, this gave us 352 scores: the cross product of the four profile classes, the eleven profile acquisition techniques (one random, nine estimated, and aggregated real profiles), and the eight values of n. As an example, Figure 3 shows the 352 key-matching scores for the gcc1 test. The fraction of the circle filled with black is the score, so a completely black circle is perfect and a completely white circle is terrible.

4.1. Key matching

We computed the 352 key-matching scores for each of the test programs, and then averaged these over all programs. Since the 352 scores are themselves averages over several program runs, this means we are taking averages of averages; this gives each program equal weight even though some had more datasets than others.

The results are shown in Figure 4. We can see that predicting which globals will be used is fairly easy, which is unsurprising since there are fewer of them than of the other profiled entities. (In some cases there are fewer than 128 globals, which lets even a random profile get a perfect score.) The call-loop estimates do rather better than the other estimates. There is relatively little difference between estimates computed using the same technique parameterized differently. As we would expect, actual profiles do considerably better than estimates, but even actual profiles are disappointingly bad at predicting which basic blocks will be executed most.

Some of the estimates are surprisingly bad, doing little better than the random profile! The random profile did so well, in fact, that we suspected that coincidence had given us an anomously good random order. We tested this by computing 50 random profiles in each case and then averaging the resulting scores. These average scores were about as good as that of our original random profile, and in many cases even better, so it would seem that we did not just happen to hit it lucky. The fact that a random profile comes so close to some of the estimated profiles suggests that these estimated profiles aren't really buying us that much.

4.2. Weight matching

We also computed the 352 weight-matching scores for each of the test programs, and then averaged these over all programs. The results are shown in Figure 5. We were rather more successful at weight matching than at key matching.* The trends, however, are much the same: globals are easy to predict, blocks are hard, call-loop estimates work better than the others, and actual profiles work best of all.

* This is not guaranteed in general: the optimizing profile in Figure 1, for example, got a better score at key matching.

4.3. Weight matching by percentages

We observed that it was easier to predict globals and procedures than calls and blocks, at least in part because there are fewer of them. To see whether there was more to it, we tried scoring each optimizing profile for values of n that are percentages of the total number of entities in the profile's domain. The results are shown in Figure 6. Here we see several interesting things. As one might expect, the random profile gets a score roughly equal to the percentage of items we tried to predict, more evidence that these random profiles are not freaks. The differences between the profile kinds are much smaller: the five rows for real profiles are very similar. On the other hand, the advantage of globals is not completely gone: we can predict the top 2% or 10% of globals a bit more accurately than we can of blocks or even of procedures. Furthermore, the similarity between the "other runs" rows does not carry over as well to the rows for estimated profiles: the estimated profiles get scores several times better for the top 2% of globals as for the top 2% of calls. Globals really do seem to be easier in some absolute sense, and not just because the domain is smaller.

4.4. Differences between test programs

There is a substantial variation in the predictability of the different programs. Figure 7 shows the average score for real (not estimated) profiles, using the weight matching criterion. This figure shows the last rows of each profile class in Figure 5, broken down by program. Emacs is astonishingly consistent from one run to another, perhaps because it is built around a Lisp interpreter, so that much of its control logic (and thus much of its variability) is hidden in the data structure. Unfortunately, this argument would lead us to suppose that gcc1, with a table-driven parser, might be more predictable than ccom, with a recursive descent parser. But in fact ccom is noticeably more predictable than gcc1. The least predictable programs are egrep, sed, and eqntott, which is quite surprising because they are also the smallest.

Figure 8 is analogous to Figure 7, but shows the results for the call-loop(3) estimate instead of for real profiles. Figures 4 and 5 suggest that this estimate was the best one we considered. Even this estimate was rarely much good at predicting the block profile, but it was quite good at predicting the global profile, particularly for n at least 16. The estimated profile was a bit more likely than a real profile to make an anomalous lucky guess, giving a higher score for low n than for high n. Although emacs was predicted quite well by real profiles, it is predicted relatively poorly by this estimate; espresso and bisim, among others, are noticeably better.

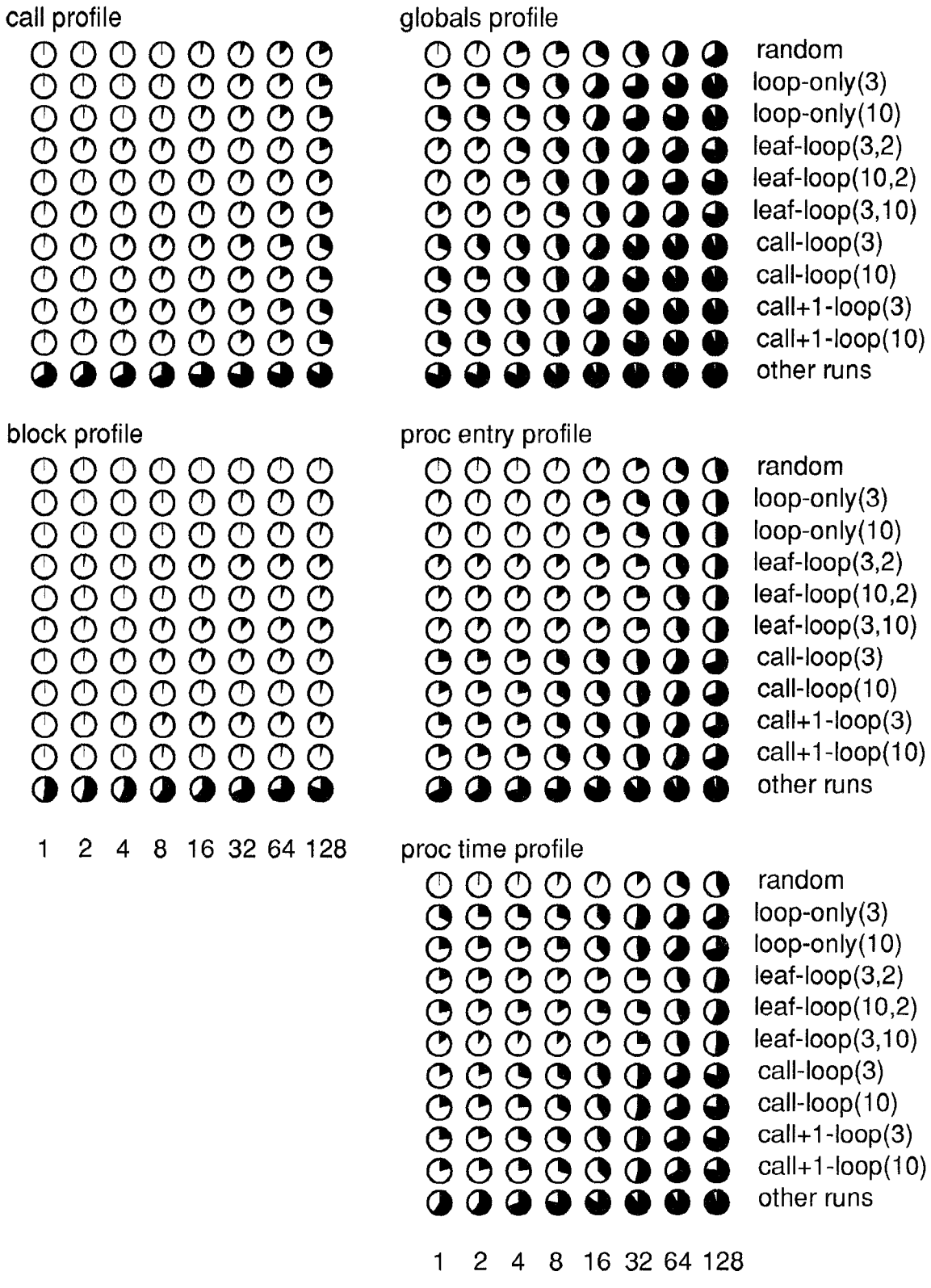

Figure 5. Average weight-matching scores.

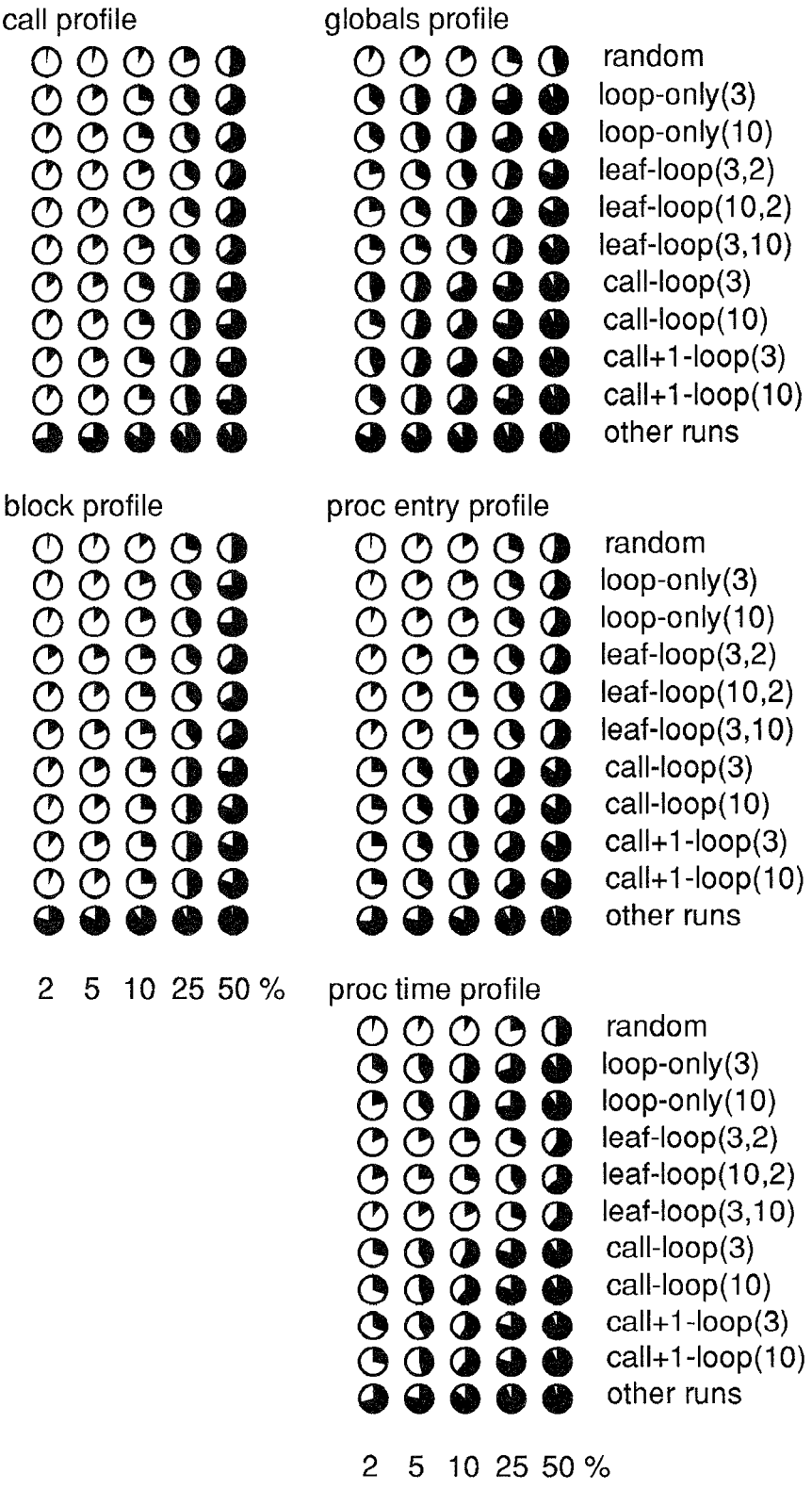

Figure 6. Average weight-matching scores for n equal to a percentage of the number of profile entries.

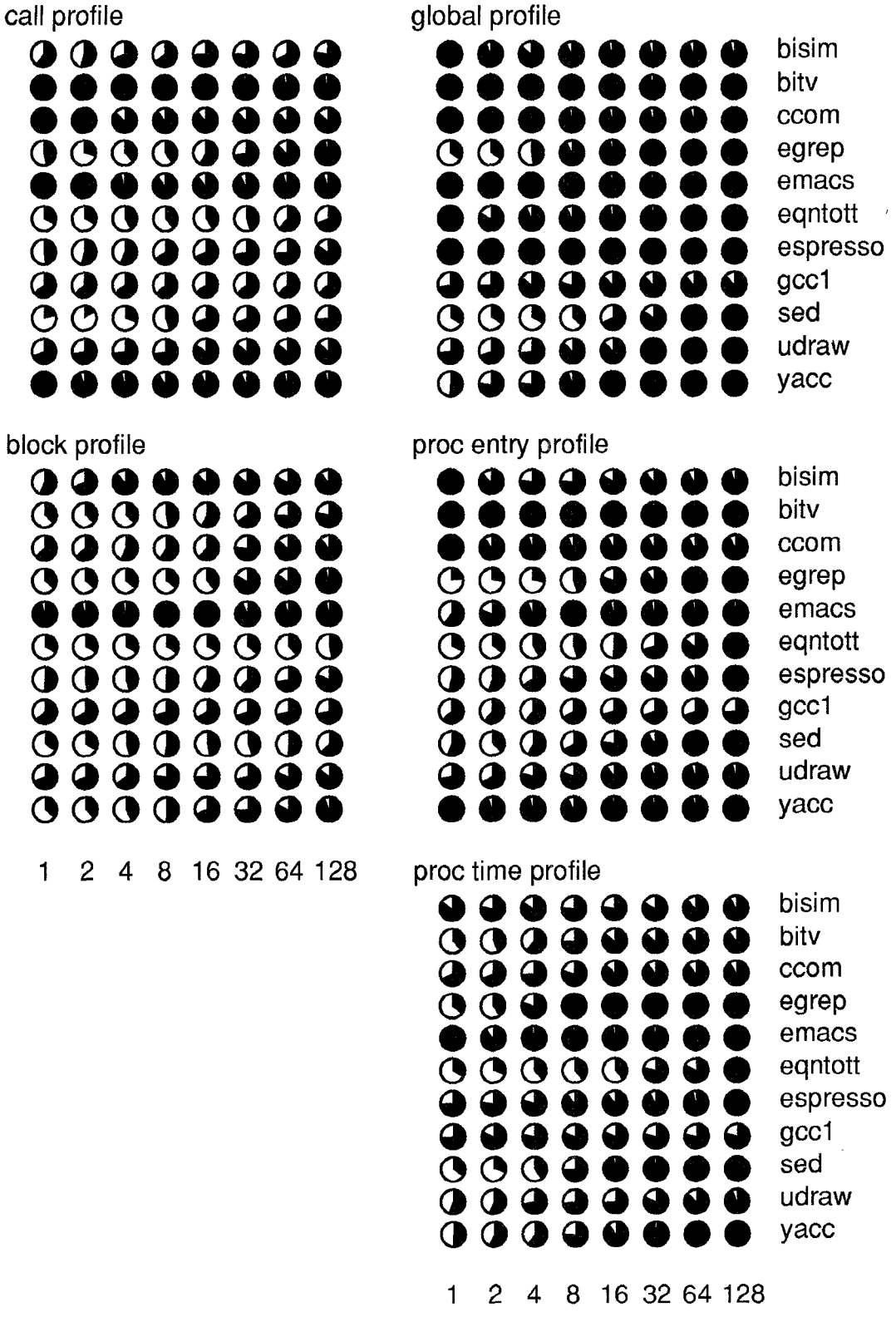

Figure 7. Average weight-matching scores for real profiles.

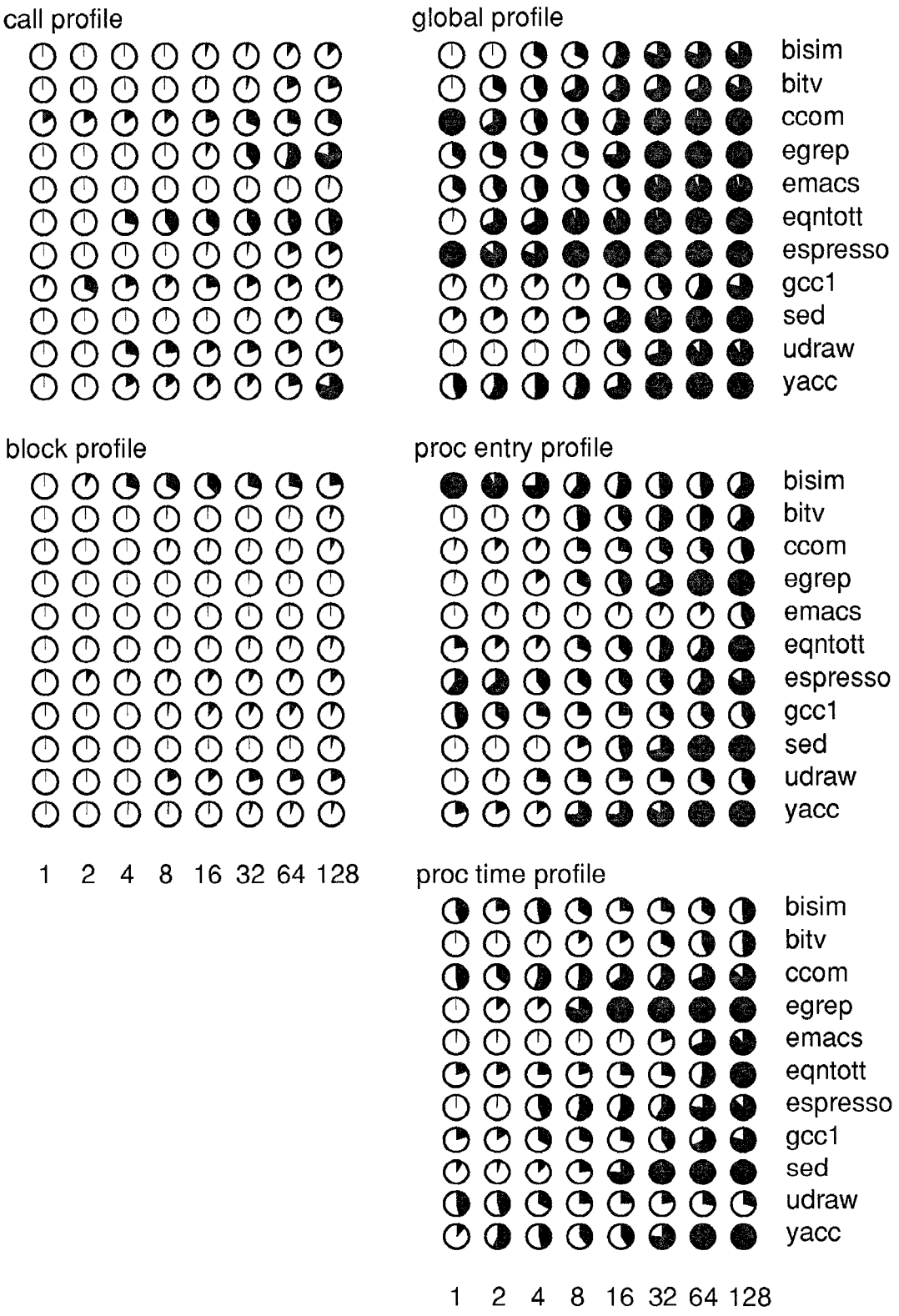

Figure 8. Average weight-matching scores for call-loop(3) estimates.

4.5. Global register allocation

To apply this technique to a realistic specific example, let us suppose that we suddenly have eight registers available that we can use to promote eight global variables or constants. They payoff of doing this is that all the loads and stores of the globals we select will be removed. We can estimate our improvement in performance by counting the executions of these loads and stores and dividing the total by the total number of instructions executed.* We did this both for a timing profile (to see how well we could possibly have done) and for an optimizing profile, in each case computing the counts using the timing profile.

The results are shown in Figure 9. This optimization by itself doesn't do a lot for performance: even if magically driven by the counts from the timing profile, the improvement in performance is only 2.7%. A good estimated profile gives us about half of the maximum possible performance improvement, and an actual profile gives us about 85% of the maximum.

	improv	max	ratio
random	0.6%	2.7%	
loop-only(3)	1.3%	2.7%	
loop-only(10)	1.2%	2.7%	
leaf-loop(3,2)	1.2%	2.7%	
leaf-loop(10,2)	1.2%	2.7%	
leaf-loop(3,10)	1.0%	2.7%	
call-loop(3)	1.2%	2.7%	
call-loop(10)	1.3%	2.7%	
call+1-loop(3)	1.3%	2.7%	
call+1-loop(10)	1.2%	2.7%	
other runs	2.3%	2.7%	

Figure 9. Improvement from global register allocation.

4.6. Selective intensive optimization

As a second specific example, let us suppose we have an excellent but expensive optimization algorithm that will cut the execution time of any procedure in half, but that is so expensive that we can apply it only to 5% of our procedures.

First we will select as the procedures to optimize those we believe will be invoked most often, by picking the first 5% of the entries in the procedure entry profile. As before, we will do this both for an optimizing profile and also for a timing profile; we will compute the improvement in performance using only the counts from the timing profile.

	improv	max	ratio
random	2.4%	31.2%	
loop-only(3)	7.4%	31.2%	
loop-only(10)	7.4%	31.2%	
leaf-loop(3,2)	3.7%	31.2%	
leaf-loop(10,2)	3.7%	31.2%	
leaf-loop(3,10)	3.7%	31.2%	
call-loop(3)	7.3%	31.2%	
call-loop(10)	7.3%	31.2%	
call+1-loop(3)	7.3%	31.2%	
call+1-loop(10)	7.3%	31.2%	
other runs	24.3%	31.2%	

Figure 10. Improvement from intensive optimization of procedures selected using procedure entry profile.

The results are shown in Figure 10. This optimization would speed up our programs by a third if it were driven by a perfect profile. A real profile gives us about three-fourths of that, but even the best estimated profile -- which oddly enough was the simple loop-only estimate -- gives us barely one-fourth. The worst estimates are hardly better than random profiles.

Picking the procedures to optimize based on how many times they are called is imprecise; we are more likely to want to optimize the procedures where we spend the most time. Let us do selective intensive optimization again, this time choosing the top 5% of the procedures in the procedure time profile instead of the procedure entry profile. Figure 11 shows the results. As one might expect, applying the optimization in this fashion is better, giving us a possible improvement of 43.7%.† As before, a real profile from a different run can get about three-fourths of that. This time, though, the best estimates give us nearly half, and the worst are still better than random profiles. This suggests that procedure times are rather easier to predict than procedure invocations, at least at the top 5% level. This is borne out by a study of Figure 6; there is somewhat more black ink in the procedure time profile than in the procedure entry profile.

* This does not take pipeline stalls into account, nor does it consider cache effects, which are likely to increase the benefit of promoting globals to registers. It also assumes that the globals selected are not ineligible because of aliasing. We are interested only in rough numbers here, as an example.

† So this would be a really great optimization if it existed!

	improv	max	ratio
random	2.0%	43.7%	
loop-only(3)	14.5%	43.7%	
loop-only(10)	15.1%	43.7%	
leaf-loop(3,2)	8.3%	43.7%	
leaf-loop(10,2)	9.4%	43.7%	
leaf-loop(3,10)	6.3%	43.7%	
call-loop(3)	18.4%	43.7%	
call-loop(10)	19.3%	43.7%	
call+1-loop(3)	18.6%	43.7%	
call+1-loop(10)	19.3%	43.7%	
other runs	34.5%	43.7%	

Figure 11. Improvement from intensive optimization of procedures selected using procedure time profile.

5. Conclusions

Real profiles from different runs worked much better than the estimated profiles discussed in this paper. The best estimations were usually those that combined loop nesting level with static call counts. Basing the estimate on the procedure's distance from leaves of the call graph was less effective. The worst estimates were hardly better than random profiles. There may of course still be better ways to estimate a profile: this is an interesting open question both in the general case and in specific applications.

Even a real profile was never as good as a perfect profile from the same run being measured. It was often quite close, however, and was usually at least half as good. Profile-based optimization would seem to have a future, but we must be careful how we measure it, lest we expect more than it can really deliver.

Acknowledgements

My thanks to Alan Eustace for goading me into finally doing this study, to Patrick Boyle, Mary Jo Doherty, Ramsey Haddad, and Joel McCormack for helping me obtain some of the data, to Anita Borg, Joel McCormack, and Scott McFarling for helpful comments on the draft, and to Joel Bartlett for the ezd tool that let me draw the 2508 pie-charts in this paper.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*, pp. 602-605. Addison-Wesley, 1986.

[2] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel processing: A smart compiler and a dumb machine. *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pp. 37-47. Published as *SIGPLAN Notices 19* (6), June 1984.

[3] Scott McFarling. Program optimization for instruction caches. *Third International Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 183-191, April 1989. Published as *Computer Architecture News 17* (2), *Operating Systems Review 23* (special issue), *SIGPLAN Notices 24* (special issue).

[4] MIPS Computer Systems, Inc. *Language Programmer's Guide*, 1986.

[5] Scott McFarling and John Hennessy. Reducing the cost of branches. *Proceedings of the 13th Annual Symposium on Computer Architecture*, pp. 396-403. Published as *Computer Architecture News 14* (2), June 1986.

[6] Karl Pettis and Robert C Hansen. Profile guided code positioning. *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 16-27. Published as *SIGPLAN Notices 25* (6), June 1990.

[7] Vatsa Santhanam and Daryl Odnert. Register allocation across procedure and module boundaries. *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 28-39. Published as *SIGPLAN Notices 25* (6), June 1990.

[8] David W. Wall. Global register allocation at link-time. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pp. 264-275. Published as *SIGPLAN Notices 21* (7), July 1986. Also available as WRL Research Report 86/3.