Predicting Program Behavior Using Real or Estimated Profiles

David W. Wall
The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There is a second research laboratory located in Palo Alto, the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a technical note. We use this form for rapid distribution of technical material. Usually this represents research in progress. Research reports are normally accounts of completed research and may include material from earlier technical notes.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution  
DEC Western Research Laboratory, UCO-4  
100 Hamilton Avenue  
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:  
DECWRL::WRL-TECHREPORTS

DARPA Internet:  
WRL-Techreports@decwrl.dec.com

CSnet:  
WRL-Techreports@decwrl.dec.com

UUCP:  
decwrl!wrl-techreports

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word “help” in the Subject line; you will receive detailed instructions.
Predicting Program Behavior
Using Real or Estimated Profiles

David W. Wall

December, 1990
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 statically by examining the program itself? 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. 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 pred-
icion of common control paths for optimizations across basic block boundaries [2,5].

When such work is presented, two questions are often asked but seldom ade-
quately answered. How well does a profile from one run predict the behavior of
another? And how well can you do with an estimated profile derived from static
analysis of the program? 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 kind. The optimization may require identi-
fying the specific program entities that are most used, or it may require only identifying
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 another run.

2. Methodology.

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 infor-
mation from the executable file, we derive four kinds of profiles. The first is the basic
block profile, which is just the mapping from each basic block to its execution count.
The second is the procedure profile, which maps each procedure to the number of
times it is entered. The third 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 maps
each global variable to the number of times it is directly referenced.

If we don’t have basic block counts from pixie, we can try to estimate them. We
first 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 each of whose tail dominates its
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

* 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; this graph will not include calls through procedure variables.

Given this information, we considered four different ways of estimating basic blocks counts. The first is the *loop-only* estimate, in which a block’s count is initially 1 and is multiplied by 3 for each loop that contains it; this ignores the effects of the call graph. The second is the *leaf-loop* estimate, in which the loop-only count is multiplied by 1024 if the block is contained in a leaf procedure, 512 if it is no more than one from a leaf procedure, and so on with powers of 2 up to 1. The third is the *call-loop* estimate, in which the loop-only count is multiplied by the static number of direct calls of the block’s procedure. The fourth is the *call+1-loop* estimate, which is the loop-only count is 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, except that procedures that are called only indirectly will not be shut out altogether; unfortunately procedures that are never called are similarly readmitted.

An optimizer would use a profile by selecting the most frequent entries in it and doing something special to them: promoting them to registers, optimizing them extra hard, or whatever. The question is how well a candidate profile, real or estimated, predicts the behavior described by a reference profile. For this study we considered two abstract methods of evaluating a candidate profile.

The first method, *specific matching*, is to take the top \( n \) entries of the candidate profile and see how many of them are also in the top \( n \) entries of the reference profile. For instance, consider the procedure profiles in Figure 1. If we let \( n = 8 \), we see that the first 8 members of the candidate profile include 5 of the first 8 members of the reference profile. Thus the candidate profile gets a score of 5/8, or 0.625.

```
306068  full_row  878373  cdist0
242254  force_lower  245657  d1_order
190252  malloc  138058  force_lower
190250  free  72374  setp_disjoint
126993  set_or  48672  cdist01
86450  setp_implies  47029  malloc
71835  d1_order  47027  free
60790  set_clear  36491  full_row
  • • •  • • •
  19131  set_or
  15065  set_clear
  • • •
  4792  setp_implies
  • • •
```

Figure 1. Candidate profile (left) and reference profile.

The second method, *frequency matching*, is to take the top \( n \) entries of the candidate profile and look up their frequencies in the reference profile, and then compare the total to the total of the top \( n \) entries of the reference profile. For example, taking the profile in Figure 1 and again assuming \( n = 8 \), the total of the candidate’s top 8 entries
as revealed by the reference profile is 553250, while the total of the reference profile’s top 8 entries is 1513681. By this measure, then, the candidate profile gets a score of 553250/1513681, or 0.365. Note that specific matching is symmetric (we get the same score comparing A to B as comparing B to A), but frequency matching is asymmetric.

Applying this approach to all four kinds of profiles, for different values of \( n \), should give us some notion of how well one profile might predict another. To apply this understanding more specifically, we also did some rough computations of the stability of the profiles when applied in two specific ways. One application is the promotion of global variables to registers. The other is intensive optimization of the most frequently called procedures.

We should note one important limitation of this approach. 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. One might argue that a program under development will not be run enough times to merit profile-based optimization, but 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.

<table>
<thead>
<tr>
<th>program</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bisim</td>
<td>multi-level machine simulator</td>
</tr>
<tr>
<td>bitv</td>
<td>timing verifier</td>
</tr>
<tr>
<td>udraw</td>
<td>drawing editor</td>
</tr>
<tr>
<td>egrep</td>
<td>file searcher</td>
</tr>
<tr>
<td>sed</td>
<td>stream editor</td>
</tr>
<tr>
<td>Gosling emacs</td>
<td>text editor</td>
</tr>
<tr>
<td>yacc</td>
<td>parser generator</td>
</tr>
<tr>
<td>ccom</td>
<td>Titan C front end</td>
</tr>
<tr>
<td>gcc1</td>
<td>gnu C front end</td>
</tr>
<tr>
<td>eqntott</td>
<td>truth table generator</td>
</tr>
<tr>
<td>espresso</td>
<td>set operation benchmark</td>
</tr>
</tbody>
</table>

Figure 2. The eleven test programs.

Wherever possible, we gave the programs 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 both run with two source files written by humans and two source files generated by the C++ front end. The eqntott and espresso benchmarks from SPEC were run with inputs provided by SPEC.

<table>
<thead>
<tr>
<th>procedure profile</th>
<th>global profile</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="chart1.png" alt="Procedure Profile Chart" /></td>
<td><img src="chart2.png" alt="Global Profile Chart" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>call profile</th>
<th>block profile</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="chart3.png" alt="Call Profile Chart" /></td>
<td><img src="chart4.png" alt="Block Profile Chart" /></td>
</tr>
</tbody>
</table>

Figure 3. Average specific matching score.

4. Results

4.1. Specific matching

Our first result assumes that we score candidate profiles by the specific matching criterion, for \( n = 1, 2, 4, 8, 16, 32, 64, \) and 128. Given a test program and a value of \( n, \) we proceeded as follows. 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 profiles. For each test program and each value of \( n, \) this gave us 20 scores: the cross product of four profile classes and five estimate classes (more precisely, four estimate classes and also real profiles from other runs). We then averaged these scores over all programs; this double averaging gave each program equal weight even though some had more datasets than others.

The results are shown in Figure 3. 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. We can see that predicting which globals will be used is fairly easy, probably because there are fewer of them than there are of the other profiled entities. The call-loop estimates do rather better than the other estimates. As we would expect, actual
profiles do considerably better than estimates, but even actual profiles do disappointingly badly at predicting which basic blocks will be executed most.

![Procedure profile](image1)

![Global profile](image2)

![Call profile](image3)

![Block profile](image4)

Figure 4. Average frequency matching score.

### 4.2. Frequency matching

Our next result has the same structure as the previous result, but it assumes frequency matching instead of specific matching. Again, we used \( n = 1, 2, 4, 8, 16, 32, 64, \) and 128. Each profile’s scores were again averaged over all the profiles it was compared against, and the resulting averages were again averaged over the eleven test programs.

The results are shown in Figure 4. We were rather more successful at frequency matching than at specific 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.

### 4.3. Differences between test programs

There is a substantial variation in the predictability of the different programs. Figure 5 shows the average score for real (not estimated) profiles, using the frequency matching criterion. This is the fifth and tenth rows of Figure 4, broken down by program. Emacs is astonishingly predictable, 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. 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.

* This is not guaranteed in general: the candidate profile in Figure 1, for example, got a better score at specific matching.
But in fact ccom is noticeably more predictable than gcc1. The least predictable programs are sed and eqntott, which is a little surprising because they are among the smallest.

Figure 5. Average frequency matching scores for real profiles.
4.4. 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 reference profile (to see how well we could possibly have done) and for a candidate profile, in each case computing the counts using the reference profile.

<table>
<thead>
<tr>
<th></th>
<th>improv</th>
<th>max</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop-only</td>
<td>1.3%</td>
<td>2.7%</td>
<td></td>
</tr>
<tr>
<td>leaf-loop</td>
<td>1.1%</td>
<td>2.7%</td>
<td></td>
</tr>
<tr>
<td>call-loop</td>
<td>1.2%</td>
<td>2.7%</td>
<td></td>
</tr>
<tr>
<td>call+1-loop</td>
<td>1.3%</td>
<td>2.7%</td>
<td></td>
</tr>
<tr>
<td>other runs</td>
<td>2.3%</td>
<td>2.7%</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6. Improvement from global register allocation.

The results are shown in Figure 6. This optimization by itself doesn’t do a lot for performance: even if magically driven by the counts from the reference 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.

4.5. 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. 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 invocation profile. As before, we will do this both for a candidate profile and also for a reference profile; we will compute the improvement in performance using only the counts from the reference profile.

The results are shown in Figure 7. 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.

---

* 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.
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. 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, and to Patrick Boyle, Mary Jo Doherty, Ramsey Haddad, and Joel McCormack for helping me obtain some of the data.

References


WRL Research Reports

‘Titan System Manual.’
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

‘Global Register Allocation at Link Time.’
David W. Wall.

‘Optimal Finned Heat Sinks.’
William R. Hamburgen.

‘The Mahler Experience: Using an Intermediate Language as the Machine Description.’
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.

‘Fragmentation Considered Harmful.’
Christopher A. Kent, Jeffrey C. Mogul.

‘Cache Coherence in Distributed Systems.’
Christopher A. Kent.

‘Register Windows vs. Register Allocation.’
David W. Wall.

‘Editing Graphical Objects Using Procedural Representations.’
Paul J. Asente.

‘The USENET Cookbook: an Experiment in Electronic Publication.’
Brian K. Reid.

‘MultiTitan: Four Architecture Papers.’
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.

‘Fast Printed Circuit Board Routing.’
Jeremy Dion.

‘Compacting Garbage Collection with Ambiguous Roots.’
Joel F. Bartlett.

‘The Experimental Literature of The Internet: An Annotated Bibliography.’
Jeffrey C. Mogul.

‘Measured Capacity of an Ethernet: Myths and Reality.’
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.

‘Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.’
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.

‘SCHEME->C A Portable Scheme-to-C Compiler.’
Joel F. Bartlett.

‘Optimal Group Distribution in Carry-Skip Adders.’
Silvio Turrini.

‘Precise Robotic Paste Dot Dispensing.’
William R. Hamburgen.
“Simple and Flexible Datagram Access Controls for Unix-based Gateways.”
Jeffrey C. Mogul.

V. Srinivasan and Jeffrey C. Mogul.

“Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.”
Norman P. Jouppi and David W. Wall.

“A Unified Vector/Scalar Floating-Point Architecture.”
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.

“Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.”
Norman P. Jouppi.

“Integration and Packaging Plateaus of Processor Performance.”
Norman P. Jouppi.

“A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.”
Norman P. Jouppi and Jeffrey Y. F. Tang.

“The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.”
Norman P. Jouppi.

“Long Address Traces from RISC Machines: Generation and Analysis.”
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.

“Link-Time Code Modification.”
David W. Wall.

“Noise Issues in the ECL Circuit Family.”
Jeffrey Y.F. Tang and J. Leon Yang.
WRL Research Report 90/1, January 1990.

“Efficient Generation of Test Patterns Using Boolean Satisfiability.”
Tracy Larrabee.

“Two Papers on Test Pattern Generation.”
Tracy Larrabee.

“Virtual Memory vs. The File System.”
Michael N. Nelson.

“Efficient Use of Workstations for Passive Monitoring of Local Area Networks.”
Jeffrey C. Mogul.

“A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.”
John S. Fitch.
WRL Research Report 90/6, July 1990.

“1990 DECWRL/Livermore Magic Release.”
WRL Research Report 90/7, September 1990.

“Pool Boiling Enhancement Techniques for Water at Low Pressure.”
WRL Technical Notes

“TCP/IP PrintServer: Print Server Protocol.”
Brian K. Reid and Christopher A. Kent.

“TCP/IP PrintServer: Server Architecture and Implementation.”
Christopher A. Kent.

“Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.”
Joel McCormack.

“Why Aren’t Operating Systems Getting Faster As Fast As Hardware?”
John Ousterhout.

“Mostly-Copying Garbage Collection Picks Up Generations and C++.”
Joel F. Bartlett.

“Limits of Instruction-Level Parallelism.”
David W. Wall.

“The Effect of Context Switches on Cache Performance.”
Jeffrey C. Mogul and Anita Borg.

“MTOOL: A Method For Detecting Memory Bottlenecks.”
Aaron Goldberg and John Hennessy.

“Predicting Program Behavior Using Real or Estimated Profiles.”
David W. Wall.