

On Proebsting's Law

Kevin Scott

Department of Computer Science

Technical Report CS-2001-12

University of Virginia

Charlottesville, VA 22904

jks6b@cs.virginia.edu

Abstract

In 1965 Gordon Moore observed that the capacity of semiconductor ICs doubled every 18 to 24 months. This trend, now known as Moore's Law, has held for over 25 years and is responsible for the exponential increase in microprocessor performance over this period. In 1998 Todd Proebsting made a similar-in-spirit, but altogether less optimistic observation about optimizing compilers. His observation, henceforth known as Proebsting's Law, is that improvements to compiler technology double the performance of *typical* programs every 18 years. Proebsting has suggested an experiment to evaluate the veracity of his observation. This paper presents the results of this experiment and some comments on what Proebsting's Law portends for compiler research.

1. Introduction

Proebsting's Law asserts that improvements to compiler technology double the performance of *typical* programs every 18 years. According to Proebsting's Law, a program compiled by a 25-year-old optimizing compiler would run about 2.5 times slower than the same program compiled with a modern, state-of-the-art compiler. According to Moore's Law, a program executed on a 25-year-old computer would run almost 80,000 times more slowly than it would on a modern machine. The difference between the performance gains due to better compilers versus those due to better hardware is enormous. If the only goal of compiler writers and researchers is to improve the performance of computing systems, then we are fighting a battle against hardware designers that we will never win. An end user desiring nothing but performance will be much better served by next generation hardware than by next generation compilers.

2. Evaluating Proebsting's Law

Proebsting suggests a simple experiment to evaluate the veracity of his law [1]. Compile and run your favorite benchmark programs with and without optimizations enabled. The ratio between the optimized and unoptimized running times entirely reflects the optimizing compiler's ability to improve the benchmark's performance. If we assume that this ratio is around 4 and that compiler research has been going on for roughly 36 years, we get Proebsting's Law—compilers double program performance every 18 years. The constants 4 and 36 were chosen to yield a performance-doubling period of 18 years—a period similar to that of Moore's Law, modulo unit of measure.

Naturally, we should ask ourselves whether or not this experiment can effectively validate Proebsting's Law. There are a number of issues to consider.

Issue 1: The experiment only evaluates the veracity of Proebsting's Law with respect to a specific set of benchmark programs. This is certainly true. However, we feel that by carefully choosing a representative benchmark, we can be reasonably confident that the experiment's results will hold for a wide range of similar programs. This is a tradeoff that many computer systems researchers must make in order to evaluate their hypotheses. We chose to use SPECint95 and SPECfp95 as the benchmarks for the experiments conducted for this paper [2]. This benchmark suite represents a wide variety of applications and scientific programs and has long been accepted as a basis for comparison in systems research.

Issue 2: The experiment only evaluates the veracity of Proebsting's Law with respect to specific vendors' compiler technology. Again, this is true. The vendor compiler(s) used in this experiment may not reflect the entirety of performance enhancing compiler optimization research conducted since the invention of compilers. However, we are comfortable concluding that vendors, in order to show their systems in the most positive light, use "good" optimization technology and incorporate the best ideas from compiler optimization research into their products.

Issue 3: Extracting the best performance improvements from an optimizing compiler requires optimization options to be set very carefully. This is another true statement. We deal with this problem by gathering "optimized" performance numbers from the vendors themselves. Since we are using SPEC95 as our benchmark suite, we can look up the best, reported performance numbers for our platform. We are confident that vendors set optimization flags to achieve the highest attainable performance on the SPEC95 benchmarks.

Issue 4: Even with optimizations disabled the compiler may be using techniques to produce "good" code, e.g., code generation via dynamic programming [3]. Said another way, it may be the case that modern compilers, even with optimizations disabled, still do a better job producing good code than the earliest non-optimizing compilers. If this is true then our experiment will be at least partially biased against optimizing compilers by showing a smaller performance difference between optimized and unoptimized programs.

Our experiments were conducted on a Compaq/Digital Personal Workstation 600au (600 MHz Alpha 21164) with 320M of RAM and a 4.1G SCSI disk. The "with optimization" base and peak numbers were reported by Compaq for a nearly identical machine [4]. The "without optimization" base ratios for SPECint95 were obtained by compiling the benchmark with the vendor cc compiler (DEC C V5.8-009 on Digital UNIX V4.0 Rev. 1091)

using the compiler options “-std1 -O0”. The “without optimization” base ratios for SPECfp95 were obtained by compiling the benchmark with the GNU Fortran 77 compiler using the compiler option “-O0”. The numbers obtained are an average of three consecutive runs on an unloaded machine with all SPEC95 program and data files located on local storage.

Even though we disable optimizations by using the “-O0” compiler option, the DEC C compiler still does peephole optimization and employs a cost-driven tree pattern matching code generator [7]. Thus, even though we don’t have optimizations enabled, the compiler is still doing some non-trivial work to improve the quality of the generated target code. As we mentioned before this creates the appearance of bias in our experiments—the unoptimized numbers that we get are influenced by compiler technology that wasn’t present at the birth of compiler optimization research.

Despite this appearance of bias, we still feel that our experiments produce valid evidence for Proebsting’s Law. Why? Proebsting’s Law is a general statement about the performance improvements that we can expect from compiler optimization research over time. Peephole optimization is a very simple code improvement technique that was discussed in the literature as early as 1965 [8] and quite possibly used in practice before then. Sophisticated code generation algorithms for trees have been in use since the late 1950’s [9]. In section 4, we will show that Proebsting’s Law demonstrates a rather low rate of optimization-based performance improvement over time even when we take the mid 1960’s as the beginning of optimization research.

3. Results

3.1 Integer Intensive Benchmarks

Table 1 summarizes the results of our experiments on the integer intensive applications in SPECint95. The optimized column contains the SPEC peak numbers reported by the manufacturer for our workstation configuration. The unoptimized column contains the SPEC base numbers that we measured on our local machine with optimizations disabled. SPEC numbers are themselves ratios of measured running times to some baseline running time. Higher numbers indicate higher performance.

SPEC peak numbers may be obtained through application specific setting of optimization flags and are typically higher than SPEC base numbers. We show optimized peak numbers since they reflect the highest reported performance for our platform. The ratio between optimized and unoptimized performance averages 3.3 with a standard deviation of 0.9. This is somewhat lower than the ratio of 4 originally hypothesized by Proebsting. This lower ratio means that optimization is even less effective at improving the performance of integer intensive benchmarks than Proebsting posited.

| Benchmark | Optimized (Peak) | Unoptimized (Base) | Ratio |
|-----------|------------------|--------------------|-------|
| go | 19.5 | 5.3 | 3.7 |
| m88ksim | 21.2 | 4.2 | 5.0 |
| gcc | 15.6 | 5.5 | 2.8 |
| compress | 16.3 | 4.3 | 3.8 |
| li | 16.5 | 6.8 | 2.4 |

| | | | |
|--------|------|-----|-----|
| ijpeg | 17.7 | 6.3 | 2.8 |
| perl | 21.7 | 8.7 | 2.5 |
| vortex | 19.3 | 5.3 | 3.6 |

Table 1: SPECint95 Results. The average ratio between optimized and unoptimized performance is 3.3 with a standard deviation of 0.9.

3.2 Floating-point Intensive Benchmarks

Table 2 summarizes the results of our experiments on the floating-point intensive applications in SPECfp95. The SPECfp95 experiments yielded larger differences in performance between optimized and unoptimized code than did the SPECint95 experiments. For the programs in SPECfp95 the ratio between optimized and unoptimized performance averages 8.1 with a standard deviation of 5.4. This result confirms our intuition that optimizing compilers are more effective at improving the performance of scientific codes than they are at improving the performance of ordinary applications.

| Benchmark | Optimized (Peak) | Unoptimized (Base) | Ratio |
|-----------|------------------|--------------------|-------|
| tomcatv | 28.3 | 6.1 | 4.6 |
| swim | 36.9 | 6.3 | 5.9 |
| su2cor | 10.9 | 3.1 | 3.5 |
| hydro2d | 9.8 | 2.6 | 3.8 |
| mgrid | 18.6 | 1.1 | 16.9 |
| applu | 10.4 | 0.6 | 17.3 |
| turb3d | 24.4 | 2.7 | 9.0 |
| apsi | 25.3 | 3.7 | 6.8 |
| fp95 | 51.3 | 9.7 | 5.3 |

Table 2: SPECfp95 Results. The average ratio between optimized and unoptimized performance is 8.1 with a standard deviation of 5.4.

4. Discussion

The results of our experiment suggest that Proebsting’s Law is probably true. The reality is somewhat grimmer than Proebsting initially supposed. Research in optimizing compilers has been ongoing since 1955. The compiler technology developed over this 45-year period is able to improve the performance of integer intensive programs by a factor of 3.3. This corresponds to uniform performance improvements of about 2.8% per year. Even if we assume that the beginning of useful compiler optimization research began in the mid 1960’s [5], the uniform performance improvement on integer intensive codes due to compiler optimization is still only 3.6% per year. This lies in stark contrast to the 60% per year performance improvements we can expect from hardware due to Moore’s Law.

The performance difference between optimized and unoptimized programs is larger for the floating-point intensive codes in SPECfp95. This indicates that compiler research has had a larger effect on improving the performance of scientific codes than on improving the performance of ordinary, integer intensive applications. Again, if we assume compiler research has been ongoing since 1955, we get a doubling of performance every 16 years. This corresponds to uniform performance improvements of about 4.9% per year over this 45 year period. This is only slightly better than the results for integer intensive programs.

So what does Proebsting's Law mean for compiler researchers? Proebsting has suggested that the compiler and programming language research communities should focus less on optimization and more on programmer productivity [1]. Bill Pugh suggests that code optimization is relevant—nobody is going to forgo a speedup of 3.3 when it is available [6]. Pugh goes on to suggest that code optimization research itself isn't irrelevant, especially when optimization facilitates programmer productivity, e.g., permits efficient use of high-level language constructs.

5. References

- [1] Proebsting, T. A. Proebsting's Law: Compiler Advances Double Computing Power Every 18 Years. <http://www.research.microsoft.com/~toddp/papers/law.htm>
- [2] SPEC CPU 95 Benchmarks. <http://www.spec.org/osg/cpu95/>
- [3] Fraser, C. W., Hanson, D. R., Proebsting, T. A. Engineering a Simple, Efficient Code Generator Generator. *ACM Letters on Programming Languages and Systems* 1(3), 213-226, September 1992.
- [4] SPEC CPU 95 Report for Digital Personal Workstation 600au. <http://www.spec.org/osg/cpu95/results/res97q2/cpu95-970512-01871.html>
- [5] Aho, A. V., Sethi R., Ullman, J. D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.
- [6] Pugh, W. W. Is Code Optimization (Research) Relevant? <http://www.cs.umd.edu/~pugh/IsCodeOptimizationRelevant.pdf>
- [7] Muchnick, S. S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
- [8] McKeeman, W. M. Peephole Optimization. *Communications of the ACM* 8(7), 443-444, 1965.
- [9] Ershov, A. P. On Programming of Arithmetic Operations. *Communications of the ACM* 1(8), 3-6, August 1958.