

# Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation

Stephen E. Richardson

SMLI TR-92-1

September 1992

## Abstract:

This paper discusses **trivial** computation, where simple operands trivialize potentially complex operations. An example of a trivial operation is integer division, where the divisor is two; the division becomes a simple shift operation. The paper also discusses the concept of **redundant** computation, where some operation repeatedly does the same function because it repeatedly sees the same operands. Experiments on two separate benchmark suites, the SPEC benchmarks and the Perfect Club, find a surprisingly large amount of trivial and redundant operation. Various architectural means of exploiting this knowledge to improve computational efficiency include **detection of trivial operands** and the **result cache**. Further experimentation shows significant speedup from these techniques, as measured on three different styles of machine architecture.

This version of the technical report includes new information as of March 1993. It incorporates material included in a paper published in the 11th Symposium on Computer Arithmetic, sponsored by the IEEE.

Also included is a license agreement for receiving free software from Sun Microsystems Laboratories, Inc.

## Categories and Subject Descriptors:

B.6.1 Hardware:

[**Logic Design**]: Design Styles-*Memory used as logic*

F.2.0 Theory of Computation:

[**Analysis of Algorithms and Problem Complexity**]: General

G.0 Mathematics of Computing:

[**General**]

**Keywords and Phrases:** Memoization, Multiplication, Result cache

 **Sun Microsystems**  
**Laboratories, Inc.**

A Sun Microsystems, Inc. Business

M/S 29-01  
2550 Garcia Avenue  
Mountain View, CA 94043

**email address:**

stever@eng.sun.com

# Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation

Stephen E. Richardson (stever@sun.com)

Sun Microsystems Laboratories, Inc.

*This paper discusses **trivial** computation, where simple operands trivialize potentially complex operations. An example of a trivial operation is integer division, where the divisor is two; the division becomes a simple shift operation. The paper also discusses the concept of **redundant** computation, where some operation repeatedly does the same function because it repeatedly sees the same operands. Experiments on two separate benchmark suites, the SPEC benchmarks and the Perfect Club, find a surprisingly large amount of trivial and redundant operation. Various architectural means of exploiting this knowledge to improve computational efficiency include **detection of trivial operands** and the **result cache**. Further experimentation shows significant speedup from these techniques, as measured on three different styles of machine architecture. The paper includes a license for free software.*

## 1 Introduction

Computing machines execute tens of millions of operations every second. Consequently, each individual operation need not be complex. In fact, it should not be surprising that much computation consists of highly redundant sequences of simple instructions, and that many of these instructions perform trivial operations, such as multiplication by zero.

This paper explores the trivial and redundant nature of computing. The paper naturally divides in two sections. The first section explores the degree of triviality in computation, focusing on long-latency arithmetic operations, and proposes a means for exploiting this triviality to increase execution speed. The second section discusses the redundant side of computation and, building on the results of the earlier section, attempts to derive further benefit.

Experimental data shows significant speedup for each of three styles of machine architecture.

---

This paper is a synthesis of the original SMLI technical report SMLI-TR-1 and a later article, *Exploiting Trivial and Redundant Computation*, published in *Proceedings of the 11th Symposium on Computer Arithmetic*, edited by Swartzlander, Irwin, and Jullien, pages 220–227. The symposium was sponsored by the IEEE Computer Society Technical Committee on VLSI and took place in Windsor, Ontario, June 29–July 2, 1993

## 2 The trivial nature of computation

### What is trivial computation?

Complex operations such as multiplication and division of fixed-width binary numbers involves a significant amount of computation, such as adds, shifts, and combinatorial logic. However, certain operands that might be presented to the operation can obviate much of this computation, thus trivializing the operation. Attempts to exploit this phenomenon often involve such techniques as counting the leading zeroes of an operand. An eight- or sixteen-bit integer divide, for instance, would take less time to complete than a full 32-bit division.

This paper uses a much stricter definition for triviality, searching for operations so simple that they could complete in one cycle on even the simplest of machines. Figure 1 displays more precisely the conditions for triviality.

operation	conditions for triviality
multiply $x \times y$	$(x \text{ or } y) = (0, 1, \text{ or } -1)$
division $x \div y$	$(x = y, x = -y, \text{ or } x = 0)$
square root $\sqrt{x}$	$(x = 0 \text{ or } x = 1)$

Figure 1: Conditions for triviality.

### Why is computation sometimes trivial?

For generality, a scientific algorithm might be designed using three-dimensional rectangular coordinates, although a large class of interesting problems may be two-dimensional. For this class of problems, approximately one-third of the computation (that concerning the z-component) will turn out to be operations on zero. For instance, rectangular-to-spherical coordinate transformation uses the formula

$$r = \sqrt{x^2 + y^2 + z^2}.$$

For two-dimensional problems, the equation becomes

$$r = \sqrt{x^2 + y^2 + 0.0^2}$$

Heat transfer problems may make heavy use of the equation for specific heat capacity  $\Delta Q = cm \Delta T$ , where  $\Delta Q$  is a quantity of heat that, applied to a substance of mass  $m$  and heat capacity  $c$ , changes its temperature by an amount  $\Delta T$ .

The equation is often set up such that for the most interesting substance, water, the value of  $c$  is 1.0. A program for heat transfer, used to calculate cooling by water, would thus wind up doing a fair amount of multiplication by 1.0.

Is it possible that a significant amount of computation involves complex operations on trivial data? What about non-scientific programs? Take the example of a typesetting algorithm. To justify its margins, such a program must calculate the width of each word within a line. Involved in this computation might be the width of each character in per-point units, its point size, and a magnification factor:

$$\text{charwidth} \times \text{pointsize} \times \text{magnificationfactor}$$

Typically, the magnification factor will be 1.0 or 2.0, resulting in a significant amount of trivial computation.

### How can trivial computation speed program execution?

If a sufficient amount of computation were indeed trivial, some obvious changes in the style of computation could make programs run faster. Consider the following algorithm for multiplying two operands  $a$  and  $b$  to yield a result  $c$ :

```
OVERHEAD: if (|a| == 1.0 or b == 0.0) then
            c = sign(a) × b;
            else if (|b| == 1.0 or a == 0.0) then
            c = sign(b) × a;
            else
            goto MULTIPLY;
            goto END;
MULTIPLY:  c = mult(a, b);
END:
```

A *trivial* multiply—multiplication by 1.0, 0.0, or -1.0—will exit after passing through only the OVERHEAD portion of the algorithm. All other multiply operations will have the extra cost of the OVERHEAD portion added to the regular MULTIPLY portion of the algorithm. Because the conditions for triviality are so specific, a scheme for detecting them can add efficiency to generic “early-out” schemes requiring a “count-leading-zeroes” and/or a “count-leading-ones” type of operation.

Provided that the OVERHEAD cost is smaller than the MULTIPLY cost, a sufficiently large ratio of trivial multiply operations to nontrivial multiply operations will justify the cost of adding the OVERHEAD.

Although not new, the idea of exploiting trivial computation has not seen wide dissemination, due in large part to a lack of knowledge concerning its usefulness in typical programs. This paper presents real data on the degree to which real programs contain trivial computation, and the potential benefit to be derived by current processors.

### How much trivial computation in real programs?

A tool called *shade* [Cmelik92] can help determine the ratio of trivial to nontrivial operations in some benchmarks of current interest. Shade analyzes programs on an instruction-by-instruction basis as they execute. Each time shade sees a targeted operation, it can note whether the operands render the operation trivial. The table in Figure 1 shows the target operations, along with the conditions for triviality. The experiment will *not* detect cases where one or more of the operands is constant; the compiler optimizes these away, as shown in Figure 2.

The data to be presented comes from two different benchmark suites. The first, known as the SPEC floating-point benchmark suite, is a group of large FORTRAN and C programs. The second, called the Perfect Club, consists of a set of statically large and dynamically very large numeric FORTRAN programs. Appendices A and B provide a more complete description of the SPEC and Perfect Club benchmarks. The benchmarks NA, SM, and TF were omitted from the Perfect Club results because of a difficulty in attaining accurate results.

```

Y = X * 1.0
DO 200 J=0,200
  A(J) = X * J
200 CONTINUE

```

Not a trivial multiply;  
compiler will optimize to "Y = X."

Trivial when J==0 or J==1.

Figure 2: Trivial multiplication in FORTRAN.

Figure 3 shows, for each of the SPEC benchmarks, what percentage of targeted operations were found by the shade analyzer to be trivial. Figure 4 shows data for the Perfect Club benchmarks.

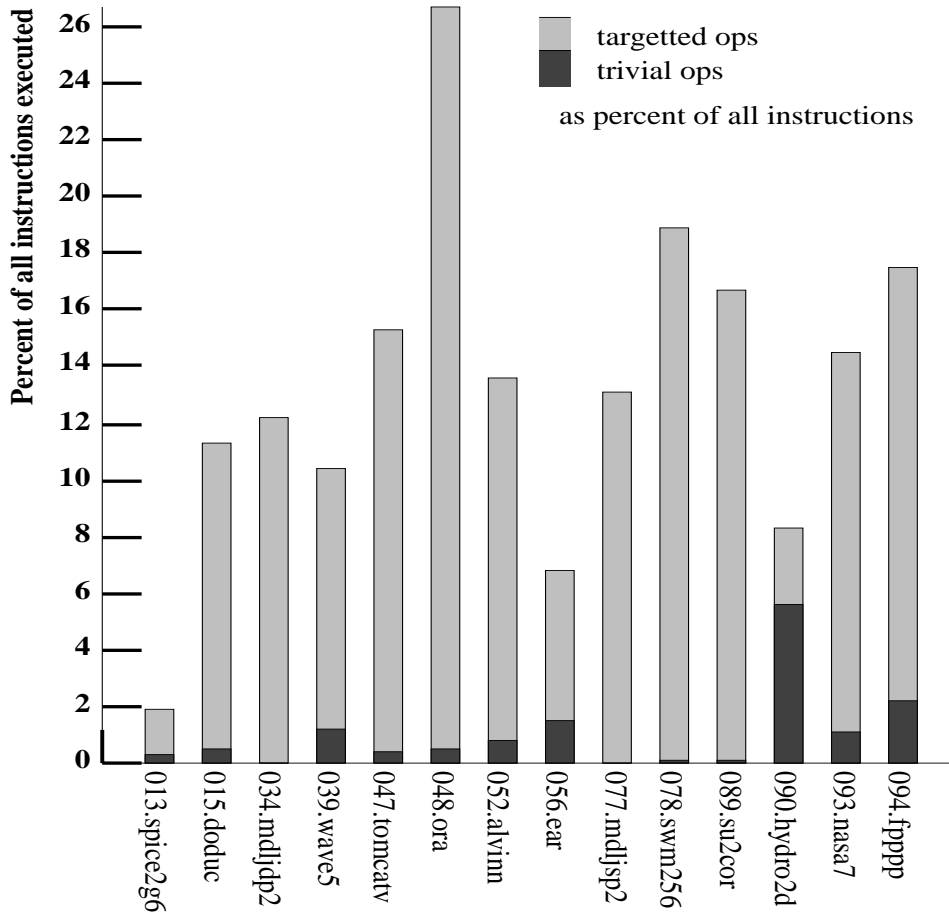


Figure 3: Trivial operations in SPEC benchmarks.

The percentage of trivial operations per program ranged from near zero to as high as 7.3 percent for the Perfect Club benchmark "SD." The relatively large percentage of trivial operations in SD results from repetitive matrix multiplication of sparse arrays such as diagonal transformation

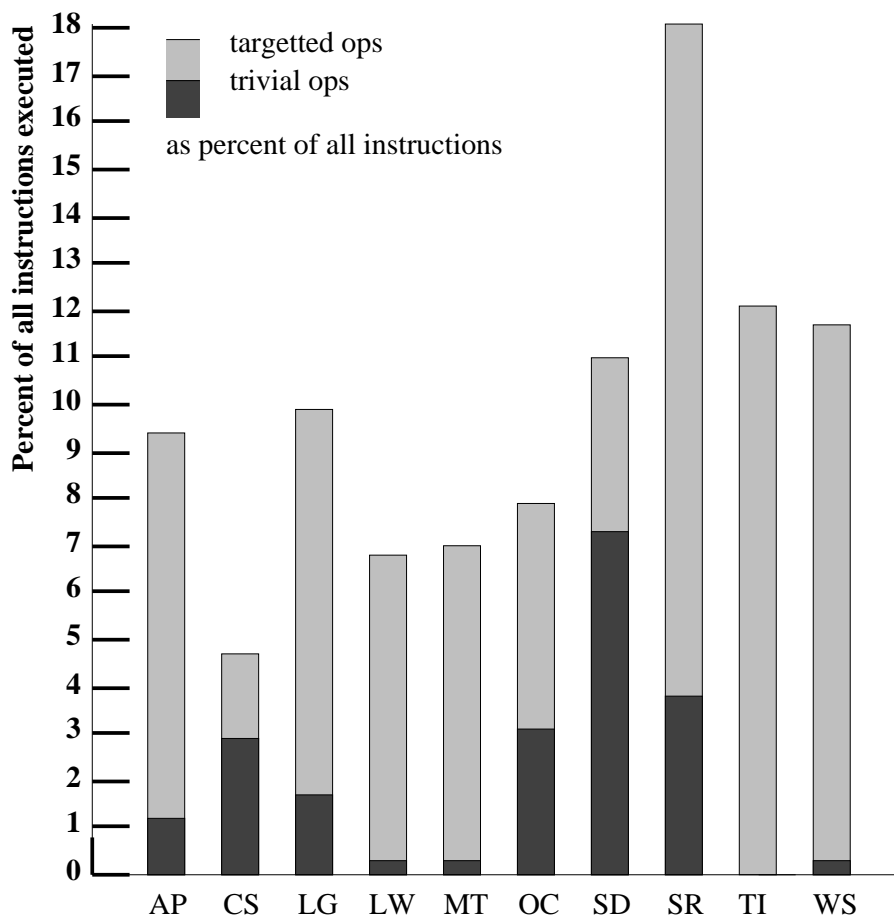


Figure 4: Trivial operations in Perfect Club benchmarks.

matrices. By far, most of the trivial operations were single- and double-precision floating-point multiply operations.

The speedup achievable from detecting trivial operations can vary, depending on the cost of each operation in a given architecture. The next section explores this cost versus speedup for various machine styles.

### How much speedup?

The table in Figure 5 gives sample times for certain long-latency operations on various implementations. Most of the numbers were derived from data books and other literature [Grohoski90, Olsson90, Motorola88, NEC91, Intel88, Intel89]. Experimental data provided numbers for the SPARCStation 2 (SS2) and the HP9000/720. The SPARCStation 2 contained a Cypress CYC602 integer unit and a Texas Instruments TMS390C602A floating-point unit. Numbers in the table do not reflect anomalously long latencies; for example, the HP machine required over three hundred cycles to compute single- and double-precision floating-point divides of the form  $0 \div x$ .

Now posit a set of three test machines: the Aggressor, an aggressive design with latencies for the targeted operations comparable to the shortest of those in Figure 5; the Normol, with somewhat intermediate values for the latencies; and the Wemp, a cost-effective machine with very long-

	integer		floating-point					
	mul	div	multiply		divide		square-root	
			sp	dp	sp	dp	sp	dp
RS/6000	5	19	2	2	17	17	*	*
HP720	12	20	3	3	10	12	120	120
SS2	20	30	4	6	16	26	26	40
MC88100	4	38	6	7	30	59	*	*
VR4000	10	69	7	8	23	36	54	112
i486	13	24	14	14	73	73	85	85
80960KB	18	37	20	36	35	77	104	104

Figure 5: Cycle times for long-latency operations, on various systems.

latency operations. The table in Figure 6 gives characteristics for each machine. Assume that non-targeted operations have no excess latency and execute at the rate of one per cycle.

	integer		floating-point					
	mul	div	multiply		divide		square-root	
			sp	dp	sp	dp	sp	dp
Aggressor	4	18	2	2	10	10	25	40
Normol	10	24	4	5	15	20	50	80
Wemp	20	70	20	40	75	80	120	120

Figure 6: Cycle times for long-latency operations, on test machines

The table in Figure 7 shows the overall performance improvement for each test machine resulting from a hardware implementation of a trivial-operand detect scheme. Each performance improvement number represents the geometric average of the improvement of the individual benchmarks in the set. The table assumes that detection of trivial operands, and the subsequent emission of the appropriate result, is a simple operation that should take no more than a single cycle on even the crudest of implementations. As one might expect, the long latency machine “Wemp” showed greatest improvement: 10.4 percent on the SPEC benchmark set and 22.0 percent on the Perfect Club. Even the short-latency Aggressor benefitted, although to a lesser degree: 2.1 percent and 4.4 percent, respectively, for the two sets of benchmarks.

A subsequent study of operand distributions shows certain missed opportunities [Richardson93a]. Specifically, the study shows a significant number of floating-point divisions of the form  $x \div 1.0$  and  $x \div 2.0$ . Aside from a few trivial operands such as 1 and 0, however, the most common oper-

Machine	Benchmark Suite	
	Spec92	Perfect Club
Aggressor	2.1%	4.4%
Normol	4.0%	8.0%
Wemp	10.4%	22.0%

Figure 7: Geometric average of overall program speedup as a result of trivial-operation detect.

ands for a given operation tend to differ from program to program. For example, the most common multiplicative operands in the SPEC benchmark *alvinn* are 0.99 and 0.01, respectively.

Now that we have seen some of the benefit to be derived from recognizing the trivial nature of computation, let's turn our attention to the concept of redundant computation. To what degree is computation redundant? Can we use the redundant nature of computation to gain additional speedup?

### 3 The redundant nature of computation

Computation typically involves the input of an initial data set, the transformation of these data through one or more states, and convergence on a final data output. Sometimes the data mimics physical quantities, such as time, distance, or voltage; other times the data consists of more abstract items, such as lexical tokens or character strings.

Such input data are by nature redundant. Take the example of a simulator for CMOS VLSI circuits, or a compiler for FORTRAN. Think how many nodes in the circuit will begin at either 0.0 or 5.0 volts. Think how often the compiler will process the keywords "FOR" or "CONTINUE," or the identifier name "I," as compared to the identifier name "XYZ123."

Similar data tends to flow through similar states. Data read as "inches" and "seconds" may be converted, one datum at a time, to "centimeters" and "hours." This involves redundant multiplication by the same conversion constants. Programs often run multiple times with the same or very similar inputs, such as the typesetter that runs over and over on a progressively refined document.

Acknowledgment of this redundant nature can speed the task of computation in many ways. Cache memory, for instance, works so well because the same areas of memory get accessed over and over during a sufficiently short time period. As another example, incremental compilation takes advantage of the fact that programs in development seldom vary much from one run to the next [Quong91, Burke90].

#### 3.1 Memoization

The technique of *memoization*, or tabulation, takes advantage of the redundant nature of computation. It allows a computer program to run faster by trading execution time for increased memory storage. Once calculated, the result of a function is stored in a table called a memoization cache. The cache traditionally exists as a software data structure. Cache lookup then replaces later calls to the function [Bentley82, Harbison82, Abelson85, Hughes85]. Tabulation can be extended to



apply not only to functions, but also statements, groups of statements, or any given region of a program that has limited side effects and a high degree of recurrence.

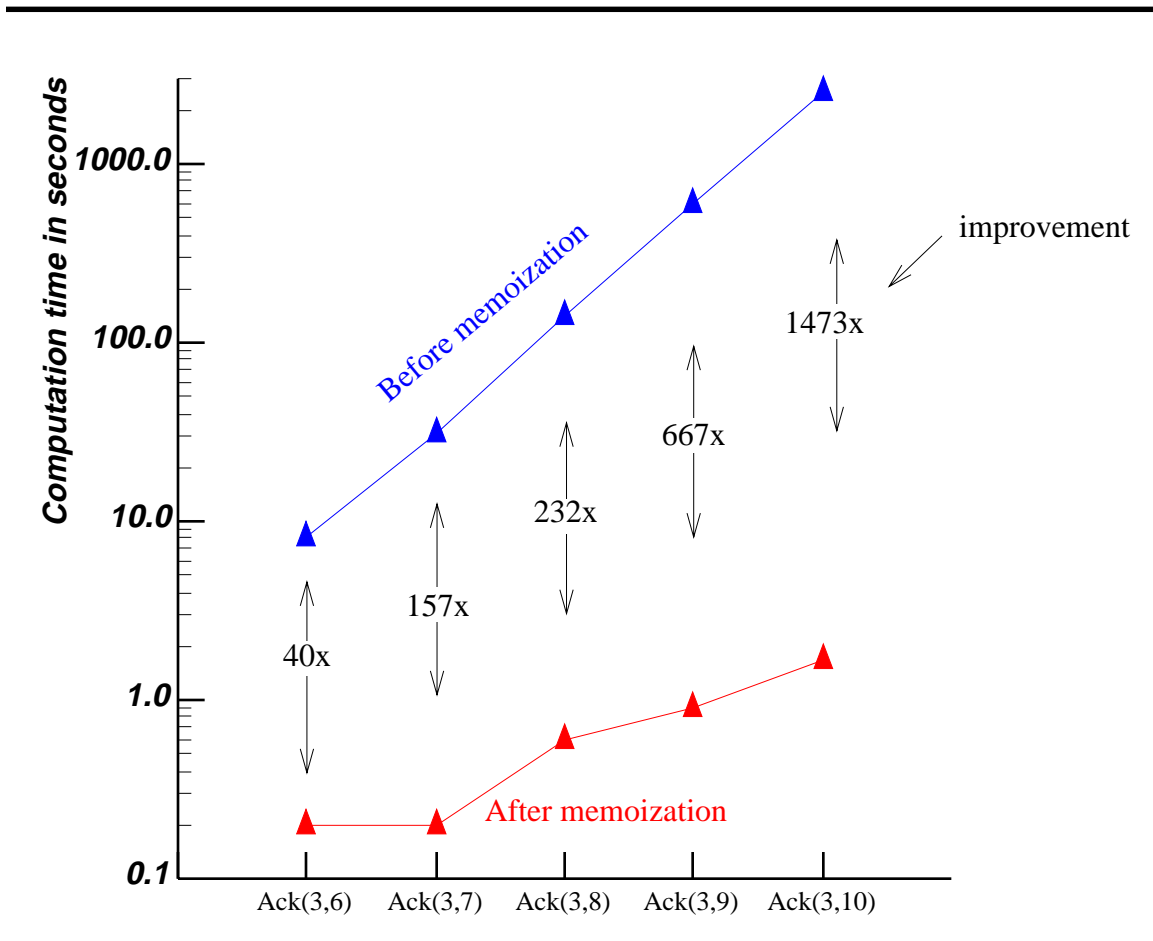


Figure 8: Optimizing Ackerman's function using memoization.

### Memoization by Region: Ackerman's Function

Figure 8 shows the dramatic improvement possible with memoization in a functional language program. By caching calls to the recursive Ackerman's function, and replacing subsequent calls with table lookup, we achieve real speedup as much as 1,473 times. Appendix C shows the modified Ackerman's function. Can the same kind of modification achieve speedup in a real, non-functional-language type program?

### A SPECmark: doduc

The table of Figure 9 summarizes actual runtime improvement found by applying memoization to the SPECmark "doduc." This experiment cached all calls to the function EWV, which in turn includes two calls to function SI and one to EXP. The table presents results for various configurations of the lookup cache. In this experiment, we look only at direct-mapped memoization caches having from 64 to 32,768 entries. A hash on the EWV function parameters formed an appropriate index into the cache. The unhashed parameters are then used as the "tags."

As shown in the partial doduc profile of Figure 10, each call to EWV (which, as noted earlier, calls SI twice and EXP once) results in the execution of  $(149 \times 2 + 74 + 43) = 415$  instructions.

Cache size	hits	misses	unopt	opt	improvement
64x1	45200	100900	49.64 sec	49.00 sec	1.3%
256x1	58300	87800	48.56 sec	47.76 sec	1.7%
1024x1	63100	82900	49.68 sec	49.40 sec	0.6%
4096x1	64400	81600	51.56 sec	48.50 sec	6.3%
16384x1	64800	81200	47.48 sec	46.60 sec	1.9%
Average			49.59 sec	48.41 sec	2.8%

Figure 9: Measured memo improvement for doduc.

**Before:**

function	calls	instrs/call	instrs	%
<b>SI</b>	<b>292,332</b>	<b>149</b>	<b>43,557,468</b>	<b>14.75</b>
POW	78,505	193	15,151,465	5.13
<b>EXP</b>	<b>146,166</b>	<b>74</b>	<b>10,816,284</b>	<b>3.68</b>
SQRT	87,213	120	10,465,560	3.53
<b>EWV</b>	<b>146,166</b>	<b>43</b>	<b>6,285,138</b>	<b>2.12</b>
(Total)			295,304,867	100.00

**After:**

function	calls	instrs/call	instrs	%
<b>SI</b>	<b>162,480</b>	<b>151</b>	<b>24,534,480</b>	<b>8.75</b>
POW	78,505	193	15,151,465	5.40
SQRT	87,213	120	10,465,560	3.73
<b>EXP</b>	<b>81,240</b>	<b>74</b>	<b>6,011,760</b>	<b>2.14</b>
<b>CS1</b>	<b>146,166</b>	<b>30</b>	<b>4,384,980</b>	<b>1.56</b>
<b>EWV</b>	<b>81,240</b>	<b>43</b>	<b>3,493,320</b>	<b>1.24</b>
<b>CI1</b>	<b>81,240</b>	<b>17</b>	<b>1,381,080</b>	<b>0.49</b>
(Total)			280,394,057	100.00

Items in **blue boldface** directly relate to calls to EWV. Other items are included as a matter of interest.

Figure 10: Execution profiles for doduc, before and after memoization.

Memoization calls CS1 before each call to EWV. CS1 searches the cache. A cache hit avoids the call to EWV. If the cache misses, we must call EWV and then call CI1 to insert the new value into the cache.

Inspection of the *after* profile shows

$$\begin{aligned} \text{Cost of hit} &= 30 \text{ instructions} \\ \text{Cost of miss} &= (30 + 415 + 17) = 462 \text{ instructions.} \end{aligned}$$

Further, the profile shows that this particular implementation hit  $(146,166 - 81,240) = 64,926$  times and missed 81,240 times. Instruction overhead from EWV was thus reduced

$$\begin{aligned} \text{before:} & 146,166 \times 30 &= 60,658,890 \\ \text{after:} & (64,926 \times 30) + (81,240 \times 462) &= 39,480,660 \end{aligned}$$

by 35 percent, and overall instruction count reduced

$$1 - \left( \frac{295,304,867}{280,394,057} \right) = 5 \text{ percent.}$$

If cache penalty could be made smaller, perhaps by implementing the cache in hardware, speedup would approach

$$(\text{original overhead}) \times \frac{\text{hits}}{\text{hits} \times \text{misses}} = (20\%) \times \frac{64,926}{146,166} = 8.8\%$$

To summarize, performing memoization on a single procedure (EWV) in *doduc* can potentially yield near 8.8 percent reduction in dynamic instruction count. A simple software scheme yields about 5 percent. As a final exercise, note that a 50 percent hit rate on the functions POW and Sqrt would yield further reduction of as much as 5 percent more.

### Tomcatv

Profiling reveals that each of the following computations represents approximately 10 percent of dynamic instructions executed for the SPECmark *tomcatv*.

$$\begin{aligned} A &= 0.25 \times (XY^2 + YY^2) \\ B &= 0.25 \times (XY^2 + YY^2) \end{aligned}$$

Value tracing shows that, of 65,031 computations, only 769 unique operand pairs are presented to each expression. Thus, a perfect cache of sufficient size with no overhead should reduce the dynamic instruction count by approximately 20 percent.

### Other SPECmarks

Other SPECmarks were not scrutinized as closely, but none seem so obviously amenable to memoization as *doduc* and *tomcatv*.

### How to memoize regions automatically

Efficient application of this technique involves these steps:

- Find a much-used statement of the program, using profile or other technique;
- find largest enclosing region having limited side-effects;
- use value tracing to verify that significant recurrence actually does occur.

As an example of this procedure, inspection of Figure 10 showed that the most time-consuming function in *doduc* was SI. A compiler with access to a callgraph would find that SI is called only by procedure EWV. Thus, we cache EWV.

## 3.2 Result Caching

A special hardware cache could perform tabulation without the need for compiler or programmer intervention. Access to this *result cache* could be initiated at the same time as, for instance, a floating point divide operation. If the cache access results in a hit, the answer appears quickly and the floating point operation can be halted. On a miss, the divide unit can write the result into the cache at the same time as it sends the result on to the next pipeline stage.

In the experiments described here, we look at direct-mapped result caches for the set of targeted operations described earlier in Section 2. As before, benchmarks from the SPEC floating-point and Perfect Club suites form the test case. A filter detects and handles trivial operations, sending

only non-trivial operations on to the result cache. Appendix C provides further details concerning the experimental setup.

Figures 11 and 12 show the percentage of all instructions captured by each of a variety of result caches. The bottom bar for each benchmark tells what percentage of instructions were trivial tar-

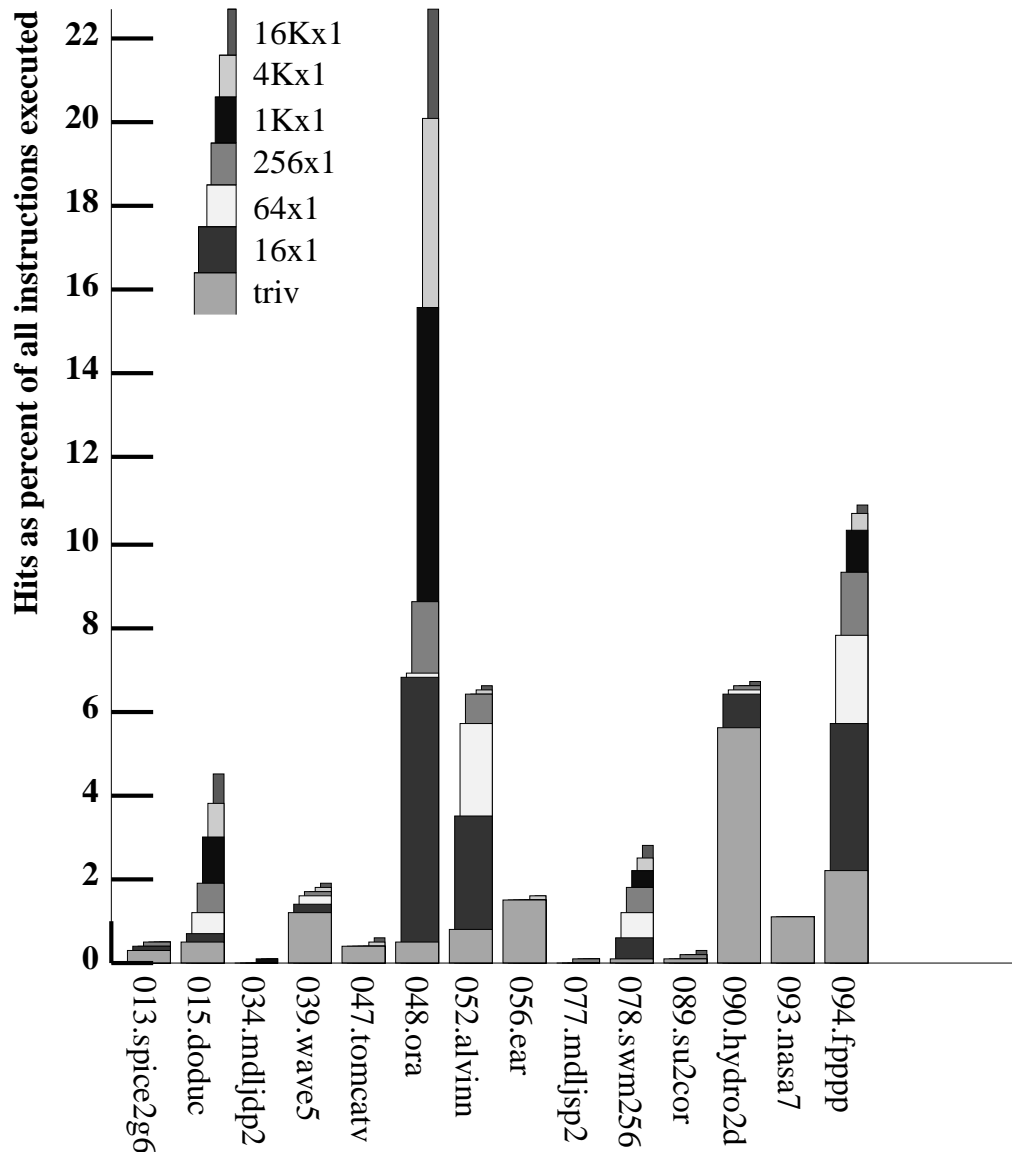


Figure 11: Percent of hits in result cache—SPEC.

geted operations. This portion of the graph represents the same information we saw earlier in Section 2. Successively taller bars show the number of instructions that hit in successively larger direct-mapped result caches. In the graph, “256x1” means the cache contains 256 direct-mapped entries.

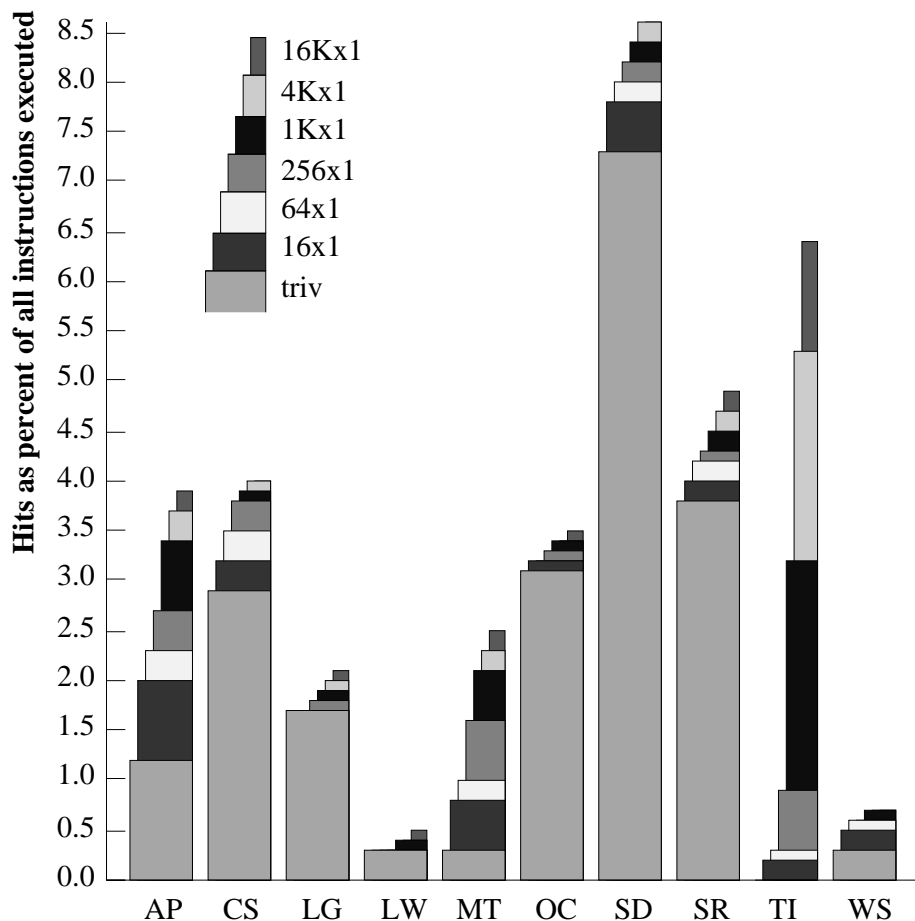


Figure 12: Percent of hits in result cache—Perfect Club.

Figure 11, for instance, shows that of all instructions executed by the SPEC benchmark *048.ora*, 0.5 percent were trivial targeted operations—that is, trivial multiplies, divides, and square roots as defined in the table of Figure 1. An additional 6.3 percent of all instructions executed were targeted operations that would hit in a direct-mapped sixteen-entry result cache, for a cumulative total of 6.8 percent. Going from sixteen to sixty-four entries captures another 0.1 percent, for a total of 6.9 percent. And a 16K cache with trivial-operand detect effectively removes the latency from 22.7 percent of all instructions executed—a significant accomplishment, considering that targeted operations comprise only 26.7 percent of all instructions.

The SPEC benchmarks in Figure 8 show a wide range of hit rates, from near zero for the *mdlj* programs to over 20 percent for the floating-point intensive *ora*. The Perfect Club benchmarks in Figure 12 show similar variation over a smaller range, from less than one percent for *LW* and *WS* to over seven percent for *SD*. Note that *TI*, while not gaining any advantage from trivial operations, responds well to the result cache.

While the detection of trivial operands seemed primarily to benefit multiplication, the result cache also captures a fair number of divides and square roots. In the application *048.ora*, for instance, the largest result cache captured 81.9 percent of all double-precision square root operations. As seen earlier in Figure 5, this advantage gets multiplied by 40x to 120x, depending on the imple-

mentation of the square root function. This, along with the other operations captured, results in enormous speedup.

Figure 13 gives geometric means for whole-suite improvements on each of the Aggressor, Normol, and Wemp test machines. Speedup corresponds to reduction in program run time, ignoring

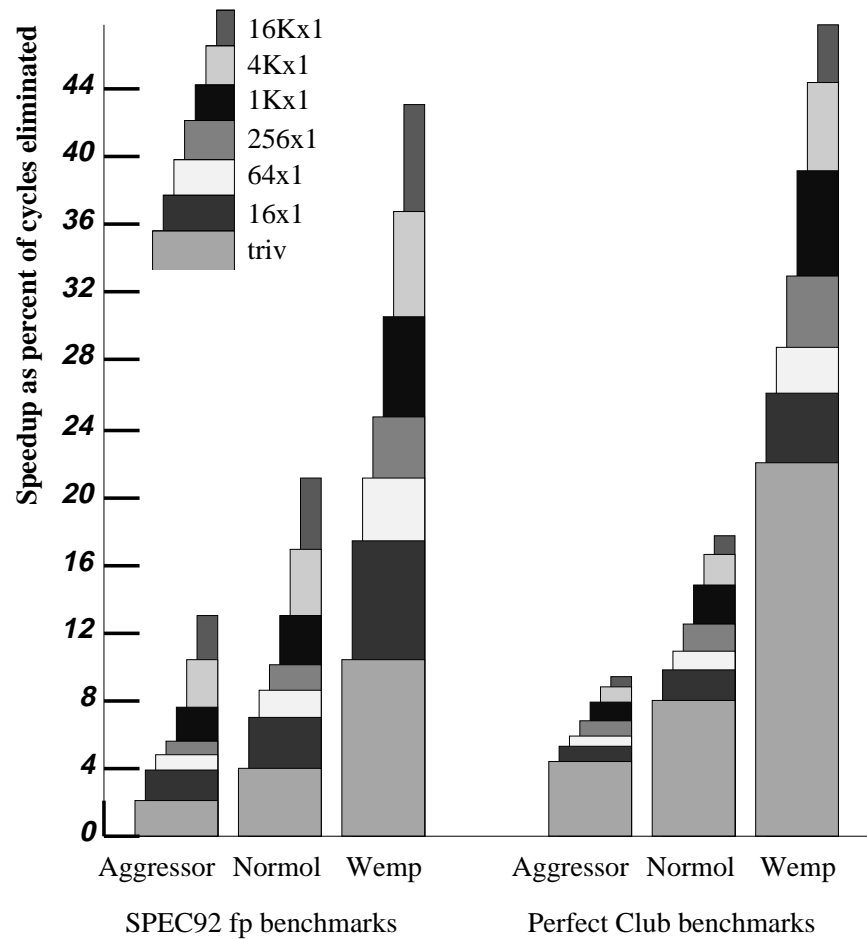


Figure 13: Machine speedups using result cache.

memory and system effects. Improvements from a given cache show remarkable similarity across benchmark suites. With caches ranging from 16 to 16K entries, the Aggressor achieved a 4 to 13 percent speedup, the Normol got 7 to 21 percent, and the Wemp a 17 to 48 percent speedup. The chart shows a fairly constant improvement of about 2x with each 4x increase in cache size, indicating that a *knee* has not yet developed; still larger caches would probably get still more improvement.

## 4 Conclusions

Experiments indicate a high percentage of *trivial* operations. Algorithms for complex arithmetic functions should always provide an early-out for such cases. For certain programs studied, trivial

operations accounted for as much as 67 percent of targeted operations. Fast evaluation of these operations yielded significant speedup in execution time, as seen in Figure 7.

Figure 13 showed that memoization of individual instructions via result cache provides further benefit, yielding more and more speedup as the result cache size increases.

Both schemes showed best results in floating-point-intensive programs, probably because most of the targeted operations were long-latency floating-point functions. Obviously, any long-latency instruction could become a candidate for speedup using these shortcut techniques.

The simplicity of the techniques presented make them particularly attractive alternatives to expensive complex-operation support in low-cost designs. The long-latency *Wemp* composite machine showed speedups of up to 43.1 percent on the set of SPEC benchmarks studied, and 47.8 percent on the Perfect Club set. Machines with shorter latency also benefitted, improving by ten to twenty percent.

## 5 Future work

The conditions for triviality captured few divide or square root operations. Closer observation of these functions might reveal a high frequency of some simple operand, such as divide-by-two, that has not yet been considered.

Different hashing algorithms for producing an index given an operand or pair of operands might raise the hit rate of the result cache. Furthermore, a cache associativity greater than one might prove useful.

A persistent result cache would exhibit “warm-start” characteristics across successive iterations of the same program. To what extent would this improve speedup?

The result cache presented here targeted only multiply, divide and square root operations. The scope could be expanded to contain others. Furthermore, the cache could support general memoization through use of specialized “check-result-cache” instructions.

Finally, means could be explored for gaining optimum hit rate per area of result cache. Such means might include data compression or limiting the type, size, or precision of operand considered for inclusion in the cache.

## Acknowledgments

Shade, a comprehensive instruction-level simulator written by Bob Cmelik, greatly facilitated the experimental results described in this paper with a minimum of user effort. I also thank my colleagues Mark Santoro and Dave Patterson for their encouragement and comments on early drafts of the paper. Thanks also go to reviewer E for challenging me to do useful studies that I did not really want to do. Finally, I thank Dave Ditzel for providing the resources necessary to do scientific research in an industrial environment.

## Appendix A: SPEC benchmarks

The SPEC (Systems Performance Evaluation Cooperative) benchmarks consist of twenty CPU-intensive programs variously written in FORTRAN and C [Weiss90]. The suite is broken up into six integer benchmarks and fourteen floating-point benchmarks. Because this paper focussed mostly on long-latency floating-point operations, it used only the floating-point portion of the suite.

Benchmark	Description
013.spice	famous circuit simulation program;
015.doduc	Monte Carlo simulation of a portion of a nuclear reactor;
034.mdljdp2	simulates the interaction of 500 atoms;
039.wave5	simulation of particles in a plasma;
047.tomcatv	vectorized version of a mesh generation program;
048.ora	traces rays through spheres and planes;
052.alvinn	trains a neural network to drive a vehicle;
056.ear	simulates sound in the human cochlea;
077.mdljsp2	single-precision version of mdljdp2;
078.swm256	solves a system of shallow water equations;
089.su2cor	computes masses of elementary particles;
090.hydro2d	uses Navier Stokes equations to compute galactic jets;
093.nasa7	heavily floating-point-intensive FORTRAN kernels;
094.fpppp	computes a “two electron integral derivative” for a given number of atoms.

## Appendix B: Perfect Club Benchmarks

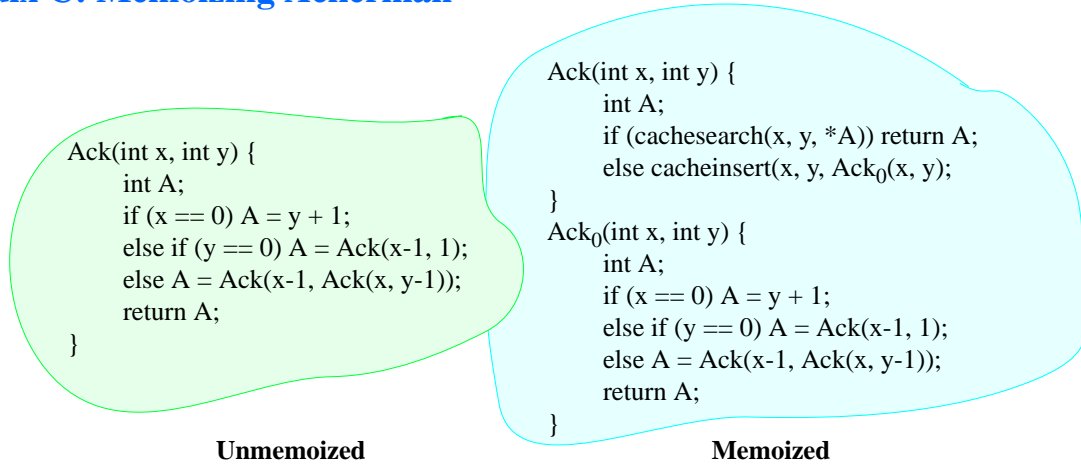
The Perfect Club is a set of computationally-intense, highly numeric FORTRAN programs for benchmarking scientific computers [Cybenko90]. Each of the thirteen programs is designated by a unique combination of two alphabetic characters. The benchmarks are described below. The benchmarks average about 129,000 characters of FORTRAN source code each.

Name	Description	Size in Bytes	
		Source	Input Data
AP	A mesoscale model for air pollution.	204307	9639
CS	The well-known circuit simulator <i>spice</i> .	579496	5292
LG	Simulation of the gauge theory of the strong interaction that binds quarks and gluons into hadrons.	64380	67
LW	A molecular dynamics program for the simulation of liquid water.	38796	50060
MT	Determines the course of a set of an unknown number of targets, such as missiles or rocket boosters.	128753	137408
NA <sup>1</sup>	A molecular dynamics package for the simulation of nucleic acids.	125540	2495739
OC	A two-dimensional ocean simulation.	134296	0
SD	A structural dynamics benchmark, solves for displacements and stresses, along with velocities and accelerations at each time step.	259473	3501
SM <sup>1</sup>	A seismic migration code used to investigate the geological structure of the earth.	79363	76551
SR	A two dimensional fluid flow solver.	131577	184
TF <sup>1</sup>	Analysis of a transonic inviscid flow past an airfoil.	61815	2013
TI	A kernel simulating a two-electron integral transformation.	10852	0
WS	A global spectral model to simulate atmospheric flow.	122084	4794767

<sup>1</sup> The benchmarks NA, SM, and TF were omitted from this paper because of a difficulty in attaining accurate results.



## Appendix C: Memoizing Ackerman



## Appendix D: Details of Result-Cache Experiment

We simulated only direct-mapped caches. More complex strategies, such as multiple set-associative, are of course possible. Discussion of the algorithms used to produce cache indices will use the following symbols for bit-level operations:

$\otimes$  = xor                      + = or                      >> = right-shift  
 $\cdot$  = and                           $\neg$  = not

For simplicity's sake, the algorithm converts all operands to double-precision before generating a cache index. A more time- and space-efficient algorithm would calculate a separate hash function depending on the operand type. The sign, exponent, and the most significant 20 bits of each double-precision operand  $x_0$  and  $y_0$  were combined using an exclusive-or operation; the two resulting numbers  $x_2$  and  $y_2$  were then exclusive-or'ed together. The appropriately-masked result of this operation formed the cache index  $i$ . For the unary square-root operation, the algorithm always set the unused operand to 0.0.

$$\text{hash}(x_2, y_2, i, m) = (x_2 \otimes y_2) \cdot m, \forall i \in \{4, 6, 8, 10, 12, 14\}$$

$$x_2 = (x_1 \gg 31) \otimes (x_1 \gg 20) \otimes (x_1 \gg (20 - i))$$

$$x_1 = (x_1 \gg 32) \cdot \text{FFFF FFFF}_{16}$$

$i$	cache size = $2^i$	mask $m = 2^i - 1$
4	16	F <sub>16</sub>
6	64	3F <sub>16</sub>
8	256	FF <sub>16</sub>
10	1,024	3FF <sub>16</sub>
12	4,096	FFF <sub>16</sub>
14	16,384	3FFF <sub>16</sub>

Each cache line included the two 64-bit operands as tags, as well as the 64-bit result and an 8-bit field to designate the operation. Again, the experiment used this space-expensive layout for simplicity only. Single-precision or unary operations do not need so much room in the cache line.

## References

- [Abelson85] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts, and McGraw-Hill Book Company, New York, New York, 1985.
- [Bentley82] Jon Louis Bentley. *Writing Efficient Programs*. Prentice-Hall, Englewood Cliffs, New Jersey, 1982. Memoization is described in Appendix C as “Space-For-Time Rule 2.”
- [Burke90] Michael Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):341–395, July 1990.
- [Cmelik92] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. Unpublished as yet.
- [Cybenko90] George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputer performance evaluation and the perfect benchmarks. In *International Conference on Supercomputing*, pages 254–266, June 1990. Also in ACM SIGARCH Computer Architecture News (CAN) Volume 18, Number 3 September 1990.
- [Grohoski90] G. F. Grohoski, J. A. Kahle, L. E. Thatcher, and C. R. Moore. *Branch and Fixed-Point Execution Units*, pages 24–33. IBM, Austin, Texas, 1990.
- [Harbison82] Samuel P. Harbison. An architectural alternative to optimizing compilers. In *Proc. Sym. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 10, 2 of *Computer Architecture News* pages 57–65, Palo Alto, California, March 1982. ACM.
- [Hughes85] R. J. M. Hughes. Lazy memo functions. In Jouannaud, editor, *Proc. IFIP Conference on Functional Programming Languages and Computer Architecture, Nancy*, pages 129–146. Springer-Verlag, September 1985.
- [Intel88] Intel Corp., Santa Clara, California. *Intel 80960KB Programmer’s Reference Manual*, 1988.
- [Intel89] Intel Corp., Santa Clara, California. *i486 Microprocessor*, April 1989.
- [Motorola88] Motorola Inc. *MC88100 RISC User’s Manual*, 1988.
- [NEC91] NEC, Mountain View, California. *VR4000 Microprocessor User’s Manual*, 1991.
- [Olsson90] Brett Olsson, Robert Montoye, Peter Markstein, and MyHong Nguyen Phu. *RISC System/6000 Floating-Point Unit*, pages 34–43. IBM, Austin, TX, 1990.
- [Quong91] Russell W. Quong and Mark A. Linton. Linking programs incrementally. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):1–20, January 1991.
- [Richardson92] Stephen E. Richardson. Caching function results: Faster arithmetic by avoiding unnecessary computation. Technical Report SMLI TR-92-1, Sun Microsystems Laboratories, Inc., September 1992.
- [Richardson93] Stephen E. Richardson. Smarter analysis of operand distributions. Unpublished documents SMLI 93-0114 and 93-0126, March 1993.
- [Weiss90] Ray Weiss and Stan Baker. Spec adds benchmark for multiple processors. *EE Times*, April 1990.

## Shade: Free software

Sun Microsystems, Inc. ("Sun") has developed a set of software tools and related documents referred to as the SPARC Performance Analysis Tools ("the Software").

The Software allows an existing SPARC program to be monitored to collect dynamic instruction characteristics. The Software can be used to collect standard instruction-level performance analysis tools (such as cache or pipeline simulators). The method used provides significant speedups over traditional trace and simulation based approaches.

The Software is experimental in nature and is provided as is with no warranties of any kind, expressed or implied, including without limitation, the warranties of merchantability or fitness for a particular purpose.

Sun is willing to license the Software in binary and limited source form solely for the Licensee's internal use. This Software may not be re-distributed by the Licensee in either source or binary form.

In no event will Sun be liable for any indirect, incidental, special or consequential damages including, without limitation, loss of revenues, profits, data or use incurred by the Licensee or any third party, even if Sun has been advised of the possibility of such damages. Sun's maximum liability for damages shall not exceed the amount of fees paid by the Licensee for the Software.

This Software is not supported by Sun, except that Sun may answer limited inquiries and may, in its sole discretion, respond to requests for bug fixes. The Sun contact for such inquiries or requests is Bob Cmelik.

To receive a copy of the Software, please return an executed (i.e. signed) copy of this license to the address listed below, and the Software will be mailed to you. The license portion of the document must be included in your request!

Bob Cmelik  
Sun Microsystems Laboratories, Inc.  
2550 Garcia Avenue, MS 29-225  
Mountain View, CA 94043  
(415) 336-1709  
rfc@sun.com

The foregoing is agreed to and accepted.

Signature: \_\_\_\_\_

Name: \_\_\_\_\_

Title: \_\_\_\_\_

Institution/Company: \_\_\_\_\_

Address: \_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_

E-mail: \_\_\_\_\_

Telephone: \_\_\_\_\_

Date: \_\_\_\_\_

© Copyright 1992 Sun Microsystems, Inc. The SMLI Technical Report Series is published by Sun Microsystems Laboratories, Inc.  
Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

#### TRADEMARKS

Sun, Sun Microsystems, and the Sun logo are trademarks or registered trademarks of Sun Microsystems, Inc. UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc. All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCworks, and SPARCcompiler are licensed exclusively to Sun Microsystems, Inc. All other product names mentioned herein are the trademarks of their respective owners.