Exploiting Trivial and Redundant Computation

Stephen E. Richardson (stever@sun.com)

Sun Microsystems Laboratories, Inc.

This paper discusses the notion of trivial *computation, where the appearance of simple operands renders potentially complex operations simple. 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 surprising 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.*

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 taken for each of three styles of machine architecture shows significant speedup using these techniques.

operation		conditions for triviality
multiply	$x * y$	$(x \text{ or } y) = (0, 1, \text{or } -1)$
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.

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.

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

1063-6889/93 $03.00 © 1993 IEEE

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

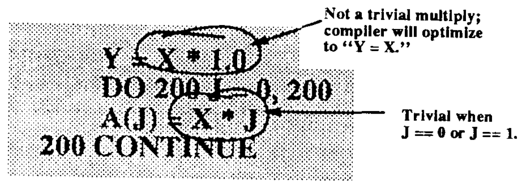
If a sufficient amount of computation were indeed trivial, some obvious changes in the style of computation could make programs run faster. Consider the following algorithm for multiplying two operands a and b to yield a result c:

```
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.



```
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.

How much trivial computation in real programs?

A tool called *shade* [4] 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.

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. 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 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

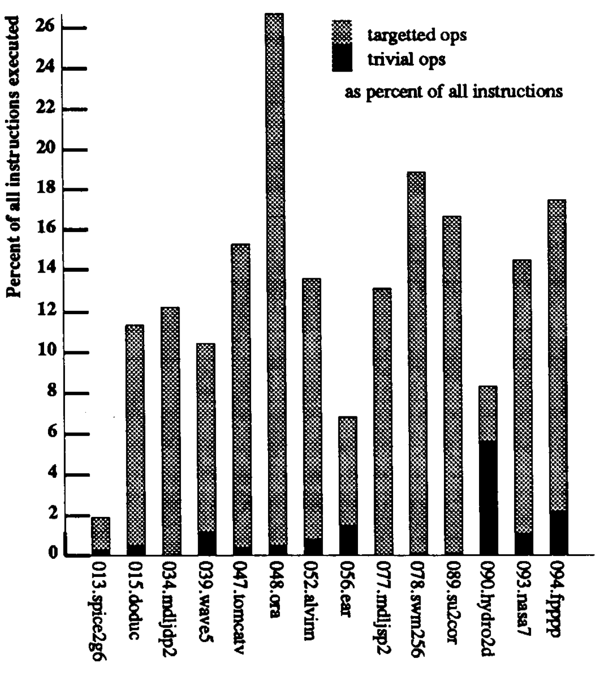


Figure 3: Trivial operations in SPEC benchmarks.

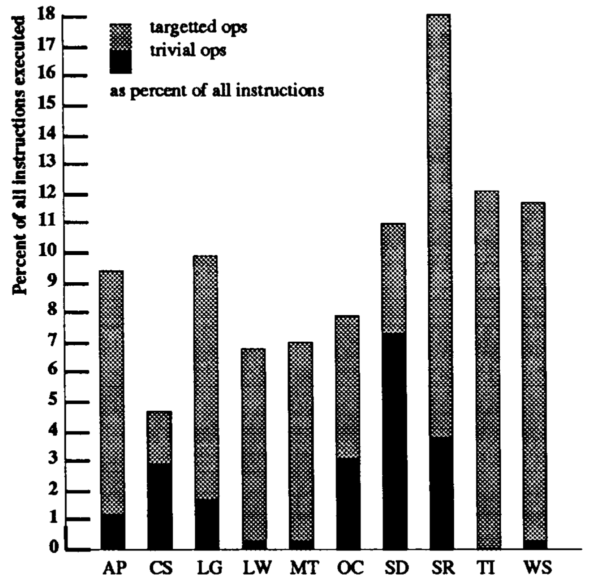


Figure 4: Trivial operations in Perfect Club benchmarks.

	integer		floating-point					
			multiply		divide		square-root	
	mul	div	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

*Numbers not available.

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

	integer		floating-point					
			multiply		divide		square-root	
	mul	div	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 ops on test machines.

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 [6, 13, 11, 12, 9, 10]. 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 $0/x$.

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-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.

The table in Figure 7 shows the overall performance im-

	Benchmark Suite	
Machine	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.

provement 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 [16]. Specifically, the study shows a significant number of floating-point divisions of the form $x/1.0$ and $x/2.0$. Aside from a few trivial operands such as 1 and 0, however, the most common operands 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 [14, 3].

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 [2, 7, 1, 8]. 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. (Furthermore, a sophisticated compiler could apply the method to make programs run faster [15].)

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 8 and 9 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 targeted operations. This portion of the graph represents the same information we saw earlier in Section 2. Successively taller bars show the number of instructions

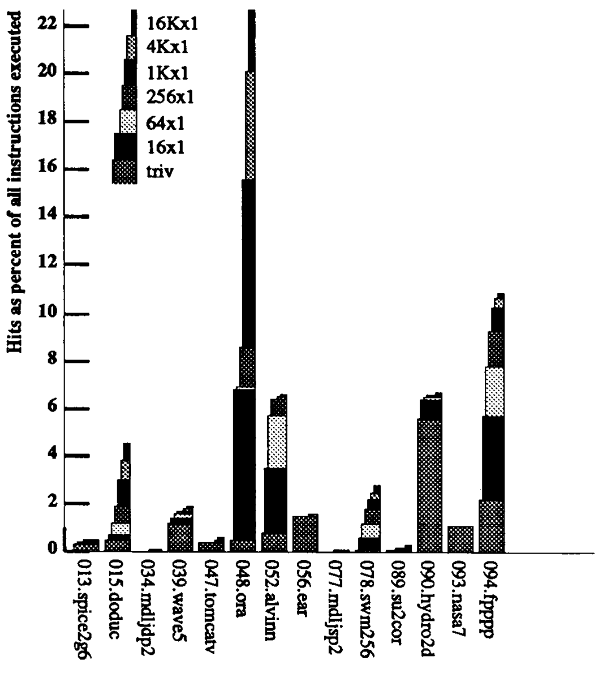


Figure 8: Percent of hits in result cache—SPEC.

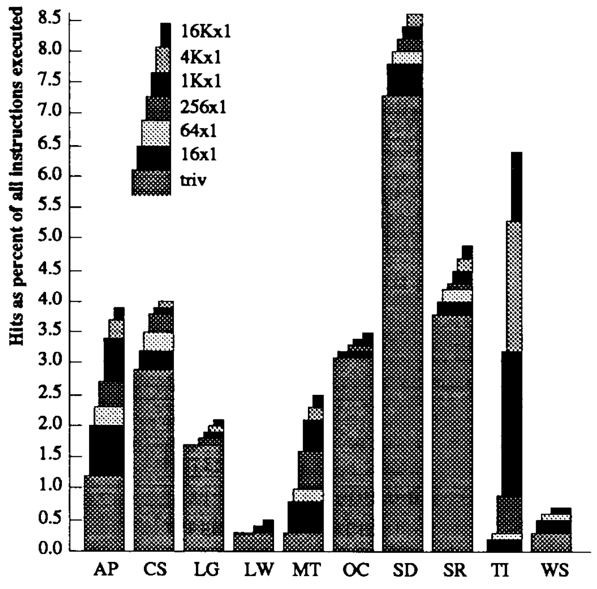


Figure 9: Percent of hits in result cache—Perfect Club.

that hit in successively larger direct-mapped result caches. In the graph, "256x1" means the cache contains 256 direct-mapped entries.

Figure 8, 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 9 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 primar-

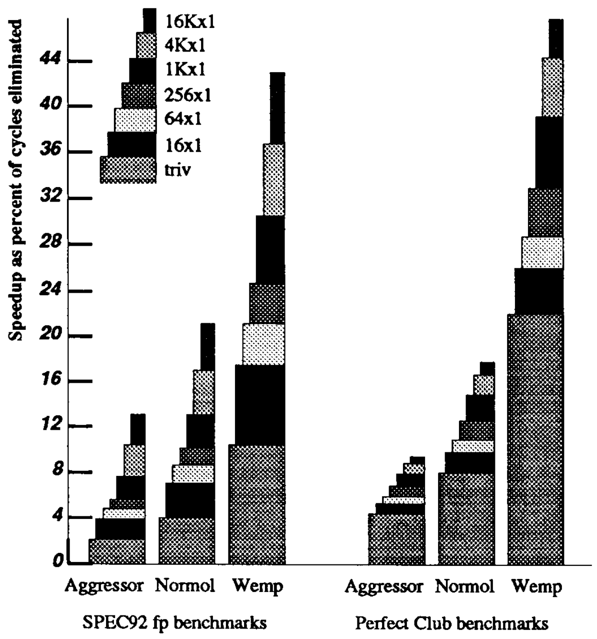


Figure 10: Machine speedups using result cache.

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

Figure 10 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 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 10 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 tchniques 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.

Appendix A: SPEC benchmarks

The SPEC (Systems Performance Evaluation Cooperative) benchmarks consist of twenty CPU-intensive programs variously written in FORTRAN and C[17]. 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.

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 galactical 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 [5]. 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.

AP:	A mesoscale model for air pollution.
CS:	The well-known circuit simulator *spice*.
LG:	Simulation of the gauge theory of the strong interaction that binds quarks and gluons into hadrons.
LW:	A molecular dynamics program for the simulation of liquid water.
MT:	Determines the course of a set of an unknown number of targets, such as missiles or rocket boosters.
NA[1]**:**	A molecular dynamics package for the simulation of nucleic acids.
OC:	A two-dimensional ocean simulation.
SD:	A structural dynamics benchmark, solves for displacements and stresses, along with velocities and accelerations at each time step.
SM[1]**:**	A seismic migration code used to investigate the geological structure of the earth.
SR:	A two dimensional fluid flow solver.
TF[1]**:**	Analysis of a transonic inviscid flow past an airfoil.
TI:	A kernel simulating a two-electron integral transformation.
WS:	A global spectral model to simulate atmospheric flow.

Appendix C: 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:

$$\otimes = \text{xor} \qquad + = \text{or} \qquad \gg = \text{right-shift}$$
$$\cdot = \text{and} \qquad \neg = \text{not}$$

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.

$$\begin{aligned}
\text{hash}(x_2, y_2, i, m) &= (x_2 \otimes y_2) \cdot m, \ \forall i \in \{4,6,8,10,12,14\} \\
x_2 &= (x_1 \gg 31) \otimes (x_1 \gg 20) \otimes (x_1 \gg (20 - i)) \\
x_1 &= (x_0 \gg 32) \cdot \text{FFFF FFFF}_{16}
\end{aligned}$$

[1]The benchmarks NA, SM, and TF were omitted from this paper because of a difficulty in attaining accurate results.

i	cache size = 2^i	mask $m = 2^i - 1$
4	16	F_{16}
6	64	$3F_{16}$
8	256	FF_{16}
10	1,024	$3FF_{16}$
12	4,096	FFF_{16}
14	16,384	$3FFF_{16}$

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.

Acknowledgements

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.

References

[1] 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.

[2] 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."

[3] 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.

[4] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. Unpublished as yet.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] Intel Corp., Santa Clara, California. *i486 Microprocessor*, April 1989.

[10] Intel Corp., Santa Clara, California. *Intel 80960KB Programmer's Reference Manual*, 1988.

[11] Motorola Inc. *MC88100 RISC User's Manual*, 1988.

[12] NEC, Mountain View, California. *VR4000 Microprocessor User's Manual*, 1991.

[13] Brett Olsson, Robert Montoye, Peter Markstein, and MyHong Nguyen Phu. *RISC System/6000 Floating-Point Unit*, pages 34–43. IBM, Austin, TX, 1990.

[14] Russell W. Quong and Mark A. Linton. Linking programs incrementally. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):1–20, January 1991.

[15] Stephen E. Richardson. Caching function results: Faster arithmetic by avoiding unnecessary computation. Technical Report SMLI TR-92-1, Sun Microsystems Laboratories, Inc., September 1992.

[16] Stephen E. Richardson. Smarter analysis of operand distributions. Unpublished documents SMLI 93-0114 and 93-0126, March 1993.

[17] Ray Weiss and Stan Baker. Spec adds benchmark for multiple processors. *EE Times*, April 1990.