ON DIVISION AND RECIPROCAL CACHES

Stuart F. Oberman and Michael J. Flynn

Technical Report: CSL-TR-95-666

April 1995

This work was supported by NSF under contract MIP93-13701.

ON DIVISION AND RECIPROCAL CACHES

by

Stuart F. Oberman and Michael J. Flynn

Technical Report: CSL-TR-95-666

April 1995

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305-4055

pubs@shasta.stanford.edu

Abstract

Floating-point division is generally regarded as a high latency operation in typical floating-point applications. Many techniques exist for increasing division performance, often at the cost of increasing either chip area, cycle time, or both. This paper presents two methods for decreasing the latency of division. Using applications from the SPECfp92 and NAS benchmark suites, these methods are evaluated to determine their effects on overall system performance. The notion of recurring computation is presented, and it is shown how recurring division can be exploited using an additional, dedicated division cache. Additionally, for multiplication-based division algorithms, reciprocal caches can be utilized to store recurring reciprocals. Due to the similarity between the algorithms typically used to compute division and square root, the performance of square root caches is also investigated. Results show that reciprocal caches can achieve nearly a 2X reduction in effective division latency for reasonable cache sizes.

Key Words and Phrases: Floating-point, division, reciprocal, square root, caches, area tradeoffs

Copyright © 1995
by
Stuart F. Oberman and Michael J. Flynn

Contents

1 Introduction **1**

2 Division Caches **1**
2.1 Description 1
2.2 Experiment 2
2.3 Performance 3
2.4 Efficiency 6

3 Reciprocal Caches **7**
3.1 Description 7
3.2 Iterative Division 8
3.3 Performance 9

4 Square Root Caches **12**

5 Conclusions **14**

List of Figures

1 Format of a division cache entry 3
2 Hit rates for infinite division cache 3
3 Average hit rates for division caches 4
4 Excess CPI due to division 5
5 CPI and area vs division latency 6
6 CPI vs area with and without division caches 7
7 Hit rates for infinite reciprocal cache 10
8 Average hit rates for reciprocal caches 10
9 Performance/area tradeoffs 12
10 Hit rates for infinite square root cache 13
11 Hit rates for shared division/square root cache 13

List of Tables

1 Performance of functional units 11
2 Performance/area tradeoffs of reciprocal caches 11

1 Introduction

Modern computer applications have increased in their computation complexity in recent years. The industry-wide usage of performance benchmarks, such as SPECmarks, forces processor designers to pay particular attention to floating-point computation. Furthermore, special purpose applications, such as high performance graphics rendering systems, have placed further demands on the computation abilities of processors. The development of high speed floating-point (FP) hardware is a requirement to meet these increasing computation demands.

Floating point units often comprise hardware implementations of addition, multiplication, division, and square root. Emphasis is typically placed on designing high performance adders and multipliers. As a result, division and square root often receive less design attention. While division is typically an infrequent operation even in floating-point intensive applications, ignoring its implementation can result in system performance degradation [8]. Many methods for implementing high performance division have appeared in the literature. Because of the low frequency of division operations, any proposed divide performance enhancement should be analyzed in terms of its possible silicon area and cycle time effects.

This paper investigates two techniques for decreasing the latency of floating-point division. Both techniques are based on recurring or redundant computations that can be found in applications. When the same divide calculation is performed on multiple occasions, it is possible to store and later reuse a previous result without having to repeat the lengthy computation. For multiplication-based divide implementations, the reciprocal can be reused rather than the quotient, increasing the likelihood of the computation being redundant. Additionally, due to the similarity between division and square root computation, the quantity of redundant square root computation is investigated.

Due to the probabilistic nature of caches, the true latency of each divide instruction is not constant. Variable latency instructions are, in general, undesirable when implementing a statically scheduled pipelined processor. However, modern CPUs are more frequently using the technique of *dynamic scheduling* to increase overall performance. Dynamic scheduling reduces the cost of data dependencies, and it also allows code that was compiled for one pipeline to continue to run efficiently on a different pipeline [2]. Thus, a divide instruction, when using one of the described techniques, becomes similar to a memory load in terms of processor scheduling. Thus, these methods are practical means of reducing the effective latency of division in high performance processors.

The remainder of this paper is organized as follows. Section 2 presents and analyzes division caches. Section 3 evaluates reciprocal caches. Section 4 analyzes square root caches. Section 5 is the conclusion.

2 Division Caches

2.1 Description

Computer applications typically perform computations on input data, and produce final output data based on the results of the computation. Often the input operands for a calcu-

lation are the same as those in a previous calculation due to the nature of the application. In matrix inversion, for example, each entry must be divided by the determinant. By recognizing and taking advantage of this redundant behavior, it is possible to decrease the effective latency of computations.

Richardson [10] discusses the technique of result caching as a means of decreasing the latency of otherwise high-latency operations, such as division. This technique exploits the redundant nature of certain computations by trading execution time for increased memory storage. Once a quotient is calculated, it is stored in a result cache. When a divide operation is initiated, the result cache can be simultaneously accessed to check for a previous instance of the computation. If the previous computation result is found, the quotient is available immediately from the cache. Otherwise, the operation continues in the divider, and the result is written into the cache upon completion of the computation.

2.2 Experiment

In this section, we analyze double precision floating-point divide operations and the effects of using a division cache to increase system performance. To obtain the data for the study, ATOM [12] was used to instrument several applications from the SPECfp92 [11] and NAS [7] benchmark suites. These applications were then executed on a DEC Alpha 3000/500 workstation.

All double precision floating-point divide operations were instrumented. An IEEE double precision operand is a 64-bit word, comprising a 1 bit sign, an 11 bit biased exponent, and 52 bits of mantissa, with one hidden mantissa bit [6]. For division, the exponent is handled in parallel with the mantissa calculation. Accordingly, the quotient mantissa is independent of the input operands' exponents, and only the input mantissas need be stored in the cache.

The reuse of previous division computations is a consequence of temporal locality. Spatial locality, a characteristic often found in memory references, is not applicable in this form of cache. In order to maximize hit rates of these caches, only associative caches with random replacement were considered in this study to take advantage of the temporal locality. Direct mapped or lower associativity caches could be used. However, to avoid lower hit rates due to conflict misses to the same cache entry, it would be necessary to sufficiently randomize the indexing into the cache. This was the technique chosen in [10], where a complex hash function is incorporated to perform this randomization. Thus, a tradeoff exists between complexity of the hash function and that of implementing higher associativity.

In this experiment, the tag is composed of the concatenation of the dividend and divisor mantissas, and a valid bit, forming 105 bits. Because the leading one is implied for the mantissas, only 52 bits per mantissa need be stored. The double precision quotient mantissa, with implied leading one, is stored in the data memory, along with guard, round, and sticky bits for a total of 55 bits. These extra bits are required to allow for correct rounding on subsequent uses of the same quotient, with possibly different rounding modes. The total storage required for each entry is therefore 160 bits. This is less than the 192 bits used in [10], due to the exclusion of exponents and hidden ones. The format of each cache entry is shown in figure 1.

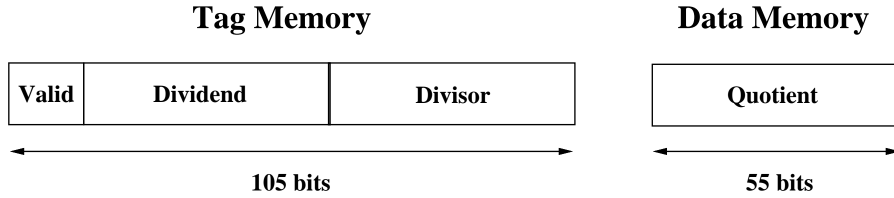

Figure 1: Format of a division cache entry

2.3 Performance

To place an upper bound on the amount of available redundant computation, hit rates were measured for each of the applications assuming an infinite, fully-associative division cache was present. These results are shown in figure 2.

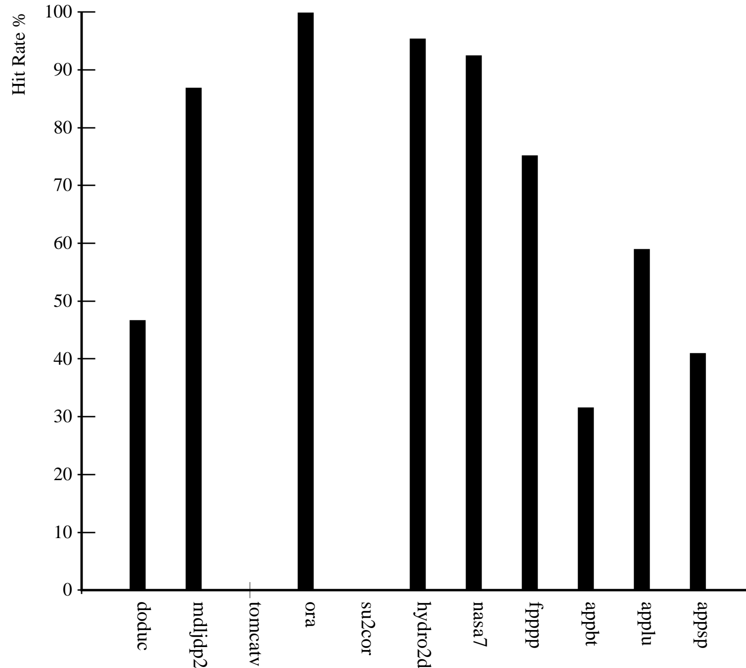

Figure 2: Hit rates for infinite division cache

These applications showed a wide variation in amounts of redundant divide computation, as demonstrated by the range of hit rates. However, the distribution is clearly bimodal. The majority of the applications exhibited some amount of redundant division. However, the applications tomcatv and su2cor from the SPEC suite exhibited very little or no redundant division, even though FP division accounted for 0.45% and 0.65% of the executed instructions respectively. Several of the applications exhibited a very large amount of redundant divide computation. The applications ora, hydro2d, and nasa7 all had hit rates

greater than 90%. The standard deviation of the hit rates for all 11 applications was 36.5%. When only those applications that exhibited some redundant computation are analyzed, thus excluding tomcatv and su2cor, the standard deviation decreased to 25.8%.

Divide caches with many configurations were then simulated, and the resulting hit rates are shown in figure 3. These hit rates were formed by taking the arithmetic average of the hit rates for each of the applications, weighted by dynamic divide instruction frequency, at each cache size.

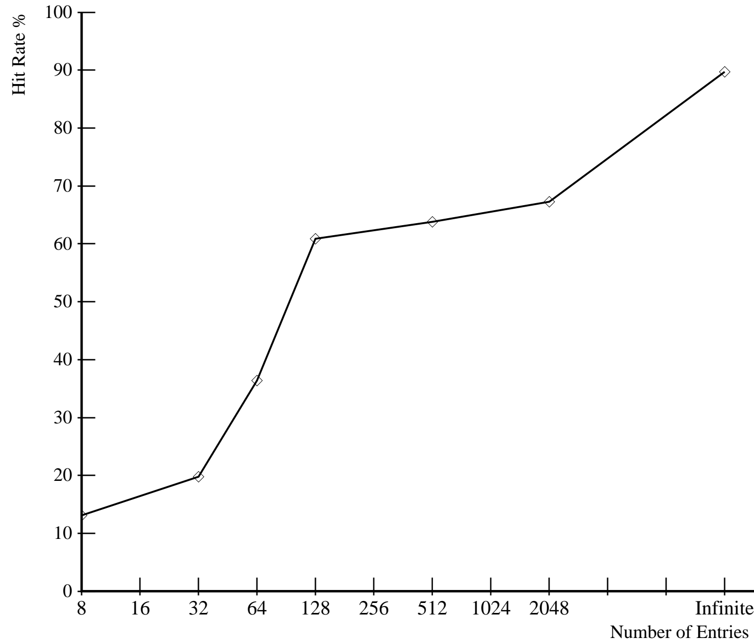

Figure 3: Average hit rates for division caches

The results of figure 3 demonstrate a knee near a division cache of 128 entries, with an average hit rate of about 60%. Additional entries over 128 provided only marginal increases in the hit rates.

A better measure of the effectiveness of a division cache is to determine its impact on system performance. While hit rates demonstrate the quantity of redundant computation present in an application, they do not provide much insight into how a division cache can increase overall performance. The excess CPI due to FP division interlocks in an application can be measured as a function of functional unit latency, both with and without a division cache.

The performance degradation due to divide can is a function of three variables, f, u, and l, where f is the dynamic frequency of divide instructions, u is the urgency of divide results, and l is the functional unit latency of divide. It is clear that f is solely a function of the application, u is a function of the application and the compiler, and l is a function of the hardware. For this analysis, a division cache size of 128 entries was chosen. A miss

to the division cache would suffer the standard divide latency. A hit to the cache, though, would have an effective latency of only two cycles: one cycle to return the result from the cache, and one cycle to perform any necessary rounding. Thus, it would be known one cycle in advance exactly when the computation would complete. This knowledge would allow a dynamically-scheduled processor to properly schedule the usage of the divide result.

In this way, the quantity l was formed. The value of u was determined by averaging the interlock distances, i.e. the distances between the production and consumption of divide results, across all of the applications. To minimize this quantity, all applications were compiled with a high degree of optimization, at the level of O3. These performance effects are presented in figure 4. This figure also shows the effect of increasing the number of instructions issued per cycle on excess CPI due to division. As the width of instruction issue increases, u increases for divide data proportionally. In the worst case, every divide result consumer could cause a stall equal to the functional unit latency.

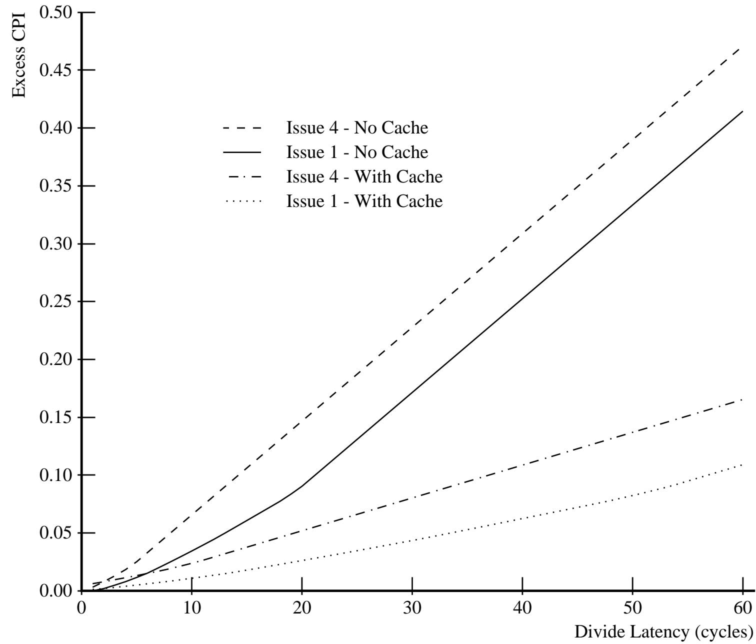

Figure 4: Excess CPI due to division

For scalar processors with high latency division units, for example a simple radix-2 SRT implementation with a latency near 60 cycles, the excess CPI due to division is decreased by a factor of four. Lower latency dividers, such as a radix-16 SRT divider with a functional unit latency of 15 cycles, have their CPI's reduced by about a factor of three. Similar speedups are obtained for the multiple issue processor. Thus, from a purely performance perspective, division caches seem to be an attractive method for reducing effective division latency.

2.4 Efficiency

An important metric for FPU designers is the amount of performance gained per additional unit of silicon area used. As mentioned previously, many techniques exist for reducing the latency of division. Thus, while division caches provide significant speedups, it is necessary to evaluate them as an alternative to other, possibly less area-demanding, alternatives. These alternatives include directly increasing the radix of SRT dividers, using multiplication-based Goldschmidt dividers, self-timed dividers, and look-up table based series approximations [8]. Using layout-extracted areas from several published dividers [9, 14, 15], along with the register bit equivalent (**rbe**) area model of Mulder [3], the relationship between functional unit latency, system performance, and silicon area can be derived. Figure 5 presents this relationship.

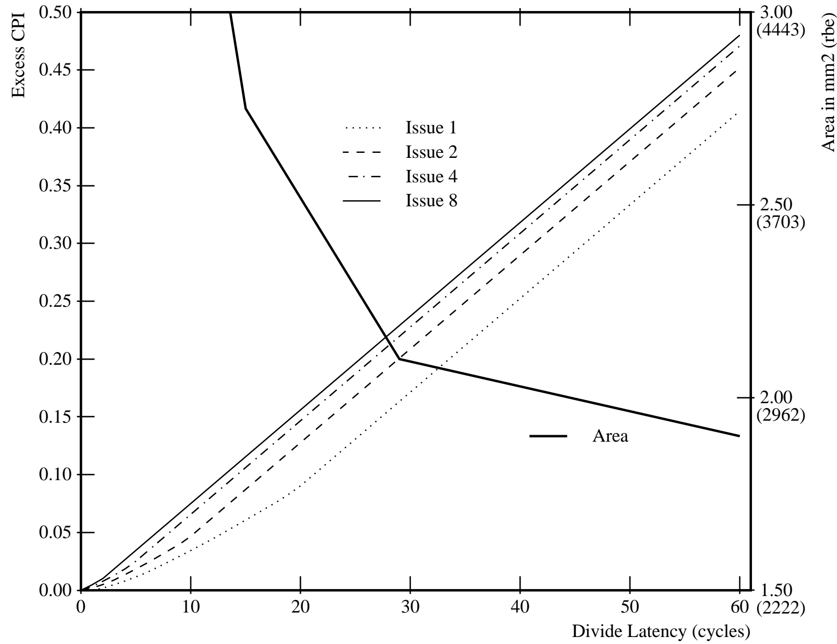

Figure 5: CPI and area vs division latency

Using the data from figure 5, it is possible to analyze the performance/area tradeoffs of using a division cache to supplement the performance of an existing divider. Figure 6 shows these tradeoffs. In this figure, excess CPI due to divide interlocks is graphed as a function of total silicon area devoted to divide computation. The bold curve is the set of dividers obtained where any additional area is used to directly decrease the latency of the functional unit. The other two curves show the effect of adding division caches to existing dividers, using the average hit rates from the previous analysis.

From figure 6, it is apparent that if the existing divider has a high latency, as in the case of the radix-4 SRT curve, the addition of a division cache is not area efficient. Only when the base divider already has a very low latency can the use of a division cache be as

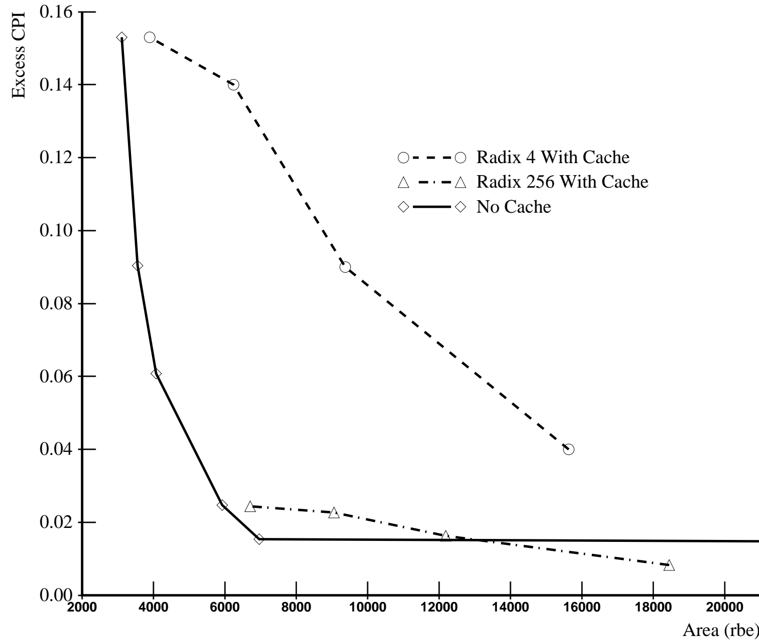

Figure 6: CPI vs area with and without division caches

efficient as simply improving the divider itself. However, many FPU designs do not have such a large area budget to devote to the divide hardware. Thus, from this analysis, the use of a cache to store the quotients for recurring divisions is not an efficient use of silicon area. Further, due to the large standard deviation in division cache hit rates, any reduction in latency is highly variable across different applications. Thus, if additional area is available for enhancing division performance, other alternatives for increasing the performance of the base divider should be considered.

3 Reciprocal Caches

3.1 Description

If redundancy exists in quotient computation, it is apparent that similar redundancy should exist in reciprocal computation, if not more. The calculation of a reciprocal requires only one operand. Hence, the likelihood of redundant computations based on a single operand would be equal to or greater than one that must match two operands. At the high level, redundant reciprocals might occur when many numbers are all divided by the same divisor, as in the case of matrix inversion. One solution might be to rewrite the algorithm to take this situation into account. The reciprocal of the divisor could be calculated directly, and each subsequent division operation would be converted into a faster multiplication. However, this would require the compiler or programmer to recognize this situation and rewrite the algorithm.

An alternative to algorithmic modification is to implement a reciprocal cache, which would store frequently used reciprocals. This would have two distinct advantages over the use of a division cache. First, the tag for each cache entry would be smaller, as only the mantissa of the divisor needs to be stored. Accordingly, the total size for each reciprocal cache entry would be approximately 108 bits. Second, intuitively, the hit rates should be larger, as it should be easier to find computations which share only the same divisor, rather than both the same divisor and dividend. The functionality of the reciprocal cache would be similar to that of the division cache. When a divide operation is initiated, the reciprocal cache can be simultaneously accessed to check for a previous instance of the reciprocal. If the result is found, the reciprocal is returned and multiplied by the dividend to form the quotient. Otherwise, the operation continues in the divider, and upon computation of the reciprocal, the result is written into the cache.

3.2 Iterative Division

Division can be implemented in hardware using the following relationship:

$$Q = \frac{a}{b} = a \times (\frac{1}{b}),$$

where Q is the quotient, a is the dividend, and b is the divisor. Certain algorithms, such as the Newton-Raphson and Goldschmidt iterations, are used to evaluate the reciprocal [5]. These two algorithms can be shown to converge quadratically in precision. The number and type of operations performed in the iterations for both the Goldschmidt and Newton-Raphson iterations are the same; they differ only in their ordering. The Goldschmidt iteration has a performance advantage in that the successive multiplications are independent, allowing for the efficient use of a pipelined multiplier. In general, iterative division algorithms are used due to their lower latency than typical subtractive methods. Additionally, the cycle time of each iteration is tightly coupled to the cycle time of the multiplier, which is typically tuned for low latency. Thus, iterative division allows for high radix division at achievable cycle times.

The choice of which iteration to use has a ramification on the use of a reciprocal cache. Whereas Newton-Raphson converges to a reciprocal and then multiplies by the dividend to compute the quotient, Goldschmidt's algorithm prescales the numerator and denominator by an approximation of the reciprocal, and converges directly to the quotient. Thus, Goldschmidt, in its basic form, is not suitable for reciprocal caching. However, a modification of Goldschmidt's can be made where this algorithm, too, converges to the reciprocal of the divisor. Then, it is necessary to multiply the reciprocal by the dividend to compute the quotient. This has the effect of adding one additional multiplication delay into the latency of the algorithm. This is a tradeoff that will be analyzed shortly.

Given an initial approximation for the reciprocal, typically from a ROM look-up table, the algorithms converge to the desired precision. Thus, higher performance can be achieved by using a higher precision starting approximation. Das Sarma [1] has shown the relationship between the accuracy of reciprocal ROM tables and the number of input bits to the table. The reciprocal tables described are of the form k-bits-in, $k+g$-bits-out, where g is

the number of guard digits in the input. The size of such tables is shown to be

$$2^k \times (k+g)$$

To guarantee error less than one ulp, g is chosen such that $g \geq 1$. In order to compute 60 bits of reciprocal precision in 3 iterations, the reciprocal table must provide at least 7.5 bits of precision. The smallest table that would provide this precision is a 7-bits input, 8-bits output table, which would yield 7.678 bits of precision. After 3 iterations, this would yield 61.424 bits of precision. The total size of this table would be 1,024 bits. However, depending on the rounding scheme implemented, it might be desired to calculate slightly more bits of precision. Hence, an 8-bits input table with no guard bits in the output, i.e. with 8-bits of output stored, will provide 8.415 bits of precision, and after 3 iterations would yield 67.32 bits precision. This table has a total size of 2,048 bits. For a 16-bits input, 16-bits output, the total size is 1M bits. For double precision division, due to the quadratic convergence of the iterative algorithm, an 8-bit table requires 3 iterations, while the 16-bit table requires only 2. This results in a tradeoff between area required for the initial approximation ROM and the latency of the algorithm. In this study, we present the additional tradeoff between larger initial approximation ROM's and cache storage for redundant reciprocals.

3.3 Performance

To place an upper bound on the quantity of redundant reciprocals present in the applications, hit rates were first measured assuming an infinite, fully-associative reciprocal cache was present. These results are shown in figure 7. The standard deviation for the hit rates of all 11 applications is 27.7%. However, when tomcatv is excluded, the standard deviation drops to only 6.2%. When compared with figure 2, it is readily apparent that for these applications, the reciprocal cache hit rates are consistently larger and less variable than the division cache hit rates.

Reciprocal caches with many configurations were then simulated, and the resulting hit rates are shown in figure 8, along with the hit rates of the division caches. The results of figure 8 demonstrate that a knee exists near a reciprocal cache of 128 entries. In general, the shape of the reciprocal cache hit rate tracks that of the division cache. At each cache size, though, the reciprocal cache hit rate is larger than that of the division cache by about 15%.

To determine the effect of reciprocal caches on overall system performance, the effective latency of division is calculated for several iterative divider configurations. For this analysis, the comparison is made with respect to the modified implementation of Goldschmidt's algorithm discussed previously. It is assumed that a pipelined multiplier and adder are present with performance as shown in table 1. These latencies correspond to a high-clock rate, high performance CPU.

The latency for a divide operation can be calculated as follows. An initial approximation ROM look-up is assumed to take 1 cycle. The initial prescaling of the numerator and the denominator requires 2 cycles. Each iteration of the algorithm requires 2 cycles for the multiply operations. Two cases arise for those schemes that use a reciprocal cache. A hit to the cache would have an effective latency of only three cycles: one cycle to return

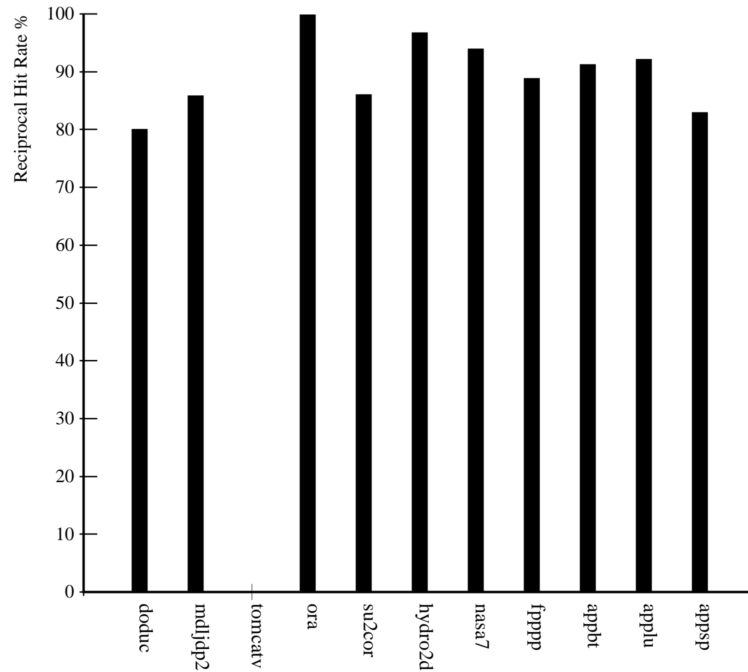

Figure 7: Hit rates for infinite reciprocal cache

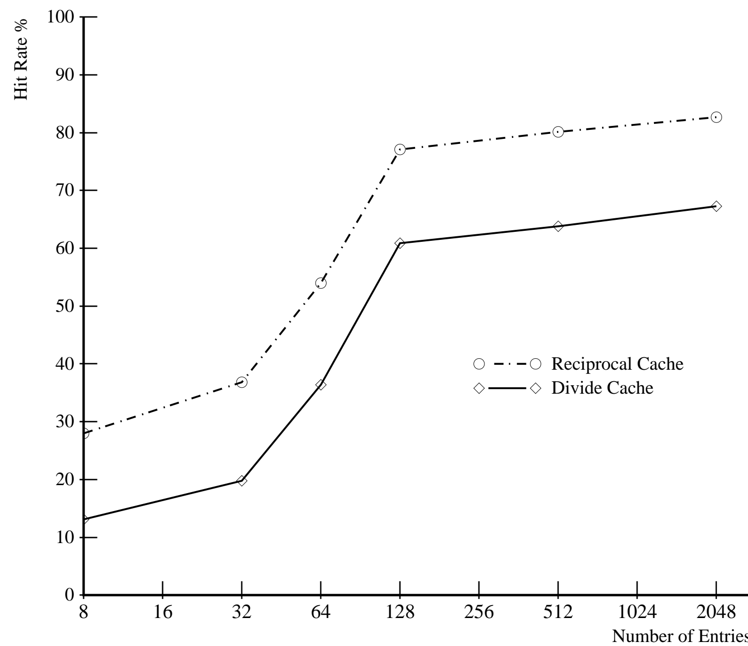

Figure 8: Average hit rates for reciprocal caches

Functional Unit	Latency (cycles)	Throughput (cycles)
Adder	2	1
Multiplier	2	1

Table 1: Performance of functional units

the result from the cache, and two to perform the multiplication by the dividend. A miss to the cache, though, would suffer the standard latency, plus an additional two cycles to multiply the reciprocal by the dividend, as per the modified Goldschmidt implementation. The results of this analysis are shown in table 2.

ROM Size	Cache Entries	Latency (cycles)	Extra Area (bits)
8-bit	-	9	-
16-bit	-	7	1,048,576
8-bit	8	8.76	864
8-bit	32	8.06	3,456
8-bit	64	6.68	6,912
8-bit	128	4.84	13,824
8-bit	512	4.60	55,296
8-bit	2048	4.38	221,184

Table 2: Performance/area tradeoffs of reciprocal caches

To better understand the tradeoff, figure 9 shows the performance of the different schemes relative to a no-cache, 8-bits-in 8-bits-out ROM reciprocal table implementation. Here, the speedups are measured against the increased area required, expressed as a factor of the 8-bit ROM reciprocal table size.

This graph demonstrates that as long as the total storage is less than about eight times that of an 8-bit implementation with no cache, or 2,048 bits, reciprocal caches can provide reasonable increases in division performance, achieving a latency speedup of 1.86. When the total area exceeds eight times the base area, the marginal increase in performance does not justify the increase in area. This can be compared to the use of a 16-bit reciprocal table, with a total storage of 1M bits. This yields an area factor of 512, with a speedup of only 1.29. The use of various reciprocal table compression techniques can reduce this storage requirement. However, the best case speedup with no reciprocal cache and requiring 2 iterations is still 1.29.

Performing IEEE rounding with a reciprocal cache could be done in a manner similar to the Texas Instruments TMS390C602A [4]. In this instance, the result would be computed to extra precision and would require a few more guard bits in the cache. Thus, the main effect would be slightly greater area for the same number of entries.

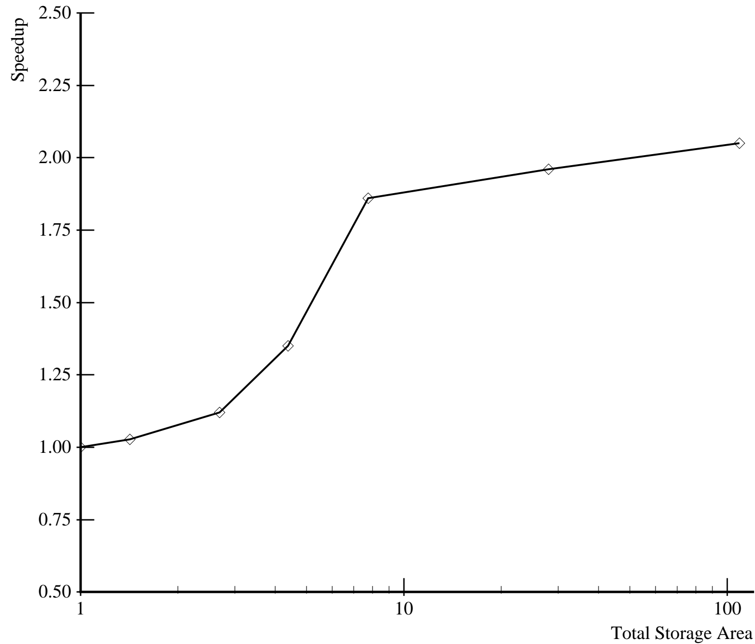

Figure 9: Performance/area tradeoffs

4 Square Root Caches

The implementation of square root in an FPU often shares the same hardware used for division computation. It can be shown that a variation of Goldschmidt's algorithm can be used to converge to the square root of an operand [13]. Thus, the question arises as to the quantity of redundant square root computation available in applications. Because both the reciprocal and square root operations are unary, they could easily share the same cache for their results.

A similar experiment was performed for square root as was done for division and reciprocal operations. All double precision square roots were instrumented along with double precision divide operations. The resulting hit rates for a dedicated infinite square root cache are shown in figure 10. The analysis showed that for an infinite cache, some applications contained a large amount of redundant square root computation. However, about half of the applications contained no redundant computation. The standard deviation of the square root hit rates was 48%.

The experiment was repeated for finite shared reciprocal/square root caches. The results are shown in figure 11. The shared cache results show that for reasonable cache sizes, the square root result hit rates are low, about 50% or less. Although the frequency of square root was about 10 times less than division, the inclusion of square root results did cause interference with the reciprocal results. This had the effect of decreasing the reciprocal hit rates, especially in the cases of 64 and 128 entries. Thus, this study suggests that square

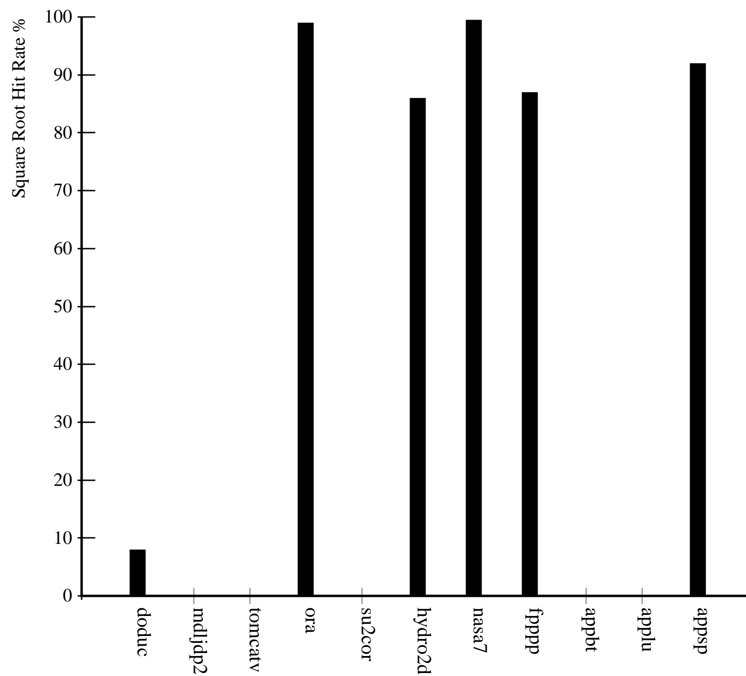

Figure 10: Hit rates for infinite square root cache

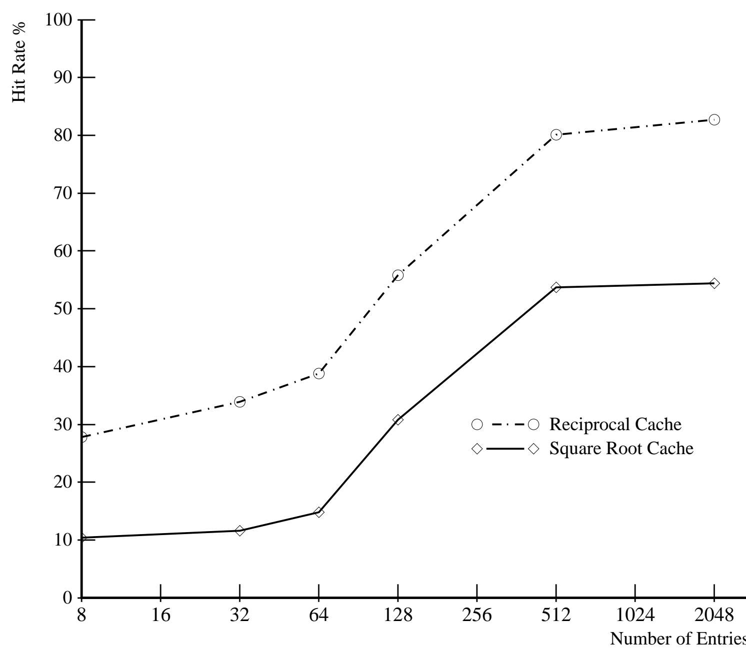

Figure 11: Hit rates for shared division/square root cache

root computations should not be stored in either a dedicated square root cache or a shared reciprocal cache, due to the low and highly variant hit rate of square root and the resulting reduction in reciprocal hit rate.

5 Conclusions

This study indicates that redundant division computation exists in many applications. Both division caches and reciprocal caches can be used to exploit this redundant behavior. When using a low radix SRT divider, division caches might initially seem attractive. However, this study has shown that rather than including a division cache to increase system performance, any additional area may better utilized in directly increasing the radix of the divider, so long as the cycle time of the processor can still be met.

For high performance implementations, where a multiplication-based algorithm is used, the inclusion of a reciprocal cache is an efficient means of increasing performance. In this scenario, too, a division cache could be used. However, the high standard deviation of a division cache's hit rates compared with that of a reciprocal cache argues against its usage and for the use of a reciprocal cache. Additionally, the analysis has shown that these applications do not contain a consistently large quantity of redundant square root computation. Thus, the caching of square root results as a means for increasing overall performance is not recommended.

The primary alternative previously to decrease latency of multiplication-based division algorithms has been to reduce the number of iterations by increasing the size of the initial approximation reciprocal table. This study has demonstrated that a reciprocal cache is an effective alternative to large reciprocal tables. The inclusion of a reasonably sized reciprocal cache can consistently provide a significant reduction in division latency.

References

[1] D. DasSarma and D. Matula. Measuring the accuracy of ROM reciprocal tables. *IEEE Transactions on Computers*, 43(8), August 1994.

[2] J. Hennessy et al. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 1990.

[3] J. Mulder et al. An area model for on-chip memories and its application. *IEEE Journal of Solid-State Circuits*, 26(2), February 1991.

[4] M. Darley et al. The TMS390C602A floating-point coprocessor for Sparc systems. *IEEE Micro*, 10(3), June 1990.

[5] M. Flynn. On division by functional iteration. *IEEE Transactions on Computers*, C-19(8), August 1970.

[6] ANSI/IEEE std 754-1985, IEEE standard for binary floating-point arithmetic.

[7] NAS parallel benchmarks 8/91.

[8] S. Oberman and M. Flynn. Design issues in floating-point division. Technical Report No. CSL-TR-94-647, Computer Systems Laboratory, Stanford University, December 1994.

[9] S. Oberman, N. Quach, and M. Flynn. The design and implementation of a high-performance floating-point divider. Technical Report No. CSL-TR-94-599, Computer Systems Laboratory, Stanford University, January 1994.

[10] S. E. Richardson. Exploiting trivial and redundant computation. In *Proceedings of the 11th IEEE Symposium on Computer Arithmetic*, pages 220–227, July 1993.

[11] SPEC benchmark suite release 2/92.

[12] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[13] S. Waser and M. Flynn. *Introduction to Arithmetic for Digital Systems Designers*. Holt, Rinehart, and Winston, 1982.

[14] T. E. Williams and M. A. Horowitz. A zero-overhead self-timed 160-ns 54-b CMOS divider. *IEEE Journal of Solid-State Circuits*, 26(11), November 1991.

[15] D. Wong and M. Flynn. Fast division using accurate quotient approximations to reduce the number of iterations. *IEEE Transactions on Computers*, 41(8), August 1992.