Reliable Computing 2 (2) (1996), pp. 147–153

Reducing division latency with reciprocal caches

Stuart F. Oberman and Michael J. Flynn

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 reducing 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. For multiplication-based division algorithms, reciprocal caches can be utilized to store recurring reciprocals. Results show that reciprocal caches can achieve nearly a two-times speedup in division performance for reasonable cache sizes.

Ускорение деления с помощью кэширования обратных значений

С. Оберман, М. Флинн

Деление значений с плавающей точкой в приложениях, использующих арифметику с плавающей точкой, обычно требует больших затрат времени. Для повышения эффективности деления предложено немало методов, многие из которых требуют увеличения площади кристалла, снижения тактовой частоты или и того, и другого. Представлены два метода ускорения операции деления. Приводятся данные о влиянии этих методов на общую производительность системы, полученные с помощью тестовых программ из пакетов SPECfp92 и NAS. Приводится понятие рекуррентных вычислений и предлагается способ реализации рекуррентного деления с помощью дополнительной кэш-памяти, отведенной специально для этой операции. В алгоритмах деления, основанных на умножении, можно использовать кэш-память для хранения рекуррентных обратных значений. Результаты свидетельствуют, то кэш-память для обратных значений может обеспечить почти двукратное увеличение скорости деления при сравнительно небольшом ее размере.

1. Introduction

Floating-point division has received increasing attention in recent years. Division has a higher latency, or time required to perform a computation, than addition or multiplication. While division is an infrequent operation even in floating-point intensive applications, its high latency can result in significant system performance degradation [4]. Many methods for implementing high performance division have appeared in the literature. However, any proposed division performance enhancement should be analyzed in terms of its possible silicon area and cycle time effects.

Richardson [6] discusses the technique of result caching as a means of decreasing the latency of otherwise high-latency operations, such as division. Result caching is based on recurring or redundant computations that can be found in applications. Often, one or both of the input operands for a calculation are the same as those in a previous calculation. In matrix inversion, for example, each entry must be divided by the determinant. When such recurring computation is present, it is possible to store and later reuse a previous result without having to repeat the computation.

This study investigates the use of a reciprocal cache as a method for reducing the latency of floating-point division. By recognizing and taking advantage of redundant division computations, it is possible to reduce the effective division latency. The performance and efficiency of reciprocal caches is compared with division caches. Additionally, due to the similarity between division and square root computation, the performance of shared reciprocal/square root caches is investigated.

2. Reciprocal caches

2.1. Iterative division

Division can be implemented in hardware using the following relationship:

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

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 [1]. These two algorithms can be shown to converge quadratically in precision. 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 can be made where this algorithm, too, converges to the reciprocal of the divisor. It is then 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.

Given an initial approximation for the reciprocal, typically from a ROM look-up table, the algorithms converge to the desired precision. Each iteration in these algorithms comprises 2 multiplications and a two's complement operation. Goldschmidt has the advantage that its multiplications are independent and can take advantage of a pipelined multiplier. Higher performance can be achieved by using a higher precision starting approximation. Due to the quadratic convergence of these iterative algorithms, the computation of 53-bit double precision quotients using an 8-bit initial approximation table requires 3 iterations, while a 16-bit table requires only 2 iterations. This results in a tradeoff between area required for the initial approximation table and the latency of the algorithm. In this study, we present the additional tradeoff between larger initial approximation tables and cache storage for redundant reciprocals.

2.2. Experimental methodology

To obtain the data for the study, ATOM [8] was used to instrument several applications from the SPECfp92 [7] and NAS [3] benchmark suites. These applications were then executed on a DEC Alpha 3000/500 workstation. All double precision floating-point division 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 [2]. For division, the exponent is handled in parallel with the mantissa calculation. Accordingly, the quotient mantissa is independent of the input operands' exponents.

For reciprocal caches, the cache tag is the concatenation of the divisor mantissa and a valid bit, for a total of 53 bits. Because the leading one is implied for the mantissas, only 52 bits per mantissa need be stored. The cache data is the double precision reciprocal mantissa, with implied leading one, and the 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 reciprocal, with possibly different rounding modes. The total storage required for each entry is therefore 108 bits.

When a division operation is initiated, the reciprocal cache is 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.

2.3. Performance

Reciprocal cache hit rates were first measured assuming an infinite, fully-associative cache. These results are shown in Figure 1(a). The average hit rate of all 11 applications is 81.7%, with a standard deviation of 27.7%. From Figure 1(a), it can be seen that the application tomcatv is unusual in that it has no reciprocal reuse, as demonstrated by its 0% hit rate. When tomcatv is excluded, the average hit rate is 89.8%, and the standard deviation is only 6.2%. Reciprocal caches of finite size were then simulated, and the resulting hit rates are shown in Figure 1(b). Figure 1(b) shows that most of the redundant reciprocal computation is captured by a 128 entry cache, with a hit rate of 77.1%.

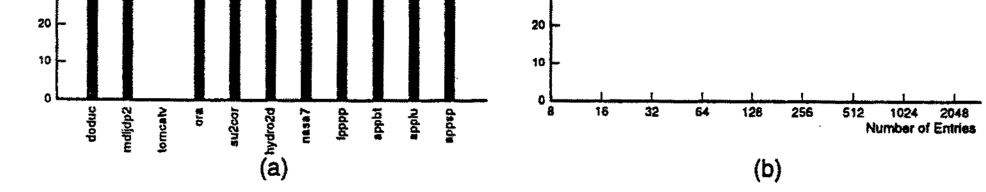

Figure 1. Hit rates for (a) infinite and (b) finite reciprocal caches

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 is present with a latency of 2 cycles and a throughput of 1 cycle.

The latency for a division operation can be calculated as follows. An initial approximation table 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 2 overlapped multiplications. The final result is available after an additional cycle to drain the multiplier pipeline. Thus, a base 8-bit Goldschmidt implementation without a cache requires 10 cycles to compute the quotient. Two cases arise for a scheme using a reciprocal cache. A hit in the cache has an effective latency of only 3 cycles: 1 cycle to return the reciprocal and 2 to perform the multiplication by the dividend. A miss in the cache suffers the base 10 cycle latency plus an additional 2 cycles to multiply the reciprocal by the dividend, as per the modified Goldschmidt implementation. The results of this analysis are shown in Table 1.

ROM Size	Cache Entries	Latency (cycles)	Extra Area (bits)
8-bit	0	10	0
16-bit	0	8	1,046,528
8-bit	8	9.48	864
8-bit	32	8.69	3,456
8-bit	64	7.14	6,912
8-bit	128	5.06	13,824
8-bit	512	4.79	55,296
8-bit	2048	4.56	221,184

Table 1. Performance/area tradeoffs for reciprocal caches

Figure 2 shows the performance of the different cache sizes relative to an 8-bit initial approximation table implementation. Here, the speedups are measured against the total storage area required, expressed as a factor of the 8-bit initial approximation table size, which is 2048 bits. This graph demonstrates that when the total storage is approximately eight-times that of an 8-bit implementation with no cache, a reciprocal cache can provide a significant increase

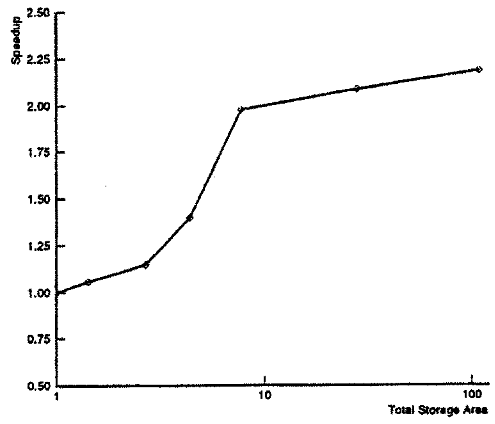

Figure 2. Speedup from reciprocal caches

in division performance, achieving approximately a two-times speedup. When the total area exceeds eight-times the base area, the marginal increase in performance does not justify the increase in area. A reciprocal cache implementation can be compared to the use of a 16-bit initial approximation table, with a total storage of 1M bits. This yields an area factor of 512, with a speedup of only 1.25. The use of various table compression techniques could reduce this storage requirement. However, the best case speedup with no reciprocal cache and requiring 2 iterations is still 1.25.

3. Division caches

An alternative to a reciprocal cache to reduce division latency is a division cache. A division cache can be used for any form of divider implementation, regardless of the choice of algorithm. For a division cache, the tag is larger than that of a reciprocal cache, as it comprises the concatenation of the dividend and divisor mantissas, and a valid bit, forming 105 bits. Accordingly, the total storage required for each division cache entry is 160 bits. The functionality of the division cache is similar to that of the reciprocal cache. When a division operation is initiated, the division cache can be simultaneously accessed to check for a previous instance of the exact dividend/divisor pair. If the result is found, the correct quotient is returned. Otherwise, the operation continues in the divider, and upon computation of the quotient, the result is written into the cache. The number of computations reusing both operands at best will be equal to and will be typically less than the number reusing only the same divisor. However, reciprocal caches restrict the form of algorithm used to compute the quotient, while division caches allow any divider implementation.

Hit rates were measured for each of the applications assuming an infinite, fully-associative division cache. The average hit rate was found to be 57.1%, with a standard deviation of 36.5%. When analyzing only those applications that exhibited some redundant computation, excluding tomcatv and su2cor, the average hit rate is 69.8%, with a standard deviation of 25.8%. Thus, the quantity of redundant division in the applications compared with redundant reciprocals was lower and more variant.

Finite division caches were simulated, and the resulting hit rates are shown in Figure 3(a), along with the hit rates of the reciprocal caches with the same number of entries. The results of Figure 3(a) demonstrate a knee near a division cache of 128 entries, with an average hit rate of 60.9%. In general, the shape of the reciprocal cache hit rate tracks that of the division cache. For the same number of entries, though, the reciprocal cache hit rate is larger than that of the division cache by about 15%. Thus, the quantity of redundant division in the applications compared with redundant reciprocals was lower and more variant. Additionally, a division cache requires approximately 50% more area than a reciprocal cache with the same number of entries. Further performance and efficiency analysis of division caches is presented in [5].

4. Square root caches

The implementation of square root 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 [9]. Thus, the question arises as to the quantity of redundant

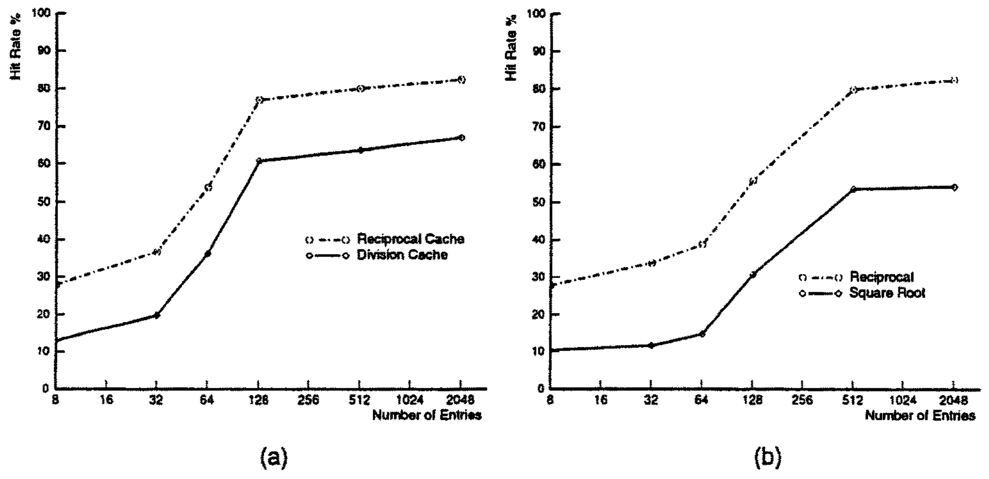

Figure 3. Hit rates for (a) division and (b) reciprocal/square root caches

square root computation available in applications. Because both the reciprocal and square root operations are unary, they can 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. Hit rates were measured for finite shared reciprocal/square root caches, where both reciprocals and square roots reside in the same cache. The results are shown in Figure 3(b). 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 caused 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 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. 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 table. This study demonstrates 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] Flynn, M. *On division by functional iteration*. IEEE Transactions on Computers **C–19** (8) (1970).

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

[3] *NAS parallel benchmarks release*. August, 1991.

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

[5] Oberman, S. and Flynn, M. *On division and reciprocal caches*. Technical Report No. CSL–TR–95–666, Computer Systems Laboratory, Stanford University, 1995.

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

[7] *Spec benchmark suite release*. February, 1992.

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

[9] Waser, S. and Flynn, M. *Introduction to arithmetic for digital systems designers*. Holt, Rinehart, and Winston, 1982.

Received: October 20, 1995
Revised version: November 29, 1995

Computer Systems Laboratory
Department of Electrical Engineering
Stanford University
Stanford, CA 94305–9030
USA