Reducing the Latency of Division Operations with Partial Caching

Edward Benowitz, Miloš Ercegovac
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024

Farzan Fallah
Fujitsu Laboratories of America, Inc.
595 Lawrence Expressway
Sunnyvale CA 94085

Abstract

We seek to reduce the latency of division operations. In many programs arithmetic instructions are frequently executed on the same inputs. Previous work has exploited this property by caching the quotient, bypassing the divider in the case of a cache hit. We propose caching a portion of the quotient, allowing a reduction in the cache size and an increase in cache hit rates. We call this approach partial caching. We present modifications to digit-recurrence division methods to accommodate partial caching. Using several benchmarks, we measure the hit rate and speedup which can be obtained.

1 Introduction

This research is about reducing the number of cycles needed to execute division operations. Our approach exploits the findings in [2, 1, 8, 7], which state that arithmetic instructions are frequently executed on the same set of inputs. If these recurring instances of instruction execution are stored in a cache-like memory, the latency is reduced from many execution cycles to a few cycles needed to access this cache and retrieve the result. We investigate the use of this approach in digit-recurrence division, develop designs, and evaluate the resulting cost and latency.

1.1 Related Work

Previous research [1, 2, 6, 7, 8] in caching division operations has focused on bypassing an entire division operation, without changing the arithmetic algorithm. A cache is indexed using the dividend and divisor. If a cache hit occurs, the quotient may be obtained. We will refer to this as full caching.

The idea of applying caching, in hardware, to arithmetic operations was introduced in [8]. He also bypasses the computation of trivial operands. Citron and Rudolph [2, 1] investigated design parameters for arithmetic caches, such as cache size and associativity, as well as the interaction with a processor's architecture.

Oberman and Flynn [7, 6] investigated the use of division caches and reciprocal caches. Reciprocal caching is used in conjunction with a multiplicative method. Reciprocal caches tend to have higher hit rates and less variability.

1.2 Our Approach

We investigate the design and performance of caching of the most-significant parts (*prefix* parts) of the operands and the result. We call this approach *partial caching*. We define the prefix to be the p most significant digits of the mantissa. Let

- X be the dividend,

$$X = X_a + X_b = \sum_{i=1}^{p} x_i r^{-i} + \sum_{i=p+1}^{m} x_i r^{-i}$$

- D be the divisor,

$$D = D_a + D_b = \sum_{i=1}^{p} d_i r^{-i} + \sum_{i=p+1}^{m} d_i r^{-i}$$

- Q be the quotient

$$Q = Q_a + Q_b = \sum_{i=1}^{p-\delta} q_i r^{-i} + \sum_{i=p-\delta+1}^{m} q_i r^{-i}$$

with X and D represented in a radix-r sign-and-magnitude system, and Q represented in two's complement after conversion. In the case of a cache hit, we avoid computing Q_a and reduce correspondingly the number of division iterations. The choice of prefix p allows trade-offs between cache area (i.e., cache hit rates) and the number of division iterations.

We will examine partial caching within the context of digit-recurrence methods. Our derivation of partial caching is based on the principle of on-line arithmetic [3]: to obtain the first quotient digit, $\delta + 1$ digits of the dividend and the divisor are processed. Thereafter, one digit of the dividend and divisor is used per

cycle to produce the next quotient digit. This implies that if p most-significant digits of the dividend and the divisor in a division operation are the same as in another one, the quotient of $p-\delta$ digits and the resulting remainder are also the same, and can be obtained from a cache.

2 Division with Partial Caching

We now describe our approach for using the cached prefix in digit-recurrence division. Our goal is to reduce the number of cycles required. If the prefices of independent operations are the same, it is wasteful to recompute those bits of the quotient dependent only upon the prefix. Instead, we retrieve the part of the division operation dependent upon the prefix from a cache, and compute the remaining digits of the quotient. Upon a cache miss, the state at the end of the prefix computation is stored in the cache and the remaining iterations are completed.

2.1 Algorithm

Algorithm 1 Division with Partial Caching

W[0]$\leftarrow X_a$; Q[0] $\leftarrow$ 0
{Stage 1}
for $j = 0$ to $p - \delta - 1$ **do**
  q_{j+1} = select($\hat{y}$)
  Q[j+1]$\leftarrow$ convert(Q[j], q_{j+1})
  W[j+1] $\leftarrow$ rW[j] - $q_{j+1}D_a$
end for
{Stage 2}
$W[p-\delta]_{corrected} \leftarrow$ W$[p-\delta] + X_b r^{p-\delta}$ - $Q_a D_b r^{p-\delta}$
{Stage 3}
for $j = p - \delta$ to $n - 1$ **do**
  q_{j+1} = select($\hat{y}$)
  Q[j+1]$\leftarrow$ convert(Q[j],q_{j+1})
  W[j+1] $\leftarrow$ rW[j] - $q_{j+1}D$
end for

We now discuss the algorithm for radix-r digit-recurrence division with partial caching, which is summarized in Algorithm 1. Let p be the length of the prefix in digits, $W[j]$ be the residual at iteration j, δ be the online delay in digits, and $\hat{y}$ be an estimate of the shifted residual $rW[j]$.

The algorithm consists of three basic stages: 1) Perform a p-digit by p-digit division, producing only $p-\delta$ quotient digits. This requires that $p \geq \delta$. 2) Correct the residual, and extend the operands. 3) Produce the remaining digits of the quotient.

In the case of a cache miss, Stage 1 is performed, and the residual $W[j]$ and Q_a, are stored in the cache. Then, Stages 2 and 3 are performed. In the case of a cache hit, we obtain Q_a and the residual from the cache, allowing us to bypass Stage 1. Only Stages 2 and 3 are performed.

One may question why we only produce $p - \delta$ digits in Stage 1, rather than producing p quotient digits of output. The residual must be taken into account. When proceeding from Stage 1 to Stage 3, one must correct the residual from Stage 1 with correction terms derived from the full precision operands, so that the algorithm will converge. On-line division provides these correction terms. If we produce $p - \delta$ digits of quotient, we maintain the on-line property, and are thus guaranteed convergence.

Consider Stage 1, which produces $p-\delta$ digits of quotient, given p digits of X and p digits of D. Throughout Stage 1, we are guaranteed to have at least as many input digits as the corresponding on-line algorithm. An on-line operation producing j digits of output requires $j + \delta$ digits of input. So for the on-line algorithm to produce $p - \delta$ digits of quotient, it requires at least p digits of input. We require Stage 1 to have p digits of input, and only allow the stage to progress through $p - \delta$ iterations.

To understand Stage 2, the correction step, it is necessary to first examine those correction terms present in the on-line algorithm. The correction term for X is $x_{j+\delta+1} r^{-\delta}$. As an additional digit of X arrives, we must insert this new digit into the residual. In a pure digit-recurrence algorithm, the residual would have all of X at the start of the algorithm. The online algorithm corrects itself by inserting one digit at a time. But for the prefix approach, we insert all of the remaining digits of X at once, with the term $X_b r^{p-\delta}$. Since $p - \delta$ iterations preceded the correction phase, we multiply by $r^{p-\delta}$.

The second correction term uses the newly introduced digits of D. Consider the definition of $W[j]$, $W[j] = X - Q * D$. We have already accounted for X; now we need to take into account the $Q * D$ term. After producing $p - \delta$ digits of quotient, we need to account for $Q_a * D$, but these iterations only provided $Q_a D_a$. Therefore, when we have the remaining digits of D_b, we subtract the correction term $Q_a D_b r^{p-\delta}$. Since $p-\delta$ iterations preceded the correction phase, we multiply by $r^{p-\delta}$. We have now corrected the residual to use the full-precision operands.

After the correction, Stage 3 proceeds with full-precision operands. Using an argument similar to that for Stage 1, we are guaranteed to have at least as many bits as the equivalent on-line algorithm. Since we have the full precision operands, this phase is analogous to a standard digit-recurrence division. Stage 1 is analo-

gous to a p by p digit-recurrence division, iterated for $p - \delta$ cycles. Only Stage 2 requires the extra terms needed for an online division.

2.2 Implementation

The block diagram for our implementation is shown in Figure 1. The diagram shows a radix-2 implementation. The additional components required include: addition multiplexers, a multiplier accumulator for computing the corrected residual (the output remains in carry-save represenation), and a short CPA used to assimilate $W[j]$ for cache storage and to compute QM_a after a cache hit. An on-line selection function is required. Stage control units select between a full precision value, and the corresponding prefix.

Each entry in the cache has a tag portion and a data portion. The tag, which consists of X_a and D_a, is used to index the cache. Each of these quantities requires p digits of storage. For the data section of a cache entry, we store both Q_a and the residual $W[j]$. Q_a requires $p - \delta$ digits of storage. Let Q and QM constitute the state of the on-the-fly converter. We can explicitly compute QM_a using the existing CPA. This step can be performed simultaneously to the multiplication in the correction step. Thus, we only need to store Q_a in the cache, but not QM_a. The residual $W[j]$ is obtained in carry-save form. It is converted to the conventional form during the remaining steps of the algorithm. The conversion is not in the critical path and a simple CPA is sufficient. The total number of digits required for one cache entry is $4p - \delta$.

2.3 Comparison

We now compare a number of divider implementations. We perform a technology-independent evaluation, with timing and area reported in terms of 2-input NAND gates. Let t_{check} be the time to check for a cache hit, t_{read} the time to read data from the cache upon a cache hit, t_{cor} the time to execute the correction step of the algorithm, t_{cycle} the cycle time, $T_{div\,H}$ the latency of a division with cache hit, and $T_{div\,M}$ the latency of a division with cache miss.

For the cache tag matching t_{check}, and cache read/write, t_{read}, we assume a 1 cycle delay. We find that t_{cor} is approximately 1 cycle. We consider both radix-2 and radix-16 implementations. For radix-2, $p = 10$, under a hit,

$$\begin{aligned} T_{div\,H} &= t_{cycle}(1 + n - p + \delta) + t_{cor} \\ &\approx t_{cycle}(n - 4) \end{aligned}$$

while under a cache miss,

$$T_{div\,M} = t_{cycle}(n) + t_{cor} \approx t_{cycle}(n + 1)$$

Table 1: Digit-recurrence algorithm comparison for n=24*

Algorithm	latency T_{div}	datapath area	cache entry size
Radix-2	1	1	-
Radix-2 full	.29	1	72
Radix-2 partial	.93	1.3	36
Radix-16	.42	1.6	-
Radix-16 full	.16	1.6	72
Radix-16 partial	.41	3.3	66

*For all cached division methods, 2^8 cache entries were used. Latency and datapath area are normalized. Cache entry size is given in bits.

For radix-16, two radix-4 stages are overlapped. We choose δ to be 3 radix-4 digits, and redundancy factor ρ to be $2/3$. We assume a prefix of 18 bits, allowing 3 radix-16 quotient digits to be cached. Under a cache hit,

$$\begin{aligned} T_{div\,H} &= t_{cycle}(1 + n/4 - 3) + t_{cor} \\ &\approx t_{cycle}(n/4 - 1) \end{aligned}$$

while under a cache miss,

$$T_{div\,M} = t_{cycle}(n/4) + t_{cor} \approx t_{cycle}(n/4 + 1)$$

As shown in Table 1, our method provides a modest decrease in latency, and decrease the cache entry size. We compute the latency as

$$T_{div} = (hit\,rate)T_{div\,H} + (1 - hit\,rate)T_{div\,M}$$

for those algorithms which use caching.

3 Experimental Results

In this section, we investigate the hit rates of prefix caches. We traced the floating-point division operations from several applications, using this data as input to our cache simulator. Our choice of benchmark applications consists of two multimedia benchmarks, MediaBench [5] and Khoros [4].

3.1 Available Operand Redundancy

First we investigate the amount of available redundancy. A simulation of a fully associative cache with infinite size was created. Hit rates are shown in Figure 2. Single-precision floating-point instructions were traced. Application dependent behavior may cause decreases in hit rates as the prefix length increases. The Rasta application has few floating-point instructions and performed poorly. For an infinite cache, hit rates increase as we decrease the prefix.

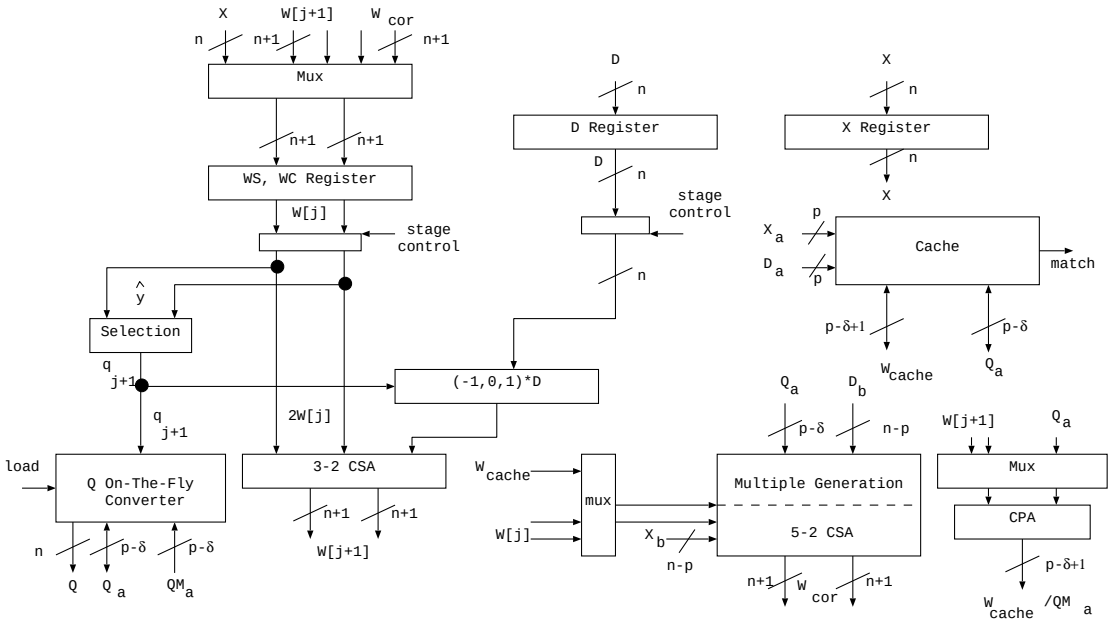

Figure 1: Block diagram of radix-2 division with partial caching

3.2 Hit Rates on Real Caches

Figure 3 shows the hit rates for a variety of caches on the "MesaTexgen" benchmark. We vary both the prefix and the number of cache entries, and examine the hit rate. A fully associative cache was used. For prefix lengths greater than 10-12 bits, the cache hit rates do not decrease further. Hit rates range from 60% to 80%, depending upon cache size. As we increase the cache size by a power of 2, the hit rate increases by approximately 5%.

In Figure 4, we show the effects of associativity upon the hit rate. This is for the "MesaTexgen" benchmark, with a cache containing 2^9 entries. Changing the associativity does not have a large impact on the hit rate for this particular application.

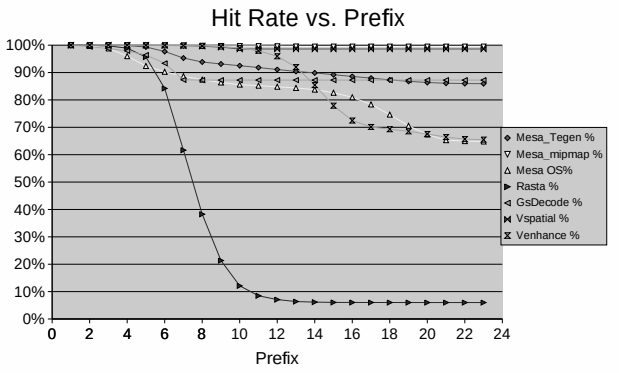

Figure 2: Hit rate vs. Prefix (bits) for an infinite, fully associative cache

3.3 Speedup

We show the speedup obtained for radix-2 in Figure 5. If one is not concerned about area, the greatest speedup will be obtained from using a larger cache size, and by caching in the full precision. However, if area is limited, one may choose an appropriate curve from Figure 5. Given a fixed increase in cache area, it is better to increase the number of entries than to increase the prefix.As one moves to a larger prefices, greater increases in speedup occur for larger caches. But for small prefices, small caches are adequate. There is a break-even point when the prefix is 8 bits.

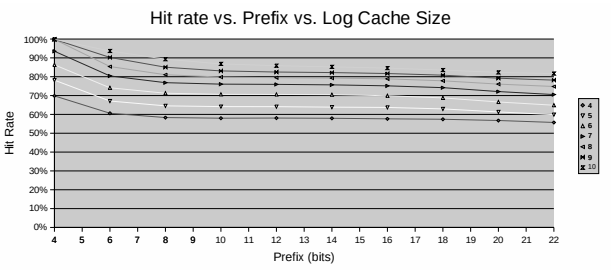

Figure 3: Hit rate vs. Prefix vs. Log cache size

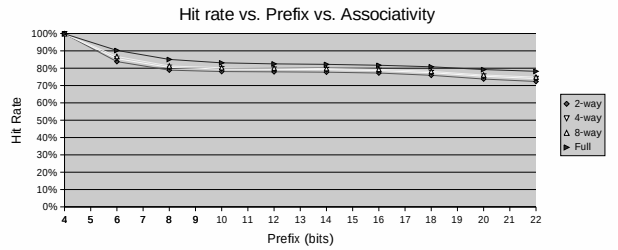

Figure 4: Hit rate vs. Prefix vs. Associativity

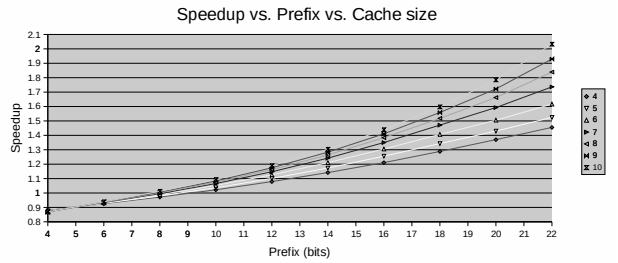

Figure 5: Speedup vs. Prefix vs. Log cache size

4 Conclusion

4.1 Summary

We have proposed and investigated the use of partial caching in reducing the latency of division operations. We allow the designer more flexibility, since the designer may chose the number of quotient digits to cache, based on the characteristics of a specific application. We have demonstrated that our digit-recurrence implementation of partial caching allows a modest speedup, with a smaller cache entry size than full caching. Depending upon the prefix, speedups can range from 1.1 with a prefix of 10 bits to a speedup of 2 with a prefix of 22 bits.

We have also observed the hit rates of partial caching. Hit rates tend to level off for prefices greater than 10-12. Despite this fact, increasing the prefix will lead to an increase in speedup, at the cost of additional area. The designer can choose the appropriate prefix for a given application and set of area constraints.

4.2 Limitations

Increases in cache hit rates only occur for $p < 12$ bits. Because of this property, partial caching is more appropriate for use with single-precision floating-point values. If one cannot achieve a high hit rate, partial caching may be slower than uncached division. Because caching depends upon the application, we do not recommend partial caching for a general purpose processor.

Partial caching imposes additional constraints on the choice of radix. Only $p - \delta$ quotient digits may be cached. As one increases the radix, the number of bits in δ becomes larger, and one can cache fewer quotient digits. Also, the on-line selection function becomes increasingly complex.

Acknowledgments

This work was supported in part by the State of California and the University of California under the MICRO research grant #01-030 "Reducing Latency of Division Operation."

References

[1] D. Citron, D. Feitelson, and L. Rudolph, "Accelerating Multi-Media Processing by Implementing Memoing in Multiplication and Division Units". Proc. 8th International Conference on ASPLOS, pp. 252-261, 1998.

[2] D. Citron, *Instruction Memoization: Exploiting Previously Performed Calculations to Enhance Performance*. Ph.D. Thesis, School of CS and Engineering, Hebrew University of Jerusalem, 2000.

[3] M.D. Ercegovac and T. Lang, "On-Line Arithmetic: A Design Methodology and Applications in Digital Signal Processing". Proc. IEEE Workshop on VLSI Signal Processing III, pp. 252-263, 1988.

[4] http://www.khoral.com

[5] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, "MediaBench". Proc. 30th International Symposium on Microarchitecture, 1997.

[6] S.F. Oberman and M.J. Flynn, "On Division and Reciprocal Caches". Technical Report No. CSL-TR-95-666, Computer Systems Laboratory, Stanford, April 1995.

[7] S.F. Oberman and M.J. Flynn, "Reducing Division Latency with Reciprocal Caches". *Reliable Computing*, 2(2) 147-153, 1996.

[8] S. Richardson, "Exploiting Trivial and Redundant Computation". Proc. 11th IEEE Symposium on Computer Arithmetic, pp. 220-227, 1993.

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