Accelerating Multi-Media Processing by Implementing Memoing in Multiplication and Division Units*

Daniel Citron Dror Feitelson
Department of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, Israel
E-mail:citron,feit@cs.huji.ac.il

Larry Rudolph
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
E-mail:rudolph@lcs.mit.edu

Abstract

This paper proposes a technique that enables performing multi-cycle (multiplication, division, square-root ...) computations in a single cycle. The technique is based on the notion of memoing: saving the input and output of previous calculations and using the output if the input is encountered again. This technique is especially suitable for Multi-Media (MM) processing. In MM applications the local entropy of the data tends to be low which results in repeated operations on the same datum.

The inputs and outputs of assembly level operations are stored in cache-like lookup tables and accessed in parallel to the conventional computation. A successful lookup gives the result of a multi-cycle computation in a single cycle, and a failed lookup doesn't necessitate a penalty in computation time.

Results of simulations have shown that on the average, for a modestly sized memo-table, about 40% of the floating point multiplications and 50% of the floating point divisions, in Multi-Media applications, can be avoided by using the values within the memo-table, leading to an average computational speedup of more than 20%.

1 Introduction

Many of the time-consuming machine instructions in Multi-Media-based applications are repeatedly applied to the same operands, and so they can be eliminated by recording the results of these operations the first time they are performed and replacing the operation with a table-lookup. Instructions such as multiplication, division, and square-root that usually complete in multiple cycles can be made to complete in a single cycle when certain conditions are met.

Many mathematical functions are computed in hardware using iterative algorithms[1] that by their nature are time consuming. Table 1 show the cycle times for floating point multiplication and division on several processors. It is important to note that these numbers are the latencies for the given instructions; the multiplication unit is pipelined itself, which leads to a throughput of one cycle for consecutive multiplication instructions. However, since none of these processors pipeline their division units, their execution can "throw a wrench" in the execution pipeline by introducing structural and data hazards and by resulting in *out-of-order completion*[1].

	Multiplication	Division
Pentium Pro [2]	3	39
Alpha 21164 [3]	4	31
MIPS R10000 [4]	2	40
PPC 604e [5]	5	31
UltraSparc-II[6]	3	22
PA8000 [7]	5	31

Table 1: *Cycle times of leading microprocessors*

We propose to mitigate the effect of floating point divisions by reducing their frequency via a *memoing*[8] technique: The input (operands) and output (result) of particular instruction types are stored into a cache-like lookup table. The table is accessed in parallel to the conventional computation. A successful lookup gives the result of a multi-cycle computation in a single cycle, and a failed lookup doesn't necessitate a penalty in computation time. Figure 1 shows a schematic layout of the idea. The operands are forwarded in parallel both to a division unit and its adjacent MEMO-TABLE.

This paper restricts its attention to Multi-Media due to the low local entropy of values displayed in the data sets of these applications and the nature of Multi-Media applications where computations are performed on local areas of an image or signal.

The rest of this section will overview related work (1.1). The next section describes the MEMO-TABLE. Section 3 summarizes the experimental results that supporting the use of memoing. Section 4 summarizes the results and suggests some future directions.

1.1 Related Work

The concept of *memoing* was introduced by Michie[8]. The idea is to save the inputs and results of side-effect-free functions in a table and reuse the results for matching inputs. Since then it has been used mainly in the context of declarative languages like Prolog, Lisp, and ML [9, 10, 11].

*Funding for this work is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Ft. Huachuca contract DABT63-95-C-0150, for work done at Lab for CS at Massachusetts Institute of Technology and in part by the Israeli Ministry of Science for work done at Hebrew University.

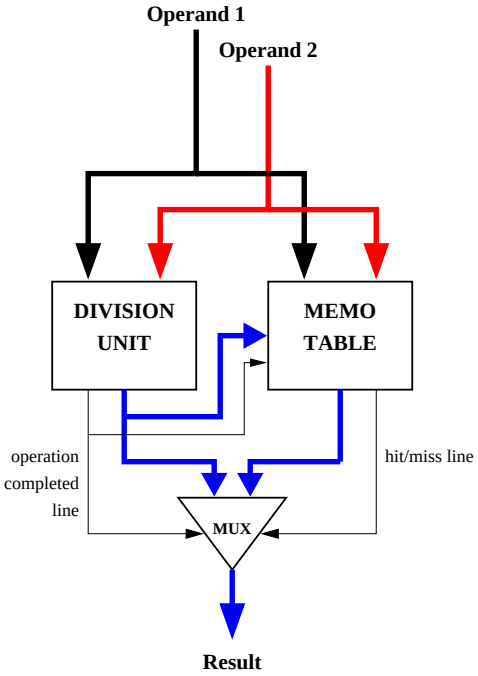

Figure 1: *A division unit using a* MEMO-TABLE

In computer arithmetic "Look Up Tables" are used in division and square-root [12]. For instance high-radix SRT[13] division uses a static table with predefined values to "guess" the next bits of the quotient (in fact this table caused the infamous "Pentium fdiv bug"). Our work isn't directly related to computer arithmetic in the sense that we don't propose any new algorithms or techniques for accelerating arithmetic computations. Where successful, our technique enables the processor to bypass these computations altogether.

The idea of exploiting redundant computation was introduced by Stevens [14] and further expanded by Flynn and Oberman [15]. Their simulations were performed on the SPEC, Perfect and NAS benchmarks [16, 17]. Our tests revealed that other families of applications produce better results, specifically Multi-Media with an emphasis on applications utilizing Image Processing.

Sodani and Sohi have introduced the concept of Dynamic Instruction Reuse[18], where all instructions executed are inserted into a table called the Reuse Buffer (RB). If the address of the instruction being fetched matches the address of an instruction in the RB and the operands stored match the current operands, the instruction isn't executed and the values in the RB are used instead (except in the case of memory stores). Our scheme differs in several ways. First, we only record certain types of instructions, thus it is less likely for multiple-cycle instructions to be bumped out of the RB by single-cycle instructions. Second, we do not care about the address of the instruction, rather the instruction itself. Thus, if the compiler unrolls a loop, our scheme will have more hits.

2 Accelerating Computations Using MEMO-TABLES

This section describes how the MEMO-TABLES accelerates multi-cycle operations. A stand-alone MEMO-TABLE is first presented and followed by a description of how a MEMO-TABLE can work in tandem with a Computation Unit (CU). The section ends with showing how an enhanced CU can be incorporated into an execution pipeline and a small discussion of the cycle time and die size of a MEMO-TABLE.

2.1 The MEMO-TABLE

A MEMO-TABLE is a cache-like Look Up Table (LUT), that receives an index into the table and returns the value in that position. The likeness to a cache is due to the fact that the values in the LUT change over time with the most recently used values present in the MEMO-TABLE.

Just like in a conventional cache when a value is forwarded to the MEMO-TABLE, a subset of its bits are used to form an index into the LUT. The remaining bits (or some subset of them) are compared to a tag that is sitting in the indexed entry. If they match, we say that we have a "hit" and the value sitting in the entry is returned. If they do not match, we say that we have a "miss" and no value is returned. Other cache-like properties like overwriting an entry that has returned a miss and having an input value compared to more than one entry (using associative sets) are used in the design of MEMO-TABLES. Specifically we suggest MEMO-TABLES of 32 entries and 4-way associativity.

Unlike a conventional cache where each line contains more than one word and relatively small associated tag, the MEMO-TABLE contains a large tag and just the one word result in each line. To emphasize this distinction, we shall use *entry* instead of the traditional *line*. In most cases the tag will be larger than the value stored (the entry) due to the fact that our input values are the two operands of a binary function while the output value is its unary result.

We have studied two variations. First, it is not necessary to enter the full floating point value of the operands into a MEMO-TABLE. Since floating point computations are computed separately on the mantissa and exponent, one alternative is to just store the mantissa of the operands. Second, it isn't necessary to store all instances. Trivial operations like multiplying by 1 or 0 and dividing by 1 or dividing 0 need not be computed at all, the MEMO-TABLE can detect them and forward the result immediately.

2.2 MEMO-TABLES in the Computation Units

The execution stage (EX) of the CPU pipeline is performed by various units. The opcode of the instruction defines which execution will be used, such as the integer ALU, the integer multiplier, the floating point adder etc. Instructions that execute in more than one cycle stay in the EX stage until the computation is completed.

A MEMO-TABLE is added to each computation unit that takes multiple cycles to complete. The Instruction Decode (ID) stage forwards the operands to the appropriate computation unit and in parallel, to the corresponding MEMO-TABLE. If there is a hit in the MEMO-TABLE, its value is forwarded to the next pipeline stage (the write back (WB) stage), the computation unit is aborted and signals it is free to receive the next set of operands. If there is a miss from the MEMO-TABLE, the computation is allowed to complete, the result obtained forwarded to WB stage and in parallel entered into the MEMO-TABLE.

The MEMO-TABLE does not increase latency along the critical path: The calculation and the lookup are performed in parallel. Updating the MEMO-TABLE with the new result in the case of a miss is also done in parallel with the forwarding of it to the WB stage. Thus on the next cycle a new lookup/computation can be performed.

In the cases where a CU computes a commutative operation (addition, multiplication) the lookup in the MEMO-TABLE must compare the operands both in the order as specified in the instruction and in their reverse order. Although this waste of space could be avoided by, say, first sorting the operands, this extra computation would introduce extra latency.

2.3 The MEMO-TABLE in the pipeline

The use of MEMO-TABLES in the CUs that implement the EX stage of the pipeline can greatly accelerate the speed in which multi-cycle instructions complete, and thus reduce the number of occurrences of *out-of-order completions*. Unfortunately, a compiler or run-time scheduler sometimes expects an instruction to complete in multiple cycles. Since with our technique it may complete much sooner than expected, there is no instruction that uses the same CU that is ready to be issued. This problem is compounded in multiplication units which are themselves pipelined in order to achieve high throughput. New compiler design and run-time scheduling is beyond the scope of this work.

In the case where a processor implements several instances of the same CU, having a MEMO-TABLE adjacent to each CU could degrade performance. Recurring calculations might be dispatched to different CUs and thus be calculated more than once and reside in more than one MEMO-TABLE. The solution is to create a larger multi-ported MEMO-TABLE that will be shared by the above CUs enabling one CU to take advantage of work performed by another.

It is possible to extend this concept and use MEMO-TABLES not only in tandem with computation hardware but as CUs themselves. Instead of having, for instance, two floating point dividers, only one will be integrated and the second will be an interface to a multi-ported MEMO-TABLE in the division unit. In the case where two fp divisions are issued together, the second one is issued to the MEMO-TABLE interface. In the case of a miss it will be stalled until the divider is free and then issued to it. Since a MEMO-TABLE is much smaller than a divider that incorporates the high-radix SRT [13] technique, it is possible for VLIW and superscalar processors to increase their issue rate by using MEMO-TABLES.

2.4 Cycle time & Die size

The cycle time of a MEMO-TABLE lookup is comparable to that of a cache lookup. An 8K cache with a line size of 32 bytes contains 256 entries, first level on-chip caches are reaching sizes of up to 64K[4, 5]. Our experiments show that just a 32 entry MEMO-TABLE is sufficient and so addressing an entry should take less time than in a conventional cache. However, the size of a tag in a cache is smaller than the size of a MEMO-TABLE tag. While a cache's tag can be up to 64 bits a MEMO-TABLE tag is composed of 2 double precision numbers, 128 bits. But due to the fact that the MEMO-TABLE comparator performs its operand comparisons in parallel, we assume that a MEMO-TABLE lookup should take one cycle, on par with most on-chip caches.

The size of a MEMO-TABLE can be compared to the size of a conventional cache. A 32 entry MEMO-TABLE holds $32 \times 3 = 96$ double precision values which is $96 \times 8 = 768$ bytes. With second-level caches being integrated on-chip[3], adding several MEMO-TABLES should be possible without draining resources meant for other units.

The size of a MEMO-TABLE is even smaller than the size of some computation units. For instance the SRT divisor on the Pentium has a 2048 entry lookup table (although only 1066 of them are used), each entry can be any of 5 values so the lookup table alone takes up one KiloByte.

3 Experiments and Results

To verify the usefulness of the MEMO-TABLE technique, we performed a series of experiments with an architecturally detailed simulator: Shade [19] a SPARC (versions 8 & 9) instruction level simulator. Shade receives as input a binary executable and executes it natively on a SPARC compatible processor. Statistics are collected by breaking on specific instructions and storing register values in software simulated MEMO-TABLES. Statistics of all multiplication (both integer and fp) and division instructions were collected. In addition the frequency breakdown of all instructions in the benchmarks were collected.

The two indicators that measure the success of the MEMO-TABLE technique are:

Hit Ratio The hit ratio of a MEMO-TABLE will show how many multiple cycle operations were avoided. A higher hit ratio implies that less instances of multiple cycle operations are performed. This ratio is measured by the simulations in Section 3.1 and the results are shown in Section 3.2.

Speedup The end goal of using MEMO-TABLES is to accelerate processing; if the enhancement has no impact on performance, the extra complexity of adding it isn't worth the effort. Section 3.3 measures the speedup of applications using MEMO-TABLES.

Naturally, the hit-ratio and speedup depend on the specific design of the MEMO-TABLE. The larger the LUT, the better the expected hit-ratio & speedup.

3.1 Simulations & Traces

The hit ratio is a function of the size of the MEMO-TABLE, its associativity, and the number of bits stored (full value or mantissa in fp numbers). We have simulated MEMO-TABLE with as its size varying from 8 to 8K entries and the spectrum of associativity from direct mapped to 8-way associativity. We have also ran the benchmarks through an "infinitely" large fully associative MEMO-TABLE for comparison.

The floating point MEMO-TABLES are simulated both with the whole value and with only the mantissa. Integer operands are hashed by by performing an exclusive or (XOR) on the n least significant bits of the two operands (where n is the number of sets in the MEMO-TABLE). For floating point operations, the n most significant bits of the mantissas of both operands are XORed in order to receive an index into the MEMO-TABLE.

ADM	Air Pollution, fluid dynamics
QCD	Lattice gauge, quantum chromodynamics
MDG	Liquid water simulation, molecular dynamics
TRACK	Missile tracking, signal processing
OCEAN	Ocean simulation, 2-D fluid dynamics
ARC2D	Supersonic reentry, 2-D fluid dynamics
FLO52	Transonic flow, 2-D fluid dynamics
TRFD	2-electron transform integrals, molecular dynamics
SPEC77	Weather simulation, fluid dynamics

Table 2: *Description of the Perfect Benchmark applications*

tomcatv	Vectorized mesh generation
swim	Shallow water equations
su2cor	Monte-Carlo method
hydro2d	Navier Stokes equations
mgrid	3d potential field
applu	Partial differential equations
turb3d	Turbulence modeling
apsi	Weather prediction
fpppp	Gaussian series of quantum chemistry
wave5	Maxwell's equation

Table 3: *Description of the SPEC CFP95 applications*

The simulated system consists of MEMO-TABLES adjacent to the integer multiplier, fp multiplier and fp divider. The traces were taken from three sources: The first two are the Perfect Benchmarks and SPEC CFP95 benchmarks (the floating point component of the SPEC CPU95 suite) [16, 17] These applications are described in tables 2 and 3. The third is the Khoros development environment [20] that consists of a suite of Image Processing (IP) and Digital Signal Processing (DSP) applications. These applications showed much higher hit ratios than the other applications, thus our targeting of Multi-Media applications. The specific applications are described in table 4. Each application was run on 8 to 14 inputs.

vspatial	Statistical spatial feature extraction
vcost	Surface arc length from a given pixel.
vslope	Slope and aspect images from elevation data.
vsqrt	Square root of each pixel.
vdiff	Differentiation using two N×N weighted ops.
vdetilt	Best-fit plane subtracted from the image.
vgauss	Generates Gaussian distributions.
venhance	Local transformation (mean & variance).
vgef	Edge detection.
vwarp	Polynomial geometric transformation (warp).
vrect2pol	Conversion of rectangular to polar data.
vmpp	2-D information from COMPLEX images.
vbrf	Band-reject filtering in the frequency domain.
vbpf	Band-pass filtering in the frequency domain.
vsurf	Surface parameters (normal and angle).
vkmeans	Kmeans clustering algorithm.
vgpwl	Two dimensional piecewise linear image.
venhpatch	Stretches contrast based on a local histogram.

Table 4: *Description of MM applications*

3.2 Results

The basic configuration of a MEMO-TABLE that we have chosen is one with 32 entries arranged in 8 rows with set associativity of 4. Floating point numbers are stored fully. Tables 5 and 6 show the results of the general scientific benchmarks and Table 7 shows the results of the Multi-Media applications. We compare the results of using an "infinitely" large fully associative MEMO-TABLE to the results of using a much smaller 32 entry 4-way associative MEMO-TABLE. All suites show a large potential for data reuse but only the MM suite can scale down to a size and associativity that are practical. The numbers exclude all trivial operations.

The low hit ratios on the Perfect and SPEC suites may be explained by the work of Franklin and Sohi [21]. A *register instance* is defined as each time a datum is written into a register. Reads to that register use that *register instance*. Franklin and Sohi show that for the SPEC [16] benchmarks, a large number of register instances are used only once and the average use being about 2. Most of the register instances are replaced with a new datum within 30-40 instructions.

On the other hand, the much higher hit ratios of MM applications can be understood by considering the entropy of the data values of the images. Table 8 describes the input images along with some of their characteristics (size, type, number of bands), their entropies and the averages of the hit ratios for the applications that used the image as an input. The lower the entropy, the higher the hit-ratio.

The entropy of an image is related to the amount of information it contains. An image can be described using fewer bits when its entropy is lower. The entropy of an image is calculated by the following equation:

$$E = -\Sigma_{k=1}^{L} p_k * \log_2(p_k)$$

Where L is the number of possible values of each pixel and p_k is the probability of the value appearing in the image. This probability is calculated by the histogram of the image. Thus, an image in which each pixel represents a level of grey between 0 to 255 and the values are evenly distributed throughout the image will have an entropy of:

$$-\Sigma_{k=1}^{256} 1/256 * \log_2(1/256) = \Sigma_{k=1}^{256} 1/256 * -8 = 8$$

In most cases the distribution is not evenly distributed, yielding an entropy of less than 8. When looking at small images or at windows of an image (16×16 and 8×8 pixels per window) most values have a probability (p_k) of 0 so the entropies are even smaller. Thus the number of different pixel values in a small area is low, this leads to our belief that the same calculations are being performed over again.

Figure 2 shows the relationship between hit-ratio and entropy [22]. Specifically, the hit-ratios of floating point division and multiplication are plotted against a parameter of the entropies of 8×8 windows and of whole images. Although the actual points are all over the graphs, we have also drawn a best-fit line (nonlinear least squares fitting using the Marquardt-Levenberg Algorithm) to show that, on average, for each bit of entropy a 5% decrease in the hit-ratio is observed. In other words the lower the entropies the higher the hit ratios, this indicates that the same calculations are being performed over and over in a localized area.

Storing only the mantissas of floating point numbers raises the hit ratios, albeit not by much (Table 10). On the other hand, in order to store only mantissas, the MEMO-TABLE has to be capable of computing the results' exponent and normalize the results' mantissa if necessary. This tradeoff between simplicity of design and enhanced hit-ratio has to be made when implementing a MEMO-TABLE. In all our subsequent experiments, the full floating point value is stored in the MEMO-TABLE.

In our experiments, we differentiate between "trivial" and "non-trivial" operations. Although entering trivial operations into the MEMO-TABLE might raise the hit ratio, these operations can complete in a few cycles anyhow. On the other hand, more calculations are being entered into the MEMO-TABLE which can lower the hit-ratio. Table 9 shows examples of both these behaviors. Trivial operations usually have shorter latencies so not entering them into the MEMO-TABLE enables non-trivial computations with long latencies to reside longer in the MEMO-TABLE and have a better chance of being reused.

In order receive the best results, trivial operations should be detected before being forwarded to a MEMO-TABLE and

application	*32 entries*			*"infinite"*		
	int mult	*fp mult*	*fp div*	*int mult*	*fp mult*	*fp div*
ADM	.98	.13	.15	.99	.41	.56
QCD	.02	.00	.00	.07	.04	.00
MDG	-	.00	.02	-	.04	.03
TRACK	.98	.17	.09	.99	.46	.89
OCEAN	.15	.03	.03	.99	.30	.99
ARC2D	.94	.15	.23	.99	.45	.26
FLO52	.86	.02	.06	.97	.11	.20
TRFD	.60	.18	.85	.99	.59	.99
SPEC77	.06	.28	.01	.97	.37	.15
average	**.57**	**.11**	**.16**	**.70**	**.31**	**.45**

Table 5: *Hit ratios for the Perfect benchmarks, LUT has 32 entries in sets of 4, or is infinitely large and associative (a '-' indicates operations that don't appear in this application).*

application	*32 entries*			*"infinite"*		
	int mult	*fp mult*	*fp div*	*int mult*	*fp mult*	*fp div*
tomcatv	.14	.01	.00	.99	.16	.00
swim	-	.16	.00	-	.93	.74
su2cor	.26	-	-	.99	-	-
hydro2d	.15	.75	.78	.98	.97	.97
mgrid	.83	.00	-	.99	.01	-
applu	.97	.25	.25	.99	.66	.64
turb3d	.80	.16	.03	.99	.86	.99
apsi	.95	.16	.13	.99	.39	.57
fpppp	.53	.29	.15	.99	.55	.62
wave5	-	.05	.02	-	.11	.16
average	**.58**	**.20**	**.17**	**.99**	**.52**	**.59**

Table 6: *Hit ratios for the SPEC CFP95 benchmarks, LUT has 32 entries in sets of 4, or is infinitely large and associative.*

application	*32 entries*			*"infinite"*		
	int mult	*fp mult*	*fp div*	*int mult*	*fp mult*	*fp div*
vdiff (sobel)	.49	.54	-	.96	.99	-
vcost	.99	.34	.44	.99	.81	.93
vgauss	-	.50	.79	-	.87	.95
vspatial	.61	.62	.94	.92	.99	.99
vslope	.34	.15	.25	.99	.60	.83
vgef	.37	.33	-	.99	.99	-
vdetilt	-	.23	-	-	.46	-
vwarp	.27	.57	.38	.99	.63	.68
venhance	-	.57	.12	-	.96	.47
vrect2pol	-	.42	.61	-	.97	.80
vmpp	-	.41	.56	-	.89	.98
vbrf	.72	.01	.05	.99	.64	.88
vbpf	.72	.54	.52	.99	.52	.80
vsurf	.48	.25	.33	.93	.65	.83
vgpwl	-	.50	.58	-	.99	.99
venhpatch	.99	.68	-	.99	.99	-
vkmeans	-	.39	.58	-	.99	.97
average	**.59**	**.39**	**.47**	**.95**	**.82**	**.85**

Table 7: *Hit ratios for Multi-Media applications, LUT has 32 entries in sets of 4, or is infinitely large and associative.*

their results returned immediately. Table 9 shows that integrating this technique within a MEMO-TABLE gives the highest hit-ratios. We decided to focus on non-trivial operations as it isn't clear what speedup is obtained by not performing trivial operations. So except for the experimental results in Table 9, all the experiments cached only non-trivial operations.

The final two experiments performed test the attributes of the LUT itself, its size and associativity. Figure 3 shows the average hit-ratios (min-max results shown by the vertical lines at each LUT size) of the MM applications when the size of the LUT ranges from 8 to 8192 entries, and its associativity is 4. Figure 4 shows the average hit-ratio (min-max results shown by the vertical lines at each set size) when the associativity ranges from direct mapped to 8-way associativity. In both these experiments five sample Multi-Media applications were used (vcost, venhance, vgpwl, vspatial & vsurf).

Figure 3 shows that it is possible to use different size MEMO-TABLES for different Computing Units. While for a fp division unit a size 8 MEMO-TABLE may be sufficient, a MEMO-TABLE adjacent to a fp multiplier must be of size 32 at least. This shows that repeating division operations are performed closer together and aren't "bumped" out of the

image	*characteristics*			*entropy window size*			*hit ratios*		
	size	*type*	*bands*	*full*	*16×16*	*8×8*	*imul*	*fmul*	*fdiv*
mandrill	256×256	BYTE	1	7.34	6.03	5.10	.31	.30	.29
nature	256×256	BYTE	1	7.38	5.64	4.72	.31	.34	.35
Muppet1	240×256	BYTE	1	7.04	4.78	4.16	.31	.45	.50
guya	128×128	BYTE	1	6.99	4.77	3.91	.36	.76	.37
star	158×158	BYTE	1	5.93	5.22	4.62	.96	.32	.33
chroms	64×64	BYTE	1	4.82	4.04	3.29	.58	.43	.40
airport1	256×256	BYTE	1	4.47	3.15	2.56	.31	.46	.45
lablabel	243×486	INTEGER	1	3.37	0.93	0.84	.93	.66	.75
fractal	450×409	BYTE	1	1.42	0.78	0.58	.88	.61	.82
head	228×256	FLOAT	1	-	-	-	.39	.29	.33
spine	228×256	FLOAT	1	-	-	-	.39	.27	.32
lenna.rgb	480×512	BYTE	3	7.75	6.84	6.25	.19	.35	.58
mandril.rgb	480×512	BYTE	3	7.75	6.22	5.64	.36	.36	.52
lizard.rgb	512×768	BYTE	3	7.60	5.66	5.17	.32	.40	.60

Table 8: *Description of the images used in IP applications.*

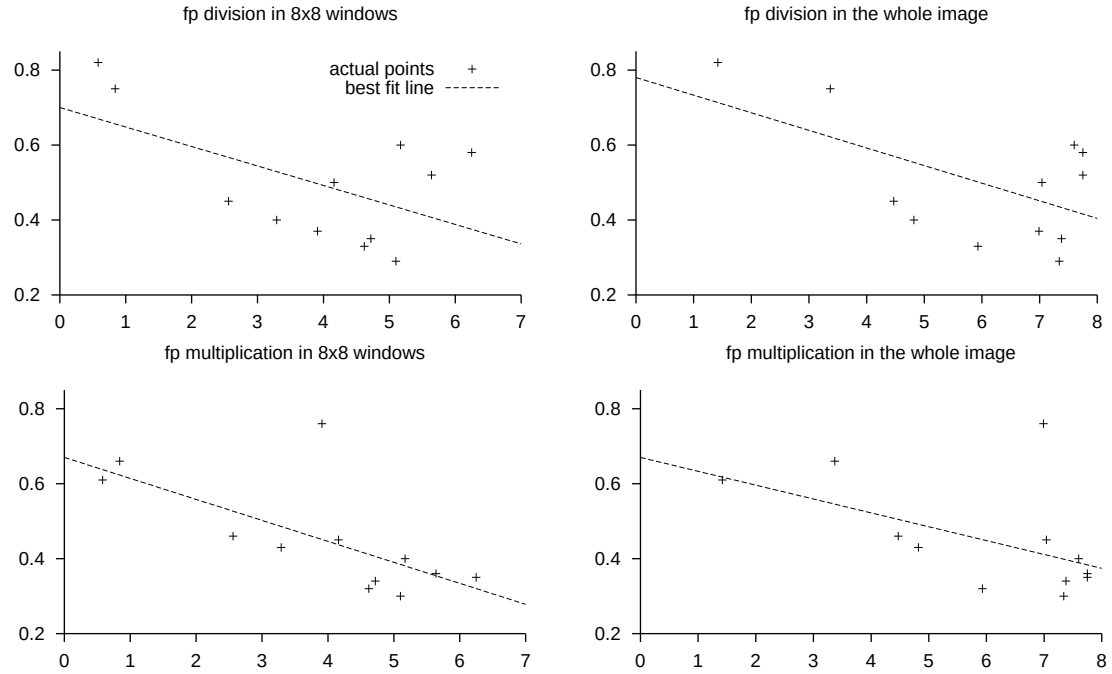

Figure 2: *Hit Ratios (vertical axis) of fp division and multiplication as a parameter of Entropy (horizontal axis) in 8×8 windows and whole images*

suite	*fp mult*		*fp div*	
	full	*mant*	*full*	*mant*
Perfect	.11	.11	.16	.17
Multi-Media	.39	.43	.47	.50

Table 10: *Comparison of storing only mantissa's or the whole floating point number (averages of 32 entry 4-way associative* MEMO-TABLES*).*

MEMO-TABLE as often as fp multiplication operations. The figures show as well that performance improves up to about 1024 entries. More entries hardly improves the hit ratio as can be seen by the curve that flattens out in the above interval. Many more entries are needed for further improvements as shown using "infinite" sized MEMO-TABLES.

Figure 4 shows that both for fp division and multiplication a set size of over 4 hardly improves the hit ratio. In fact, a set size of 2 suffices for division. These results shows that a MEMO-TABLE with 16 entries and an associativity of 2 gives results almost as good as a 32/4 MEMO-TABLE. In fp multiplication a 32/4 is the minimum required. It would be simpler to let the MEMO-TABLE be direct-mapped and save on the space taken by a set of comparators but Figure 4 shows that conflict misses lower the hit ratio in both MEMO-TABLES. The conflicts are caused by the hashing scheme, in some applications (vsqrt, vcost, vgauss) nearly identical values are entered into the MEMO-TABLE alternately causing a conflict miss on every lookup. A set size of 2 avoids this problem.

3.3 Speedup

Amdahl's law [1] states that the speedup obtained by using an enhancement depends on two factors:

application	*int mult*				*fp mult*				*fp div*			
	trv	*hit ratios*			*trv*	*hit ratios*			*trv*	*hit ratios*		
	%	*all*	*non*	*intgr*	*%*	*all*	*non*	*intgr*	*%*	*all*	*non*	*intgr*
vdiff	.34	.48	.49	.67	.62	.63	.54	.81	-	-	-	-
vcost	.66	.62	.99	.99	.20	.30	.34	.43	.00	.44	.44	.44
vgauss	-	-	-	-	.23	.39	.50	.60	.00	.23	.23	.23
vspatial	.61	.71	.61	.87	.06	.57	.62	.65	.00	.94	.94	.94
vslope	.53	.54	.34	.70	.44	.22	.15	.50	.15	.28	.25	.30
vgef	.34	.43	.37	.58	.23	.41	.33	.48	-	-	-	-
vdetilt	-	-	-	-	.04	.26	.23	.27	-	-	-	-
venhance	-	-	-	-	.15	.52	.57	.57	.00	.12	.12	.12
average	**.50**	**.55**	**.56**	**.76**	**.25**	**.41**	**.41**	**.54**	**.03**	**.40**	**.40**	**.40**

Table 9: *Hit ratios for several Multi-Media applications, LUT has 32 entries in sets of 4. This table shows:*
trv *the ratio between trivial operations and all operations*
all *the hit ratio when all operations (trivial and non-trivial) are stored in a* MEMO-TABLE
non *the hit ratio when ony non-trivial operations are stored in a* MEMO-TABLE
intgr *the hit ratio when checking for trivial ops is integrated into the* MEMO-TABLE*, and only non-trivial operations are stored in the* MEMO-TABLE*(trivial operations are counted as "hits")*

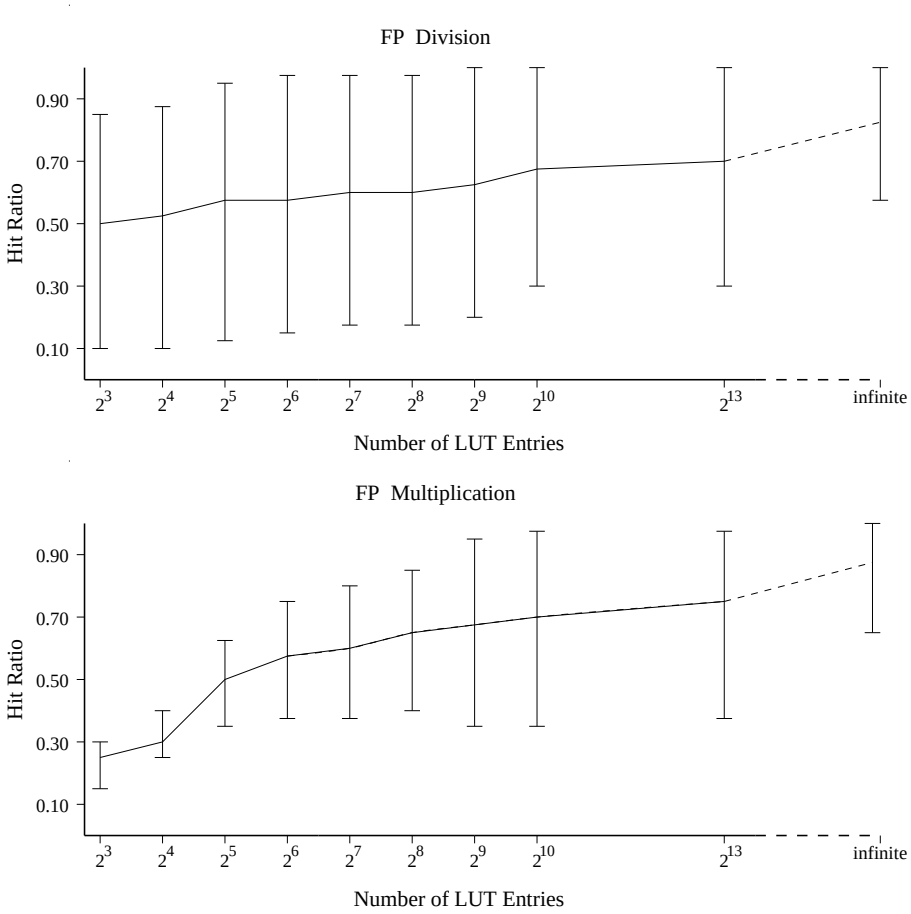

Figure 3: *Hit ratios of floating point division and multiplication in MM applications as a function of the LUT size (set size is 4).*

1. The fraction of computation time in the original machine that can use the enhancement. This is called *Fraction Enhanced (FE)*, it is always smaller than 1.
2. The improvement gained if *only* the enhancement mode could be used. This is called the *Speedup Enhanced(SE)*, it is always greater that 1.

The new execution time when using the enhancement is:

$$T_{new} = T_{old} * ((1 - FE) + FE/SE).$$

Taking fp division as an example, SE is equal to:

$$\frac{dc}{(1 - hr)dc + hr}$$

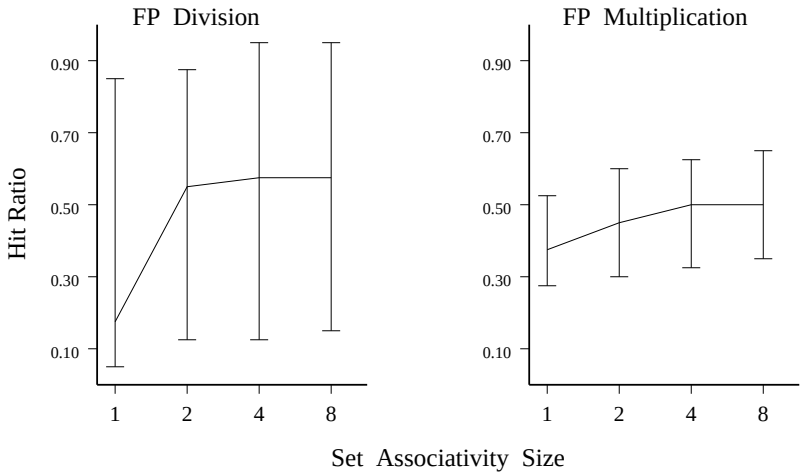

Figure 4: *Hit ratios of MM applications as a function of the LUT associativity (LUT size is 32 entries).*

app	*hit ratio*	*13 cycles*			*39 cycles*		
		FE	*SE*	*Speedup*	*FE*	*SE*	*Speedup*
venhance	.12	.036	1.12	1.00	.101	1.13	1.01
vbrf	.05	.062	1.05	1.00	.180	1.05	1.01
vsqrt	.54	.017	1.99	1.01	.049	2.11	1.03
vslope	.25	.074	1.30	1.02	.204	1.32	1.05
vbpf	.52	.089	1.92	1.04	.226	2.02	1.13
vkmeans	.58	.094	2.15	1.05	.237	2.30	1.15
vspatial	.94	.101	7.55	1.10	.252	11.89	1.30
vgauss	.79	.150	3.69	1.12	.346	4.34	1.36
vgpwl	.58	.208	2.15	1.13	.440	2.29	1.33
average	**.48**	**.092**	**2.85**	**1.05**	**.226**	**3.16**	**1.15**

Table 11: *Speedup of applications when fp division is memoized (division takes 13 or 39 machine cycles).*

app	*hit ratio*	*3 cycles*			*5 cycles*		
		FE	*SE*	*Speedup*	*FE*	*SE*	*Speedup*
venhance	.57	.061	1.61	1.02	.101	1.84	1.05
vbrf	.01	.021	1.00	1.00	.052	1.01	1.00
vsqrt	.39	.015	1.36	1.01	.025	1.45	1.01
vslope	.15	.111	1.11	1.01	.147	1.14	1.02
vbpf	.54	.026	1.56	1.01	.038	1.76	1.02
vkmeans	.39	.101	1.35	1.03	.133	1.45	1.04
vspatial	.62	.050	1.70	1.02	.071	1.98	1.04
vgauss	.50	.125	1.50	1.04	.172	1.67	1.07
vgpwl	.50	.071	1.50	1.02	.083	1.67	1.03
average	**.28**	**.057**	**1.40**	**1.02**	**.091**	**1.55**	**1.03**

Table 12: *Speedup of applications when fp multiplication is memoized (multiplication latency is 3 or 5 machine cycles).*

where *dc* is the number of machine cycles it takes to perform a division and *hr* is the hit ratio in the MEMO-TABLE. *FE* is equal to the number of cycles used by division instructions divided by the cycle count of the application. In order to compute the total cycle count of an application vs. the number of cycles of multiplication and division instructions the simulator was enhanced to incorporate a memory hierarchy of two caches and take into account annulled instructions in the pipeline. Thus the indicator of speedup is total cycle count executed by all instructions. Enhancements like multiple issue and pipelining aren't taken into consideration at this point. This enables us to bypass the problem of compiler changes that have to be made in order to take advantage of MEMO-TABLES and focus on the number of superfluous cycles avoided.

Table 11 shows the MEMO-TABLE hit ratio, *Speedup Enhanced*, *Fraction Enhanced* and Speedup for the nine MM applications that use a fdiv MEMO-TABLE (32 entries in sets of 4). It is assumed that each division instruction takes 13 machine cycles or 39 machine cycles. A look at table 1 shows that no modern microprocessors even comes close to completing a double precision division instruction in less that 13 cycles and that on at least one processor it takes more that 39 cycles. The table shows an average speedup of between 5% to 15%.

Table 12 shows the same information for applications using a fmul MEMO-TABLE (32 entries in sets of 4). It is assumed that each multiplication instruction has a latency of 3 or 5 machine cycles. It is possible that some of the cycles avoided by using a MEMO-TABLE would be avoided in any case due to a pipelined multiplication unit. In our simu-

app	*3(fmul), 13(fdiv) cycles*			*5(fmul), 39(fdiv) cycles*		
	FE	*SE*	*Speedup*	*FE*	*SE*	*Speedup*
venhance	.097	1.42	1.03	.201	1.49	1.07
vbrf	.083	1.03	1.00	.232	1.04	1.01
vsqrt	.032	1.69	1.01	.074	1.88	1.04
vslope	.185	1.19	1.03	.351	1.24	1.07
vbpf	.115	1.83	1.06	.264	1.98	1.15
vkmeans	.195	1.73	1.09	.370	1.99	1.23
vspatial	.151	5.61	1.14	.323	9.71	1.41
vgauss	.275	2.70	1.21	.518	3.45	1.58
vgpwl	.279	1.98	1.16	.523	2.19	1.39
average	**.156**	**2.13**	**1.08**	**.317**	**2.77**	**1.22**

Table 13: *Speedup of applications when fp multiplication and division are memoized (latencies are 3 and 13 or 5 and 39 machine cycles).*

lations we didn't take into account the possibility that sequential multiplication instructions will be pipelined, which will cause a throughput of one cycle for each instruction. So it is possible that the speedup figures for fp multiplication are somewhat biased in our favor. But as mentioned above we focus on total cycles per application. Future work will take into account multiple units, pipelined units and multiple MEMO-TABLES. A look at table 1 shows that these latencies are consistent with most modern microprocessors. The table shows an average speedup of between 2% to 3%.

The fact that memoizing division gives better speedups than memoizing multiplication is due to the long latencies of division instructions and the benefit of avoiding some of them. This shows that future work should be integrating MEMO-TABLES into other long latency functions such as sqrt, log and the trigonometric functions.

Table 13 shows the speedups when both fdiv and fmul are memoized on two type of processors. The first has very fast floating point units that complete fp multiplication and division in 3 and 13 cycles respectively. The second is slower and completes fp multiplication and division in 5 and 39 cycles respectively. The table shows an average speedup of between 8% to 22%. Even taken at its face value a 8% speedup is comparable with the speedups attained by enhancing branch prediction and cache & TLB hit ratios [23, 24, 25].

4 Conclusions

This paper investigates a technique to reduce the average CPI of multiplication and division instructions by using the concept of *memoing*. Previous computations are stored in look up tables and access in parallel to computing an operation. If the result of a computation already resides in the table, it is obtained in a single cycle as opposed to the multiple cycles needed to perform multiplication and division.

The technique is based on the temporal locality of the data used in computations. Unfortunately not all applications show such locality. A suite of "general" benchmarks tested have shown poor hit ratios on the MEMO-TABLES. But applications that do display this locality benefit from the memoing. Image Processing & Digital Signal Processing that are used in Multi-Media applications show high hit ratios and are clearly candidates for enhanced execution using MEMO-TABLES. Our tests show that an average of 59% of the integer multiplications, 43% of the floating-point multiplications and 50% of the floating-point divisions in MM applications can be performed in a single cycle using small look up tables. These hit ratios lead to an average speedup of up to 22% in Multi-Media applications using MEMO-TABLES.

A 32 entry, 4-way associative MEMO-TABLE is comparable in size to 1KB of on-chip cache, and no machine cycles are lost in the case of an unsuccessful lookup. Thus MEMO-TABLES can be implemented in general purpose processors and not only in dedicated MM processors.

Future work will be to extend the MEMO-TABLE technique to sqrt, log, trigonometric and other mathematical functions based on the success and promise of this work. Another avenue of research is to quantify the benefits of using several MEMO-TABLES instead of duplicating functional units.

It might be said that we are "throwing hardware" at the problem and that is indeed so. The scales of integration mentioned above enable putting in the excess of 7 million transistors on chip. A large amount of these transistors are dedicated to on-chip caches which are becoming larger and larger. The R10000 and PPC 604e both have 64K of on-chip cache with the Alpha 21164 having 112K of cache in two levels on-chip. This large amount of hardware is used to bridge the growing gap between levels of the memory hierarchy. A fraction of this transistor space can be diverted to the MEMO-TABLES and enable the bridging of the gap between the cycle times of different instructions.

References

[1] Hennessy J. L. and Patterson D. A., *"Computer Architecture: A Quantitative Approach,"* Morgan Kaufmann Publishers, San Mateo CA, 1990.

[2] http://www.intel.com/design/

[3] http://www.digital.com/info

[4] http://www.sgi.com/MIPS/products/r10k

[5] http://www.mot.com/SPS/PowerPC/products

[6] http://www.sun.com/microelectronics/datasheets

[7] http://www.hp.com/wsg/strategies

[8] Michie D., "Memo Functions and Machine Learning," *Nature* 218, pp 19–22, 1968.

[9] L. Sterling and E. Shapiro, *"The Art of Prolog, 2nd Ed.",* MIT Press Cambridge MA, 1992.

[10] Abelson, H. and Sussman, G.J. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Mass. 1985.

[11] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML (Revised)* .MIT Press, Cambridge, Mass. 1997.

[12] P. Soderquist and M. Leeser, "An area/performance comparison of subtractive and multiplicative divide/square root implementations," *Proc. 12th IEEE Symp. Computer Arithmetic,* pp. 132–139, July 1995.

[13] Atkins, D.E. "Higher-radix division using estimates of the divisor and partial reminders," *IEEE Trans. on Computers* C-17:10, 925–934,1968.

[14] S. Richardson, "Exploiting Trivial and Redundant Computation", *Proc. of the 11th Symp. on Computer Arithmetic,* pp. 220–227, July 1993.

[15] S. Oberman, M. Flynn, "Reducing Division Latency with Reciprocal Caches", *Reliable Computing,* Vol 2, no. 2, pages 147–153, April 1996.

[16] Price W.J. , "A Benchmark Tutorial," *IEEE Micro,* pp. 28–43, October 1989.

[17] http://www.netlib.org/benchweb

[18] A. Sodani, G. Sohi, "Dynamic Instruction Reuse", *Proc. of the 24th Int. Symp. on Computer Architecture*, June 1997.

[19] Cmelik R. and Keppel D., *Shade: A Fast Instruction-Set Simulator for Execution Profiling*, Sun Microsystems Laboratories.

[20] D. Argiro and C. Gage, *"Khoros User's Manual,"* U. of New Mexico, 1991.

[21] M. Franklin and G.Sohi, "Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors," *Proc. of Micro 25*, pp 236–245, 1992.

[22] A. K. Jalin, *"Fundamentals of Digital Image Processing,"* Prentice Hall, Englewood Cliffs NJ, 1989.

[23] T. Yeh and Y. Patt, "A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History," *Proc. of the 20th Int. Symp. on Computer Architecture*, pp 191–201, 1993.

[24] N. Jouppi, "Cache Write Policies and Performances," *Proc. of the 20th Int. Symp. on Computer Architecture*, pp 191–201, 1993.

[25] J. Chen, A. Borg, N. Jouppi, "A Simulation Based Study of TLB Performance," *Proc. of the 18th Int. Symp. on Computer Architecture*, pp 114–123, 1991.