Hardware Memoization of Mathematical and Trigonometric Functions

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

March 26, 2000

Abstract

Memoization is saving the input and output of previous calculations and using the output if the input is encountered again. MEMO-TABLES are cache-like tables that store the operands and results of calculations that are candidates for memoization. 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. Thus even a modest success rate can entail a computational speedup.

This paper shows how the concept of memoization may be successfully used in the hardware implementations of common mathematical and trigonometric functions (log, exp, sin, cos ...) on general-purpose or application-specific floating point processors. Alternatively we introduce two new instructions to lookup and update a generic MEMO-TABLE. The use of these instructions enables the memoization of software implemented functions written by the user or existing in pre-written libraries. Integrating the new instructions into existing code is possible without altering the function code or calling conventions, with small or non-existent compiler changes, and with a minimal computational overhead.

Results of simulations have shown that hardware memoization of the software implemented common mathematical and trigonometric functions leads to a computational speedup greater than 10%. Comparison with software based memoization shows that using software entails a slowdown of 7% (0.93 speedup).

1 Introduction

The concept of memoization is as follows: saving the input and output of previous calculations and using the output if the input is encountered again. Several recent studies [1, 2, 3, 4] have shown that this notion can be applied to instruction-level calculations and be implemented on chip with modest sized tables (less than one hundred entries). Thus, if the technique works for instructions with latencies of several cycles only, it should surely work for functions with latencies of tens to hundreds of cycles. Table 1 shows the latencies in cycles of several common mathematical and trigonometric functions in the Pentium II processor[5],[1] the only processor to date to include these functions in its instruction set, and the latencies of the software implementations of the same functions[2]. The numbers lead us to believe that successful memoization will be productive. The fact that these functions are common to most scientific, engineering, and Multi-Media applications encouraged us to suggest a hardware based solution rather that a software one.

Figure 1 shows a schematic layout of the idea using a hardware-implemented square root unit as an example. The operands are forwarded in parallel both to the square root unit and its adjacent MEMO-TABLE. Whichever completes first — the MEMO-TABLE lookup or the actual computation — cancels the other and produces the result. In the case of the actual computation the result is also stored in the MEMO-TABLE for future use.

What differentiates this work from other works in the immediate field is the fact that aside from Intel all other microprocessor manufacturers don't

[1] The instructions aren't executed by dedicated functional units, they use all the processor's units and block all other instructions from issuing until they complete. The latencies are input dependent, usually inputs with longer mantissas entail a longer cycle time in computing the function.

[2] The code was taken from the gnu C library version 1.09 (glibc-1.09) and run through the simple-scalar simulator. The numbers are the average of measuring the computation time for 10,000 random double precision values.

function	*Pentium II*	*software*
Square root	70	1,700
Sine	16–126	250
Cosine	18–124	230
Tangent	17–173	320
Logarithm	22–111	196
Exponent	13–57	131
Ceiling	9–20	15
Floor	9–20	15
Power	-	473

Table 1: *Latencies of mathematical functions, in cycles*

include these functions (aside from square root taking) in their instruction sets and don't have hardware units to implement them. Therefore the framework described above cannot be applied. Instead, we propose to modify the Instruction Set Architecture (ISA) by adding two new instructions to lookup and update a generic MEMO-TABLE. These instructions provide a completely general interface to the MEMO-TABLE, and allow the compiler to use it to memoize any function it chooses, be it a library function or a user function. Even inlined functions are supported. We assume that the functions are side-effect free, this is noted by the developer and enforced by the compiler. Functions with side-effects will have to be executed in any case.

The rest of this section will overview related work. The next section describes the layout of the MEMO-TABLES and functional units, and explains our proposed additions to the ISA and their use. Section 3 presents the experimental results that support the use of memoization mathematical and trigonometric functions. Section 4 Compares the hardware based technique to a software based one. Sections 3.3 and 5 discuss and summarize the results.

1.1 Related Work

The concept of *memoization* was introduced by Michie[6]. The idea is to save the inputs and results of side-effect-free functions in a table and reuse

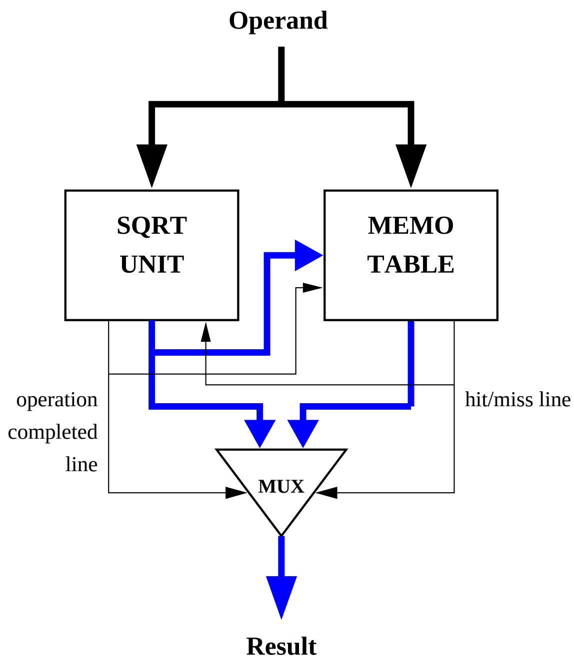

Figure 1: *A square root unit using a* MEMO-TABLE

the results for matching inputs. Until the middle 90's it had been used mainly in the context of declarative languages like Prolog, Lisp, and ML. In the past years several groups have used the concept in order to reuse the results of previously executed instructions (Harbison[7],Richardson[8], Flynn & Oberman[10], and Sodani & Sohi [1]). Other groups have focused on the aspect of value prediction in order to speculatively execute instructions that haven't had their data dependencies satisfied yet (Gabby & Mendelson[2], Lipasti, Wilkerson & Shen[3], and Sazeides & Smith[11]). Sodani & Sohi[12] discuss the differences and similarities between these two approaches. Citron, Feitelson & Rudolph[4] were the first leave the realm of SPEC and focus on Multi-Media applications and how they impact memoization.

Connors and Hwu[13] propose a general technique for reusing large regions of code which have distinct entry and exit points. If the values at the entry points match the values stored in a lookup table the results stored can be used, thus avoiding the need to recompute the region. Changes in the registers and memory locations accessed in the region invalidate the stored results. The regions are detected by a profiling-compiler which inserts new instructions that test reuse, update the table, and invalidate stored entries.

While our technique can be seen as a subset of theirs it necessitates smaller changes to the existing compiler and hardware and doesn't call for extensive profiling. The work of Calder, Feller & Eustace on value profiling will be discussed in section 4 where our hardware approach is compared to software approaches.

The difference between our work and all previous works is our focus on functions that are at this time not yet integrated into all microprocessors but rather implemented in software. Richardson[9] mentions this idea and shows results for two applications, but doesn't specify how the technique should be implemented. We show a detailed implementation of the technique and analyze the results for a specific domain.

2 Memoization of Mathematical Functions

This section describes how using MEMO-TABLES accelerates computing mathematical and trigonometric functions. A stand-alone MEMO-TABLE is first presented, followed by a description of how a MEMO-TABLE works in tandem with a Functional Unit (FU). We then show how memoization is implemented if the function is calculated in software. Finally we analyze the overhead, cycle time, and die size of implementing memoization.

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 stored in the indexed entry. If they match, we say that we have a "hit" and the value stored in the entry is returned. If they do not match, we say that we have a "miss", no value is returned and the table is updated with a new value. 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 64 entries and 4-way associativity. The justification for these values is presented in another paper[22].

Unlike a conventional cache where each line contains more than one word and a 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* or *block*.

Given that we want to memo multiple functions (at least the nine from table 1), two designs are possible. First, each function may have its own MEMO-TABLE. The index is derived from the input of the function alone and the tag contains only the input value. Second, all functions share one larger unified MEMO-TABLE. The index is now derived from the input value and the function name or address, and the tag contains both the input value and function name/address.

In our design, if the number of sets in the MEMO-TABLE is k then the bits used to index the MEMO-TABLE are the two Least Significant Bits (LSB) of the exponent and the $log(k)-2$ Most Significant Bits (MSB) of the mantissa. This is based on the extensive study of instruction MEMO-TABLES (performed in [22]) which shows that this mapping scheme yields the best utilization of MEMO-TABLE entries. Figure 2 shows the bit positions. In the case where both the function identifier and input value are used as an index the relevant bits from both are XORed together to create an index.

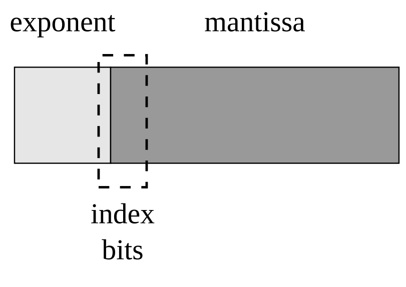

Figure 2: *The index bits are taken from the LSB of the exponent and MSB of the mantissa.*

2.2 MEMO-TABLES in the Functional Units

In this section we assume that the function is in fact an instruction (such as sqrt in most RISC microprocessors, and the rest of the mathematical and trigonometric functions in the Pentium II). We will call these instructions FUNCTION-INSTRUCTIONS. A MEMO-TABLE is added to each functional unit that performs one of the functions as in figure 1 (an alternative is to use a unified MEMO-TABLEfor all functions). The Instruction Issue stage of the pipeline forwards the operands to the appropriate functional unit, and in parallel, to the corresponding MEMO-TABLE. If there is a hit in the MEMO-TABLE, the value in the table is forwarded to the next pipeline stage, the computation taking place in the functional unit is aborted, and the unit signals it is free to receive the next set of operands. If there is a miss signal from the MEMO-TABLE, the computation is allowed to complete, and the obtained result is forwarded to the next pipeline 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 its forwarding to the next stage. Thus on the next cycle a new lookup/computation can be performed.

2.3 Memoization of Software Implemented Functions

In the previous section we assumed that there is a hardware implementation of the function on chip. In the common case where most functions are implemented in software several ISA changes must be made. Three main reasons motivated our design choice:

1. Decouple the memoization from the routine that executes the function. Not in all cases will the code be available for compilation, thus we decided to perform the lookup and update outside the routine body instead of altering the calling and return instructions to perform the lookup and update.

2. Most RISC ISAs have instruction formats of three register operands and a small (5-6 bit) immediate field (the MIPS R-format or the PPC A-Form). We will use these instruction formats for our new instructions.

3. Most of the mathematical and trigonometric functions have a single operand and single result, and a minority of them have two operands and a single result. Thus it is possible to use the same MEMO-TABLE structure used to memoize instructions[4]. The new instructions introduced support functions with one or two inputs and one output.

The ISA we will add the new instructions to is SimpleScalar[18] which is based on the MIPS instruction set. Only two new instructions (each with two variations) must be added:

- **LUPM2 (LookUP_Memo2)** - Look up a value in a MEMO-TABLE. The instruction has three operands which reside in registers and one immediate operand.

  1. IN1 - function input 1 in a register
  2. IN2 - function input 2 in a register
  3. OUT - function result in a register
  4. FID - function identifier, a 5 bit mnemonic.

  When executed the instruction uses the values in IN1, IN2 and the function identifier to index a separate or unified MEMO-TABLE (for the case of separate tables FID identifies the MEMO-TABLE and IN1 & IN2 index it, for the case of a unified table IN1, IN2 and FID index the table). If the lookup is successful the output of the function is loaded from the MEMO-TABLE into OUT and the floating point flag is set. A test instruction (such as `bc1t`) can then branch to an address beyond the function call.

- **UPDM2 (UPDate_Memo2)** - Update an entry in a MEMO-TABLE. Like **lupm2**, this instruction has three operands which reside in registers and one immediate operand.

  1. IN1 - function input 1 in a register
  2. IN2 - function input 2 in a register
  3. IN3 - function result in a register
  4. FID - function identifier, a 5 bit mnemonic.

When executed the instruction uses the values in IN1, IN2 and FID to index a MEMO-TABLE, and stores the value in IN3 in it.

Each of these instructions has an one operand version (`lupm1`, `updm1`) where the second input register (IN2) is an implicit 0.

Table 2 shows a complete assembly code excerpt which uses the new instructions. The assembly is for the SimpleScalar ISA (f* are fp registers, L* are labels, sin is the address of the Sine routine, and NSIN is its 5-bit mnemonic). The code demonstrates the memoization of a single operand function. The compiler loads f20 with the input to the `sin()` function and executes **lupm1** with f22 as the OUT register. If the lookup is successful the result in the MEMO-TABLE will overwrite f22 and set the floating point flag, causing the next instruction to branch and skip the function call. If the lookup is unsuccessful the function call will be performed and the instruction **updm1** updates the MEMO-TABLE with the value in f0 (the result of the function call).

C code	*Assembly*			*Remarks*
a = 1.1;		l.d	f20,LC	The input (1.1) is loaded into f20
		lupm1	**f20,f22,NSIN**	is 1.1 in the table?
		bc1t	**L1**	if lookup succeed skip routine call
		mov.d	f12,f20	f20 ⇒ f12 (input reg)
b = sin(a);		jal	sin	call routine
		mov.d	f22, f0	f0 (output reg) ⇒ f22
		updm1	**f12,f0,NSIN**	update table with sin(1.1)
c = b + a;	L1:	add.d	f24,f22,f20	continue execution

Table 2: *Assembly code implementing memoization of sin function. New instructions are bold faced (***lupm1***), added instructions are in sans serif (*`bc1t`*) (the $ sign before registers and variables is omitted).*

2.4 Overhead Considerations

If **lupm1** is unsuccessful no branch is performed, the routine is setup, jumped to, cleaned up (the output is moved from f0, the function's output register, into f22), the MEMO-TABLE is updated with the computed value, and execution continues at L1. Thus the overhead of a miss is three instructions: **lupm1**, **updm1**, and `bc1t` (for the case of two operand functions the penalty is the same). Of course a hit eliminates the function's setup, execution and cleanup saving tens to hundreds of machine cycles.

When the hit ratios are high the cost of the extra instructions is insignificant in comparison to the elimination of tens to hundreds of instructions due to successful memoization. When the hit ratios are low or nonexistent (see section 3.1), a penalty of three instructions per function call might seem high.

The following table shows how a processor capable of executing 4 instructions per cycle (such as the MIPS R10000) will execute the code. The processor has a Floating Point Unit, and two Integer Units (one of them executes the branchs). **lupm1** and **updm1** are executed by the FP Unit.

cycle	*FP Unit*		*Integer Unit1*		*Integer Unit2*
0	**lupm1**	**f20,f22,NSIN**			
1	mov.d	f12,f20	`bc1t`	`L1`	
2			jal	sin	
3–253	executing sin				
254	mov.d	f22,f0			
255	**updm1**	**f12,f0,NSIN**			

Due to the dependencies between `mtc1`, **lupm1** and `bc1t` and the use of the FU by both **updm1** and mov.d the overhead of a miss is three cycles.

This penalty can be reduced by adding a unit that can execute a **lupm1** or **updm1** in parallel to other FP instructions (a dedicated MEMO-TABLE Unit (MTU) or another FPU), enabling the MEMO-TABLE lookup and update to be performed in parallel to the function's setup and cleanup. This reduces the miss penalty down to a single cycle.

As mentioned above the new instructions are written in MIPS style assembly code. For other architectures the instructions would take on characteristics of the relevant ISA. For instance for the Power PC ISA the **lupm1** instruction will set a Condition Register (CR) based on the success of the lookup and the following instruction will be a conditional branch based on the value inserted into it. For the Intel 80x86 ISA the **lupm1** instruction will pop it's operands from the floating point stack and set the appropriate flag in the EFLAGS register.

2.5 Cycle Time and Die Size

The cycle time of a MEMO-TABLE lookup is comparable to that of a cache lookup. A 16-KByte cache with a line size of 32 bytes contains 512 entries, and first level on-chip caches are reaching sizes of up to 64-KByte[15, 16]. Our experiments show that a 64 (or 128 for a unified MEMO-TABLE) entry MEMO-TABLE is sufficient for good results, so accessing a MEMO-TABLE should take less time than accessing an on-chip cache (which is one cycle for most modern microprocessors).

The size of a MEMO-TABLE can be compared to the size of a conventional cache. A two operand unified (contains function identifier) 128 entry MEMO-TABLE holds $128 \times 4 = 512$ double precision values which is 4 Kbytes. With second-level caches being integrated on-chip[17], adding several MEMO-TABLES should be possible without draining resources meant for other units.

3 Experiments and Results

To verify the usefulness of memoization of mathematical and trigonometric functions, we performed a series of experiments with SimpleScalar[18], an architecturally detailed RISC instruction-level simulator based upon the MIPS ISA. SimpleScalar receives as input a binary executable compiled for the simulator and executes it down to the cycle level. All applications were compiled using gcc version 2.6.3 with the optimization flags `-O3 -finline-functions -funroll-loops`. We tailored SimpleScalar to incorporate MEMO-TABLES in it's design and thus simulate the memoization of mathematical and trigonometric functions. The new instructions were added by inserting compiler directives in the functions to be memoized. The compiler then replaced these directives with the new instructions. The simulator was altered to recognize these instructions and act upon them.

The two indicators that measure the success of the memoization are:

Hit Ratio The hit ratio of a MEMO-TABLE (number of successful lookups divided by number of lookups) will show how many multiple cycle function implementations were avoided.

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.

Naturally, the hit-ratio and speedup depend on the specific design of the MEMO-TABLE. The size, associativity and contents of the LUT, impact the expected hit-ratio and speedup.

3.1 Simulations

The hit ratio is a function of the size of the MEMO-TABLE, its associativity, and its contents (single function results or all function results).

We have simulated a MEMO-TABLE with its size varying from 16 to 1K entries and the spectrum of associativity from direct mapped to 16-way associativity. In addition we have simulated using several MEMO-TABLES, one for each function, and using a single unified MEMO-TABLE for all functions. We have also run the benchmarks through an "infinitely" large fully associative MEMO-TABLE for comparison. In section 3.5 we explore memoization of user defined functions and in section 3.6 we compare function memoization to instruction memoization.

L1 Instruction Cache	32-KBytes, 64-Byte blocks, 2-way associative
L1 Data Cache	32-KBytes, 32-Byte blocks, 2-way associative
L2 Unified Cache	1-Mbytes, 64-Byte blocks, 2-way associative
Memory Latencies (cycles)	L1 hit - 1, L2 hit - 6, L2 miss -18
Bus Interface	64-bit data, 32-bit address
Branch Prediction	256-entry BTB, 512-entry 2-bit BHT
Registers	32 GPR, 32 FP
Function Units	2 Integer Units (perform branchs)
	1 FADD unit, 1 FMUL Unit, 1 Memory unit
Out of Order Execution	4-insts fetched per cycle,
	16-entry Int, FP & Addr queues
	32-entry reorder buffer, 4-insts retired per cycle

Table 3: *Characteristic's of MIPS R10000.*

The overhead of memoization in our simulations is three machine cycles, the stricter of the two options shown in section 2.4. The simulated system is built upon the MIPS R10000 processor[15]. Table 3 shows a summary of its major features. The functions are assumed to be implemented in software except square root taking which is implemented in hardware on chip. This is the current state for most modern microprocessors. Each function has its own MEMO-TABLE or they share a unified MEMO-TABLE. The benchmarks were taken from several sources:

- **SPEC CFP95** - the floating point component of the SPEC CPU95 suite[19].
- **MediaBench** - a suite of multi-media and communication applications from UCLA[20].
- **Khoros** - Khoros Pro 2000[21] is a development environment that consists of a suite of Image Processing (IP) and Digital Signal Processing (DSP) applications.

Only benchmarks which have a nontrivial (thousands) number of mathematical and trigonometric function calls were selected for simulation. Applications that don't call the above functions aren't influenced by our enhancements to the processor.

Table 4 describes the specific applications, and table 5 shows how many instruction and cycles each application executed, and how many function calls were made by it (at least 1,000 calls)[3]. It can be seen that in most cases only two or three functions are used heavily in each application. This influences the choice of whether to use separate MEMO-TABLES for each FUNCTION-INSTRUCTION or to have a unified MEMO-TABLE for all functions.

3.2 Speedups Obtained

The basic configuration of a MEMO-TABLE that we have chosen is one with 64 entries arranged in 16 sets (set associativity of 4), each function has it's own MEMO-TABLE. Table 6 shows the results. We compare the results of using "infinitely" large fully associative MEMO-TABLES to the results of using

[3]In some cases the numbers are the sum of several applications that make up a benchmark (eg. decode and encode for mpeg2) or the sum of several runs with different inputs (the Khoros applications).The SPEC benchmarks weren't run to completion but were restricted to between 2 to 7 billion instructions. Running the benchmarks through sim-fast and only measuring the hit ratios showed that the partial results collected reflect the expected results when running the benchmarks to completion.

suite	*application*	*description*
MediaBench	rasta	Speech recognition
MediaBench	mesa	3D graphics library
MediaBench	mpeg2	Video compression
SPEC	swim	Shallow water equations
SPEC	su2cor	Monte-Carlo method
SPEC	hydro2d	Navier Stokes equations
SPEC	turb3d	Turbulence modeling
SPEC	apsi	Weather prediction
SPEC	fpppp	Quantum chemistry
SPEC	wave5	Maxwell's equation
Khoros	kfft	Fast Fourier Transform
Khoros	kgsin	Generate sinusoidal data
Khoros	khisto	Compute image histogram
Khoros	klogexp	Image logarithm taking
Khoros	vgbox	Parallelogram creation
Khoros	vpml	Fractal dim. estimation
Khoros	vmarr	Edge detection

Table 4: *Description of benchmark applications*

9 64 entry 4-way associative MEMO-TABLES. In addition we show the results of using a single 128 entry unified MEMO-TABLE (4-way associative) which holds all function values. What is shown in the table is:

- *FE - Fraction Enhanced*, the fraction of computation time in the original machine that can use the enhancement. This is shown in terms of dynamic instruction count and number of cycles.
- *HR* - the hit ratios of the MEMO-TABLES (for the infinite, separate and unified cases).
- *OSP* - the optimal speedup attainable if all functions were performed in a single cycle.

application	*insts*	*cycles*	*sqrt*	*sin*	*cos*	*tan*	*log*	*exp*	*floor*	*ceil*	*pow*
rasta	53	58	13K	12K	12K		11K	5K			15K
mesa	107	129	27K	4K	4K		77K		119K	29K	13K
mpeg2	2411	2159							4.1M	1.4M	
swim	2674	3021		526K	526K						
su2cor	5234	5462									2.0M
hydro2d	3740	4879	1.54M								96K
turb3d	6836	5996		531K	531K						
apsi	3605	5179	1.0M				49K				1.3M
fpppp	4957	6300	1.3M						856K	315K	87K
wave5	7918	8372	9.0M	750K	1.5M		1.5M				
kfft	483	582	2K	103K	103K	2K					2K
kgsin	135	135		288K							7K
khisto	107	142							1.2M	6K	3K
klogexp	52	54					80K				2K
vgbox	128	134		5K	5K						5K
vpml	1129	726					48K				50K
vmarr	21	25						84k			2k

Table 5: *Number of instructions, Number of cycles (in millions) and breakdown of function calls (in thousands) in the benchmark applications. Entries of less then 1K are ignored*

- *SP* - the actual speedup attained (for the infinite, separate and unified cases).

The results show that for most applications the hit ratio is high with an average of 57% for separate MEMO-TABLES and 58% for a unified MEMO-TABLE. The more significant number is the speedup. An average speedup of 10% is attained (11% for a unified MEMO-TABLE).

While the average hit ratios and speedup are good we find a lack of correlationbetween them. Figure 3 shows that for the SPEC and Khoros applications the hit ratios are higher than for the MediaBench benchmarks. On the other hand figure 4 shows the breakdown of the speedup according to suite. From this figure it can be seen that the Multi-Media benchmarks attain higher speedups than the SPEC benchmarks. This can be attributed to the higher percentage of execution time spent computing the functions

application	*FE*		*Hit Ratio*			*OSP*	*Speedup*		
	inst	*cycle*	*inf*	*sep*	*unif*		*inf*	*sep*	*unif*
rasta	.27	.31	.72	.67	.45	1.46	1.24	1.25	1.13
mesa	.18	.20	.69	.26	.20	1.26	1.20	1.06	1.04
mpeg2	.04	.05	.32	.17	.22	1.05	1.03	1.02	1.02
average	.16	.19	.56	.37	.29	1.26	1.15	1.12	1.06
swim	.07	.06	.99	.50	.49	1.07	1.07	1.03	1.03
su2cor	.20	.22	.99	.99	.99	1.29	1.29	1.29	1.29
hydro2d	.01	.02	.99	.70	.77	1.03	1.03	1.02	1.02
turb3d	.03	.03	.99	.99	.99	1.03	1.03	1.03	1.03
apsi	.01	.02	.89	.69	.65	1.02	1.02	1.01	1.01
fpppp	.04	.08	.62	.39	.34	1.10	1.09	1.08	1.07
wave5	.10	.10	.00	.00	.00	1.12	0.98	0.98	0.98
average	.07	.08	.78	.61	.59	1.09	1.07	1.06	1.06
kfft	.10	.09	.99	.75	.60	1.10	1.10	1.07	1.06
kgsin	.54	.50	.99	.06	.13	2.03	2.02	1.13	1.20
khisto	.21	.13	.99	.63	.83	1.15	1.14	1.05	1.08
klogexp	.41	.40	.99	.85	.92	1.67	1.66	1.53	1.60
vgbox	.02	.02	.99	.99	.99	1.02	1.02	1.02	1.02
vpml	.01	.01	.92	.86	.88	1.02	1.02	1.01	1.01
vmarr	.40	.26	.90	.26	.41	1.35	1.33	1.11	1.19
average	.24	.20	.96	.62	.68	1.33	1.32	1.13	1.16
average	**.16**	**.15**	**.82**	**.57**	**.58**	**1.21**	**1.19**	**1.10**	**1.11**

Table 6: *Performance enhancement with memoization ofmathematical and trigonometric functions.* MEMO-TABLES *are either infinitely large, of size 64 for each function or a 128-entry unified for all functions.*

(FE). While the Multi-Media benchmarks spend 19% (MediaBench) to 20% (Khoros) of their execution time in mathematical and trigonometric functions the SPEC benchmarks only spend 8% of their execution time in these functions.

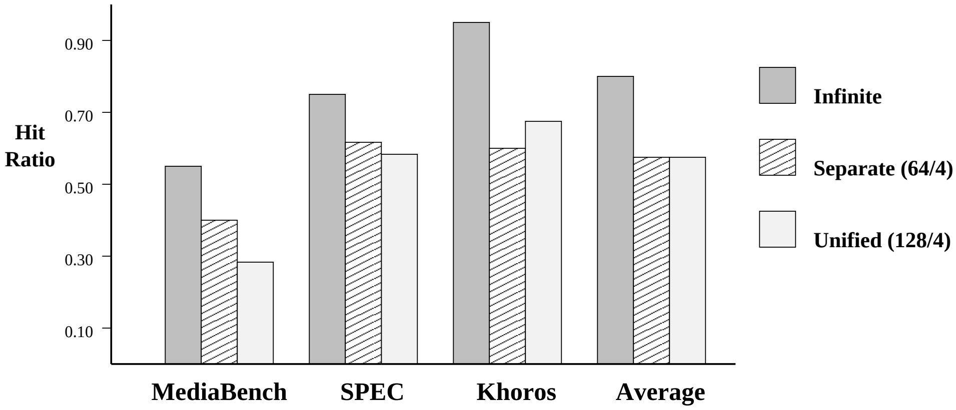

Figure 3: *Breakdown of hit ratios by suite*

Thus it is essential to show the FE. The current norm of showing only the HR isn't enough to understand the speedups obtained. The product of both HR and FE is necessary as shown in the following graphs. Figure 5 contains three graphs. The top two show the correlation between Hit-Ratio (HR) and Fraction Enhanced (FE) to speedup. Shown are the actual points and the

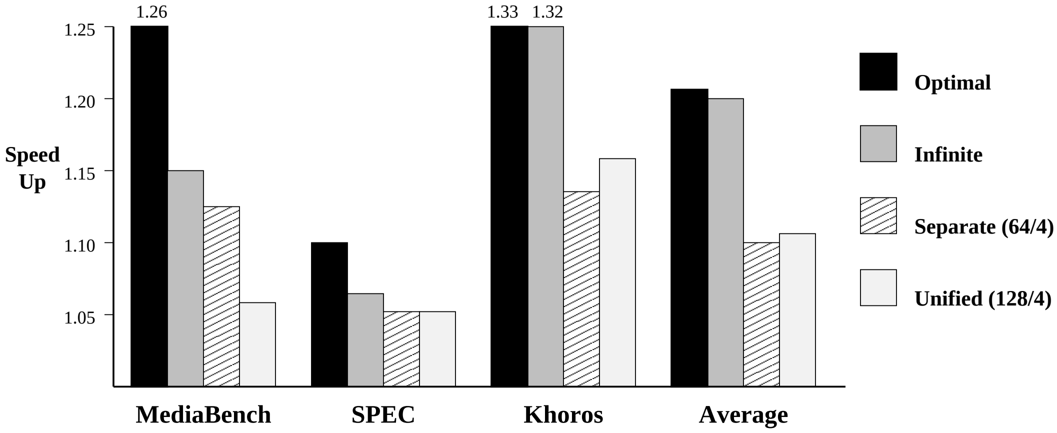

Figure 4: *Breakdown of speedups by suite*

best-fit lines (nonlinear least squares fitting using the Marquardt-Levenberg Algorithm). The graphs show that the correlation of HR and speedup and of FE and speedup are both poor. Applications such as vgbox and su2cor have the same HRs but have vastly different speedups, while mpeg2 shares the same speedup as vgbox but has a different HR. The same is true of the second graph for the applications kfft, wave5 and mesa.

The third graph shows the correlation between the product of Hit-Ratio (HR) and Fraction Enhanced (FE) to speedup. The line shows that by raising only the hit-ratio or only the FE a modest speedup increase is obtained as shown by the applications vgbox (high HR, low FE) and kgsin (low HR, high FE). Klogexp is the best performer with high values in both categories. The benchmark khisto which has an average hit-ratio (63%) and an average FE (13%) achieves only a low speedup (5% as opposed to the average 10%). A look at table 5 shows that the application utilizes mainly the `floor()` and `ceil()` functions. Table 1 shows that these functions have low latencies, thus the relatively low speedup in applications that utilize them.

Only one application, wave5, shows a slowdown (0.98 speedup). All other applications show a real speedup. In cases where a slowdown is encountered the developer can instruct the compiler to disallow memoization. In cases where the slowdown is input dependent the developer must use intuition to decide whether to memoize the application or not.

3.3 Discussion

It is important to understand why the technique works. Why do benchmarks such as vgbox, turb3d and klogexp display such high speedups. A look at a simplified excerpt from the source code of vgbox shows (table 7) that it is computing in a loop the Sine and Cosine of a variable. Profiling showed that this variable hardly changes and when it does the new value is used extensively. However the compiler can't perform Common Subexpression Elimination (CSE) and move it out of the loop body due to a condition that might change the variable's values. The compiler can't detect the fact that the value actually hardly changes. Using a MEMO-TABLE solves the problem by saving the redundant computations.

The benchmark turb3d contains code that performs a complex Fast Fourier Transformation (FFT). The main loop of the fortran subroutine is shown in table 8. In the inner loop half the values of `TI` are the same as from the pre-

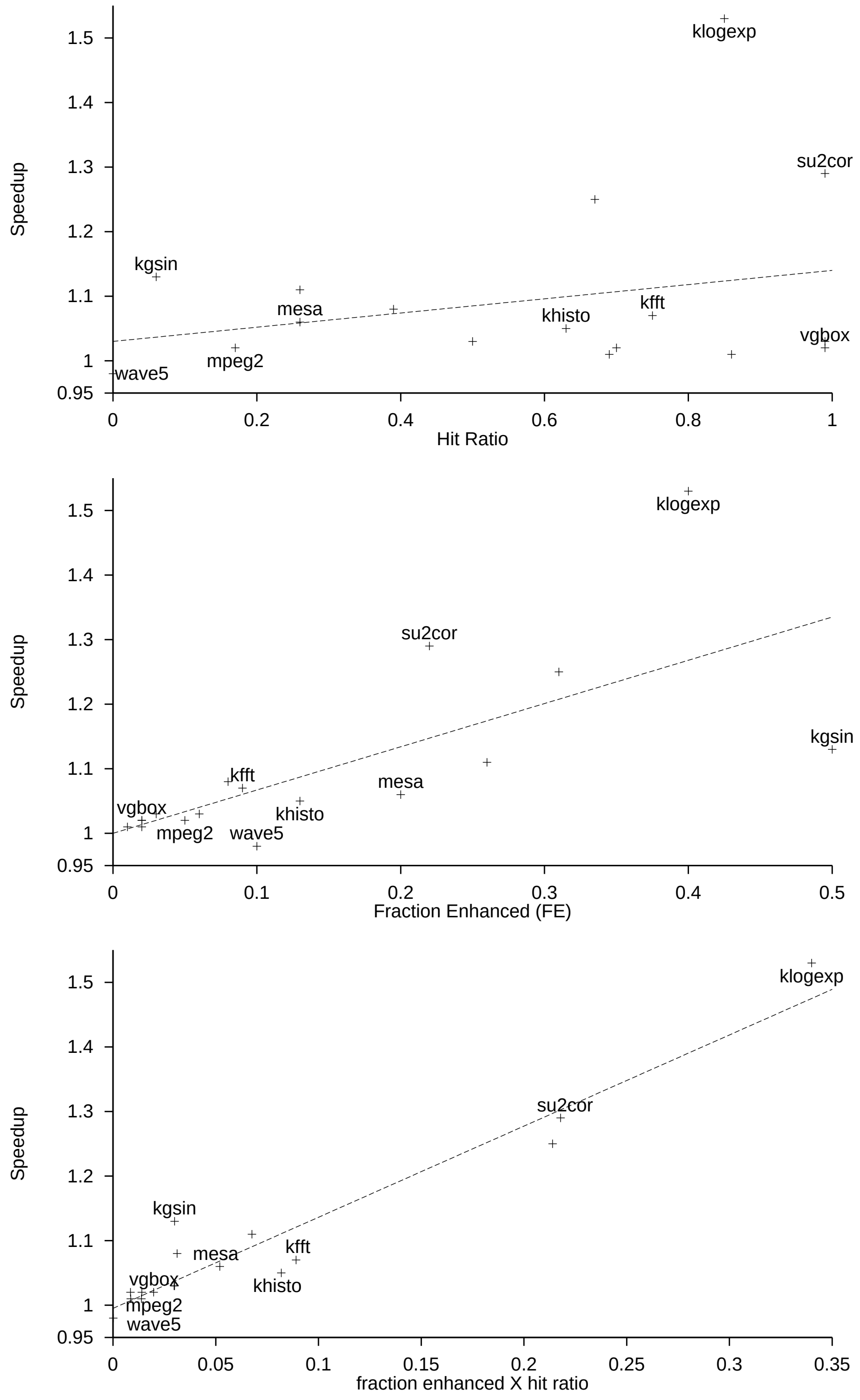

Figure 5: *Speedup (vertical axis) as a function of hit-ratio, FE, and their product (horizontal axis). The results are for separate* MEMO-TABLES.

vious iteration of the outer loop. The MEMO-TABLE easily takes advantage of this.

The application klogexp takes the logarithm of all pixels in an image. Neighboring pixels in an image tend to have the same values, leading to a high hit-ratio in the MEMO-TABLE[4].

```
for(i=0;i<N;i++){
    xp = i/px;
    std[i] = xp*cos(teta)/sin(teta);
    if(std[i] >= KPI - EPS && std[i] <= KPI + EPS)
        teta += KPI;
}
```

Table 7: *Simplified vgbox code.*

```
       DO 110 J = 1, M
          T = PI / LN
          DO 100 I = 0, LN - 1
            TI = I * T
            U(I+KU) = COS (TI)
            U(I+KN) = SIN (TI)
100       CONTINUE
          KU = KU + LN
          KN = KU + NU
          LN = 2 * LN
110    CONTINUE
```

Table 8: *FFT routine from turb3d.*

3.4 MEMO-TABLE Configuration

The next three experiments performed test the attributes of the LUT itself, its size, associativity and contents (unified MEMO-TABLE or separate MEMO-TABLES for each function). For these tests we used only 11 out of

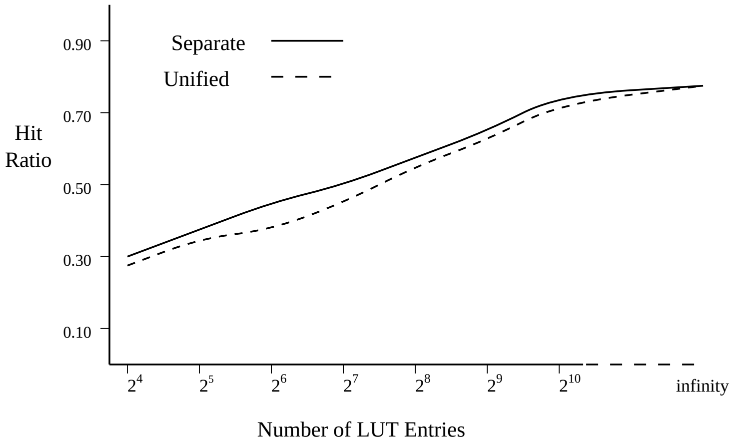

Figure 6: *Hit ratios as a function of LUT size (set size is 4).* MEMO-TABLE *is unified (dashed line) or separate* MEMO-TABLES *are used (solid line).*

17 application used in the previous tests. We have omitted the benchmarks su2cor, turb3d, wave5, vpml and vgbox where the hit-ratios are almost the same regardless of the MEMO-TABLE size. And fpppp was omitted due to its long run time.

Figure 6 shows the average hit-ratios of the chosen applications when the size of the LUT ranges from 16 to 1024 entries, and its associativity is 4. The figure shows the difference in hit-ratios between using a unified MEMO-TABLE (dashed line) to having a separate MEMO-TABLE for each function (solid line). We see that performance improves up to about 1024 entries after which the line starts to flatten towards infinity. The figure shows that almost the same hit-ratios are obtained for a unified table of size n and separate tables of size $n/2$. In our case, where 9 separate MEMO-TABLES are implemented, using a unified MEMO-TABLE gives the same results at less than 1/4 of the area cost. Table 6 corroborates this by showing that the average speedup achieved using a unified MEMO-TABLE of size 128 (11%) is slightly greater than the average speedup achieved when using 9 separate MEMO-TABLES of size 64 (10%). This is due to the fact that most applications heavily use only two or three functions (table 5).

Figure 7 shows the hit-ratios as a function of set associativity. For sep-

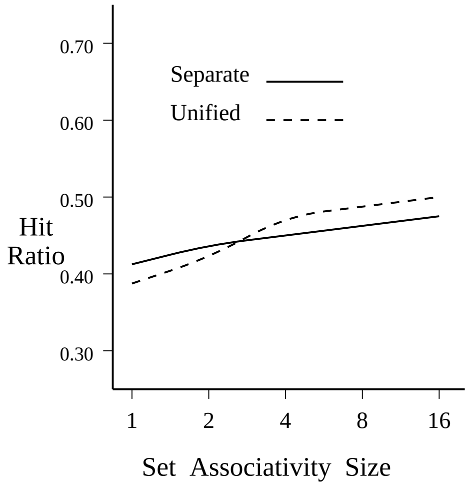

Figure 7: *Hit ratios as a function of set associativity size.* MEMO-TABLE *is unified (dashed line, 128-entry) or separate* MEMO-TABLES *are used (solid line, 64-entry per* MEMO-TABLE*).*

arate MEMO-TABLES any set associativity higher than one (direct-mapped) hardly influences the hit-ratio. For a unified MEMO-TABLE the curve starts straightening out only for a set size of 4. These results can be explained by the contents of the MEMO-TABLES. The separate tables are only mapped by the input value(s), leading to a greater spread of values throughout the entries in the table. In a unified MEMO-TABLE the mapping is by the input value(s) and the function identifier. In some applications (kfft, swim) the same values are being computed for several functions. This leads to conflict misses in the case of a direct mapped or even 2-set associative table. A set associativity of 4 alleviates this problem and enhances the hit-ratios.

3.5 Memoization of User Functions

It is possible to memoize user defined, application specific, functions in addition to the common mathematical functions. In the benchmarks we used we found only two applications that heavily use side-effect free functions, apsi and wave5. However the results of memoizing them are encouraging.

The table 9 shows the hit ratios and speedups of memoizing user defined instructions (in addition to the mathematical instructions) compared with only memoizing mathematical functions.

application	*math*		*math + user*	
	hr	*spdp*	*hr*	*spdp*
apsi	.69	1.01	.20	1.05
wave5	.00	0.98	.55	1.02

Table 9: *Speedup and hit ratio comparison of math function memoization with user and math function memoization. The results are for separate* MEMO-TABLES.

The table clearly shows that there is an advantage to memoizing user defined functions as well (when possible). The hit ratio for user defined functions is lower (for apsi) due to the fact that a successful user function lookup avoids many mathematical functions. When the user function lookup is unsuccessful the mathematical functions are called with new values, causing a lower hit ratio. However the run-time is reduced due to many other instruction avoided execution.

3.6 Memoization of Functions and Instructions

As mentioned in the introduction memoization was first introduced for long-latency instructions such as FP division and multiplication[8, 10, 4]. In this section we will compare 3 implementations::

1. The implementation explored in this paper where functions are memoized.

2. Multiple-cycle instructions (without loads/stores) are memoized. Functions are implemented in software and benefit from the memoized instructons.

3. combined approach where both functions and instructions are memoized.

The MEMO-TABLES used to memoize instructions are of size 256 entries and are 4-way set associative. There are 5 MEMO-TABLES: (i) for long-latency instructions (FP div and mult), (ii) integer instructions (INT div and mult), (iii) FP comparisons, (iv) FP⇔INT conversions, (v) and all other FP instructions (add, sub, ...). A full description of these tables is available in [22]. Table 10 shows the speedups each implementation yields, in addition the percentage of cycles spent in the mathematical functions is displayed. Figure 8 shows the speedups per suite.

application	*FE*	*func*	*inst*	*comb*
rasta	.31	1.25	1.06	1.27
mesa	.20	1.06	1.06	1.13
mpeg2	.05	1.02	1.10	1.11
average	.19	1.12	1.07	1.17
swim	.06	1.03	1.03	1.05
su2cor	.22	1.29	1.03	1.29
hydro2d	.02	1.02	1.23	1.23
turb3d	.03	1.03	1.05	1.07
apsi	.02	1.01	1.01	1.09
fpppp	.08	1.08	1.02	1.09
wave5	.10	0.98	1.17	1.15
average	.07	1.06	1.06	1.12
kfft	.09	1.07	1.02	1.08
kgsin	.50	1.13	1.11	1.14
khisto	.13	1.05	1.09	1.09
klogexp	.40	1.53	1.25	1.53
vgbox	.02	1.02	1.04	1.04
vpml	.01	1.01	1.02	1.02
vmarr	.26	1.11	1.05	1.13
average	.20	1.13	1.08	1.14
average	**.15**	**1.10**	**1.08**	**1.15**

Table 10: *Speedup comparison between function memoization, instruction memoization, and a combined implementation. The percentage of cycles spent in the mathematical functions is shown as well.*

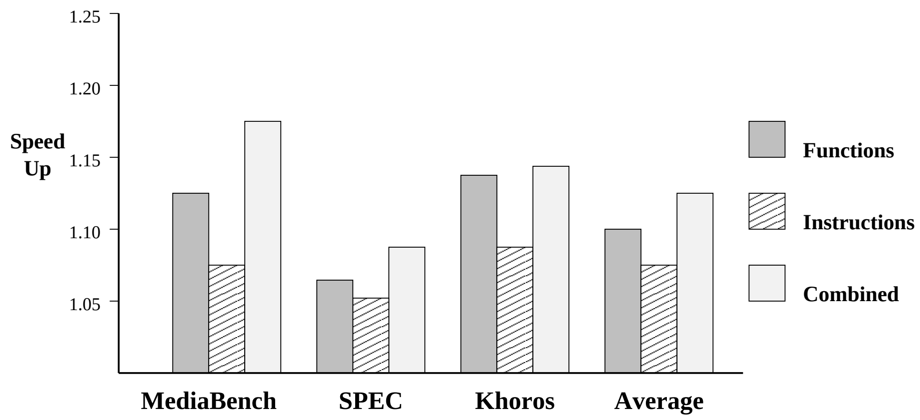

Figure 8: *Breakdown of function, instruction, and combined memoization speedups by suite.*

The table shows that applications that heavily use mathematical functions (Multi-Media) benefit more by memoizing functions than by only memoizing instructions. A large amount of the instructions memoized are in the memoized functions (this was verified by analyzing the source code of the applications), leading to their execution being avoided when the function is memoized. On the other hand applications that use the mathematical functions sparingly benefit from instruction memoization which can catch instructions not in the mathematical functions.

Obviously the combined approach is superior with an average 15% speedup. In choosing between the function to instruction implementations we might be mislead to choose function memoization due to the higher speedup (10% vs. 8%). However we must remember that instruction memoization is effective for a broader scope of applications and is compiler transparent. As opposed to function memoization which is limited in its scope to specific applications and needs compiler support for most architectures. From a design point of view instruction memoization is more complex as it necessitates adding several MEMO-TABLES adjacent to the processors execution units. Function memoization is simpler as the MEMO-TABLES (or unified MEMO-TABLE) can be located in one area of the chip and only 2 instructions access them.

4 Comparing Hardware to Software Memoization

A fundamental question about memoization is: "Why can't it be done in software?" Memoization is, of course, possible to implement in software but there are several reasons why a hardware-based approach is superior:

- the most compelling reason for using hardware-based MEMO-TABLE is the penalty of an unsuccessful lookup. For terminal cases where the hit-ratio on the MEMO-TABLE is low, the penalty for a software test and update is several memory accesses and tens of extra instructions. The penalty of a hardware-based MEMO-TABLE miss is from one to three machine cycles as shown in section 2.4.

- Initializing, accessing and updating the software based MEMO-TABLE complicates programming and compiler design. Global MEMO-TABLES will have to be recognized by code that was written by different teams of developers or by a third party company. A hardware-based MEMO-TABLE access is simple (**lupm1** and **updm1**) and all modules of an application access the same table. A developer need not know of the existence of memoization and the compiler writer needs to add only three extra instructions for each memoized function call (of 1 or 2 operands and 1 result).

- A memory based MEMO-TABLE demands resources such as registers, cache lines, and memory ports. These resources are deducted from the original application. On the other hand a hardware-based MEMO-TABLE uses none of the above.

Another software approach suggested by Calder, Feller & Eustace[14] uses value profiling to identify instructions that have invariant or predictable values at run-time. By inserting specialized code they can compare the inputs of functions or code segments to the values that have been found to be most common. If they match the results are obtained immediately, if not the function or code segment is executed.

The only advantage this technique has over hardware memoization is that no hardware and instruction set changes are needed. On the other hand our technique has the following advantages:

- It doesn't need the extensive profiling necessary for value profiling.
- It can capture reuse that value profiling doesn't detect such as a large number of data values each used only a few times, or data that is input variant.
- If there are even two values to compare to the overhead of a miss is greater than the overhead of a hardware miss (although it is much smaller than the overhead of software memoization).

Table 11 compares hardware based to software based memoization. Shown are the speedups when using separate MEMO-TABLES for each function. The size of each MEMO-TABLE is 64 entries and the associativity is either direct-mapped or 4-way set associativity. In addition to the speedups we show the table size needed in order to achieve the same speedup of using a hardware 64/4 MEMO-TABLE, or the maximal speedup if it is impossible to obtain the same speedup. This happens when the miss ratio is so high that the miss penalty is larger than the number of cycles avoided by successful memoization, or the software tables are so large that they dominate the L1 data cache and degrade execution.

The table shows that only 5 out of 17 applications achieve any speedup (with software-memoing & 64-entry tables). The average speedup is -11% (-7% for a direct-mapped table), in other words a slowdown. It is interesting to point out that while the hardware based scheme favors the higher associativity (10% compared to 8%), the software favours the direct-mapped approach. This is due to serialization of the 4-way lookup in software. Further simulations have shown that for software a 2-way lookup is the best tradeoff between hit-ratio and lookup overhead.

When trying to find the software based table that yields the best results we observed that a table larger than 2K entries will always cause a degradation in performance. This is caused by the doubling of the miss ratio of the L1 data cache over the case where a 1K entry table is used (for some applications a smaller table size already causes degradation). The average speedup obtained is 1%, with only six applications being slowed down, and 11 achieving some degree of speedup.

application	*64-entry tables*				*best achieved soft memoization*	
	4-way		*direct*			
	hard	*soft*	*hard*	*soft*	*size/assoc*	*spdp*
rasta	1.25	1.10	1.22	1.12	512/2	1.15
mesa	1.06	0.76	1.06	0.83	1024/2	0.93
mpeg2	1.02	0.71	1.01	0.74	1024/2	0.78
average	1.12	0.86	1.10	0.90	1024/2	.95
swim	1.03	0.97	1.03	0.98	1024/2	1.03
su2cor	1.29	1.20	1.29	1.23	128/1	1.24
hydro2d	1.02	1.01	1.01	0.99	1024/2	1.01
turb3d	1.03	1.01	1.02	0.99	2048/2	1.02
apsi	1.01	0.74	1.01	0.78	2048/2	0.94
fpppp	1.08	0.86	1.07	0.90	2048/1	1.03
wave5	0.98	0.73	0.98	0.76	2048/2	0.82
average	1.06	0.93	1.05	0.95	2048/2	1.01
kfft	1.07	0.99	1.06	1.02	512/2	1.06
kgsin	1.13	0.76	1.12	0.82	2048/1	1.13
khisto	1.05	0.48	1.04	0.60	2048/2	0.72
klogexp	1.53	1.26	1.51	1.29	1024/2	1.40
vgbox	1.02	1.02	1.02	1.02	64/4	1.02
vpml	1.01	0.99	1.01	1.00	512/2	1.01
vmarr	1.11	0.67	1.09	0.73	1024/1	0.91
average	1.13	0.88	1.12	0.92	1024/2	1.03
average	**1.10**	**0.89**	**1.08**	**0.93**		**1.01**

Table 11: *Speedup comparison between hardware based to software based memoization. Hardware* MEMO-TABLES *are separate and of size 64/1 and 64/4.*

5 Conclusions

This paper investigates the technique of memoization in the framework of the mathematical and trigonometric functions. The results of previous function invocations are saved (along with their inputs) in lookup tables. If the result of a function call already resides in a table, it is obtained in a single cycle as opposed to the tens to hundreds of cycles it would take to compute the function. Our tests have shown that an average success rate of 58% is achieved for applications that utilize the mathematical and trigonometric functions.

Our main conclusion is that with hardware support in the form of a small and simple to design lookup table (a unified, 128-entry, 4-way associative MEMO-TABLE's size is 4KBytes) it is possible to attain an average speedup of 11% for applications which utilize the aforementioned functions. This is 60% of the maximal speedup achievable which would require using MEMO-TABLES with millions of entries.

The overhead for unsuccessful lookups is from one to three cycles for each function call, thus an almost negligible penalty is paid for applications that don't display a large degree of "value locality". Such a low overhead is impossible to duplicate using software memoization techniques.

As most mathematical and trigonometric functions aren't included in the instruction sets of most microprocessors (square root taking being the exception) we suggest adding two new instructions to the ISA. **lupm1** (**lupm2**) and **updm1** (**updm2**), which lookup and update a generic MEMO-TABLE. Thus it is simple for program developers and compiler writers to reduce the execution time of frequently used functions. This is opposed to other techniques (value prediction, value profiling) which rely heavily on pre-execution profiling.

In comparing function memoization with instruction memoization we saw that for applications which heavily utilize mathematical and trigonometric functions, function memoization yields better results. For almost all applications both approaches complement each other leading to a 14% speedup using a combined implementation.

Acknowledgments

We would like to thank Prof. Larry Rudolph of MIT, for our discussions with him, which established the basis for this work.

References

[1] A. Sodani and G. Sohi, "Dynamic Instruction Reuse", *Proc. of the 24th Int. Symp. on Computer Architecture*, pp. 194–205, June 1997.

[2] F. Gabbay and A. Mendelson, "Using Value Prediction to Increase the Power of Speculative Execution Hardware", *ACM Transactions on Computer Systems*, Vol. 16, No. 3, pp. 234-270, August 1998.

[3] M. Lipasti, C. Wilkerson and J. Shen, "Value Locality and Load Value Prediction", *Proc. of the 7th Int. Conf. on Architectural Support for Programming Languages and Operationg Systems*, pp. 138–147, October 1996.

[4] D. Citron, D. Feitelson and L. Rudolph, "Accelerating Multi-Media Processing by Implementing Memoing in Multiplication and Division Units", *Proc. of the 8th Int. Conf. on Architectural Support for Programming Languages and Operationg Systems*, pp. 252–261, October 1998.

[5] http://developer.intel.com/design/processor

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

[7] S. Harbison, "An Architectural Alternative to Optimizing Compilers", *Proc. of the 1st Int. Conf. on Architectural Support for Programming Languages and Operationg Systems*, pp. 57–65, March 1982.

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

[9] S. Richardson, "Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation", *Sun Microsystems Laboratories*,Technical Report TR-92-1, September 1992.

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

[11] Y. Sazeides and J. Smith, "The Predictability of Data Values", *Proc. of the 30th Int. Symp. on Microarchitecture*, pp. 138–148, December 1997.

[12] A. Sodani and G. Sohi, "Understanding the Differences Between Value Prediction and Instruction Reuse", *Proc. of 31st Int. Symp. on Microarchitecture*, November 1998.

[13] D. Connors and W. Hwu, "Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results",*Proc. of 32nd Int. Symp. on Microarchitecture*, pp. 158–169, November 1999.

[14] B. Calder, P. Feller, A. Eustace, "Value Profiling and Optimization", *Journal of Instruction-Level Parallelism, Vol. 1*, 1-6 1999.

[15] http://www.sgi.com/MIPS

[16] http://www.motorola.com/PowerPC

[17] http://www.compaq.com/alphaoem

[18] D. Burger and T. Austin, "The SimpleScalar Tool Set, Version 2.0", *Technical Report TR-CS-97-1342*, University of Wisconsin-Madison, June 1997.

[19] http://www.specbench.org

[20] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", *Proc. of 30th Int. Symp. on Microarchitecture*, December 1997.

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

[22] D. Citron and D. Feitelson, "The Organization of Lookup Tables in Instruction Memoization", *Technical Report, 2000-4*, Hebrew University of Jerusalem, March 2000.