The Organization of Lookup Tables in Instruction Memoization

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

Instruction Memoization is a technique that enables performing multiple-cycle computations in a single cycle by exploiting redundant instructions. 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. Several recent papers have proven that non-trivial speedup (over 10%) can be obtained by using this technique.

The speedup can be attributed to three major factors: (i) the percentage of instructions that can benefit from memoization, (ii) the integration of the the Lookup Table (LUT) into the datapath of the processor, and (iii) the percentage of successful lookups. This paper focuses on the latter factor and explores how the organization of a Lookup Table (LUT) influences the percentage of successful lookups (hit-ratio).

The best design for a LUT — which we call a MEMO-TABLE — that yields the highest hit-ratio is described. The design chosen is based on experimental simulations that minimize the influence of the other two factors of speedup. After the MEMO-TABLE design is fixed the first two factors can be dealt with, without having to alter the

MEMO-TABLE to accommodate changes in (i) the instruction mix of an application and (ii) changes in the microprocessors datapath.

The characteristics of the MEMO-TABLES explored are size, set associativity, indexing, replacement methods, and contents (instruction mix). Results of simulations have shown that the optimal design is to use 5 MEMO-TABLES each which holds a "family" of instructions. Each MEMO-TABLE contains 256 entries in sets of 4. Entries are replaced randomly and are indexed by using a mix of bits from the exponent and mantissa of FP numbers or the Least Significant Bits of integers. Trivial operations are detected using dedicated circuitry and aren't entered into the MEMO-TABLES.

Integrating the proposed design into the datapaths of the MIPS R10000 and PPC 604e results in an average 8-10% speedup for applications that heavily utilize multiple-cycle instructions.

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] have established that Instruction Memoization (IM) is a technique that shows great potential for increasing processor performance. The technique exploits the redundancy of instruction results by storing the operands and results of executed instructions in a Lookup Tables (LUT), which we will call a MEMO-TABLE. When the same instruction type with matching operands is encountered the result is obtained from the MEMO-TABLE and instruction execution is avoided. This leads to less instructions being executed which in turn lowers the total cycle time of the program.

The speedup obtained is composed of three major factors:

1. The percentage of instructions that can benefit from memoization. Instructions that have a latency of a single-cycle and instructions that must be executed (stores to memory) are examples of instructions that aren't candidates for memoing. This factor is decided by the application's instruction mix and by the implementation of the microprocessor (latencies of instructions).

2. The integration of MEMO-TABLES in the datapath of the processor. The following issues affect the speedup: The stage of the pipeline that MEMO-TABLES are accessed, Multiple-issue of instructions, Long-latency instructions have their latency cut down to one cycle, and the penalty of an unsuccessful lookup.

3. The percentage of successful lookups, the hit-ratio of the MEMO-TABLE. This is influenced by the nature of the program being executed, how much redundancy it contains. [1, 3] have shown that for infinitely large MEMO-TABLES most programs show redundancy. To obtain high hit-ratios for modest size MEMO-TABLES the best design possible must be chosen.

The focus of this paper is on the third factor. The first factor is almost totally beyond the sphere of Computer Architecture.[1] The second factor is of utmost importance and may very well decide if IM is a viable feature that will be integrated into future microprocessors or just remain as a research effort that hasn't been able to justify its existence.

However, before effects of IM on the datapath (and vice-versa) are explored, we believe that the third factor of speedup, the hit-ratio should be explored. Finding the near optimal design for MEMO-TABLES will enable us to receive the maximal hit-ratio possible (for finite MEMO-TABLE sizes), fix this design and then explore the integration of MEMO-TABLES in the datapath.

The characteristics of the MEMO-TABLES explored are its cache-like traits: size, associativity, and replacement method, and characteristics that are unique to memoization such as indexing methods (which bits of the values compose the index into the MEMO-TABLE), contents (which instructions are in each MEMO-TABLE) and the detection of trivial calculations that can be computed easily ($x + 0$, $y * 1$, ...).

The rest of this section will overview related work (1.1). The next section describes the design of the MEMO-TABLE, and reviews IM. Section 3 presents the experimental results that support our design decisions. Section 4 presents the speedups achieved on two microprocessors, the MIPS R10000[4] and the PPC 604e[5]. Section 5 discusses and summarizes the results.

[1] Except where technological advances reduce the latencies of certain instructions to one and thus drops them from the list of memoization candidates.

1.1 Prior and Related Work

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

in a table and reuse the results for matching inputs. Until the middle 90's it has 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 previous executed instructions. Richardson[7] and Flynn & Oberman[8] have shown that the notion is feasible for long latency instructions such as multiplication, division and square root taking. Citron, Feitelson & Rudolph[3] have focused on Multi-Media applications and how they impact memoization of the aforementioned instructions. Citron & Feitelson[9] have taken instruction memoization one step further and have explored memoization of mathematic and trigonometric functions, both as instructions and as library functions. Our work broadens the scope of the aforementioned works and explores memoizing all multiple-cycle latency instructions.

On the other hand Sohi & Sodani[1] try to reuse all instructions, even single-cycle instructions. This technique was coined *Instruction Reuse (IR)*. Their table is called a *Reuse Buffer (RB)* and is accessed as early as the Instruction Fetch stage in the pipeline. Thus the RB can be accessed by the Program Counter (PC) only, as operand values aren't available at this stage. Three reuse schemes are presented: (i) each entry contains the operand values and the result, (ii) each entry contains the operand register names and the result, and (iii) each entry

contains pointers to entries that supply the operands of the instruction in addition to the contents of (ii). The first scheme shows a higher reuse rate and a higher speedup but a more detailed simulation [10] shows that due to the fact that the reuse test can be performed only in the Instruction Issue stage, performance improvement is hampered. For single-cycle instructions no cycle reduction is made.

Molina, González & Tubella[2] have recognized this and try to create links between instructions that produced the same result, resulting in instructions with different PCs accessing the same entry. Their conclusion is that a hybrid scheme which uses the operand values as indexes as well is necessary in order to boost performance.

IM differs from IR in several ways. First, we only store multiple-cycle latency instructions, thus a MEMO-TABLE lookup has a higher potential for eliminating cycles. Second, we store instructions in several MEMO-TABLES, thus it is less likely for a long latency instruction to be bumped out by a short latency one. Third, we never use the PC to index the MEMO-TABLE, only the operand values. Thus, for instance, if the compiler unrolls a loop our scheme will be able to reuse calculations. This is impossible with the scheme used by [1] and partially possible using the scheme of [2]. The last difference is the fact that IM is totally independent of the results of other instructions. IR must invalidate entries if registers are written to or if dependent instructions are replaced in the RB.

Other groups have focused on the neighboring field of value prediction in order to speculatively execute instructions that haven't had their data dependencies satisfied yet (Gabby & Mendelson[11], Lipasti, Wilkerson & Shen[12], and Sazeides & Smith[13]). Sohi & Sodani[10] discuss the differences and similarities between these two approaches. The main difference between VP and IM is the speculative nature of VP. As our research is totally un-speculative we don't compare our work to these studies.

2 Instruction Memoization

This section presents a brief overview of the IM technique, [3] provides a more detailed description. A basic MEMO-TABLE is first presented followed by a description of how IM fits into the datapath.

2.1 The MEMO-TABLE

A MEMO-TABLE is a cache-like Look Up Table (LUT), that is added to each computation unit that takes multiple cycles to complete. The likeness to a cache is due to the fact that the values in the LUT change dynamically 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 are compared to the value 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 (evicting an "older" entry). Which subset of bits to use is one of the characteristics explored in section 3.6.

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*. Figure 1 shows a MEMO-TABLE with n entries. The shaded area contains the results, the

unshaded areas contains the operands and opcode (in the case where several instruction types reside in the same MEMO-TABLE) which are compared to the operands and opcode of the instruction being memoized. Note that no valid bit is necessary, data is valid at all times even across context switches due to the fact that the instructions stored in MEMO-TABLES are context free, the result depends only on the operands. The only time invalid data is in a MEMO-TABLE is during startup, loading the opcode field with an invalid opcode solves this problem.

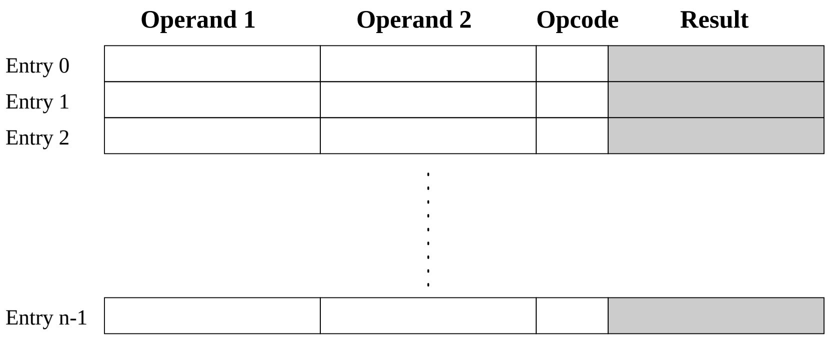

Figure 1: *Layout of a n entry* MEMO-TABLE.

During execution the operands are forwarded 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 computation unit is aborted and signals it is free to receive the next set of operands. If there is a miss in the MEMO-TABLE, the computation is allowed to complete, the result obtained forwarded to the next stage and in parallel entered into the MEMO-TABLE. Figure 2 shows the layout of a MEMO-TABLE adjacent to a division unit.

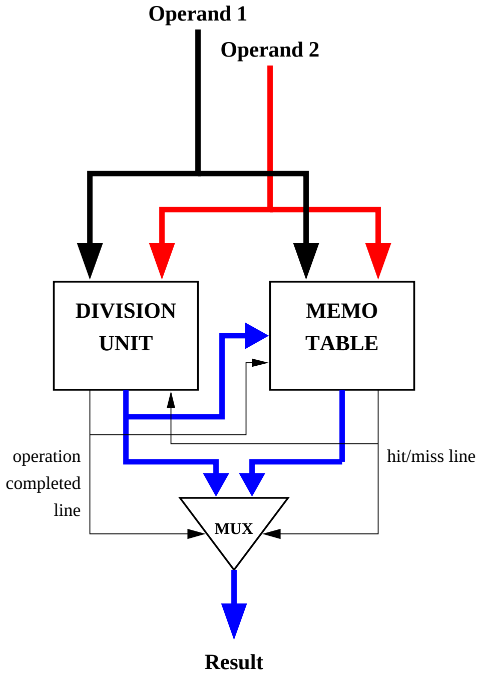

Figure 2: *Layout of a* MEMO-TABLE *adjacent to a division unit.*

2.2 Instruction Memoization in the Datapath

As explained in the introduction, this paper will not explore how MEMO-TABLES coexist with the microprocessor's pipeline and how they influence each other. Questions like "How does the issue width influence IM?" or "In what pipeline stage is it best to access the MEMO-TABLE?" will be deferred until we converge on the near optimal MEMO-TABLE structure.

What we will describe is the basic integration of IM in the pipeline. As opposed to IR where the RB is indexed by the PC, in IM the MEMO-TABLE is indexed by the instruction's values. Thus until the instructions dependencies are resolved (somewhere in the Issue stage) the MEMO-TABLE can't be accessed. We furthermore restrict MEMO-TABLE lookup until the required FU is free, so in the case of a miss the MEMO-TABLE can be updated with the result being computed in parallel to the lookup.

It would seem possible to access the MEMO-TABLE *before* the relevant FU is free and in the case of a hit bypass the Execute stage altogether. But in the case of a miss we would have to wait until the FU is free and

"remember" that the MEMO-TABLE must be updated. As mentioned before these questions are postponed for now. Thus the MEMO-TABLE is accessed during the Execute stage, this leads to our decision to restrict IM to multiple-cycle instructions. For single-cycle instructions no performance gain can be achieved.

3 Experiments and Results

To find the optimal design of a MEMO-TABLE we performed a series of experiments with an architecturally detailed simulator: Simple-Scalar[14], a RISC instruction-level simulator based upon the MIPS ISA. Simple-Scalar 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 Simple-Scalar to incorporate MEMO-TABLES in it's design and thus simulate IM.

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

Hit-Ratio The hit-ratio of a MEMO-TABLE (number of successful lookups divided by number of lookups) will show how many instruction executions 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.

As mentioned above the emphasis of the simulations will be on enhancing the hit-ratios of the MEMO-TABLES. We will explore how the size, associativity, replacement strategy, indexing and contents of the LUT, impact the achieved hit-ratio. We will show how the speedup is impacted by the hit-ratio but enhancement of speedup due to other factors will be explored in future work.

3.1 Simulations

The simulations were performed using the Simple-Scalar simulator. As we want to negate the influence of the datapath the programs were run through the `sim-fast` version of the simulator. This version simulates instruction execution step-by-step but doesn't simulate the memory hierarchy, pipelining, multiple-issue, branch prediction, or any other architectural enhancements (except the use of MEMO-TABLES, of course).

The benchmarks were taken from several sources:

- **SPEC CFP95** - the floating point component of the SPEC CPU95 suite[15].
- **SPEC CINT95** - the integer component of the SPEC CPU95 suite[15].
- **MediaBench** - a suite of multi-media and communication applications from UCLA[16].

The benchmark applications are either FP intensive or perform integer multiplication and/or division. Applications that don't execute large amounts of multiple-latency instructions can't benefit from IM and aren't simulated[2]

Table 1 describes the specific applications, the number of instructions executed, and the percentage of multiple-cycle instructions executed. [3] Even though less than 1% of the instructions in integer intensive applications are multiple-cycle instructions we simulate them and give them an equal standing to FP intensive applications[4]. We are exploring primarily MEMO-TABLE characteristics not overall speedup, thus the impact of these applications which have a different instruction mix than FP applications is important. The following simulation results are the average hit-ratios of the MEMO-TABLES for all the above applications.

[2]For this reason adpcm and pegwit from the MB suite and li and go from CINT95 aren't simulated. Jpeg from MB and ijpeg from CINT95 are similar so only jpeg is run. Couldn't get vortex from CINT95 to run. m88ksim from CINT95 is invariant to any MEMO-TABLE changes, 99% of all integer multiplications are reused in any configuration, thus this application was dropped from the simulations.

[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 SPEC CINT95 benchmarks were run with the test versions of the inputs in order to keep them relatively short, running them with the reference inputs gives similar results.The SPEC CFP95 benchmarks weren't run to completion but were restricted to between 1 to 5 billion instructions. Running the benchmarks to completion have shown that the partial results collected reflect the expected results when running to completion.

[4]The integer intensive applications are g721, pgp, and jpeg from MediaBench and the CINT benchmarks

suite	*application*	*description*	*# of insts*	*%*
MediaBench	rasta	Speech recognition	63M	10.4%
	mesa	3D graphics library	130M	17.8%
	mpeg2	Video compression	2493M	7.8%
	epic	Image compression	120M	15.5%
	gsm	Speech transcoding	223M	14.9%
	ghostscript	Postscript interpreter	159M	4.4%
	g721	Voice compression	529M	0.6%
	pgp	Cryptography	1296M	2.3%
	jpeg	Image compression	181M	0.3%
SPEC CFP95	tomcatv	Vectorized mesh generation	4000M	10.4%
	swim	Shallow water equations	1500M	26.3%
	su2cor	Monte-Carlo method	4500M	12.8%
	hydro2d	Navier Stokes equations	1000M	16.4%
	mgrid	3D potential field	2000M	14.5%
	applu	Partial differential equations	1500M	7.7%
	turb3d	Turbulence modeling	4000M	7.5%
	apsi	Weather prediction	1000M	22.6%
	fpppp	Quantum chemistry	1000M	32.8%
	wave5	Maxwell's equation	1000M	31.7%
SPEC CINT95	gcc	C compiler	119M	0.3%
	compress	Lempel-Ziv compression	123M	0.5%
	perl	Perl interpreter	525M	0.4%

Table 1: *Description of benchmark applications, number of instructions executed, and percentage of multiple-cycle instructions.*

3.1.1 The Instructions Memoized

All the instructions memoized have a latency of more than on e cycle, these include integer division and multiplication and all the floating point instructions. Table 2 lists the instructions memoized along with their latencies and throughputs on the R10000 and 604e[5]. For each instruction type there is a MEMO-TABLE that stores the operands and results of the instances of the instructions, there are 19 such MEMO-TABLES in use.

instruction	*MIPS R10000*		*PPC 604e*	
	lty	*thpt*	*lty*	*thpt*
Int Division	35	35	20	19
Int Multiplication	6	6	3	1
FP Add/Subtract	2	1	3	1
FP Comparison	2	1	3	1
FP→Int Conversion	2	1	3	1
Int→FP Conversion	4	1	3	1
FP Neg/Abs	2	1	3	1
FP Move	2	1	3	1
FP Multiplication	2	1	3	1
FP Division (sp)	12	14	18	18
FP Division (dp)	19	21	31	31
FP Sqrt (sp)*	18	20	50	50
FP Sqrt (dp)	33	35	60	60

* The 604e doesn't implement the `fsqrt` instruction.

Table 2: *Instruction latencies and throughputs for the MIPS R10000 and PPC 604e.*

3.2 MEMO-TABLE Structural Factors

We first measured four factors of the MEMO-TABLES. They are all related to the structure of the MEMO-TABLE rather than to its contents. The factors and their levels are:

- Size - the number of entries in each MEMO-TABLE, the levels are from 8 to

[5]The 604e doesn't implement the `fsqrt` instruction listed in its instruction set, we decided to do so in our simulator in order to compare the datapaths of both processors (a software implementation of the sqrt function can take over 1000 cycles).

16K entries, and an infinite table size.

- Associativity - the number of entries in each set. The levels are from direct-mapped (set size 1), to 8-way set associative (set size 8), and fully associative (one set).
- Replacement Strategy - Which entry is evicted from the MEMO-TABLE in the case of a miss. The levels are: replace randomly, First In First Out (FIFO), pseudo Least Recently Used (where the LRU entry is approximated) and true LRU.
- Mapping Strategy - How is an entry mapped to a set. The levels are to hash the Program Counter or hash the values. The values can be hashed using various techniques, simple ones such as hashing the Least Significant Bits (LSBs), to more complex techniques which hash the exponent, mantissa or some bit mix of them.

The number of simulations needed to perform a *full factorial design*, simulating every possible combination of all levels, would take: $n = \prod_{i=1}^{k} n_i$ simulations. In our case it is (12 levels of size)$\times$(5 levels of associativity)$\times$(4 levels of replacement schemes)$\times$(6 levels of mapping strategies) 1440 simulations for each and every application. This number is daunting and beyond our processing power.

In such cases where a full factorial design is impossible, a 2^k factorial design is used. For each factor two levels or alternatives are chosen resulting in only 16 simulations in our case.

These simulations can give us an indication which factors have a higher impact on the hit-ratios and which factors have little or no impact at all.

By using the Sign-Table[19] technique it is possible to compute the allocation of variation of each factor and the interaction between factors. The importance of a factor is measured by the proportion of the total variation in the result that is explained by the factor.

The levels chosen are: Size - 32, 1024; Associativity - direct mapped, 8-way set associative; Replacement Strategy - random, LRU; Mapping - PC, value (LSBs); The results (average hit-ratios of all applications) are shown in the following table:

The main components of the variation are: Mapping scheme - 55%, Associativity - 31%, and Size - 10%. The variation attributed to the replacement

32	1	rand	pc	0.17	32	1	rand	val	0.32
1024	1	rand	pc	0.22	1024	1	rand	val	0.39
32	8	rand	pc	0.30	32	8	rand	val	0.39
1024	8	rand	pc	0.32	1024	8	rand	val	0.51
32	1	lru	pc	0.17	32	1	lru	val	0.32
1024	1	lru	pc	0.22	1024	1	lru	val	0.39
32	8	lru	pc	0.32	32	8	lru	val	0.40
1024	8	lru	pc	0.33	1024	8	lru	val	0.51

Table 3: 2^4 *factorial design. The factors and levels are size (32, 1024), associativity (direct mapped, 8-way set associativity), replacement strategy (random, lru) and the hashing scheme (pc, value).*

strategy is 0%. ¿From these numbers and a look at the table we can make two important observations:

1. The mapping scheme is of utmost importance. The left hand side of the table which uses the PC as the index into the MEMO-TABLES shows consistently poorer results than the right hand side which uses the operand values as

   indices into the MEMO-TABLES. Thus in future simulations we will use the operand values only as indices. Section 3.6 explains this phenomena in greater detail.

2. The replacement strategy is of little importance. The top half of the table which uses a random replacement strategy has the same results as the bottom half which uses the LRU replacement strategy. The results of the direct-mapped MEMO-TABLES are self-explanatory, and the results of the set-associative MEMO-TABLES are consistent with memory caches where the replacement method has little impact on the hit-ratio[18]. Because of the simplicity of implementing a random replacement method we will use this method in future simulations.

The variation allocated to size and associativity and the results displayed prohibit us from making clean cut decisions as with the mapping and replacement method. We must investigate more levels of both size and associativity, we will do this in the next section.

3.3 Size and Associativity

The next set of simulations are targeted at determining the highest hit-ratio with the lowest MEMO-TABLE size and associativity. The levels of size are from 16 to 16K entries per MEMO-TABLE (omitting 512, 2K, and 8K sizes) and an infinitely large MEMO-TABLE, and the levels of associativity are from direct-mapped to 8-way set associative and fully associative.

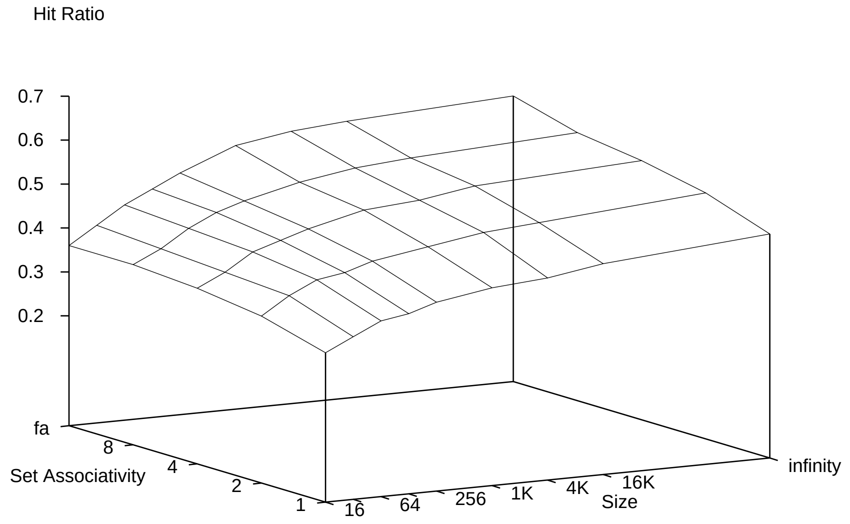

Figure 3: *hit-ratio as a factor of* MEMO-TABLE *size and set associativity.*

A two-factor full factorial design is used[19] to determine which factor influences the hit-ratio more. The total variation can be divided into parts explained by factors A (size) and B (associativity) and an unexplained part due to experimental errors. The results show that 68% of the variation is attributed to changes in the MEMO-TABLES size and 30% to changes in the associativity, 2% of the variation is unexplained.

Figure 3 is a 3-D plot of the hit-ratio (z-axis), size (x-axis), and associativity (y-axis). Looking at the lesser factor of variation, associativity, shows that raising the associativity from direct-mapped to 2-way gives a considerable hit-ratio enhancement and raising the associativity beyond 4-way hardly changes the hit-ratio. Even if it did implementing a 8-way set associative MEMO-TABLE is beyond current on-chip memory cache technology which will be used in implementing MEMO-TABLES. Current on-chip caches can perform a 4-way set associative cache lookup in a single machine cycle so there is no reason not to set the associativity of MEMO-TABLES to 4.

Looking at the plot again shows that for sizes 16 to 128 the curve rises rapidly, from MEMO-TABLE size 256 the curve starts to flatten. Table 4 shows the ratio of change between a table of size n to a table of size $2n$ and compares the hit-ratios of the above sizes to a utopian infinitely large and fully associative MEMO-TABLE. Dividing the hit-ratio of using 256 entry 4-way set associative MEMO-TABLES with the hit-ratio of using infinite fully-associative MEMO-TABLES, shows that 76% of all reusable multiple-cycle instructions can be reused with

moderate size MEMO-TABLES. This is 46% of all multiple-cycle instructions. Thus this will be the baseline MEMO-TABLE organization for the following simulations.

size	*change*	*% of inf*
$16 \rightarrow 32$	12%	0.53
$32 \rightarrow 64$	11%	0.60
$64 \rightarrow 128$	7%	0.66
$128 \rightarrow 256$	5%	0.70
$256 \rightarrow 512$	4%	0.74
$512 \rightarrow 1K$	3%	0.77
$1K \rightarrow 2K$	2%	0.79
$2K \rightarrow 4K$	2%	0.80
$4K \rightarrow 8K$	1%	0.81
$8K \rightarrow 16K$	1%	0.82
$16K \rightarrow Inf$	4%	0.83

Table 4: *Ratio of change when changing a* MEMO-TABLE *size from n to $2n$ (4-way set associativity) and the ratio between finite tables to an utopian fully associative infinite table.*

Another consideration to take into account is the *hit-time* (the time to access a MEMO-TABLE, check if the entry is resident in the MEMO-TABLE, and return the result) of a MEMO-TABLE. This time must be a single machine cycle, with most FP instructions having latencies of 2-3 cycles, a longer hit-time will reduce the effectiveness of IM. Thus the size of a MEMO-TABLE should be comparable to the size of small on-chip caches, which have a hit-time of one cycle. A 256 entry MEMO-TABLE holds $256 \times 3 = 768$ double precision values which is $768 \times 8 = 6144 = 6K$ bytes. This is less than the on-chip caches of the MIPS R10000 (32KB), Power PC 604e (32KB) and other leading microprocessors. Thus we will limit the size of MEMO-TABLES to 1024 entries (24KBytes) with a set associativity of 4.

3.4 Trivial Calculations

The result of a trivial calculation is immediately obtained from the operands of the calculation itself. No calculation is performed, just a input check is needed to detect the occurrence of triviality. In all previous simulations trivial calculations were treated as regular calculations and forwarded to the MEMO-TABLES. In this section trivial calculations will be detected in parallel to the MEMO-TABLE lookup. Thus only non-trivial calculations will be stored in the MEMO-TABLES. Table 5 shows the trivial calculations detected. Figure 4 shows the layout of a MEMO-TABLE, division unit, and trivial test unit. The calculation is tested for triviality in parallel to the MEMO-TABLE lookup and FU execution. If the calculation is a trivial one the result will be obtained from the Trivial Test Unit (TTU),and the MEMO-TABLE lookup and FU execution will be terminated. If the calculation isn't trivial the MEMO-TABLE lookup or FU execution supplies the result (for clarity each control line is drawn using a different line style).

The TTU is composed of a set of 4 comparators and a FP negative bit test

(figure 5). This covers all the triviality tests defined in table 5 in one design and enables building a uniform TTU[6].

Table 6 shows the hit-ratios for 256 entry (4-way sets) MEMO-TABLES with and without trivial calculation detection, and the percentage of trivial calculations out of all memoized instructions. An average 3% enhancement

[6] Just as an integer MEMO-TABLE is different than a FP MEMO-TABLE so is an integer TTU different than a FP TTU

is possible by just adding circuits to perform trivial calculation detection, as opposed to quadrupling the MEMO-TABLES size in order to achieve the same enhancement as shown by figure 6. For FP applications, MEMO-TABLES of size 128 with trivial calculation detection have higher hit-ratios than 1K entry MEMO-TABLES without trivial calculation detection. Our conclusions are straightforward: each MEMO-TABLE will have a TTU integrated into it, this achieves a hit-ratio enhancement comparable to a size increase of one magnitude.

Addition	$a+0, 0+a$	a
Subtraction	$a-0$	a
	$a-a$	0
Multiplication	$a\times 0, 0\times a$	0
	$a\times 1, 1\times a$	a
Division	$a/1$	a
	$0/a$	0
	$a/0$	Inf
	$0/0$	NaN
	a/a	1
Sqrt	$\sqrt{1}$	1
	$\sqrt{0}$	0
	$a<0$	NaN
Conversions	0	0
Negation	0	0
Absolute Value	0	0

Table 5: *Operation, trivial calculation, and result.*

3.5 Contents of MEMO-TABLES

Our previous simulations used a MEMO-TABLE for each instruction type. It is possible that for different applications some MEMO-TABLES won't be utilized at all, while others will suffer from capacity misses. Microprocessors have

separate Instruction and Data caches to make it possible to access them at the same cycle, not because this enhances the hit-ratio (it doesn't[18]).

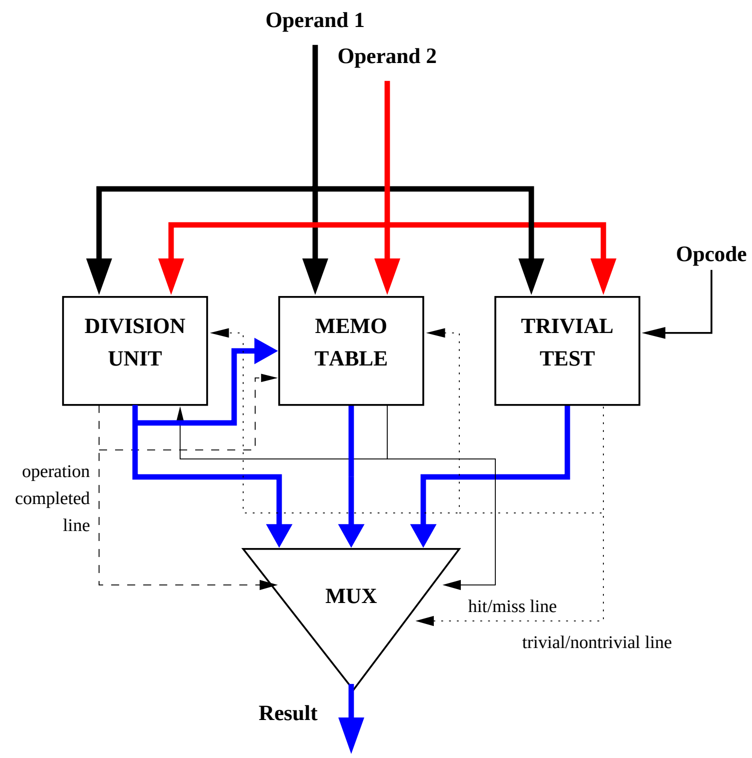

Figure 4: *Layout of a Trivial Test Unit adjacent to a* MEMO-TABLE *and Division Unit.*

On the other hand one centralized MEMO-TABLE will suffer from a longer hit-time, might have to be multi-ported, might suffer from non-uniform access due to line delays, and disallows different mapping schemes for integer and floating point values.

The average number of tables used per application is less than 12 (out of 19). When counting the number of accesses per MEMO-TABLE we discovered that the mean is lower than the standard deviation for all applications. This shows that there are many tables that are accessed relatively little and a few which are highly accessed, leading us to assume that using a unified MEMO-TABLE might enhance the hit-ratio.

Due to the problems in using a unified table mentioned earlier we added a level between MEMO-TABLE per instruction to a unified MEMO-TABLE. Five tables are used, each which contains instructions that use the same or neigh-

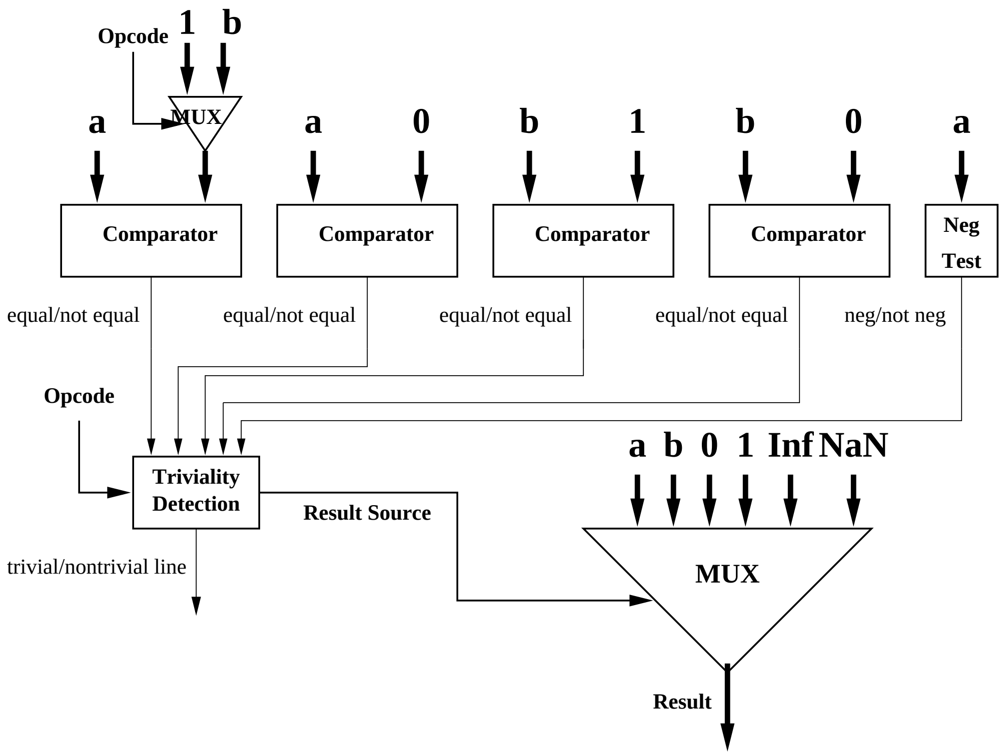

Figure 5: *Layout of a Trivial Test Unit, the opcode of the instruction determines which comparisons are used.*

boring functional units. The tables are:

1. **Integer** - integer multiplication and division.

2. **Long Latency** - floating point multiplication, division, and square root taking.

3. **Comparison** - floating point comparisons.

4. **Conversion** - conversion from single precision to double precision to integer formats.

5. **Addition** - floating point addition, subtraction, negation, and all other floating point instructions.

application	*org hr*	*new hr*	*trivial ratio*
mesa	0.42	0.51	23%
epic	0.15	0.18	4%
rasta	0.32	0.37	9%
mpeg2	0.58	0.65	51%
gsm	0.05	0.08	3%
ghostscript	0.96	0.97	57%
jpeg	0.82	0.84	54%
g721	0.49	0.51	22%
pgp	0.07	0.07	0%
tomcatv	0.19	0.28	13%
swim	0.19	0.22	7%
su2cor	0.25	0.26	5%
hydro2d	0.90	0.93	46%
mgrid	0.69	0.71	6%
applu	0.40	0.43	7%
turb3d	0.75	0.83	62%
apsi	0.35	0.40	16%
fpppp	0.40	0.44	8%
wave5	0.11	0.12	1%
gcc	0.94	0.96	72%
compress	0.13	0.13	8%
perl	0.96	0.97	1%
average	**0.46**	**0.49**	**22%**

Table 6: *Hit-ratios for 256 entry (4 entries to a set)* MEMO-TABLES *with and without trivial calculation detection, and the percentage of trivial calculations out of all memoized instructions.*

Table 7 compares using single instruction MEMO-TABLES, multiple instruction MEMO-TABLES and a unified MEMO-TABLE. Using multiple MEMO-TABLES, has the same benefits of using single MEMO-TABLES with a better utilization. Using a unified MEMO-TABLE has a better utilization but can

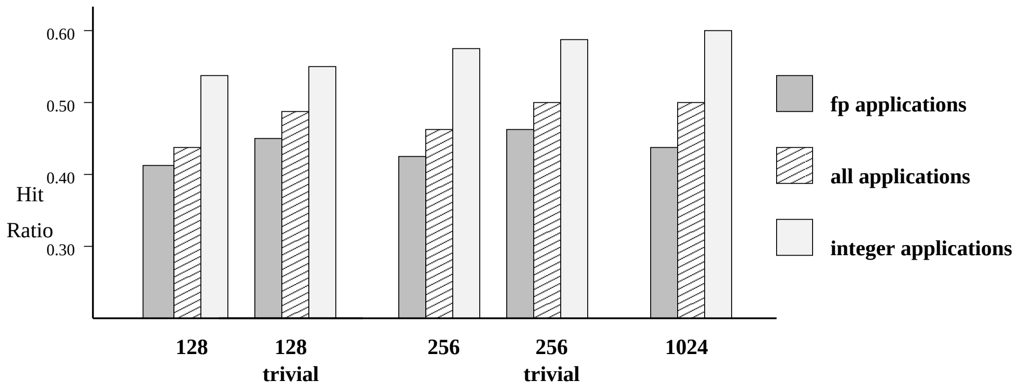

Figure 6: *Hit-ratio of* MEMO-TABLES *with and without trivial calculation detection.*

have a higher hit-time which offsets the posible hit-ratio enhancement.

trait	*single*	*multiple*	*unified*
lookup time	small table, low lookup time	small table, low lookup time	larger table, higher lookup time
table access	close to FU, uniform access	close to FU, uniform access	distant from some FUs, nonuniform access
ports	read/write 2 operands, 1 result	read/write 1 opcode, 2 operands, 1 result	read/write 1 opcode, 2 operands, 1 result, per FU
mapping	different mapping schemes	different mapping schemes	same mapping scheme for different data types
utilization	low, some tables aren't used	higher, 2-5 instruction types per table	high, all instructions use 1 table
hardware complexity	high, needs comparators and TTU per instruction	low,comparators and TTU per table	very low, one set of comparators and TTU

Table 7: *Comparison of the three* MEMO-TABLES *contents schemes.*

Figure 7 shows the hit-ratios of 19 single instruction 64 and 128-entry MEMO-TABLES, 5 multiple instruction 256 and 512-entry MEMO-TABLES, and a unified 1024 and 2048-entry MEMO-TABLE. Each level uses approximately the same amount of storage. The rest of the characteristics of the the MEMO-

TABLES are 4-way set associativity, random replacement, indexing using the LSBs of the values and opcodes, and trivial calculation detection.

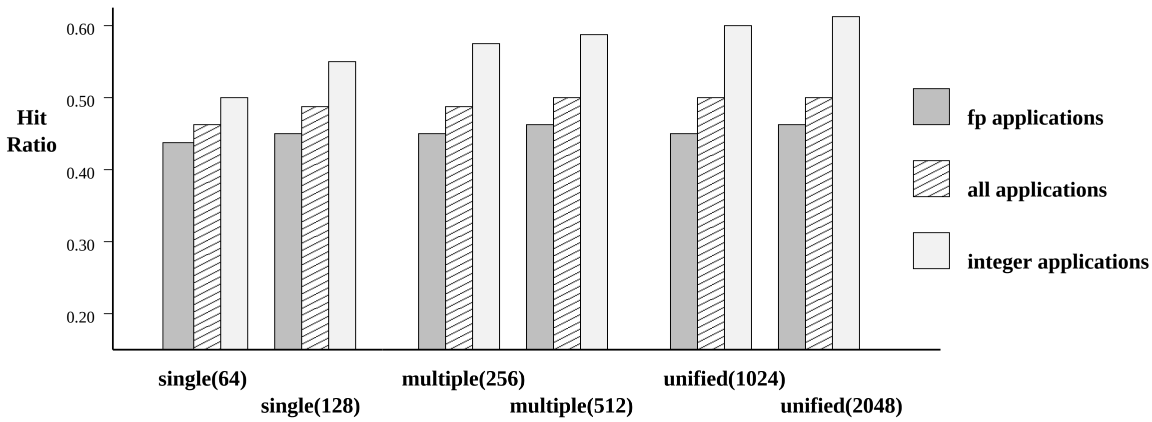

Figure 7: *hit-ratio of single, multiple, and unified contents scheme. Each* MEMO-TABLE *is 4-way set associative, uses random replacement, uses the LSBs of the operands (and opcodes) as indices to* MEMO-TABLE *entries, and performs trivial calculation detection.*

The figure shows that for floating point intensive applications, which use most of the MEMO-TABLES, the difference between the three approaches is small. For integer applications most of the MEMO-TABLES aren't used, thus the multiple MEMO-TABLE and unified MEMO-TABLE scheme just enlarge the integer multiplication and division MEMO-TABLES, which causes a higher hit-ratio. Due to the integer applications the average of all applications are slightly biased towards the unified approach.

Based on the results shown in table 7 and figure 7 we concluded that the scheme of choice is using multiple MEMO-TABLES each containing several instruction types.

3.6 Mapping Strategies

Until this point in our research we have indexed the MEMO-TABLES using two possible values, the program counter (PC) and the operand values. As shown previously using the PC as an index greatly diminishes the hit-ratio thus we will focus on indexing using the operand values (a more extensive explanation will be made in section 3.6.1). Our previous mapping strategy took the least-significant-bits (LSBs) of the value(s) (XORed them together if a two operand operation is memoized) and used them as an index into a MEMO-TABLE. The benefit of this scheme is it's simplicity and the fact that integer values and FP values can be dealt with in a similar manner.

For integer values this mapping strategy is optimal as the LSBs show the highest entropy[17]. For FP numbers this isn't necessarily true. Due to the IEEE 754 representation scheme for FP numbers, where the numbers are normalized, the most-significant-bits (MSBs) of the mantissa or the LSBs of the exponent would seem to be likely candidates for index bits.

Using these assumptions we devised four additional mapping schemes (assuming the number of sets in a MEMO-TABLE is n):

- Least Significant Bits (lsb) - The $\log_2 n$ LSBs of the mantissa.
- Mantissa (mant) - The $\log_2 n$ most-significant-bits of the mantissa are.
- Mixture 1 (mix1) - The LSB of the exponent and the $\log_2 n - 1$ MSBs of the mantissa.
- Mixture 2 (mix2) - The 2 LSBs of the exponent and the $\log_2 n - 2$ MSBs of the mantissa.
- Exponent (exp) - The $\log_2 n$ least-significant-bits of the exponent.

Figure 8 shows the schemes.

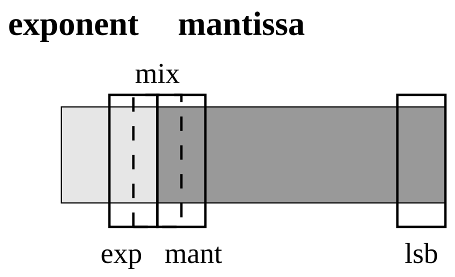

Figure 8: *The index bits are taken from the LSB of the exponent and MSB of the mantissa.*

The 5 schemes (and PC indexing) were run on the recommended MEMO-TABLES of section 3.5: multiple MEMO-TABLES of size 256 and 512 and set-associativity of 4, associativity of 1 and 2 was simulated as well, a good mapping scheme may result in having to use a lesser degree of associativity. Figure 9 shows the hit-ratios of the FP applications (as 4 of the 6 schemes aren't relevant to integer applications).

The graph shows that for a lower associativity the "middle" schemes (mant, mix1, mix2) result in noticable better hit-ratios. When the associativity is 4 the differences are much smaller with exp, mix1, and mix2 having a slight edge on the lsb and exp schemes. This is due to the flexibility of replacing entries in a set. In a direct-mapped MEMO-TABLE mapping two instructions to the same set results in conflict misses, a better mapping scheme avoids this. If the degree of associativity is higher, instructions mapped to the same set can continue to reside together in the MEMO-TABLE, thus the mapping scheme has
less impact.

For any degree of associativity and any size (the results for 512 entry MEMO-TABLES add one percent of hit-ratio to the 256 entry results) using the operand values as indices results in considerable higher hit-ratios than using the PC as an index. The conclusion of this section is that using a mix of bits from the mantissas and exponents of the operand values results in slightly better hit-ratios than the other operand value schemes and much better hit-ratios than the PC based scheme.

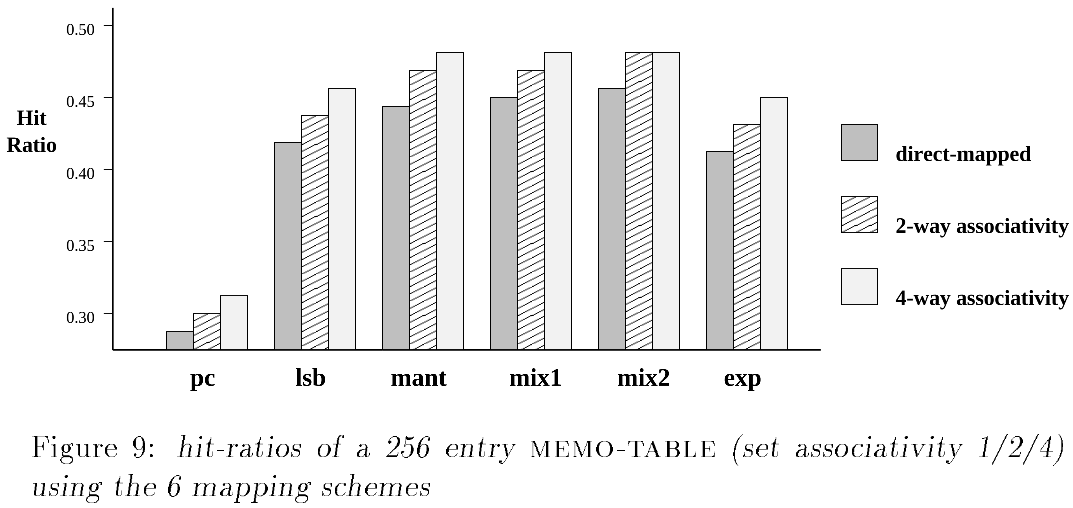

Figure 9: *hit-ratios of a 256 entry* MEMO-TABLE *(set associativity 1/2/4) using the 6 mapping schemes*

3.6.1 PC vs. Value Mapping

```
for (i=0;i<N;i++){
    for (j=0;j<N;j++){
        c[i][j] 0.0;
        for (k=0;k<N;k++)
            c[i][j] += a[i][k]*b[k][j];
    }
}
```

Table 8: *Naive matrix multiplication.*

In order to understand the differences between mapping using the PC vs. mapping using the operand values we look at a simple yet widely used application: matrix multiplication. The most naive scheme (table 8) performs N^3 multiplications when multiplying two NxN matrices. To ensure that we will have redundant multiplications we used N different coefficients which result in N^2 different multiplications. The hit-ratios (of the FP multiplication MEMO-TABLE only) for multiplying two $100x100$ matrices are shown in the top graph of figure 10 (sizes 128-1024, associativity 4, random replacement, trivial calculations stored in the MEMO-TABLE). The hit-ratios when the PC is used as an index are invariant to the size of the MEMO-TABLE, this is easily explained by looking at the code. All multiplications are executed by one instruction, thus all multiplications are mapped to a single set, leaving the rest of the MEMO-TABLE

unused.

Fixing the MEMO-TABLE size and varying the associativity is shown in the bottom graph of figure 10 (size 512, associativity 1-512, random replacement, trivial calculations stored in the MEMO-TABLE). Only when using a fully-associative MEMO-TABLE do the hit-ratios match, this is again due to the fact that all multiplications are mapped to the same set.

When more complex algorithms such as loop unrolling, tiling, and sub-blocking are used the PC indexed hit-ratios are even worse. The multiplication calculations are performed by several instructions which leads to a better MEMO-TABLE utilization but results in a lower hit-ratio. This is due to the fact that the same calculation might be performed by different instructions

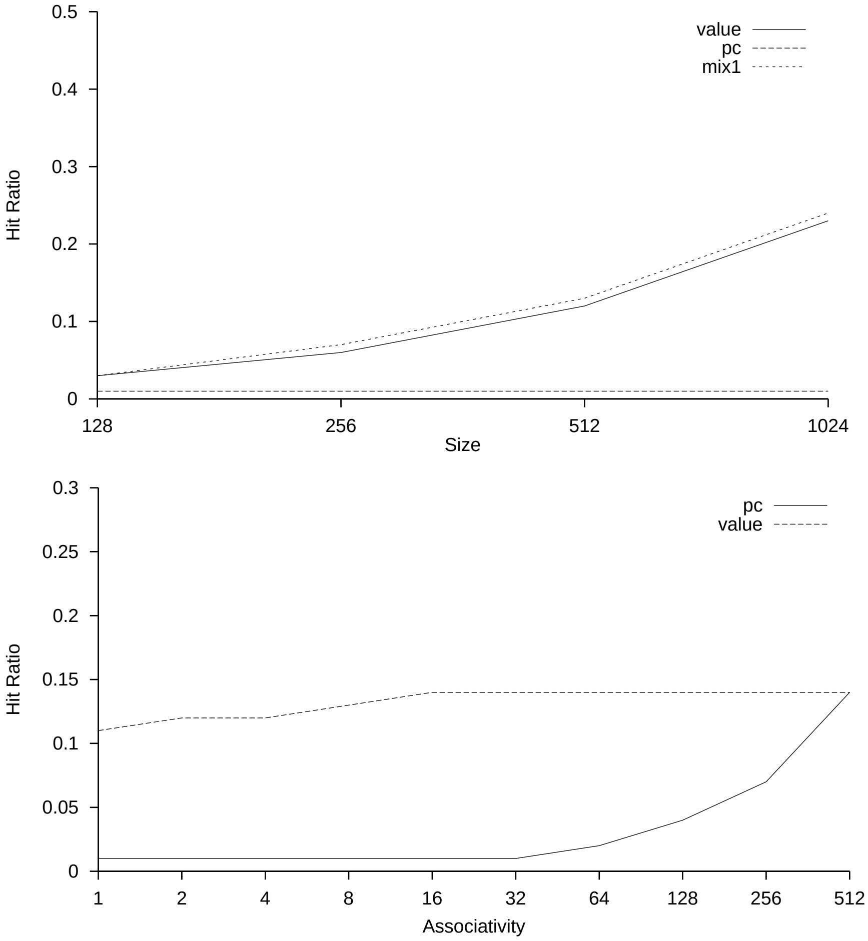

Figure 10: *Hit-ratios of the multiplication* MEMO-TABLE *in matrix multiplication.*

and thus mapped to different sets, causing MEMO-TABLE misses instead of hits.

4 Speedup

In order to gauge the extent of enhancement gained by using Instruction Memoization we must view the speedup achieved when IM is implemented.

Using Simplescalar we simulated in detail running the benchmarks on two modern microprocessors: The MIPS R10000 and PPC 604e. Table 9 shows the Fraction Enhanced (FE) the fraction of cycles attributed to multiple-cycle instructions[7], The hit-ratio on the MEMO-TABLES, and the speedup obtained.

The table shows that on the R10000 9% of the processor cycles are spent on multiple-cycle instructions, 49% of those instructions are successfully memoized which leads to an average speedup of 7%. On the 604e 13% of the processor cycles are spent on multiple-cycle instructions, 46% of those instructions are successfully memoized which leads to an average speedup of 6%. One possible reason that the R10000 exhibits a lower FE is due to the lower latency of FP instructions, 2 cycles for most instructions as opposed to 3 cycles on the 604e (table 2). Why the speedup on the R10000 is greater is beyond the scope of this paper and will be answered when the influence of the datapath on IM is explored.

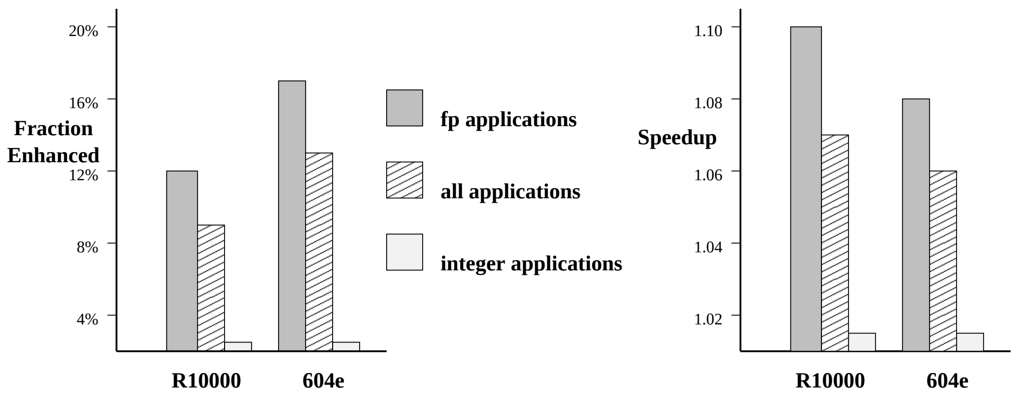

Figure 11: *Comparison of the FE and speedups between FP to integer applications.*

Figure 11 compares the FP and integer applications. As expected the FE and as a consequence the speedup of the integer applications are around 1% as opposed to the FP applications which display speedups of 8% (604e) and 10% (R10000). As a result future IM research should be focused on

[7] This fraction was obtained by simulating a processor where all multiple-cycle instructions have a latency of one cycle. The number of cycles from this run divided by the number of cycles from a regular run, subtracted from 1 is the Fraction Enhanced.

application	*MIPS R10000*			*PPC 604e*		
	FE	*hr*	*spdp*	*FE*	*hr*	*spdp*
mesa	6%	0.51	1.04	16%	0.45	1.06
epic	17%	0.24	1.04	21%	0.14	1.02
rasta	10%	0.39	1.06	13%	0.31	1.06
mpeg2	13%	0.72	1.09	14%	0.62	1.11
gsm	5%	0.09	1.00	10%	0.08	1.01
ghostscript	35%	0.86	1.40	23%	0.86	1.20
jpeg	1%	0.73	1.01	1%	0.73	1.01
g721	2%	0.49	1.01	1%	0.38	1.00
pgp	2%	0.05	1.00	0%	0.09	1.00
average	10%	0.45	1.07	11%	0.40	1.05
tomcatv	8%	0.32	1.07	12%	0.30	1.05
swim	2%	0.24	1.01	15%	0.24	1.04
su2cor	8%	0.36	1.06	11%	0.31	1.03
hydro2d	8%	0.92	1.07	21%	0.92	1.23
mgrid	3%	0.67	1.03	8%	0.67	1.07
applu	1%	0.49	1.01	3%	0.49	1.01
turb3d	6%	0.83	1.06	6%	0.82	1.05
apsi	20%	0.38	1.16	32%	0.38	1.11
fpppp	3%	0.42	1.02	9%	0.42	1.02
wave5	50%	0.14	1.44	64%	0.13	1.18
gcc	2%	0.95	1.02	1%	0.95	1.01
perl	0%	0.73	1.00	0%	0.71	1.00
compress	1%	0.16	1.00	2%	0.14	1.00
average	9%	0.51	1.07	14%	0.49	1.06
average	**9%**	**0.49**	**1.07**	**13%**	**0.46**	**1.06**

Table 9: *FE, hit-ratios, and speedups on the MIPS R10000 and PPC 604e when IM is implemented.*

the Floating Point Unit and on memoizing single-cycle instructions using the operand values as indices.

5 Conclusions

This paper investigates the structure of lookup tables (coined MEMO-TABLES) used in Instruction Memoization (IM). The characteristics of the MEMO-TABLES explored were its size, associativity, replacement method, indexing methods, contents (instruction mix in the MEMO-TABLES) and the detection of trivial calculations.

Our main conclusions from the simulations regarding the organization of MEMO-TABLES are:

- The replacement method is irrelevant, random is as good as LRU.
- A degree of set associativity higher than four is unnecessary.
- Enlarging a MEMO-TABLE beyond a certain point results in diminishing returns as the hit-time increases as well as the hit-ratio.
- Using several MEMO-TABLES for different instruction types enables accessing them concurrently but not having to implement a MEMO-TABLE for every instruction type.
- Using the Program Counter (PC) as the index into a MEMO-TABLE results in much poorer hit-ratios than when the operand values are used as indices.
- By detecting trivial calculations, and not entering the operations into the MEMO-TABLES, a hit-ratio improvement is achieved that is comparable to a four-fold size increase.

Specifically we recommend implementing IM with 5 MEMO-TABLES: (i) for long-latency instructions (FP div, mult, sqrt), (ii) integer instructions (INT div and mult), (iii) FP comparisons, (iv) FP$\Leftrightarrow$INT conversions, (v) and all other FP instructions (add, sub, ...). Each MEMO-TABLE contains 256 entries in sets of 4. Entries are replaced randomly and are indexed by the 2 LSBs of the exponent and the 6 MSBs of the mantissa XORed with the opcode.

Trivial calculations aren't entered into the MEMO-TABLES but are detected with dedicated circuitry.

This organization yields an average hit-ratio of 0.49, this is over 80% of the hit-ratio obtained when using an infinite fully-associative MEMO-TABLE (0.60 hit-ratio).

Another conclusion is that IM is much more effective for Floating Point applications than for integer applications. This is due to the fact that all FP instruction have multiple-cycle latencies. Thus in our research of the impact of IM on the datapath we must find a way to memoize single-cycle instructions as well.

Integrating the proposed design into the datapaths of two microprocessors the MIPS R10000 and PPC 604e results in an average 8% (604e) and 10% (R10000) speedup for FP applications.

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] C. Molina, A. González, and J. Tubella, "Dynamic Removal of Redundant Computations", *Proc. of the ACM Int. Conf on Supercomputing*, June 1999.

[3] 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.

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

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

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

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

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

[9] D. Citron and D. Feitelson, "Hardware Memoization of Mathematical and Trigonometric Functions", *Technical Report, 2000-5*, Hebrew University of Jerusalem, March 2000. Jerusalem.

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

[11] 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.

[12] 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.

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

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

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

[16] 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.

[17] D. Citron and L. Rudolph, "Creating a Wider Bus Using Caching Techniques", *Proc. of 1st Int. Symp. on High Performance Computer Architecture (HPCA)*, January 1995.

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

[19] Jain R., *"The Art of Computer Systems Performance Analysis"*, John Wiley & Sons, 1991.