"Instruction Memoization: Exploiting Previously Performed Calculations to Enhance Performance"

A dissertation submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

by

Daniel Citron

Submitted to the senate of the Hebrew University in the year 2000

"Instruction Memoization: Exploiting Previously Performed Calculations to Enhance Performance"

A dissertation submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

by

Daniel Citron

Submitted to the senate of the Hebrew University in the year 2000

This dissertation was compiled under the supervision of
Dr. Dror G. Feitelson.

Abstract

This thesis explores the concept named memoization: *saving the input(s) and output(s) of previously calculated (side-effect-free) functions, and using the output if the input is encountered again.* Our focus will be on the memoization of instructions. The operands and results of previous invocations of multi-cycle instructions are saved in dedicated tables named MEMO-TABLES. Successful lookups in these tables, before or parallel to instruction execution, make it possible to improve execution by reducing the latencies of these instructions to one cycle. We named this technique *Instruction Memoization (IM).*

To test this idea we used a detailed RISC processor simulator running the SPEC and MediaBench benchmarks. We first explore the organization of the MEMO-TABLES in search for an "optimal" design that will maximize hit-ratio and minimize cost. A hit-ratio of over 50% is achieved for moderate sized tables.

Next we integrated IM into a RISC super-scalar processor's datapath. We discovered that 13% of the benchmarks' execution time can be attributed to multi-cycle instructions. With a 52% hit-ratio an average (harmonic mean) speedup of 1.07 was obtained (1.10 for FP intensive applications). In our search for greater performance improvement we decided to memoize single-cycle instructions as well.

The speedup rised by 50% to 1.11 (1.13 for FP applications). However the new speedup is attributed not to instruction latency reduction but rather to the artificial addition of more Functional Units (FUs). The MEMO-TABLES act as "virtual" FUs. Adding more FUs to a processor nullifies the effect of single-cycle IM. On the other hand multi-cycle IM yields a better speedup for faster processors.

Contents

Chapter 1

Introduction

In the field of Computer Architecture the end goal of almost all innovations and enhancements is speed. We want our programs to run in less time. This can be achieved in numerous and various ways: running the processor at higher speeds, introducing changes to the design of the processor, changing the instruction set, compiler enhancements, and finally by altering the programs themselves.

This thesis will focus mainly on enhancing the design of the *datapath*. the datapath is by analogy the "blood system" of the processor. Through its stages flow the instructions fetched from memory. The instructions are decoded, their operands are obtained, they are executed, the results of the instructions are written back to memory or the register file, and finally the instructions are committed and exit the datapath. During each cycle, a tick of the processor's clock, instructions either flow through the datapath or are delayed in the datapath until previous instructions have progressed through the stages.

The less cycles it takes instructions to traverse the datapath the faster the program will execute. This thesis shows a technique that shortens the stay of some of the instructons in the datapath. Just as memory caching exploits the "Principal of Locality" in order to present the processor with a short and almost uniform memory access time, we will exploit the concept of *memoization* in order to shorten the execution time of many long latency instructions.

1.1 What is Memoization?

The concept of memoization is as follows: *saving the input(s) and output(s) of previously calculated (side-effect-free) functions, and using the output if the input is encountered again.*

Before a side-effect-free function is to be computed its input(s) are used to access (usually with a hash function) a Look Up Table (LUT). If the inputs are resident in the LUT the previously calculated output(s) is obtained from the table and recalculation of the function is averted. If the input(s) aren't in the LUT the function is calculated and its input(s) and output(s) are stored in the

LUT for future reference.

This technique can result in faster recalculations if the storage and lookup of formerly calculated functions is faster then recalculating the function again. But in the general case the LUT is a software based table residing in main memory. Thus the lookup and storage are time consuming. A successful lookup must have a lower access time than calculating the function. Every unsuccessful lookup results in a penalty. Thus a high successful lookup ratio is necessary in order to benefit from memoization. Coupled with the fact that most software based functions aren't side-effect-free, the use of memoization seems limited.

But when looking "right under your code", we find that almost all instructions are side-effect-free (except for memory accesses). And if the LUTs are dedicated tables located on-chip the lookup and storage times are now very short. Thus memoizing instructions seems a much better prospect than memoizing functions. This technique is named *Instruction Memoization (IM)* and is the topic of this thesis.

1.2 Instruction Memoization

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 Table (LUT), which we will call a MEMO-TABLE. When the same instruction type with matching operands is encountered again the result is obtained from the MEMO-TABLE and instruction execution is avoided. The "execution time" of the instruction is the access time of the MEMO-TABLE, which is a single machine cycle for a small hardware based table. When the lookup is unsuccessful the instruction must be executed in one to tens of cycles (depending on the instruction type). Thus for successful lookups the execution time of these instructions is one cycle, which in turn minimizes their occupancy in the datapath which leads to shorter execution times.

The performance improvement (speedup) obtained is dependent on four major factors:

1. The percentage of instructions that can benefit from memoization. Instructions that have a *latency* (number of cycles from execution start until the result is ready) of a single-cycle and instructions that must be executed (stores to memory) are examples of instructions that aren't candidates for memoization. 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 stage of the pipeline that MEMO-TABLES are accessed, multiple-issue of instructions, long-latency instructions completing sooner than expected, and the penalty of an unsuccessful lookup. All these issues affect the usefulness of IM.

3. The percentage of successful lookups, i.e. the hit-ratio of the MEMO-TABLE. This is influenced by the nature of the program being executed, how much redundancy it contains, and by the design of the MEMO-TABLE.

4. The physical integration of IM modules on the processor: The number of transistors needed to implement IM, the added power consumption, and the complexity of design all influence the Cost/Performance ratio of implementing IM. This thesis is an architectural research, the physical aspects of implementation are beyond the scope of this work. However the issues will be addressed, tradeoffs compared (not always quantitatively), and solutions given for the problems.

In this thesis we will explore all four factors in order to understand the impact of memoization on the processor and in order to obtain the best possible performance enhancement when using IM.

1.3 Prior and Related Work

This section will survey prior and closely related work. At this point in the thesis we won't compare our technique to these works but just present them as is. In chapter 6 after the technique of IM has been fully presented we will compare it to several of the alternate and complementing approaches of reusing previous computations.

1.3.1 Early Use of Memoization

The concept of *memoing* was introduced by Michie [1] in 1968. The idea is to save the inputs and results of side-effect-free functions in a table and reuse the results for matching inputs. Since then it has been used mainly in the context of declarative languages like Prolog, Lisp, and ML [2, 3, 4].

In 1982 Harbison [5] proposed a stack-oriented architecture called the *Tree Machine (TM)* which assumes the role of an optimizing compiler by detecting and eliminating common subexpressions (CSEs) and invariant expressions in loops. It performs this by using a *value cache*. Results of instructions are saved in the value cache. If the same instruction is to be executed and its operands haven't been changed, the result is obtained from the value cache instead of being performed again. Thus the scope of optimizations can be widened to expressions that aren't available at compile time.

The idea of exploiting redundant computation for off-the-shelf RISC architectures was introduced by Richardson [6] in 1992. The results of multiplication, division, and square-root instructions are saved in dedicated tables. When the instructions are to be executed a lookup in the table is performed and if the lookup is successful the result is obtained from the table (this is in fact memoization). This idea was further expanded by Flynn and Oberman [7] (1995) to include storing the reciprocals of division instructions.

1.3.2 Instruction Reuse

In 1997 Sodani & Sohi [8] introduced the concept of *Instruction Reuse (IR)*. All instructions, even single-cycle instructions are candidates for reuse. The instructions are inserted in a table called the *Reuse Buffer (RB)*. Instructions in the RB are accessed using the Program Counter (PC). If the operand values of the instruction match the operands values in the RB, the result is obtained from the the RB. Variations of the scheme include matching the operand register names (requires invalidation of entries if the registers were written into), and matching instructions that supply the current instruction with its operands (again requires invalidation). The technique of IR is closely related to IM and in some cases overlaps it. In chapter 6 we will describe the differences in detail.

1.3.3 Other Techniques

Other techniques such as Value Prediction (VP) (Gabbay & Mendelson [9] , Lipasti, Wilkerson & Shen [10, 11], and Sazeides & Smith [12]), Compiler-Directed Dynamic Computation Reuse (Connors and Hwu [13]), and Value Profiling (Calder, Feller & Eustace [14]) will be presented in more detail in chapter 6.

1.4 Thesis Outline

The rest of this thesis covers the following topics: Chapter 2 describes how IM works and shows the rationale behind its success. Chapter 3 explores various organizations of the Memo-Table. Chapter 4 describes the integration of IM into the processor's datapath. Chapter 5 shows how single-cycle instructions can use IM. Chapter 6 compares IM to other similar research efforts and chapter 7 concludes this thesis. Two appendices at the end of the thesis show how IM performs on real world processors (appendix A) and appendix B widens the scope of IM to include function memoization.

Chapter 2

Instruction Memoization

In this chapter we will describe in detail how Instruction Memoization (IM) works and the basic structure of the Memo-Table. The idea is to mitigate the effect of *multi-cycle* instructions (instructions with a latency of more than one cycle) by reducing their latency via IM. The input (operands) and output (result) of particular instruction types are stored in a cache-like lookup table (the Memo-Table). The Memo-Table is accessed in parallel to the conventional computation. A successful lookup gives the result of a multi-cycle computation in a single cycle, and a failed lookup doesn't necessitate a penalty in computation time. Figure 2.1 shows a schematic layout of the idea. The operands are forwarded in parallel both to a division unit and its adjacent Memo-Table.

2.1 The Memo-Table

A Memo-Table is a cache-like Look Up Table (LUT), that is placed adjacent to each Functional Unit (FU) that has a latency of multiple cycles. 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 2.2 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

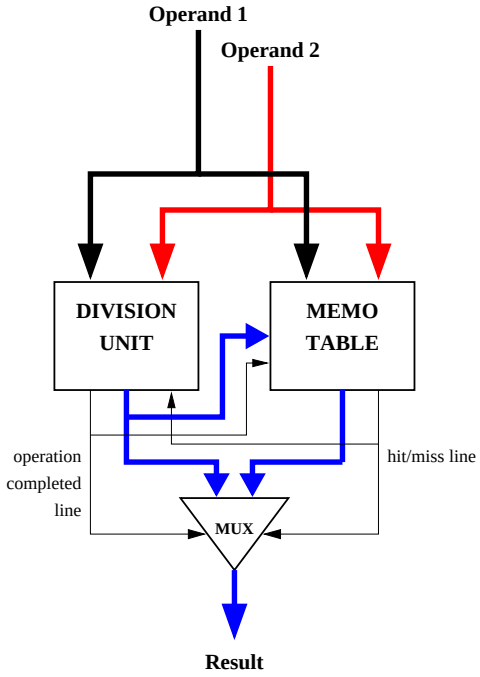


Figure 2.1: *A division unit using a* MEMO-TABLE

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, and 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[1]. The only time invalid data is in a MEMO-TABLE is during startup, initially loading the opcode fields with invalid opcode solves this problem.

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 in the FU is aborted and it 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, and the result obtained is forwarded to the next stage and in parallel entered into the MEMO-TABLE.

[1] Except if different IEEE 754 rounding modes are used.

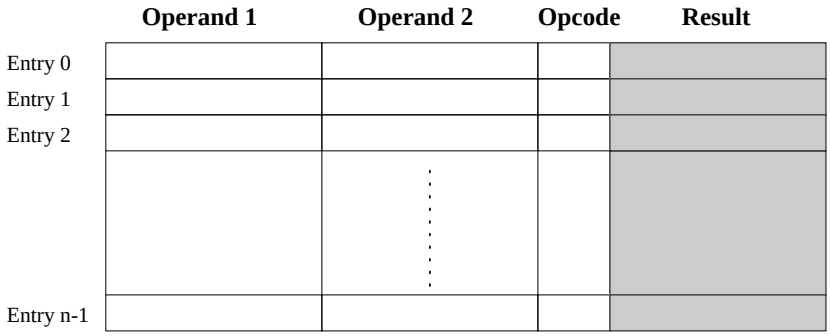


Figure 2.2: *Layout of a n entry* MEMO-TABLE.

2.2 The Rationale Behind Instruction Memoization

After we have shown the basic IM technique we will explain why it should work. To best understand the rationale a few examples will be presented:

vsqrt The application vsqrt takes the square-root of all pixels in an image. We have previously shown [15] that neighboring pixels in an image tend to have the same values, thus leading to a high hit-ratio in the MEMO-TABLE.

vspatial Performs image enhancement based on local histograms. An examination of a sample image, a self portrait of Guya (figure 2.3), shows that out of 256 possible pixel values only 161 are represented (figure 2.4). Zooming in to an 8x8 window surrounding Guya's nose (figure 2.5) shows that there are only 11 unique values. Building a histogram of this windows and running the following loop:

```
n = N*N; /* N=8 */
for(i=0;i<L;i++) /* L = # of values */
   e += (hist[i]/n) * log2(hist[i]/n);
```

results in a 94% hit-ratio when memoizing division. The same is true for color images which are composed of three "bands" (red, green, and blue images). Each band displays a similar amount of redundancy.

tomcatv In the following code excerpt [2]:

```
A = 0.25 * (XY*XY+YY*YY)
B = 0.25 * (XX*XX+YX*YX)
```

[2] This excerpt was taken from Richardson's paper [6].

The number of unique pairs is 769. Using an "infinite" multiplication MEMO-TABLE results in an almost perfect hit-ratio.

As we can see the nature of the programs and inputs causes instruction repetition. Most Multi-Media applications work on local areas of an image or signal which may result in the same calculations being performed over and over again. Of course not all programs that exhibit redundancy have source code excerpts that pinpoint the cause, most don't.

Figure 2.3: *A self portrait of Guya.*

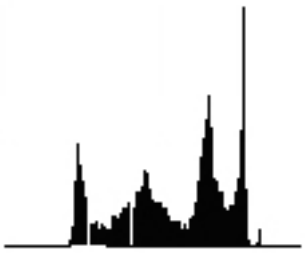


Figure 2.4: *Histogram of the Guya image.*

Sodani and Sohi [16] have performed a detailed analysis of instruction repetition for the SPEC 95 integer benchmarks and have found that most of the repetition originates from internal values of the program (immediates) or from global initialized data. Our conclusions are that for most Floating Point benchmarks the redundancy originates from the input sets of the applications [15].

Figure 2.5: *A blowup of Guya's nose.*

Chapter 3

The Organization of the Lookup Tables

This chapter is dedicated to finding the near optimal design for MEMO-TABLES that will enable us to receive the maximal hit-ratio possible (for finite MEMO-TABLE sizes). In this chapter we memoize all instructions that have a latency of more than one cycle[1]. The MEMO-TABLE we will explore is the same as proposed in chapter 2. Each entry consists of two operands, a result, and an opcode. The organization of the processor's datapath is irrelevant at this stage of the research and will be explored in chapter 4 after we fix the MEMO-TABLE characteristics.

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), detection of trivial calculations that can be computed easily ($x + 0$, $y * 1$, ...),and the relationships between instructions types ($a + b = c \rightarrow c = b - a$, ...).

3.1 Simulation Framework

To find the optimal design of a MEMO-TABLE we performed a series of experiments with an architecturally detailed simulator: SimpleScalar [17], a 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 IM.

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

[1] Except memory accesses which aren't side-effect free (stores) or aren't context free (loads).

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.

The emphasis of the simulations in this chapter will be on enhancing the hit-ratios of the MEMO-TABLES. The speedup achieved by using IM will be shown in chapter 4.

3.1.1 Simulations

The simulations were performed using the SimpleScalar 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).

3.1.2 Benchmarks

The benchmarks were taken from several sources:

- **SPEC CFP95** - the floating point component of the SPEC CPU95 suite [18].
- **SPEC CINT95** - the integer component of the SPEC CPU95 suite [18].
- **MediaBench** - a suite of multi-media and communication applications from UCLA [19].

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 3.1 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

[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. 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). The SPEC benchmarks were run with the test or train versions of the inputs in order to keep them relatively short, running them with the reference inputs gives similar results.

suite	*application*	*description*	*input*	*# of insts*	*%*
MediaBench	rasta	Speech recognition	ex5_c1.wav	23M	10.4%
	mesa	3D graphics library	hardcoded	130M	17.8%
	mpeg2	Video compression	mei16v2.m2v	1282M	7.8%
	epic	Image compression	lenna.pgm	60M	15.5%
	gsm	Speech transcoding	clinton.pcm	223M	14.9%
	ghostscript	Postscript interpreter	tiger.ps	1294M	4.4%
	g721	Voice compression	clinton.pcm	529M	0.6%
	pgp	Cryptography	pgptest.pgp	159M	2.3%
	jpeg	Image compression	monalisa.jpg	161M	0.3%
CFP95	tomcatv	Vectorized mesh generation	train.in, ITACT=20	818M	10.4%
	swim	Shallow water equations	train.in	842M	26.3%
	su2cor	Monte-Carlo method	test.in	1050M	12.8%
	hydro2d	Navier Stokes equations	test.in	1124M	16.4%
	mgrid	3D potential field	train.in, NTIMES=1	382M	14.5%
	applu	Partial differential equations	train.in, itmax=20	1000M	7.7%
	turb3d	Turbulence modeling	train.in, nsteps=1	398M	7.5%
	apsi	Weather prediction	test.in	888M	22.6%
	fpppp	Quantum chemistry	train.in	344M	32.8%
	wave5	Maxwell's equation	test.in,nsteps=2	1389M	31.7%
CINT95	gcc	C compiler	1stmt.i	119M	0.3%
	compress	Lempel-Ziv compression	test.in	35M	0.5%
	perl	Perl interpreter	scrabll.pl, train input	40M	0.4%

Table 3.1: *Description of benchmark applications, inputs, number of instructions executed, and percentage of multiple-cycle instructions.*

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 (harmonic mean) hit-ratios of the MEMO-TABLES for all the above applications[5].

3.1.3 The Instructions Memoized

All the instructions memoized have a latency of more than one cycle. These include integer division and multiplication and all the floating point instructions. Table 3.2 lists the instructions memoized along with their latencies and throughputs[6] on the R10000 and 604e[7]. For each instruction type there is a

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

[5]The average is unweighed, every benchmark, short or long running, has an equal standing. We didn't want the SPEC benchmarks, which have a longer execution time, to dominate the results.

[6]If an unit is pipelined it can complete an instruction every cycle, this is the throughput of the instruction.

[7]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

MEMO-TABLE that stores the operands and results of the instances of the instruction, for a total of 19 such MEMO-TABLES in use.

instruction type	*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↔FP Conversion	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/dp)	12/19	14/21	18/31	18/31
FP Sqrt (sp/dp)*	18/33	20/35	50/60	50/60

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

Table 3.2: *Instruction latencies and throughputs for the MIPS R10000 and PPC 604e.*

3.2 MEMO-TABLE Structural Factors

We first measured the effects of four factors 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 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), Most Recently Used (MRU) and true LRU. As memoization isn't speculative we don't explore any confidence schemes, once a value is in the MEMO-TABLE it is valid.
- Mapping Strategy - How an entry is mapped to a set. The levels are to hash the Program Counter (like [8] do) 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.

implementation of the sqrt function can take over 1000 cycles).

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)$\times$(5 levels of replacement schemes)$\times$(6 levels of mapping strategies) = 1800 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* [20] 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 for simulation are: Size - 32, 1024 entries; Associativity - direct mapped, 8-way set associative; Replacement Strategy - random, LRU; Mapping - PC, value (LSBs); The results (harmonic mean hit-ratios of all applications) are shown in table 3.3.

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 3.3: 2^4 *factorial design and resulting hit-ratios. 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).*

The results obtained are inserted into a Sign-Table. The *sample variance* of the data is calculated by computing the *Sum of Squares Total (SST)*, this number can then be broken into its components. The main components of variation are: Mapping scheme - 55%, Associativity - 31%, and Size - 10%. The variation attributed to the replacement 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. This is consistent with memory caches where the replacement method has little impact on the hit-ratio [21]. The reason is that values that are highly reused will be reentered into the MEMO-TABLE, even if they were randomly evicted. Because of the simplicity of implementing a random replacement method we 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 (1MB entries), and the levels of associativity are from direct-mapped to 8-way set associative and fully associative (for large MEMO-TABLES an associativity of 512 was used).

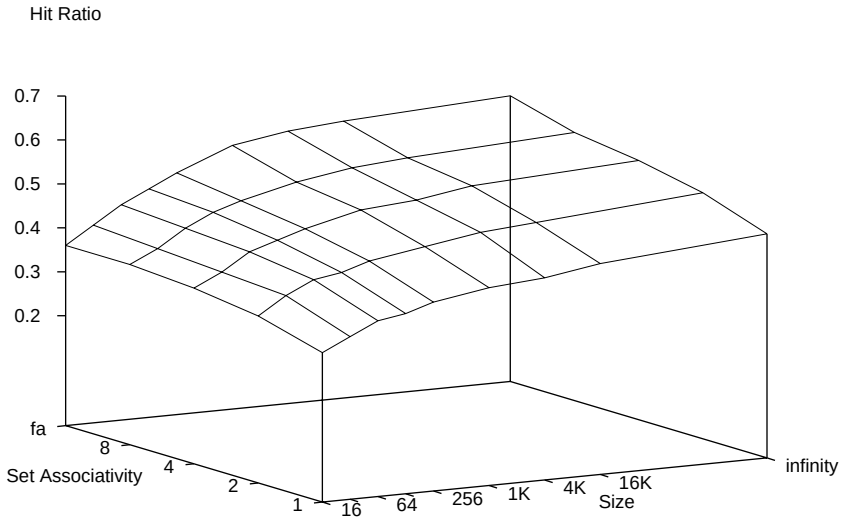


Figure 3.1: *Hit-ratio as a factor of* MEMO-TABLE *size and set associativity.*

A two-factor full factorial design is used [20] 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.

Size/Assoc	*1*	*2*	*4*	*8*	*full*
16	0.29	0.33	0.35	0.36	0.36
32	0.32	0.37	0.38	0.39	0.40
64	0.35	0.40	0.42	0.43	0.44
128	0.36	0.41	0.44	0.46	0.47
256	0.38	0.43	0.46	0.48	0.50
1K	0.40	0.45	0.49	0.51	0.55
4K	0.41	0.47	0.50	0.53	0.57
16K	0.43	0.48	0.52	0.54	0.58
infinite	0.46	0.51	0.54	0.56	0.60

Table 3.4: *Tabular version of hit-ratio as a factor of* MEMO-TABLE *size and set associativity.*

Figure 3.1 is a 3-D plot of the hit-ratio (z-axis) as a function of size (x-axis), and associativity (y-axis) (the actual results are in table 3.4). 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. This is fortunate as implementing a 8-way set associative MEMO-TABLE is the cutting-edge [22] of 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. 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.

Figure 3.2 shows the breakdown of hit-ratios per instruction (associativity: 4; size: 32–1024). It is noticeable that the hit-ratios for the integer instructions are amongst the highest and they continue to benefit from a larger MEMO-TABLE after the hit-ratios for other instructions flatten out (as does single precision to double precision conversion). For the square-root, FP comparison, and FP$\leftrightarrow$INT conversion instructions the hit-ratio is invariant to MEMO-TABLE sizes above 128 entries. For FP move a MEMO-TABLE of size 64 is sufficient. Nevertheless, in order to work with a uniform MEMO-TABLE size we will use a baseline size of 256 in future simulations.

Another consideration to take into account is the *hit-time* (the time to

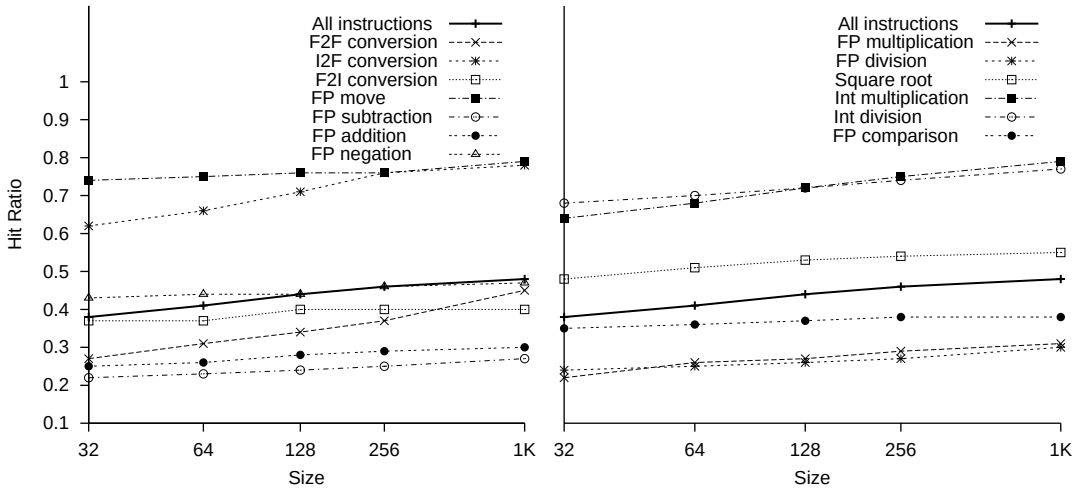


Figure 3.2: *Breakdown of hit-ratio by instruction type (4-way set associativity, random replacement, mapping by value).*

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 considerably less than the on-chip caches of the MIPS R10000 (32KB), Power PC 604e (32KB) and other leading microprocessors. Thus in any case the upper limit on the size of MEMO-TABLES will be 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 3.5 shows the trivial calculations detected. Figure 3.3 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 trivial the result will be obtained from the Trivial Test Unit (TTU),and the MEMO-TABLE lookup and FU execution will be terminated. If

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 3.5: *Operation, trivial calculation, and result.*

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, a FP negative bit test, and combinational logic to detect triviality (figure 3.4). This design covers all the triviality tests defined in table 3.5 and enables building a uniform TTU[8].

Table 3.6 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 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 3.5. 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.

Table 3.7 shows the main trivial operation contributers. For each instruction type: the trivial operation ratio, the percentage out of all trivial instructions, and the breakdown of trivial values detected is displayed. The tables shows that 93% of all trivial instructions contain the values one or zero. Thus we can simplify the triviality check by just testing for zero and one. We can further narrow down the scope of the triviality test by just checking triviality for the top contributers (multiplication, addition, subtraction, and division) but for the sake of uniformity we will check triviality (zero and one only) for all relevant instructions. Thus 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 order of magnitude.

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

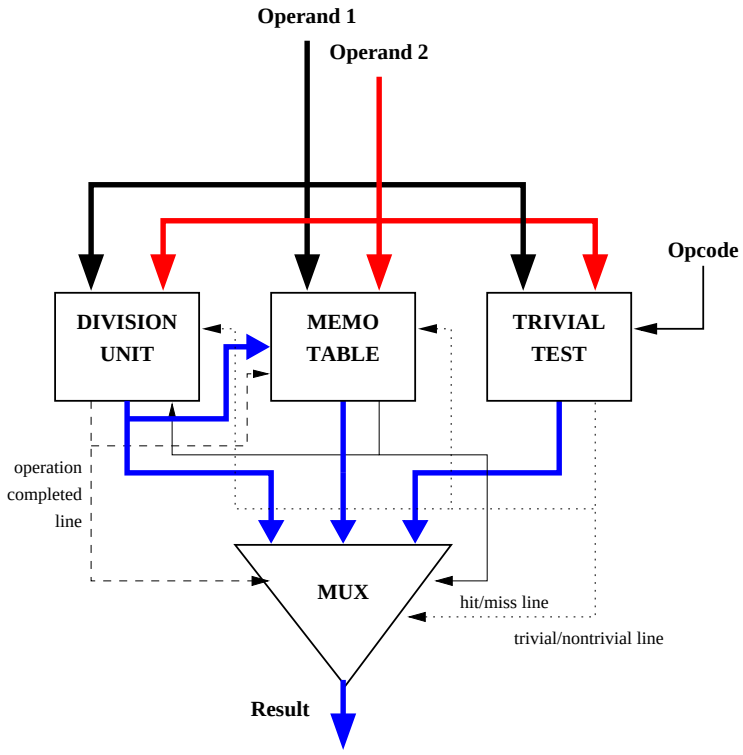


Figure 3.3: *Layout of a Trivial Test Unit adjacent to a* MEMO-TABLE *and Division Unit.*

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 [21]). 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.

Our previous simulations show that the average number of MEMO-TABLES used per application is 11.7 (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 suggest adding a level between MEMO-TABLE per instruction to a unified MEMO-TABLE.

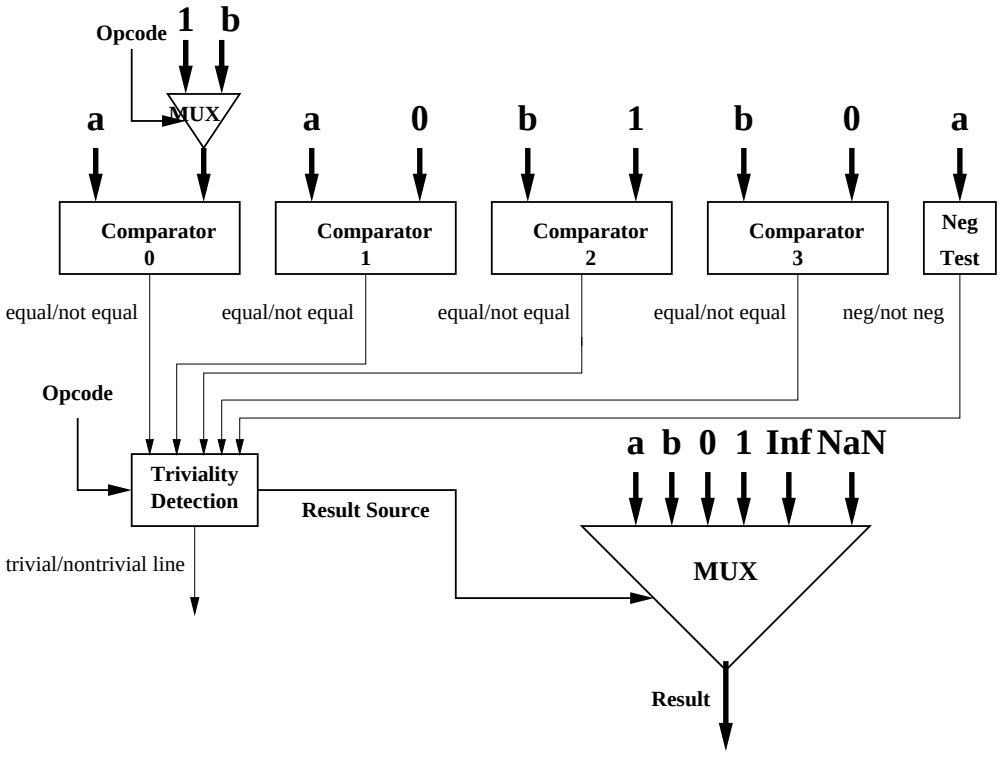


Figure 3.4: *Layout of a Trivial Test Unit, the opcode of the instruction determines which comparisons are used.*

The motivations for a specific setup are the utilization per table, the functional units that process each instruction, and the effect of the MEMO-TABLES' size on the hit-ratio. Each MEMO-TABLE will contain one heavily executed instruction and one or more under utilized instructions. Thus are choice of tables is:

1. **Integer** - integer multiplication (heavily used) and division (lightly used). Both use the same unit (604e) or adjacent units (R10000). This table will be the largest (double size) as the hit-ratio constantly rises for a larger MEMO-TABLE size (section 3.3). **Total dynamic instruction count: 35%**.

2. **Long Latency** - floating point multiplication (heavily used), division, and square root taking. Usually share circuitry in most microprocessors. **Total dynamic instruction count: 24%**.

3. **Addition** - floating point addition. **Total dynamic instruction count: 18%**.

4. **Subtraction** - floating point subtraction (moderately used), negation,

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%
harmonic mean	**0.46**	**0.49**	**22%**

Table 3.6: *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.*

instruction	*tr/ac*	*inst/all*	*value breakdown (%)*			
			0	1	=	−
Int Multiplication	0.38	0.31	45	55	0	0
FP Multiplication	0.23	0.25	86	14	0	0
FP Addition	0.26	0.22	100	0		0
FP Subtraction	0.24	0.11	42	0	52	0
Int Division	0.25	0.3	46	29	24	0
FP Division	0.13	0.2	56	30	14	
Int→FP Conversion	0.8	0.2	100	0	0	0
All Instructions	0.22	1.00	72	21	7	0

Table 3.7: *Breakdown of triviality per instruction type. Column 2 is the trivial ratio out of all* MEMO-TABLE *accesses, column 3 is the ratio between the instructions' trivial operations to all trivial operations, and the last columns show the breakdown of the trivial values.*

absolute value and move. Using MEMO-TABLES both for addition and subtraction, although they use the same circuitry, makes it possible to

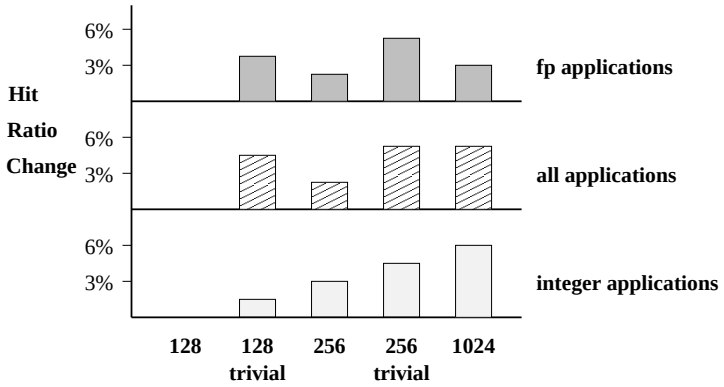


Figure 3.5: *Changes in hit-ratio of* MEMO-TABLES *with and without trivial calculation detection (base* MEMO-TABLE *of size 128/4).*

access both in the same cycle. **Total dynamic instruction count: 11%**.

5. **Comparison & Conversion** - floating point comparisons and conversions from single precision to double precision to integer formats. This table will be smaller (half size) due to the fact that the hit ratios of comparisons and conversions hardly grow with increases in MEMO-TABLE size (section 3.3). **Total dynamic instruction count: 12%**

Table 3.8 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 have a higher hit-time which offsets the possible 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	moderate, 2-5 instruction types per table	high, all instructions use 1 table
contention	low, only one instruction per table	moderate, several instructions per table	high, all instructions compete for entries
hardware complexity	high, needs comparators and TTU per instruction	moderate, comparators and TTU per table	very low, one set of comparators and TTU

Table 3.8: *Comparison of the three* MEMO-TABLES *contents schemes.*

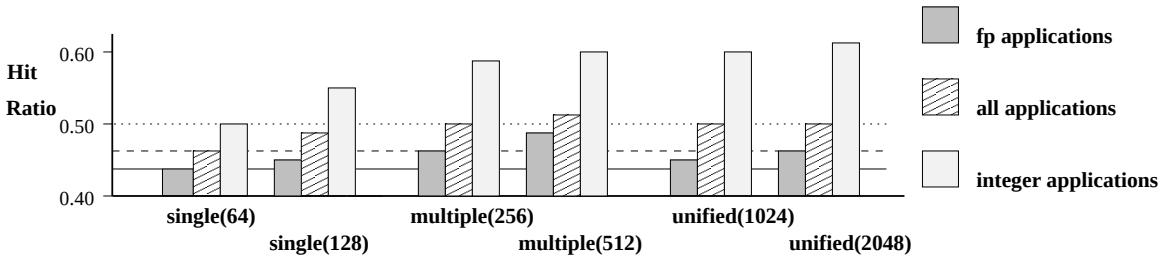


Figure 3.6: *hit-ratio of single, multiple, and unified contents scheme. Each* MEMO-TABLE *is 4-way set associative, uses random replacement, uses the LSBs of the operands (and opcodes) as indices to* MEMO-TABLE *entries, and performs trivial calculation detection.*

Figure 3.6 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.

The figure shows the multiple table approach is better than the single table approach and comparable to the unified approach. Given that using multiple MEMO-TABLES is a good compromise between single MEMO-TABLES and a unified MEMO-TABLE (table 3.8), and that the difference in hit-ratios is negligible (figure 3.6) our decision is to use multiple MEMO-TABLES each containing several instruction types.

Using multiple MEMO-TABLES also answers the question: "How does adding MEMO-TABLES impact the die size of the processor?". It is obvious that adding MEMO-TABLES requires additional transistors and wires to bring the operands and results from the FUs to the MEMO-TABLES. However, the size of 5x6KB MEMO-TABLES is 30KB. Modern microprocessors are already integrating L2 caches with sizes in the 256KB range (Intel Pentium-III, AMD Athlon) with future processors projecting onchip caches in the excess of 1MB. In fact, microprocessor designers are looking for beter uses of their transistors than just using them as caches. IM fits this role perfectly. The wire problem is solved by using multiple MEMO-TABLES located adjacent to the FUs that use them, no long, cross chip, wires are needed.

3.5.1 Exploiting Inverse and Commutative Operations

The multiplication, addition, and equality operations are commutative, for example: $a * b = c \rightarrow b * a = c$. It might be possible to exploit this trait by performing a *commutative lookup* in the MEMO-TABLE. The index created by hashing the bits of a, b are the same as for b, a. All we have to do now is compare the entries in the set to a, b and to b, a. Thus if a previous instruction calculated

$b*a$ we will receive a hit for an instruction calculating $a*b$. The disadvantage of this technique is that we now need twice the amount of comparators as before. 4-way set associativity becomes 8-way.

Another mathematical rule we can exploit is the properties of inverse operations. If $a+b=c$ were executed and inserted into the Addition MEMO-TABLE, the information to execute operations $a=c-b$ and $b=c-a$ are residing in the MEMO-TABLE. The question is could we exploit this information and memoize instructions that weren't executed even once yet? The same is true for FP multiplication and division. We can't implement the same idea for integers because $c/b=a$ doesn't necessarily imply that $a*b=c$ ($100/3=33$, $3*33=99$). The same problem exists for conversions. Converting a FP number to an integer or converting a double precision FP number to a single precision FP number results in loss of accuracy. Therefore trying to perform an inverse lookup can lead to wrong results ($1.3 \rightarrow 1$ but $1 \rightarrow 1.0$). We built an elaborate mechanism to enable *inverse lookup* and simulated it.

In addition we composed a MEMO-TABLE which we will call the Comparison MEMO-TABLE, which contains the equal, less-then, equal or less-then instructions. In order to have comparisons benefit from previous comparisons between the same two numbers we altered the MEMO-TABLE to store the relationships between two numbers in the result field. It is either -1 ($a<b$), 0 ($a=b$), or 1 ($a>b$)[9].

We ran the benchmarks on this new organization which performs commutative and inverse lookups and stores the relationships between pairs of numbers. The results were disappointing, no increase in the hit-ratio was measured. These new ideas were abandoned in future simulations.

3.6 Mapping Strategies

Until this point in our research we have indexed the MEMO-TABLES using the operand values and specifically 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. Mapping using the PC was shown to be inferior.

For integer values this mapping strategy is optimal as the LSBs show the highest entropy [23]. 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. Another reason not to use the LSBs for FP numbers is in the case where integers are the inputs. In this case the LSBs are all zero, leading to all numbers being mapped to the same entry.

[9] We are assuming that any compare instruction can provide this information, this might not be true for all architectures.

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.
- 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 3.7 shows the schemes.

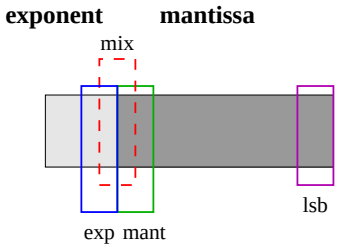


Figure 3.7: *The index bits are taken from the LSB of the exponent and MSB of the mantissa.*

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. An associativity of 1 and 2 was simulated as well, as a good mapping scheme may result in having to use a lesser degree of associativity. Figure 3.8 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 noticeable 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.

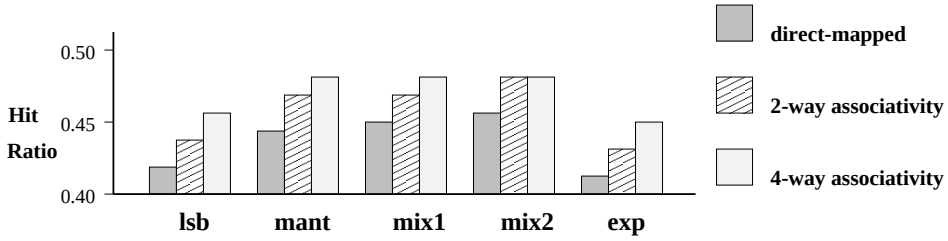


Figure 3.8: *hit-ratios of a 256 entry* MEMO-TABLE *(set associativity 1/2/4) using the 6 mapping schemes*

3.7 Summary

This chapter investigated the structure of the 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.
- Inverse and commutative operation lookup is hardly successful and isn't worth the added MEMO-TABLE complexity.
- 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 and FP⇔INT conversions, (iv) FP addition, (v) and all other FP instructions (sub, neg, ...). Each MEMO-TABLE contains 256 entries in sets of 4 (the Integer MEMO-TABLE's size is 512 and the Comp_Conv MEMO-TABLE's size is 128) . 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 0.50, this is over 80% of the hit-ratio obtained when using an infinite fully-associative MEMO-TABLE (0.60 hit-ratio).

Chapter 4

Integrating IM in a Processor's Datapath

This chapter is where we show how IM is integrated into a processor's datapath and enhances execution. We will first integrate multi-cycle instruction memoization (MCIM) in a microprocessor's datapath (section 4.1.4), show the speedup attained (section 4.2), and explore the influence of several datapath characteristics on IM (section 4.3). In the next chapter we will widen the scope of IM to include single-cycle instructions.

4.1 A Basic Microprocessor Design

4.1.1 Pipeline Stages

The SimpleScalar simulator, which is modeled after the MIPS series processors, possesses a five stage pipeline for all non Load/Store instructions (figure 4.1):

1. **Fetch:** Instructions are fetched from the Instruction Cache and stored in the *Instruction Fetch Queue (IFQ)*.
2. **Decode:** Instructions are read from the IFQ and decoded. Their operand sources are defined: either from the *Register File (RF)* or from instructions that are already in the pipeline. The instructions are entered into the *Register Update Unit (RUU)* (named also the Active List (R10000) or Reorder Buffer (604e)) where they will reside until committed.
3. **Issue:** When an instruction's operands are available it is issued to a free Functional Unit (FU) to be executed, instructions are issued out-of-order. An instruction can be delayed in this stage until it's operand dependencies are satisfied and a FU is available.
4. **Execute:** The instruction is executed by one of the FUs (there might be several types and more than one of each type). For multi-cycle instructions

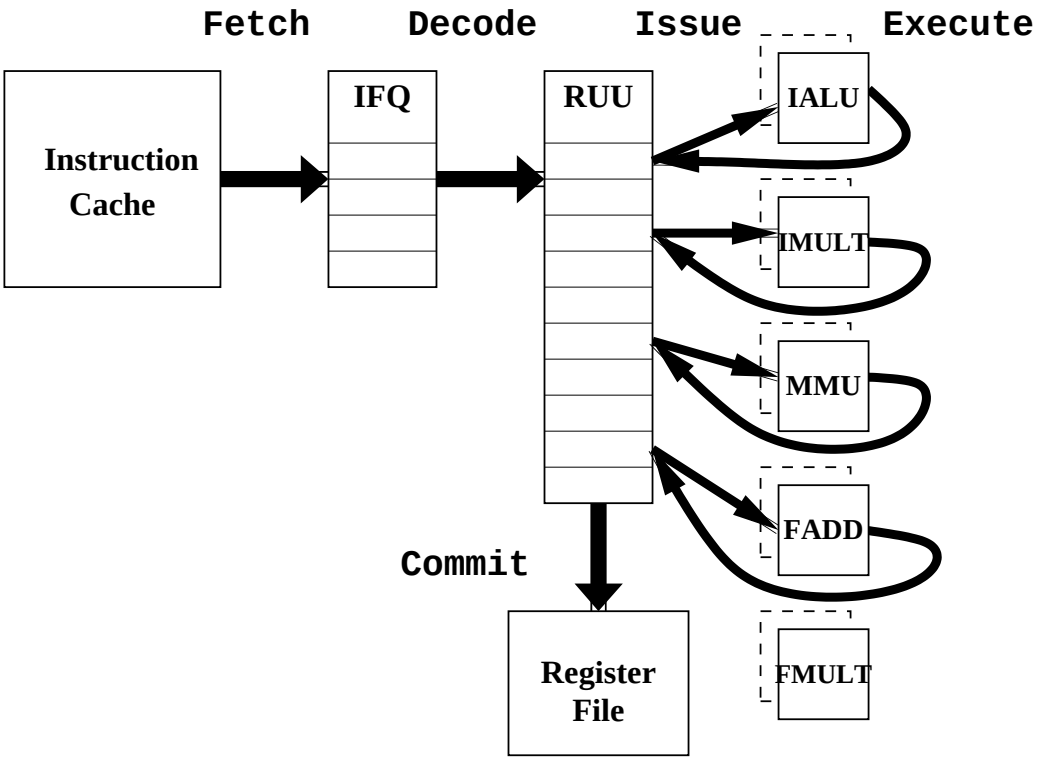


Figure 4.1: *Datapath of basic microprocessor.*

this stage takes several cycles. Results are written back into the RUU, where instructions wait to be committed.

5. **Commit:** The instruction is committed by having its result written into the RF and it is removed from the RUU. Instructions are committed in program order, thus even though an instruction has been executed it can't be committed until all previous instructions have been committed.

4.1.2 Functional Units

The processor simulated has five different FU types that execute the processor's instruction set:

1. **Integer ALU (IALU):** Executes all integer instructions (addition, subtraction, logical operations, shifts, comparisons, and branches) with the exception of multiplication and division. All instructions have a latency of one cycle.

2. **Integer Multiply Unit (IMULT):** Executes integer division and multiplication. The unit *may* be pipelined for multiplication, division isn't

pipelined.

3. **Memory Unit (MMU):** Executes Load/Stores from the L1 cache.

4. **Float Add Unit (FADD):** Executes floating point addition, subtraction, comparisons, conversions, negations, and absolute value. The unit is pipelined.

5. **Float Multiplication Unit (FMULT):** Executes floating point multiplication, division, and square-root taking. The unit is pipelined only for multiplication.

4.1.3 Processor Characteristics

L1 Instruction Cache	16-KBytes, 32-Byte blocks, direct-mapped
L1 Data Cache	16-KBytes, 32-Byte blocks, 4-way associative
L2 Unified Cache	256-Kbytes, 64-Byte blocks, 4-way associative
Memory Latencies (cycles)	L1 hit - 1, L2 hit - 6, L2 miss -18
Bus Interface	64-bit data, 32-bit address
Branch Prediction	2048-entry BTB, 2-bit counters
Registers	32 General Purpose, 32 Floating Point
Function Units	2 IALU, 1 IMULT 1 FADD unit, 1 FMULT, 2 MMU
Instruction Latencies & Throughputs	Integer multiplication: 4,1 Integer division: 20,19 All other integer instructions: 1,1 Floating point multiplication: 3,1 Floating point division: 20,20 Floating point Sqrt: 35,35 All other floating point instructions: 2,1
Pipeline attributes	4-instructions fetched, decoded, issued, and committed per cycle; 16 instructions in RUU, out-of-order execution, in-order retirement

Table 4.1: *Characteristics of basic microprocessor.*

The characteristics of the basic datapath we used in our first set of simulations is listed in table 4.1. This processor is called the *basic* processor. It's characteristic values where taken from two popular RISC processors, the MIPS R10000 [24] and PPC 604e [25], and from the default values of the SimpleScalar simulator.

4.1.4 Integrating IM

The 5 MEMO-TABLES described in the previous chapter are integrated adjacent to the relevant FUs (figure 4.2). The questions we are confronted with are: At what stage in the pipeline is memoization performed? What is the latency of

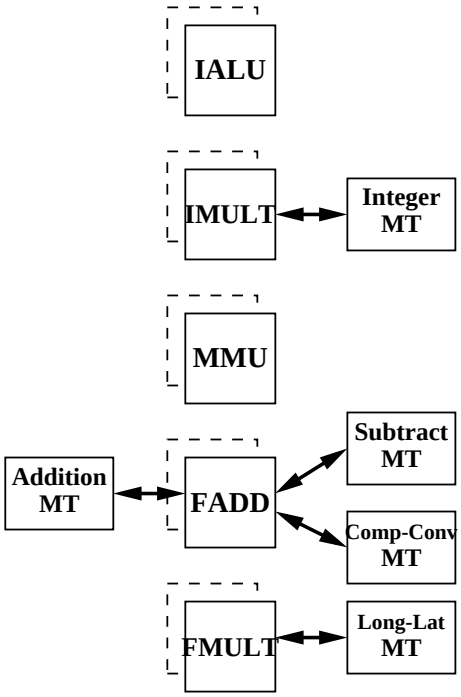


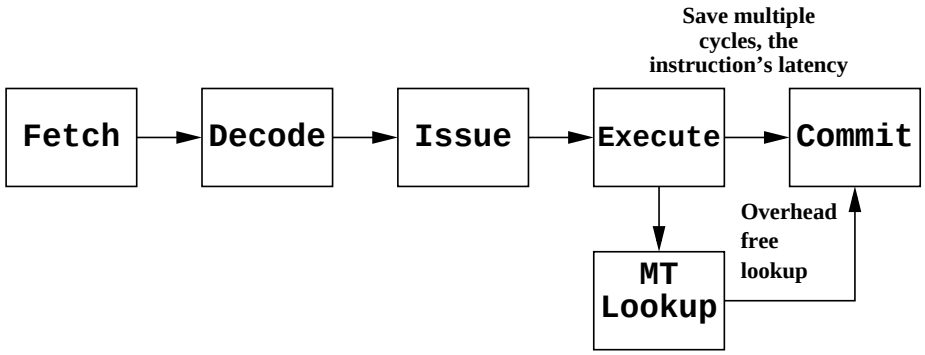
Figure 4.2: *Integration of IM in the datapath.*

a MEMO-TABLE lookup? How many lookups per cycle can a MEMO-TABLE sustain? We will answer the questions in the following sections.

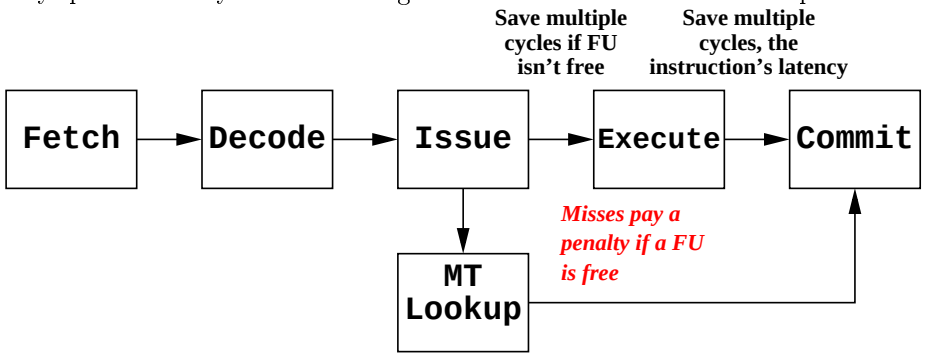
Pipeline Stage

As the instruction's operands must be ready before memoization may commence there are three alternatives:

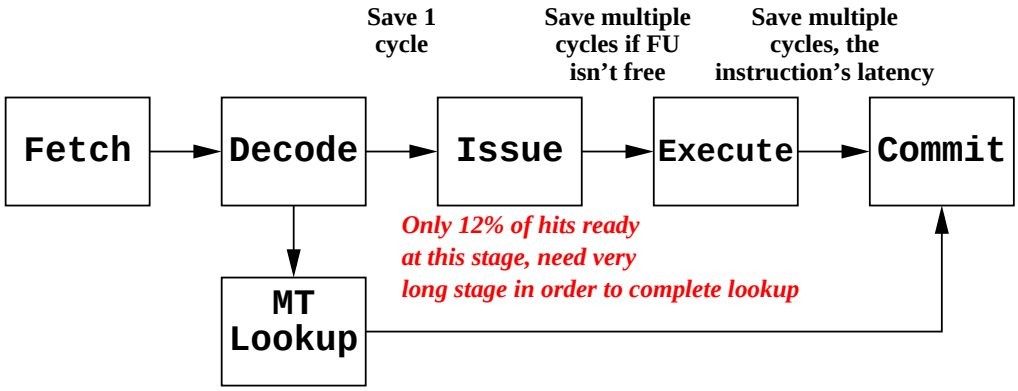
- **Execute stage:** After the instruction is allocated to a FU the MEMO-TABLE lookup and instruction execution are performed in parallel. A hit terminates the execution, a miss results in the completion of execution and updating the MEMO-TABLE with the result. Successful lookups complete in 1 cycle, unsuccessful lookups complete in the latency of the instruction.



- **Issue stage:** When the operands are ready we perform a MEMO-TABLE lookup, whether a FU is ready or not. A hit results in the instruction bypassing the execute stage, a miss results in the instruction waiting for a FU, executing, and updating the MEMO-TABLE. Successful lookups complete in 1 cycle and may gain cycles if a FU isn't available. Unsuccessful lookups lose one cycle due to the lookup, wait for a FU to be available, and then complete in the latency of the instruction. Thus an instruction may spend extra cycles in this stage due to the MEMO-TABLE lookup.



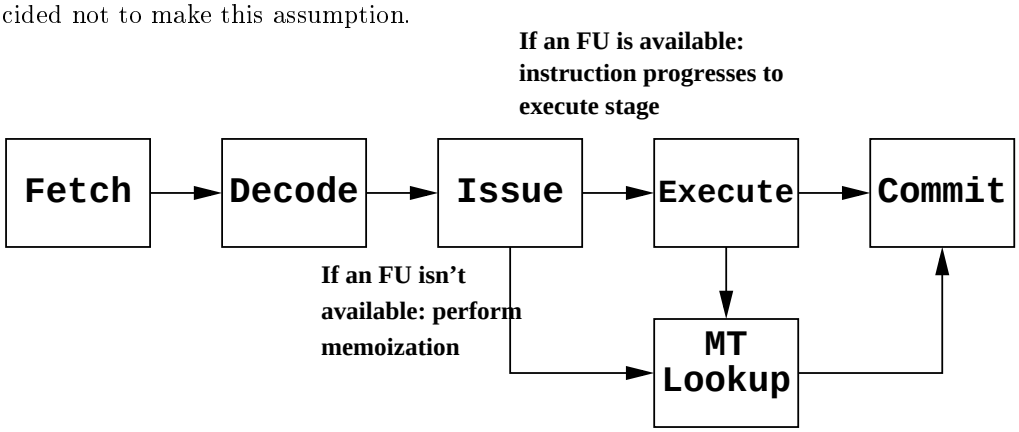
- **Decode stage:** If during the decode stage it can be determined that the operands are available, and if they can be read, and if a MEMO-TABLE lookup can be performed then memoization is possible in this stage. For high-speed processors such as the Alpha [26], which requires a pipeline stage just to access the register file, this is impossible. For other processors with longer pipeline stages this might be possible with small MEMO-TABLES (with lower lookup times). A hit completely bypasses the issue and execute stage in one cycle. A miss continues normal execution.



Memoization in the decode stage has the most potential for speedup but only 12% of all hits have their operands ready at this stage and we would need a very aggressive design to enable a MEMO-TABLE lookup at this stage. Memoization in the issue stage eliminates the need to wait for a FU and can conserve power [27] if instruction execution isn't started, however in the case of a miss there is an overhead of the lookup time if a FU was available but wasn't used. Memoization in the execute stage is overhead free but the potential gain is the lowest and limited to the instruction's latency (less one cycle for the lookup).

A hybrid solution which results in a win-win situation is to perform memoization in the execute stage if a FU is available and to perform it in the issue stage if not. This way instructions that can't issue due to a structural hazard can still benefit from memoization without paying the lookup penalty. Future references will call this scheme: *memoization in the issue stage*.

Accesses to a MEMO-TABLE in this stage **are** counted as issues even if the lookup failed and the instruction must wait in the issue stage until a FU is available. The alternative, **not** to count MEMO-TABLE lookups as issues, assumes that the processor can handle more than four instructions (the issue width) per cycle. This demands resources that aren't available to the processor. We decided not to make this assumption.



MEMO-TABLE **Latency and Parallelism**

Our assumption is that a MEMO-TABLE lookup has a latency of one cycle. This is based on the access time of on-chip caches which can perform several tag compares (in the case of a set associative cache) and retrieve the cached data in a single-cycle. Thus it should be possible to compare the operands of an instruction with a MEMO-TABLE entry and retrieve the result in a single cycle.

The only difference between the lookups is the size of data to compare. The data cache tag is at the most 64-bits wide (32-bits for most processors), the MEMO-TABLE tag may contain 2 FP numbers and an opcode (133 bits). However the comparison is a bitwise equality test so the added gate delay due to the wider tags shouldn't be much very big.

The same comparison to caches can be made in order to determine the maximum number of lookups per cycle. Most L1 caches can sustain two lookups per cycle, so we will assume that each MEMO-TABLE is limited to two accesses per cycle (both lookup or update).

4.2 Basic Processor Speedup

Our first set of experiments simulates the basic microprocessor with IM performed in the issue stage only if an FU isn't available. For all benchmarks simulated the dynamic Fraction Enhanced (FE)[1] , hit-ratio, and speedup are shown in table 4.2. The FE was measured by simulating a processor where all multi-cycle instructions have a latency of one cycle and execute without the need of a FU. The difference between this run and a regular run is the FE.

The table shows that there is a certain correlation between the FE to the speedup, while there is a lesser correlation between hit-ratio and speedup. Figure 4.3 which shows the actual points and the best-fit lines (nonlinear least squares fitting using the Marquardt-Levenberg algorithm), depicts this fact. For example, the integer benchmarks (g721, jpeg, pgp, gcc, perl, and compress, which are circled in figure 4.3), show a very low speedup, even though they have relatively high hit-ratios, due to their low FEs. Floating Point intensive benchmarks show a much higher speedup due to a higher FE. Figure 4.4 shows the breakdown of speedup by suite (SPEC, MB) and data type (Int, FP). This shows that we must widen the scope of memoization to encompass more instructions and thus enhance more applications. Chapter 5 is devoted to this task.

[1] Amdahl's law [21] states that the speedup obtained by using an enhancement is

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

Fraction Enhanced (FE) is the fraction of computation time in the original machine that can use the enhancement. *Speedup Enhanced (SE)* is the improvement gained if *only* the enhancement mode could be used.

application	*FE*	*hr*	*spdp*
mesa	20%	0.51	1.09
epic	23%	0.20	1.05
rasta	12%	0.40	1.06
mpeg2	8%	0.64	1.07
gsm	13%	0.09	1.02
ghostscript	25%	0.97	1.33
jpeg	1%	0.75	1.00
g721	1%	0.54	1.01
pgp	4%	0.12	1.01
harmonic mean	12%	0.47	1.07
tomcatv	10%	0.30	1.04
swim	24%	0.28	1.08
su2cor	14%	0.12	1.02
hydro2d	20%	0.92	1.21
mgrid	24%	0.70	1.27
applu	6%	0.58	1.04
turb3d	10%	0.46	1.04
apsi	38%	0.39	1.16
fpppp	6%	0.44	1.02
wave5	16%	0.34	1.05
gcc	1%	0.96	1.01
perl	0%	0.97	1.00
compress	3%	0.27	1.01
harmonic mean	13%	0.55	1.07
harmonic mean	**13%**	**0.52**	**1.07**

Table 4.2: *FE, hit-ratios, and speedups on the basic processor when IM is implemented.*

4.3 Measuring Attributes of the Datapath

In order to gauge the impact of different datapath attributes on the effectiveness of IM we will change attributes of the datapath and the memoization process and explore their impact on the hit-ratio, processor performance (measured in IPC), fraction enhanced, and speedup. We chose eight attributes of the datapath and MEMO-TABLES to variate:

1. **Pipeline Width:** The maximal number of instructions that can be fetched, decoded, executed, and committed each cycle. Can vary from a width of 1 (no multiple-issue at all) and upwards.

2. **Instruction Window:** The maximal number of instructions the processors "sees" in any given cycle. Only these instructions can be issued out-of-order to the FUs. Must be at least the width of the pipeline.

3. **Branch Prediction:** The scheme used to predict the outcome of branches and thus avoid control hazards. Can vary from simple taken/nottaken

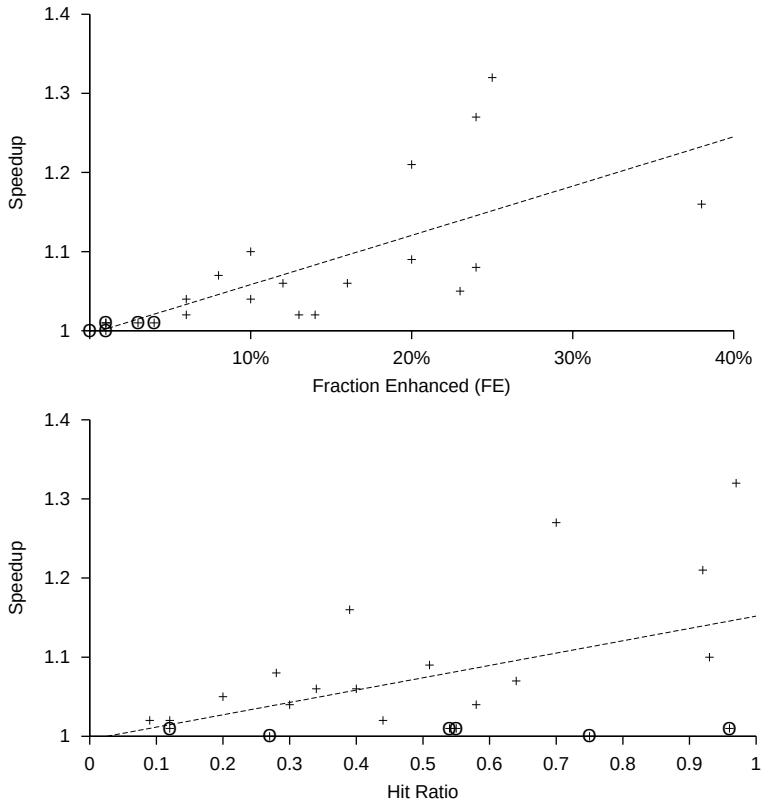


Figure 4.3: *Correlation between FE to speedup and between hit-ratio to speedup (integer applications are circled). Lines are best fit using the Marquardt-Levenberg algorithm.*

schemes to a "perfect" prediction scheme.

4. **Functional Units:** The number of FUs of each type available, must be at least one of each type.

5. **Instruction Latencies:** The number of cycles it takes to complete the Execute stage of each multi-cycle instruction.

6. **Memory Hierarchy:** The capacity, line size, associativity, hit/miss time of the caches, can vary from a perfect cache to no cache at all.

7. **Memoization Latency:** The latency of a MEMO-TABLE lookup.

8. **Memoization Stage:** Could be either at the issue (hybrid solution) or execute stages.

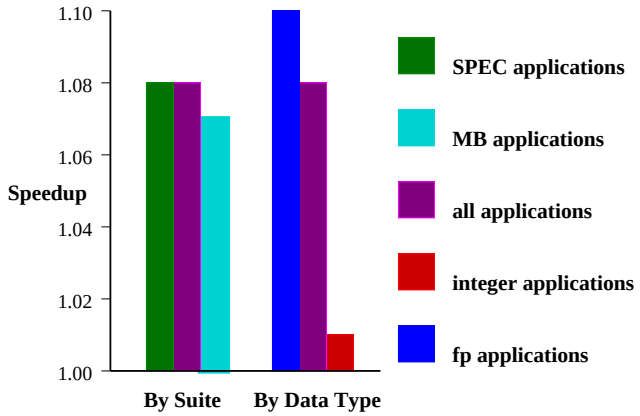


Figure 4.4: *Breakdown of speedup by application suite (SPEC, MediaBench) and by data type (FP, Int)* .

In order to perform a full factorial design we would have to perform thousands of simulations, even a 2^k factorial design (as performed in the previous chapter) would take 256 simulations. However performing a 2^{k-p} factorial design with $p = 4$, necessitates only 16 simulations but provides almost the same level of accuracy. The levels of each of the above 8 factors used are:

Factor	*Low Level*	*High Level*
Instruction Window	8	32
Pipeline Width	2	8
Branch Prediction	Predict taken	Perfect prediction
Functional Units	2 IALU, 1 IMULT, 2 MMU 1 FADD, 1 FMULT	4 IALU, 2 IMULT, 2 MMU 2 FADD, 2 FMULT
Instruction Latencies		
int multiplication	6,6	3,1
int division	35,35	20,20
fp multiplication	3,1	2,1
fp division	31,31	20,20
fp sqrt	50,50	35,35
Memory Hierarchy	Basic	Perfect memory access
Memoization Latency	2 cycles	1 cycle
Memoization Stage	Execute stage	Issue stage

We measured the hit-ratio, speedup, FE, and IPC (Instructions Per Cycle) for each simulation. The following sub-sections present and explain the results for each of the measurements.

4.3.1 Hit-Ratio

The minimal and maximal values of the hit-ratio are 0.51 and 0.63. Using the Sign Table method [20] to allocate the variation between factors shows that 47%

of the allocation is attributed to the branch prediction mechanism, 27% to the pipeline width, window size and their combination and an additional 24% to the combinations of branch prediction with pipeline width and window size. The memory hierarchy, number of FUs and their latencies and the stage and latency of memoization have no impact on the hit-ratio.

The allocation of variation is consistent with the results that show that for the runs in which the branch prediction rate is perfect the hit-ratio is the lowest. This is explained by the fact that instructions are re-executed when the branch prediction rate is low. The following code excerpt explains the phenomena:

```
/* 1 */ if (a < b)
/* 2 */     c = a + 2.5;
/* 3 */ else
/* 4 */     c = b + 2.5;
/* 5 */ d = a*b;
```

The instruction at line 5 isn't dependent on the result of the comparison at line 1. If the comparison is predicted as being taken lines 2 and 5 are executed, if later the prediction turns out to have been incorrect the pipeline is flushed and lines 4 and 5 are executed. Thus the calculation at line 5 resides in one of the MEMO-TABLES and a lookup results in a hit. This case was the primary reason Sodani and Sohi [8] started exploring instruction reuse, they named it "squash reuse".

A wider pipeline and larger instruction window raise the IPC, thus more executed but not yet committed instructions are flushed during a branch misprediction, which in turns raises the MEMO-TABLES hit-ratio. The highest hit-ratio, 0.63, is achieved when the branch prediction rate is low (a predict taken scheme is used) and the pipeline width (8) and window size (32) are large. This shows that the "true" hit-ratio attributed to program and data characteristics is around 50%. Any additional hit-ratio percentage is due to branch misprediction.

4.3.2 Instructions Per Cycle (IPC)

We will use the IPC, which is the number of committed instructions divided by the number of cycles, as our performance metric. The higher the IPC the better the processor's performance. We measured both the IPC for a run without IM and for a run with IM. The allocations of variations are almost identical. The values measured range from 0.65 to 2.63. The allocation of variation is: Branch Prediction - 58%; Window Size - 16%; Memory Hierarchy - 12%; Pipeline Width - 7%; Instruction Latency - 4%;

The results indicate that branch prediction plays a very important role in improving performance. The IPC for the basic processor is 1.19 with a BP rate of 0.94, when altering only the branch prediction scheme the IPC is 1.26 (perfect, BP rate of 1.00) and 0.82 (taken, BP rate of 0.26). This shows that standard branch prediction techniques are very close to the perfect scheme.

4.3.3 Fraction Enhanced (FE)

The part of the program that is susceptible to IM is called the Fraction Enhanced (FE). This is the part of the program that benefits from IM. The larger the FE is the larger the potential for speedup is. The minimal and maximal FE values are 3% and 28%. The allocation of variation is: Instruction Latency - 31%; Pipeline Width - 23%; Windows Size - 13%; Branch Prediction - 11%; Memory Hierarchy - 6%;

That the instruction latency is a contributing factor is obvious. A long latency instruction consumes more processor cycles, raising the fraction of the program spent executing multi-cycle instructions. However the combined effects of pipeline width and window size have an even larger part in the variation. A wide pipeline can issue more instructions per cycle, that can execute in parallel to the multi-cycle instruction "stuck" in the execute stage. Our intuition says that the wider the pipeline is the less the FE is.

However the results are counter intuitive and show exactly the opposite: If the datapath can't process more instructions due to a low pipeline width, small instruction window size, and/or a low branch prediction rate, the long latency instructions stall only a small number of instructions. Thus the IPC is lower but so is the FE, slower processors have less potential for exploiting IM. On the other hand if the processor can execute multiple instructions per cycle the long latency instructions delay the commitment of many more instructions. So although the IPC is higher the FE is as well, which lead to a higher potential for improvement using IM.

4.3.4 Speedup

Finally we arrive at the most important measurement from our point of view: speedup. A high speedup proves the viability of implementing IM in the datapath. The speedups range from 1.01 which isn't very promising to 1.18 which shows great potential. The allocation of variation is: Instruction Latency - 42%; Pipeline Width - 20%; Memoization Latency 9%; Memory Hierarchy - 7%; Branch Prediction - 6%; Windows Size - 4%; Again we see that instruction latency is, obviously, the leading speedup factor, successfully memoizing these instructions leads to considerable savings. Of the other factors pipeline width is the dominant, this is consistent with the FE factors and strengthens the relation between FE and speedup.

The memoization stage doesn't impact the results at all. Neither do the number of FUs. Both these facts are related. When simulating the basic processor the number of structural hazards caused by multi-cycle instructions are relatively low, only 10% of issue requests to multi-cycle FUs are stalled due to the lack of an appropriate unit, this is opposed to 31% for all instructions[2]. In addition only 9% of all successful memoizations occur in the Issue stage. Comparing memoization in the issue stage to memoization in the execute stage shows that the average number of cycles an instruction is resident in the RUU (RUU

[2]Multi-cycle instruction structural hazards are 8% of all structural hazards.

latency) is the same. This leads to both runs having the same IPC. The apriori advantage of memoization in the issue stage isn't used, section 4.4 elaborates this point. Another surprise is that the memoization latency contributes only 9%, we will explore this phenomena in section 4.4 as well.

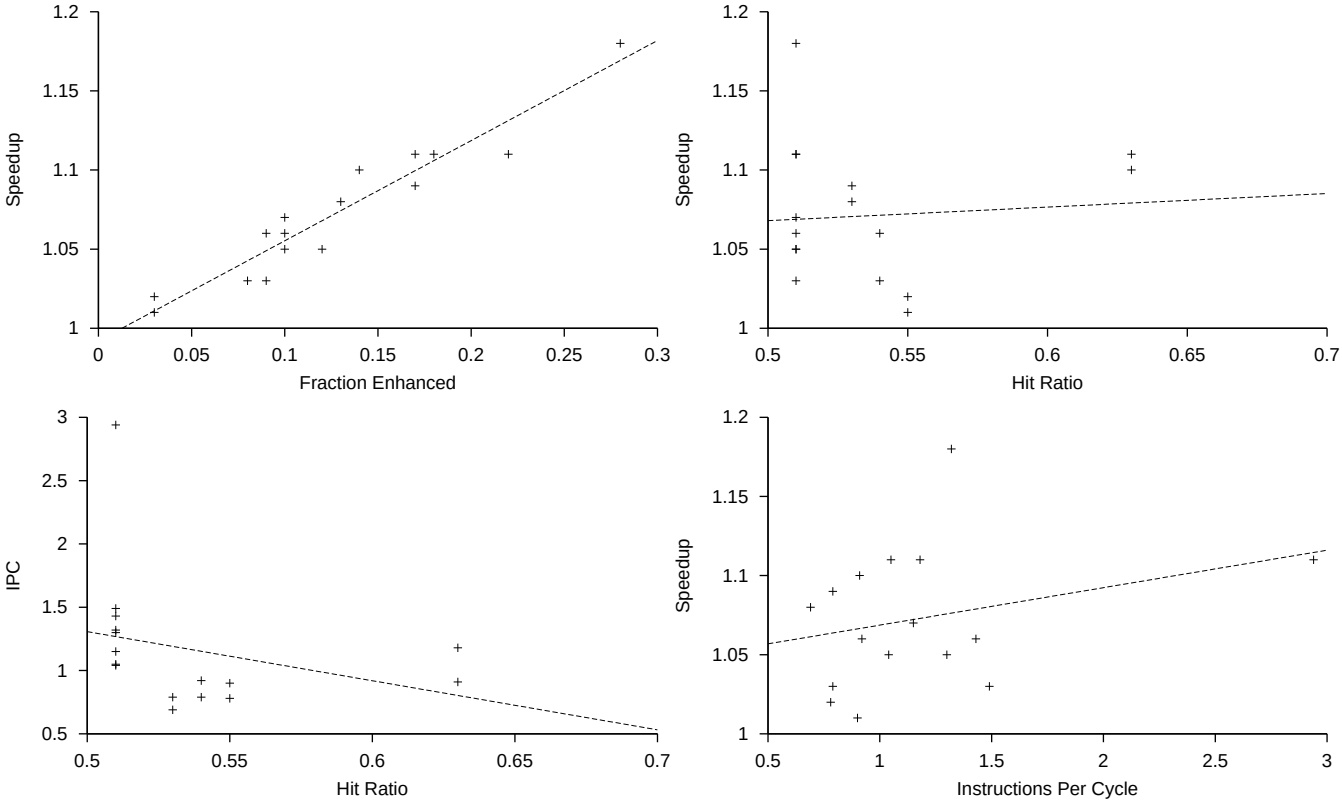


Figure 4.5: *Correlations between FE and speedup (upper-left), hit-ratio and speedup (upper-right), hit-ratio and IPC (lower-left), IPC and speedup (lower-right).*

4.3.5 Correlation Between Measurements

In order to verify the usefulness of IM we must correlate the four measurements recorded above. A high hit-ratio combined with a low speedup is useless, as is a high speedup on a slow machine. Figure 4.5 show the correlation between all four measurements. Our observations and conclusions are:

- As mentioned above there is a direct correlation between FE and speedup. The more potential there is for memoization the better the speedup is.

- There is no correlation between hit-ratio and any other measurements. This doesn't mean that a higher hit-ratio doesn't influence the speedup, it does as will be shown in section 4.4.1. It means that given a fixed MEMO-TABLE structure the FE or IPC of a processor don't alter the hit-ratio. The only influence the datapath has on the hit-ratio is through branch prediction. A poor prediction rate leads to a higer hit-ratio, but this "gain" is offset by the low performance (low IPC) of the processor.

- There is no direct correlation between IPC and speedup. This fact is encouraging, our preliminary assumption was that for powerful processors (high IPC) the speedups would be low. This isn't true, in fact the speedup on the most powerful processor is 1.11, which is higher than the speedup on the basic processor (1.07), although the powerful processor is more than twice as fast (1.19 vs. 2.63 IPC). We will explore this correlation further in section 4.4

4.4 Additional Measurements

After examining the previous results we decided to refine the simulations and concentrate on three of the eight previously simulated factors, factors for which we couldn't make any clear cut decisions. The factors and levels are:

1. **Pipeline size:** This factor condenses 3 factors (pipeline width, window size, and number of FUs) into one factor. All 3 factors are enlarged or shrunken together, a wide pipeline needs a large instruction window and a large number of FUs. In our previous simulations we saw that their combined influence surpassed their individual influences. The low level is a small pipeline with a width of 2, instruction window of 8, and 1 unit of each type. The high level is a large pipeline with a width of 8, instruction window of 64, and 4 units of each type.

2. **Memoization Stage:** In the previous simulations we couldn't discern any differences between them. The levels are memoization in the execute or issue stages.

3. **Memoization Latency:** Memoization latency contributed only 9% to the variation. The levels are 2 or 1 machine cycles for a MEMO-TABLE lookup.

For the remaining three factors we chose to target faster processors by implementing low latency instructions, perfect branch prediction and a perfect memory hierarchy. As we have only 3 factors we performed a 2^k factorial design which consists of 8 runs. The results are in table 4.3. We chose to display the results for the FP intensive applications only in order to magnify the effects of the IM stage and IM latency on the results.

The allocation of variation of the IPC (100% pipeline size) and hit-ratio (equal distribution) is trivial. The allocation of variation of the speedup is:

Factor Levels			*IPC*	*hr*	*Spdp*	*Factor Levels*			*IPC*	*hr*	*Spdp*
small	execute	2	1.02	0.47	1.04	small	issue	2	1.02	0.47	1.04
small	execute	1	1.03	0.47	1.06	small	issue	1	1.03	0.47	1.06
large	execute	2	3.54	0.47	1.09	large	issue	2	3.54	0.47	1.09
large	execute	1	3.65	0.47	1.13	large	issue	1	3.65	0.47	1.13

Table 4.3: 2^3 *factorial design and resulting IPCs, hit-ratios, and speedups. The factors and levels are pipeline size (small, large), IM stage (execute, issue), and IM latency (2 or 1 cycles).*

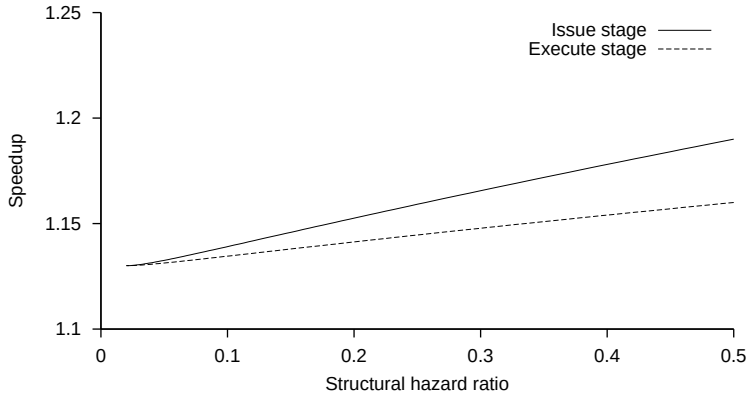


Figure 4.6: *Comparison of IM in the issue and execute stages. On a large pipeline machine the number of FUs is raised until the structural hazard ratio is 0. Speedup is shown as a function of the structural hazard ratio.*

Pipeline size - 78%; IM latency - 20%; IM stage - 0%; To understand why the results are neutral to the memoization stage we measured the structural hazard ratio (number of successful issues divided by number of issue attempts) and found it to be under 0.02. Reducing the number of FUs raises the structural hazard ratio. When this happens the differences between memoization in the issue and execute stage become apparent as displayed in figure 4.6. Following this set of experiments we can conclude:

- The most important conclusion is that IM favors fast processors. A higher IPC usually results in a higher speedup. A processor with short latency instructions, perfect memory hierarchy, perfect branch prediction, and multiple-issue capabilities still has it's performance hampered by the latencies of multi-cycle instructions. Using IM reduces this impediment and accelerates processing.

- The hit-ratio is orthogonal to the datapath design and is dependent upon the application's inherent locality and the MEMO-TABLE design.

- The memoization stage has little to no influence (as can be seen by com-

paring the right and left hand sides of table 4.3). Most instructions find a FU and progress to the execute stage, this limits the effect of performing a lookup in the issue stage. Figure 4.6 shows that only when the structural hazard ratio (number of successful issues divided by number of issue attempts) is high (due to less FUs) IM in the issue stage is better.

- A memoization latency of 2 cycles isn't "fatal" to IM, even though in our model most FP instructions have a latency of 2 cycles. This indicates that a large amount of the speedup can be attributed to long latency instructions such as division and sqrt.

4.4.1 Speedup as a Function of MEMO-TABLE Organization

The previous simulations all used a fixed MEMO-TABLE organization. In this section we shall observe the impact of varying the MEMO-TABLE organization on the hit-ratio and speedup (over the basic processor). We will use the multiple MEMO-TABLE design and vary the size, associativity, and trivial calculation detection of the MEMO-TABLES. The replacement method will be random and the mapping scheme will use the mix2 scheme (section 3.6).

Table 4.4 shows the MEMO-TABLE organizations, hit-ratios, and speedups. The hit-ratio increase rises swiftly until a size of 512 entries and then tapers out, no matter what MEMO-TABLE enhancements are introduced. This directly affects the speedup which also flattens out. The results strengthen our choice of MEMO-TABLE organization. Investing more hardware resources in MEMO-TABLES isn't worth the small improvements achieved. These results mirror the results observed in chapter 3

Size	*Assoc*	*Triv*	*hr*	*spdp*
32	1	no	0.34	1.04
64	2	no	0.41	1.05
128	2	yes	0.48	1.06
256	**4**	**yes**	**0.51**	**1.07**
512	8	yes	0.54	1.08
1024	8	yes	0.56	1.08
2048	16	yes	0.57	1.08

Table 4.4: *Different* MEMO-TABLE *organizations and the resulting hit-ratios and speedups (on the basic processor).*

4.5 Summary

In this chapter we investigated the integration of IM into the processors's datapath, the performance enhancement gained by exploiting IM, and the influence of the datapath structure on IM and vice-versa. On a basic processor whose design is similar to the MIPS R10000 and PPC 604e, two ubiquitous RISC processors,

13% of the execution time can be attributed to multi-cycle instructions. 52% of those instructions are repeated with the same operands. By implementing IM an average (harmonic mean) speedup of 1.07 is attained. This speedup is as high as 1.32 for highly intensive FP applications, and as low as 1.003 for integer applications which hardly use multi-cycle instructions.

The influence of the datapath on IM is minimal. The only datapath factor that effects the hit-ratio is the branch prediction rate. Mispredicted branches cause instructions to be flushed from the pipeline, many of these instructions may later be re-executed causing hits in the MEMO-TABLES. Thus the hit-ratio is raised , together with the total execution time.

On the other hand the influence IM has on the datapath is large. The major contribution is the reduced latency of successfully memoized instructions. Having instructions complete execution earlier enables dependent instructions to be issued earlier. The number of cycles an instruction spends, from being fetched until it is committed (RUU latency) is reduced, which directly reduces execution time. A minor contribution to enhanced execution is the virtual addition of FUs. When a structural hazard occurs a MEMO-TABLE lookup may be able to provide the instruction's result, thus the execute stage of the pipeline is circumvented.

Instruction memoization is best utilized when it reduces the latency of "critical" instructions, instructions that are prohibiting many other instructions from advancing through the pipeline. It is hard to say in what datapath design an instruction is critical and in what design it isn't. However it is clear that faster processors that can execute more instructions per cycle benefit greatly from IM. A processor with a wide pipeline, a near perfect memory hierarchy, a high rate of branch prediction, and enough FUs will encounter a bottleneck when waiting for long latency instruction to complete. IM relieves this bottleneck.

On the other hand slower processors might have their bottleneck in the memory hierarchy or issue rate. In this case IM will still speedup processing but at a lower rate. Even in the case of an inorder processor, where every instruction delays its successors, the effect of memoization is less than for an out-of-order processor which can mask the effect of long latency instructions by executing "around" them. The average speedup on an inorder basic processor is 1.05 (over an IPC of 0.70) compared with 1.07 (over an IPC of 1.27) for the same out-of-order processor.

All the above notwithstanding, the scope of multi-cycle IM is limited. Few applications spend more than 20% of their execution time computing multi-cycle instructions. Many more spend less than 1%. It is imperative that we widen the scope of IM to encompass single-cycle instructions as well. Chapter 5 is dedicated to this issue.

Chapter 5

Memoizing Single Cycle Instructions

In this research we have only memoized multi-cycle instructions. The rationale behind this decision has been that single-cycle instructions can be executed in the same cycle a MEMO-TABLE lookup is performed, thus no improvement is gained. However if instructions are memoized in the issue stage their results can be obtained even if a suitable FU isn't available, thus many structural hazards are avoided.

We added to our simulator the capability to memoize single-cycle instructions as well. The instructions memoized are integer addition and subtraction, shifts, logical instructions, moves, and set less than (`slt`) instructions. The mnemonic single-cycle IM (scIM) refers to the memoization of both multi-cycle and single-cycle instructions.

We do not memoize conditional and unconditional branches, these instructions aren't context free and their results are Program Counter (PC) dependent. In any case the branch prediction mechanism is itself a MEMO-TABLE of sorts, and performs very well. For the same reason we do not memoize loads or stores. We would have to trace all memory references and invalidate MEMO-TABLE entries that had their addresses updated. Moreover the L1 caches are themselves MEMO-TABLES which do a very good job of exploiting previous memory references.

5.1 Comparing Single and Multi-Cycle IM

For our first set of simulations we have added a 512-entry MEMO-TABLE (the Single-Cycle table) that holds the single-cycle instructions. Table 5.1 displays the single-cycle hit-ratios, the accumulated hit-ratio and the speedups, for comparison the speedups for mcIM are included in parentheses. The table clearly shows that memoizing single-cycle instructions results in a speedup that is 50% better than the speedup obtained by memoizing only multi-cycle instructions.

application	*sc hr*	*hr*	*spdp*
mesa	0.72	0.64	1.12 (1.09)
epic	0.52	0.45	1.08 (1.05)
rasta	0.68	0.62	1.09 (1.06)
mpeg2	0.49	0.49	1.10 (1.07)
gsm	0.36	0.31	1.07 (1.02)
ghostscript	0.92	0.92	1.49 (1.33)
jpeg	0.45	0.45	1.07 (1.00)
g721	0.51	0.51	1.12 (1.01)
pgp	0.41	0.39	1.07 (1.01)
harmonic mean	0.56	0.53	1.13 (1.07)
tomcatv	0.57	0.48	1.06 (1.04)
swim	0.37	0.33	1.10 (1.08)
su2cor	0.55	0.42	1.03 (1.02)
hydro2d	0.32	0.51	1.22 (1.21)
mgrid	0.84	0.80	1.27 (1.27)
applu	0.93	0.89	1.04 (1.04)
turb3d	0.59	0.55	1.09 (1.04)
apsi	0.45	0.42	1.17 (1.16)
fpppp	0.65	0.46	1.02 (1.02)
wave5	0.33	0.33	1.07 (1.05)
gcc	0.75	0.75	1.04 (1.01)
perl	0.75	0.75	1.02 (1.00)
compress	0.51	0.51	1.08 (1.01)
harmonic mean	0.58	0.56	1.09 (1.07)
harmonic mean	**0.57**	**0.55**	**1.11 (1.07)**

Table 5.1: *single-cycle hit-ratios, combined hit-ratios, and speedups on the basic processor when single-cycle IM is implemented.*

But what the table doesn't show is from where this speedup originates. The RUU latency is reduced but why? If a MEMO-TABLE lookup and the latency of a single-cycle instruction are one cycle, where is the speedup coming from? The answer is: by reducing the number of structural hazards. On the average the ratio of structural hazards out of all requests for a FU is 31%. Almost every 3rd instruction in the issue stage can't find a free FU.

Memoizing single-cycle instructions reduces the structural hazard ratio to 15%. Successful MEMO-TABLE lookups overcome the absence of enough FUs. Thus when the number of FUs a processor possesses is such that no structural hazards occur, single-cycle memoization will be useless. Figure 5.1 is similar to figure 4.6, it shows the speedup of MCIM and SCIM as a function of structural hazard ratio and IPC. When the structural hazard ratio drops the difference between multi-cycle to single-cycle narrows and then disappears. MCIM speedup is improved as the IPC of an application rises, on the other hand SCIM speedup decreases as the IPC rises. The impact of SCIM diminishes as more FUs are available, SCIM effectively becomes MCIM.

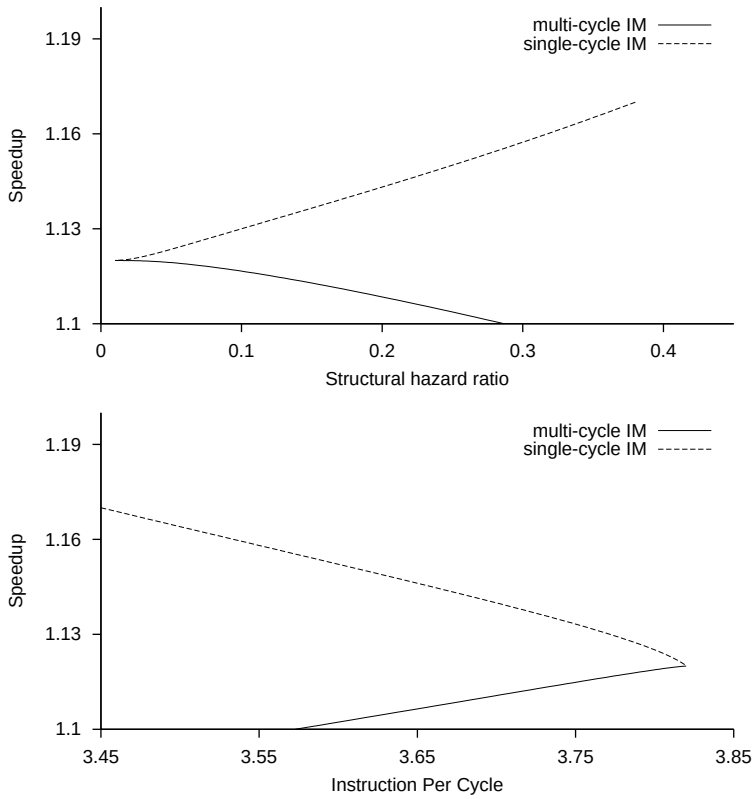


Figure 5.1: *Comparison of single to multi-cycle IM. On a large pipeline machine the number of FUs is raised until the structural hazard ratio is 0. Speedup is shown as a function of the structural hazard ratio and of IPC.*

5.1.1 scIM Compared to Other Enhancements

It isn't necessary to push ILP to its limit to see the futileness of single-cycle execution. Table 5.2 shows the speedups of several configurations over the basic unmemoized processor. The configurations add IALUs, multi-cycle units, implement mcIM and scIM, and combine all techniques. In addition we implemented a processors with double the L1 cache size and a processor with perfect branch prediction. The base we are comparing against is the basic processor which has a performance of 1.0. The table shows that:

1. Adding MCUs hardly effects performance, due to the inherent low structural hazard ratio of MCUs.

2. Adding ALUs enhances integer application performance better than implementing scIM, however FP application performance isn't improved as well as using mcIM.

3. Adding ALUs to mcIMyields, across all applications, more than a 50% improvement over scIM.

4. Using the resources dedicated to IM in order to achieve a lower L1 miss rate or a higher branch prediction rate improves integer applications more than IM. However FP applications benefit from a higher speedup when the resources are used to implement mcIM.

What we have now is a tradeoff problem. An ALU, which has a latency of 1 cycle and 100% hit-ratio (every calculation is correct), outperforms a 512-entry (12K bytes and 4 comparators) Memo-Table with a 57% hit-ratio. Which uses less transistors? Which is simpler to design? Which consumes less power? These questions are beyond the scope of this thesis. If adding an ALU is cheaper then there is no contest: scIM isn't worthwhile. However in the next section we will present several techniques that enable implementing scIM with a lower cost.

Name	*# ALU*	*# MCU*	*IM*	*Int*	*FP*	*All*
basic	2	1	no	1.00	1.00	1.00
basic + cache X 2	2	1	no	1.09	1.06	1.07
basic + perfect BP	2	1	no	1.10	1.05	1.07
basic + mcIM	2	1	mcIM	1.01	1.10	1.07
basic + scIM	2	1	scIM	1.07	1.13	1.11
basic + 3 ALUs	3	1	no	1.10	1.04	1.05
basic + 4 ALUs	4	1	no	1.12	1.05	1.07
basic + 2 MCUs	2	2	no	1.00	1.01	1.01
basic + 3 MCUs	2	3	no	1.00	1.01	1.01
basic + 3 ALUs + mcIM	3	1	mcIM	1.12	1.16	1.15
basic + 4 ALUs + mcIM	3	1	mcIM	1.13	1.17	1.16

Table 5.2: *Comparison of adding FUs, integrating IM, and combining both on the basic processor. MCU stands for Multi-Cycle Unit, any unit which executes multi-cycle instructions.*

5.2 Lowering the cost of scIM

In the previous section we suggested that scIM isn't "real" memoization, the benefits we gain are due to using the Memo-Table that contains single-cycle instructions as an additional ALU. Adding an ALU instead of the Memo-Table results in greater performance. In order to make scIM worthwhile we have to reduce the cost of the single-cycle Memo-Table or alternatively improve its performance. We suggest three schemes:

- Use existing hardware. Specifically use the existing Memo-Tables. We chose to use the Integer Memo-Table which contains integer division and multiplication instructions to hold all single-cycle instructions as well. A

variant of these scheme is to use LRU replacement and a victim cache that only contains evicted division and multiplication instructions.

- Use a simpler MEMO-TABLE. A small (32-entry), direct mapped, with no trivial detection MEMO-TABLE is used.
- Perform SCIM in the decode stage. In section 4.1.4 we described how this may be implemented. The problem is that a long machine cycle is necessary in order to determine if the operands are ready, find them, and perform a MEMO-TABLE lookup. Our solution is to *speculatively* perform a lookup using the current data in the Register File (RF). If the RF has valid data (no previous instructions are writing to the operand registers) and the lookup was successful, the instruction can progress to the commit stage, bypassing the issue and execute stages.

Scheme	*Int*	*FP*	*All*
regular MCIM	1.01	1.10	1.07
regular SCIM	1.07	1.13	1.11
sc insts. in idiv/imult MEMO-TABLE	1.05	1.12	1.10
above with lru and victim cache	1.06	1.12	1.10
small sc MEMO-TABLE (32-entry)	1.02	1.11	1.08
sc memoization in decode stage	1.02	1.12	1.09

Table 5.3: *Speedups of different schemes used to lower the overhead of* SCIM.

Table 5.3 compares the 3 suggested alternatives with regular MCIM and SCIM. The first alternative yields an average speedup of 1.10 as opposed to 1.11 when a dedicated MEMO-TABLE is used for the single-cycle instructions. The speedup attributed to integer division and multiplication is around 1.01, meaning that the single-cycle instructions swamped the Integer table and reduced the hit-ratio of integer division and multiplication. Thus we converted the Integer table to the Single-Cycle table. But this table itself is less productive than an additional ALU. In the second alternative the hit-ratio of the Single-Cycle table drops from 57% to 19% causing the speedup to drop to 1.08. The applications that "suffer" the most are the integer applications, their average speedup is 1.02 compared to 1.10 for regular SCIM.

The third alternative, memoizing single-cycle instructions in the decode stage only doesn't perform much better. Only 30% of the previously successful lookups are detected now. Morever if IM can be performed in the decode stage why not dedicate an ALU or two to speculatively perform calculations in the decode stage. Thus we could conclude that SCIM reaps no real performance gains. Nevertheless SCIM in all its variants enhances FP performance over only using MCIM. If it isn't possible to add an ALU any of the above techniques will suffice to boost performance. In appendix A we will present how MCIM and SCIM work on real world processors.

Chapter 6

Comparing IM to Other Techniques

In the introduction chapter of this thesis we surveyed previous occurrences of memoization in the literature and other related techniques. In this chapter we will compare our view of Instruction Memoization to other techniques proposed, list the advantages and disadvantages of IM over these schemes and try to qualify the differences. We won't quantify the differences as each research uses slightly different benchmarks with slightly different simulators and in some cases uses different units of measurement.

In this chapter we have not chosen to belittle the work of others. All research is built upon previous successes and failures. We will show how IM expands earlier work on memoization and differs from Value Prediction (VP). The work of Sodani & Sohi on Instruction Reuse (IR) is monumental in exploring the sources of instruction reuse and in laying out a framework that strives to reuse all instructions. We will show how IM is different and complements IR.

6.1 Early Memoization

The earliest (1982) use of instruction reuse in hardware is by Harbison's *Tree Machine (TM)* [5]. The TM is a stack-oriented architecture which evaluates instructions at the head of the stack. A *value cache* is used in order to reuse instructions that haven't had their operands written to since the last evaluation of the instruction. In this case the evaluation of the instruction is performed by obtaining the result from the value cache. The technique is limited by two factors: the instructions are identified by their PC and are invalidated by a write to their operands, thus true value memoization isn't possible. The same operation might be performed by different instructions or the same instruction will use the same values (but be invalidated by a write to one of the operands). The technique is more suited to detecting CSEs during run-time and is almost impossible to compare to due to the extraordinary machine architecture.

In 1992 Richardson [6] proposed integrating memoizaton and trivial operation testing in multiplication, division, and sqrt instructions. This work is a direct predecessor to ours and differs only in scope. Richardson used *shade* [28] an instruction-level **non**-architecturally detailed simulator. The only architectural details supplied are the latencies of the memoized instructions. His results match ours in that longer latency instructions are more susceptible to memoization than short latency instructions. Our research, of course, is richer in detail and explores all aspects of a memoizing processor. Richardson [29] mentions that functions and code areas can be memoized as well (as we do in appendix B) but aside from a few simple examples he doesn't explore the issue in depth.

Flynn & Oberman [7] expand the idea to include storing the reciprocals of division instructions. In addition they perform a detailed analysis of the traits of the *division caches* used (size, associativity) and of the cost/performance tradeoffs (silicon area vs. CPI) associated with implementing them. As with the work of Richardson our research is of a broader scope and more detailed.

Azam, Franzon & Liu [27] use memoization in order to reduce power consumption rather than enhance performance. Thus almost every aspect of their reuse technique is different from ours: The stage of memoization (only if a lookup fails is the instruction executed), the instructions memoized (only multiplication), and the characteristics of the lookup table (small and direct-mapped in order to save more power).

6.2 Value Prediction

In 1996 and 1997 a series of papers were published that introduced and discussed the technique of *Value Prediction (VP)* (Gabbay & Mendelson [9], Lipasti, Wilkerson & Shen [10, 11], and Sazeides & Smith [12]). The idea is that the results of an instruction can be obtained speculatively based on results of previous invocations of the same instruction, exceeding the dataflow limit on extractable ILP.

The values are saved in a table and if they are constant (the same value is repeatedly produced), are different by a constant stride (an increment instruction will have a stride of 1), or follow some recurring pattern the result of the current invocation can be predicted with a high-degree of accuracy. The instructions are executed specuatively and aren't committed until their dependencies are satisfied. Of course a wrong prediction will cause the erroneous instruction and all instructions dependent on it to be recalculated.

The main difference between the techniques is their reliability: VP is speculative and while it may capture redundancy that can break the ILP limit it incurs a high overhead for mis-predictions. On the other hand IM is unspeculative and can't resolve data dependencies but it carries no overhead. Sodani & Sohi perform a detailed analysis of the differences in [30]. Perhaps a hybrid VP/IM implementation can exploit the advantages of both techniques.

Gabbay and Mendelson [31] have proposed to use program profiling in order to mark instructions that have a tendency to be predicted correctly and only

predict these instructions, thus lowering the mis-prediction rate. This is yet another major difference between VP and IM. IM is software transparent and may even be used across context switches.

6.3 Comparing IM to IR

The most comprehensive work in the field of reusing previous calculations was performed by Sodani & Sohi [8] in the years 1997-2000. They introduced the concept of *Instruction Reuse (IR)*. The instructions are inserted in a table called the *Reuse Buffer (RB)*. Three reuse schemes are presented:

S_v Each entry contains the PC, operand values, and result of an instruction. If the current instruction's PC and operands match an entry the result is used.

S_n Each entry contains the PC, operand register names and the result. If the current instruction's PC and operand register names match the result is used. If a register is written into, all entries using that register are invalidated. Thus it is enough for the PC to match.

S_{n+d} In addition to the information in the previous scheme each operand name has a link to its source instruction (if it's in the RB). By building these links instructions may be kept in the RB even if their registers are written upon (due to their links).

The first scheme is similar to IM, if the operands and operation match obtain the result from the RB. However IR uses the Program Counter as the sole index to the RB. Thus instructions at different locations can't use each others previous results. We will elaborate on this in section 6.3.1. This scheme is hampered due to the fact that the reuse test can be performed only in the instruction issue stage (the operands must be ready). For single-cycle instructions no cycle reduction is made.

The second scheme is aimed at solving this problem by comparing the register names of the fetched instruction to instructions in the RB. If the register names match *and* the registers' contents haven't been altered since storage in the RB, the result can be obtained from the RB as early as the fetch stage. This is a significant gain, unfortunately only the last appearance of an instruction can be used. Previous invocations with different operand values will have been invalidated.

Molina, González & Tubella [32] 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 maps an entry both by its PC and by its operand values (doubling the size of the table) is necessary in order to boost performance. In the fetch stage the PC is used to index the table, if the lookup is unsuccessful the operand values are used in the issue stage.

The third scheme suggested by Sodani & Sohi is targeted at exploiting dependent instructions fetched together, these instructions are called dependence chains. If dependence can be determined it is enough to detect reuse of the first instruction in the chain, the linked instructions can be reused as well. This scheme performs better than the second one as all instructions are chains of one. However only 25% of all dependence chains are of a length of more than one. Thus the use of this scheme is limited.

The conclusion of Sodani & Sohi is that their first scheme is the best as it uncovers the most reuse. However the potential for speedup is diminished as most instructions can be executed during the time it takes to perform a RB lookup. For this reason IM which is streamlined to use only the operand values can outperform the S_v scheme of IR. The reasons are due to the different mapping schemes, organization of the tables, the stage at which IM is performed, and the simplicity of IM.

6.3.1 PC vs. Value Mapping

. In section 3.2 we have shown that mapping MEMO-TABLE entries using the operand values is superior to using the PC (table 3.3 displays this clearly). The differences between mapping using the PC vs. mapping using the operand values can be understood by examining a simple yet widely used application: matrix multiplication.

```
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 6.1: *Naive matrix multiplication.*

The most naive scheme (table 6.1) performs N^3 multiplications when multiplying two $N \times N$ 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 FP multiplication only) for multiplying two 100×100 matrices are shown in the top graph of figure 6.1 (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 6.1 (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.

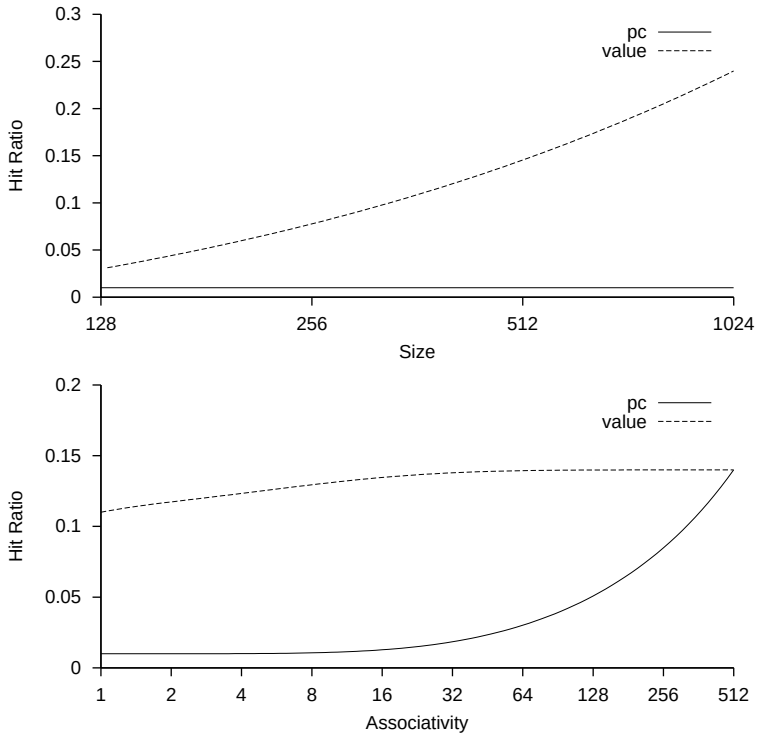


Figure 6.1: *Hit-ratios of the multiplication* MEMO-TABLE *in matrix multiplication.*

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

Using IM a 48% hit-ratio is achieved on the SPEC CFP95 benchmarks for FP instructions. Sodani & Sohi report only a 6.6% hit ratio. This is due to an inferior mapping scheme.

6.3.2 Table Organization

IR looks at all instructions as equal and uses a unified RB which contains all instructions. Instructions with longer latencies and a higher potential for improving performance are evicted from the RB by instructions whose reuse contributes much less. In section 5.2 we have shown that storing the single-cycle

instructions in the Integer MEMO-TABLE (integer division and multiplication) results in the hit-ratio of the "original" occupants of the MEMO-TABLE being suppressed.

On the other hand IM uses a set of MEMO-TABLES each containing different instruction types. Section 3.5 describes the advantages and disadvantages of using several tables or a unified one. In addition we have shown that the speedup attributed to single-cycle instructions (section 5.1) is due to creating a "virtual" ALU out of the MEMO-TABLE. In section 5.1.1 we have shown that adding an ALU is better than memoizing single-cycle instructions. In this case due to our distributed table structure we can choose to memoize choice single-cycle instructions or not to memoize them at all.

6.3.3 Lookup Stage

An IR lookup is performed in the decode stage. This is possible for the S_n and S_{n+d} schemes but not for the S_v scheme which must have the operand values available. As we have shown in section 5.2 even if a lookup is possible, only a small fraction of instructions have their operands ready at this stage.

Thus it is more likely that the lookup is performed in the issue stage. In this case due to the uniformity in which all instructions are treated in IR the instruction isn't issued to a unit until a lookup has been performed. But in this case a penalty of one cycle is paid if the lookup has failed.

Our tests have shown that IM in the issue stage results in an average speedup of only 1.05 compared to a speedup of 1.11 for performing IM in the issue stage only if a FU isn't available. If a FU is available, IM is performed in parallel to the FU execution, reducing the overhead of a miss to zero cycles. But there is no use in memoizing a single-cycle instruction in the execute stage. The distributed nature of IM which uses different MEMO-TABLES for different instructions enables us to treat single-cycle and multi-cycle instructions differently.

6.3.4 Design Simplicity

IM uses tables that store only values and operations. The entries in the tables are always valid (except on startup) even across context switches. In the case of a FP application sharing a processor with integer applications IM has a huge advantage, the FP MEMO-TABLES will remain untouched by the other applications.

IR must invalidate the RB across context switches as it is indexed by the PC. Even the first scheme of IR which stores operand values must keep track of memory references as it memoizes loads and stores. A write to a memory address invalidates other references to the same address. The other two schemes are much more complicated as every instruction executed may invalidate RB entries and links between instructions must be maintained at all times.

Chapter 7

Summary and Conclusions

This thesis explored the concept named memoization: *saving the input(s) and output(s) of previously calculated (side-effect-free) functions, and using the output if the input is encountered again.* However our focus was on very short functions: instructions. By saving the operands and results of previous invocations of executed instructions in dedicated tables (named MEMO-TABLES by us) implemented in the processor, it is possible to reduce the latencies of instructions from multiple cycles to one cycle. This is used to improve execution. We named this technique *Instruction Memoization (IM).*

A simulator (based on the SimpleScalar [17] simulator) of a RISC superscalar processor with IM integrated in its datapath has been constructed. On it we have run two sets of commonly used benchmarks (SPEC95 [18], MediaBench [19]). The simulations have been performed in three major stages:

1. The organization of the MEMO-TABLES has been explored in search for an "optimal" design that will maximize hit-ratio and minimize cost. The instructions memoized are multi-cycle instructions, instructions with latencies larger than one (chapter 3).

2. The integration of MEMO-TABLES in a RISC processor has been simulated and explored in order to quantify the speedup achieved by using IM (chapter 4).

3. The scope of IM was widened to include single-cycle instructions as well (chapter 5).

The following sections will summarize the stages and present our conclusions. We want to stress that this thesis deals with the architectural aspects of IM. The positive or negative influences of compilers, for super-scalar or EPIC[1] processors, on IM hasn't been tackled in this research. Nor has the cost of IM in

[1] An Explicitly Parallel Instruction Computing (EPIC) a.k.a Very Long Instruction Word (VLIW) computer, schedules during compile time several operations to several FUs. Reducing the latencies of instructions might not improve computation if instruction scheduling is static.

terms of number of transistors, power consumption, or design complexity been discussed. We have performed several simulations that compare IM to other architectural enhancements but not on a transistor to transistor basis, these results are presented later.

7.1 MEMO-TABLE Organization

Our first task was to prove that instruction results are reusable. This was performed by capturing the operands of all multi-cycle instructions executed in an "infinitely" large "fully associative" MEMO-TABLE (in practice 1M entries in sets of 512). The simulations have shown that 60% of all dynamic instruction appearances are repeatable, they are executed with the same operand values.

We then proceeded to characterize the "optimal" MEMO-TABLE structure. A MEMO-TABLE is "cache-like", it saves the last instructions executed. Thus the cache-like traits: size, associativity, replacement method, and mapping scheme were explored first. Then schemes like trivial calculation detection, commutative and inverse operation detection were tested. Finally the number of MEMO-TABLES and the contents of each MEMO-TABLE were investigated. The results and conclusions at this stage were:

- 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 recommended implementing IM with 5 MEMO-TABLES, each holding several of the multi-cycle instruction types. Each MEMO-TABLE contains 256 entries in sets of 4 . Entries are replaced randomly and are indexed by the operand values XORed with the opcode. Trivial calculations involving values of 0 or 1 aren't entered into the MEMO-TABLES but are detected with dedicated circuitry. This organization yields an average hit-ratio of 0.50, this is over 80% of the hit-ratio obtained when using an infinite fully-associative MEMO-TABLE.

7.2 IM in the datapath

The proposed MEMO-TABLE organization was integrated into a RISC superscalar processor with characteristics similar to the MIPS R10000 [24] and the Power PC 604e [25] processors. We discovered that 13% of the benchmarks' execution time can be attributed to multi-cycle instructions. With a 52% hit-ratio an average speedup of 1.07 was obtained. We then proceeded to alter the attributes of the datapath to check their influence on IM and vice-versa. Our results and conclusions are:

- The only datapath factor that effects the hit-ratio is the branch prediction rate. Mispredicted branches cause instructions to be flushed from the pipeline, many of these instructions may later be re-executed causing hits in the MEMO-TABLES.
- The major contribution of IM is the reduced latency of successfully memoized instructions. Having instructions complete execution earlier enables dependent instructions to be issued earlier. The number of cycles an instruction spends in the pipeline is reduced, which directly reduces execution time.
- A minor contribution to enhanced execution is the virtual addition of FUs. When a structural hazard occurs a MEMO-TABLE lookup may be able to provide the instruction's result, circumventing the execute stage of the pipeline.
- Given a fixed latency for multi-cycle instructions, IM works better for faster processors. A processor with a wide pipeline, a near perfect memory hierarchy, a high rate of branch prediction, and enough FUs will encounter a bottleneck when waiting for long latency instruction to complete. IM relieves this bottleneck. The basic processor has an IPC of 1.22, IM provides a speedup of 1.07. On a processor with an IPC of 3.65 the speedup of using IM is 1.09.
- IM is a technique that predominantly favors FP intensive applications. The speedup for FP applications is 1.10, for integer applications it is only 1.01 (for applications which heavily use integer division and multiplication). A way must be found to widen the scope of IM.

7.3 Single-Cycle Instruction Memoization (SCIM)

In order to encompass more instructions in IM we added a MEMO-TABLE that contains most integer single-cycle instructions. 57% of these instructions are reused resulting in a 1.11 speedup. However the speedup is only the result of reducing the structural-hazard ratio. The MEMO-TABLE is used as an additional FU, supplying results when no FU is available. We continued to explore this aspect of SCIM and arrived at the following conclusions:

- Adding more FUs to a processor minimizes the impact of SCIM. When the structural-hazard ratio reaches 0 the effect of memoizing single-cycle instructions is non-existent.
- Adding more FUs doesn't harm MCIM, in fact it performs even better.
- Better performance is gained by adding just one ALU and implementing SCIM, than implementing MCIM.
- Using the area dedicated to the MEMO-TABLES to enlarge on-chip caches or improve branch prediction proves better than IM for integer applications but not for FP applications.

7.4 The Bottom Line

The bottom line is that IM improves FP processing. By reusing previous calculations the latency of multi-cycle instructions is reduced 50% of the time to one cycle. Thus, in practice the latency of FP instructions is cut in half.

The more powerful the processor is the better it can utilize IM. The only enhancement that reduces the effectiveness of IM is reducing the latency (not the throughput, IM works fine with pipelined FUs), this doesn't seem to be the trend in state of the art microprocessors.

Appendix A

IM on Real Processors

In the body of this research IM has been an academic issue described and simulated in the context of an unexistent processor. We will now describe and quantify the effect of IM on two real processors: The MIPS R10000 [24] anf the Power PC 604e [25]. Tables A.1 and A.2 list the characteristics of both processors. Both processors are similar in their memory hierarchy, branch prediction capabilities, functional units and instruction latencies (slightly shorter for the R10000). The main difference is in their super-scalar capabilities. While the R10000 has three instruction queues (Integer, FP, Memory) of 16 instructions each, the 604e has only 2-instruction reservation stations for each FU. This limits the out-of-order issue capability of the 604e.

SimpleScalar was modified to simulate both processors as close to reality as possible[1]. The benchmarks were then run on the simulators with and without IM (MCIM at this stage). The IM is performed at the execute stage of the pipeline if a FU is available and at the issue stage if not. IM latency is one cycle and the MEMO-TABLE structure defined in chapter 3 is used. The results of both sets of simulations are compared to the basic processor in table A.3.

The results for MCIM are similar with the basic processor having a slight edge. The 604e is a slightly slower processor and as we have shown in section 4.4 benefits less from IM. The R10000 is almost as fast as the basic processor but has shorter instruction latencies for FP instructions which leads to a lower FE and speedup (section 4.3).

The main difference is in the results of SCIM. Single-cycle instructions may benefit from IM if at the issue stage they are ready to be issued but lack a FU to execute on. The MEMO-TABLE is then utilized as an additional FU. For the basic processor 27% of all hits are performed in the issue stage. However for the R10000 and 604e the ratio of hits in the issue stage is much lower being 17% and 8% respectively. This strengthens our claim that SCIM is of limited use.

[1] The instruction set of the R10000 is identical to the SimpleScalar ISA. The 604e ISA is different which might lead to slightly inaccurate results.

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	512-entry BHT, 2-bit counters
Registers	32 General Purpose, 32 Floating Point
Function Units	2 IALU*, 1 IMULT 1 FADD unit, 1 FMULT, 1 MMU**
Instruction Latencies & Throughputs	Integer multiplication: 6,6 Integer division: 35,35 All other integer instructions: 1,1 Floating point multiplication: 2,1 Floating point division: 19,21 (sp: 12,14) Floating point Sqrt: 33,35 (sp:18,20) All other floating point instructions: 2,1
Pipeline attributes	4-instructions fetched, decoded, issued, and committed per cycle; 32 instructions in Active List; 16 instruction INT, FP, Address queues; out-of-order execution; in-order retirement

* One of the IALUs performs idiv.

** Has a dedicated ALU for EA calculation.

Table A.1: *Characteristics of the MIPS R10000 microprocessor.*

L1 Instruction Cache	32-KBytes, 32-Byte blocks, 4-way associative
L1 Data Cache	32-KBytes, 32-Byte blocks, 4-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	512-entry BHT, 2-bit counters
Registers	32 General Purpose, 32 Floating Point
Function Units	2 IALU, 1 IMULT 1 FPU*, 1 BPU, 1 MMU**
Instruction Latencies & Throughputs	Integer multiplication: 3,1 Integer division: 20,19 All other integer instructions: 1,1 Floating point multiplication: 3,1 Floating point division: 31,31 (sp: 18,18) Floating point Sqrt***: 60,60 (sp: 50,50) All other floating point instructions: 3,1
Pipeline attributes	4-instructions fetched, decoded, issued, and committed per cycle; 16 instructions in Reorder Buffer; 2-instruction reservation stations for each FU; out-of-order execution; in-order retirement

* Performs all FP instructions.

** Has a dedicated ALU for EA calculation.

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

Table A.2: *Characteristics of the PPC 604e microprocessor.*

Processor	*IPC*	*hr*	*FE*	*Speedup*
MCIM				
Basic	1.27	0.51	13%	1.07
R10000	1.23	0.51	9%	1.06
604e	1.06	0.51	11%	1.06
SCIM				
Basic	1.27	0.55	-	1.11
R10000	1.23	0.55	-	1.08
604e	1.06	0.54	-	1.06

Table A.3: *Comparison of R10000, 604e, and "basic" processors (*MCIM *and* SCIM *integrated into pipeline).*

Appendix B

Memoization of Functions

We have shown in the previous chapters that IM works for instructions and enhances execution. 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 B.1 shows the latencies in cycles of several common mathematical and trigonometric functions in the Pentium II processor [33],[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. We will call this scheme *Function Memoization (FM)*.

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 B.1: *Latencies of mathematical functions, in cycles*

Figure B.1 shows a schematic layout of the idea using a hardware-implemented

[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.

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.

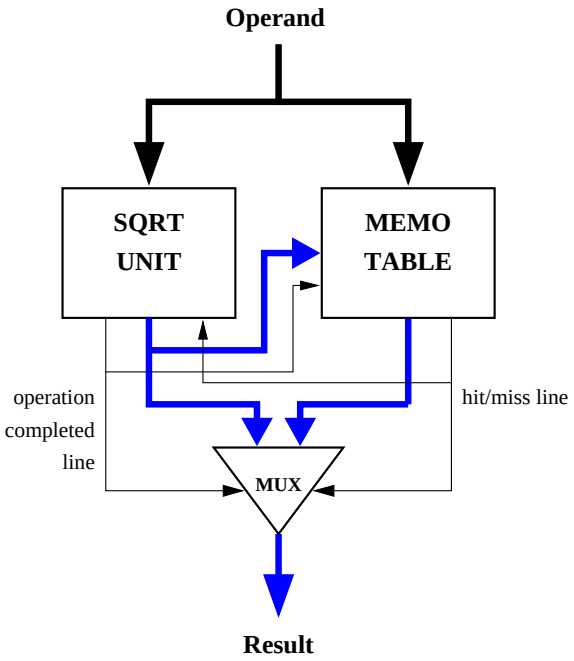


Figure B.1: *A square root unit using a* MEMO-TABLE

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

B.1 Memoization of Mathematical Functions

This section describes how using MEMO-TABLES accelerates computing mathematical and trigonometric functions. The MEMO-TABLE used is identical to the MEMO-TABLE described in the previous chapters. Each entry contains two operands, a result, and a field that identifies the function. A description of how a MEMO-TABLE works in tandem with a FU was covered in the previous chapters and won't be covered her. What we will show is how memoization is implemented if the function is calculated in software (section B.1.1) and analyze the overhead of FM (section B.1.2).

B.1.1 Memoization of Software Implemented Functions

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 function 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 the MEMO-TABLE.

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. 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 [17] 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 code.

  When executed the instruction uses the values in IN1, IN2 and the function identifier to index a separate or unified MEMO-TABLE (separate tables: FID identifies the MEMO-TABLE and IN1 & IN2 index it, 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 code.

  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 B.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 B.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).*

B.1.2 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 B.2.1), 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 an Integer Unit which executes the branches. **lupm1** and **updm1** are executed by the FP Unit.

cycle	*FP Unit*		*Integer Unit*	
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 **lupm1** and `bc1t` and the use of the FU by both **updm1** and mov.d the overhead of a miss is two 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 update to be performed in parallel to the function's cleanup. This reduces the miss penalty 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.

B.2 Experiments and Results

To verify the usefulness of memoization of mathematical and trigonometric functions, we performed a series of experiments with SimpleScalar [17] (the same simulator used for the simulations in chapter 3) , 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 the hit-ratio and speedup. Naturally, they depend on the specific design of the MEMO-TABLE. The size, associativity and contents of the MEMO-TABLE, impact the expected hit-ratio and speedup.

B.2.1 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) as we have seen in chapter 3. 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 B.2.4 we explore memoization of user defined functions, in section B.2.5 we compare function memoization to instruction memoization, and in section B.2.6 we compare function memoization to using the same hardware to implement the functions in hardware on-chip.

The overhead of memoization in our simulations is two machine cycles, the stricter of the two options shown in section B.1.2. The simulated system is built upon the MIPS R10000 processor [24]. 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 [18].
- **MediaBench** - a suite of multi-media and communication applications from UCLA [19].
- **Khoros** - Khoros Pro 2000 [34] 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 B.3 describes the specific applications, and table B.4 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

[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).

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

Table B.3: *Description of benchmark applications*

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.

B.2.2 Speedups Obtained

The basic configuration of a MEMO-TABLE that we have chosen is one with 256 entries arranged in 64 sets (set associativity of 4), each function has it's own MEMO-TABLE. Table B.5 shows the results. We compare the results of using "infinitely" large fully associative MEMO-TABLES to the results of using 9 256 entry 4-way associative MEMO-TABLES. In addition we show the results of using a single 512 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).
- *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.

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 B.4: *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*

The more significant number is the speedup. An average speedup of 10% (harmonic mean) is attained (11% for a unified MEMO-TABLE).

While the average hit ratios and speedup are good we find a lack of correlation between them, as is the case for IM (sections 4.2 and 4.3). Figure B.2 shows that for the SPEC and Khoros applications the hit ratios are higher than for the MediaBench benchmarks. On the other hand figure B.3 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 (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.

B.2.3 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[4] out of 17 application used in the previous tests. Figure B.4 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.

[4]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.

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

Table B.5: *Performance enhancement with memoization of mathematical and trigonometric functions.* MEMO-TABLES *are either infinitely large, of size 256 for each function or a 512-entry unified for all functions.*

The figure compares a unified MEMO-TABLE (dashed line) to separate MEMO-TABLES 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 B.5 corroborates this by showing that the average speedup achieved using a unified MEMO-TABLE of size 512 (11%) is slightly greater than the average speedup achieved when using 9 separate MEMO-TABLES of size 256 (10%). This is due to the fact that most applications heavily use only two or three functions (table B.4).

Figure B.5 shows the hit-ratios as a function of set associativity. For separate 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

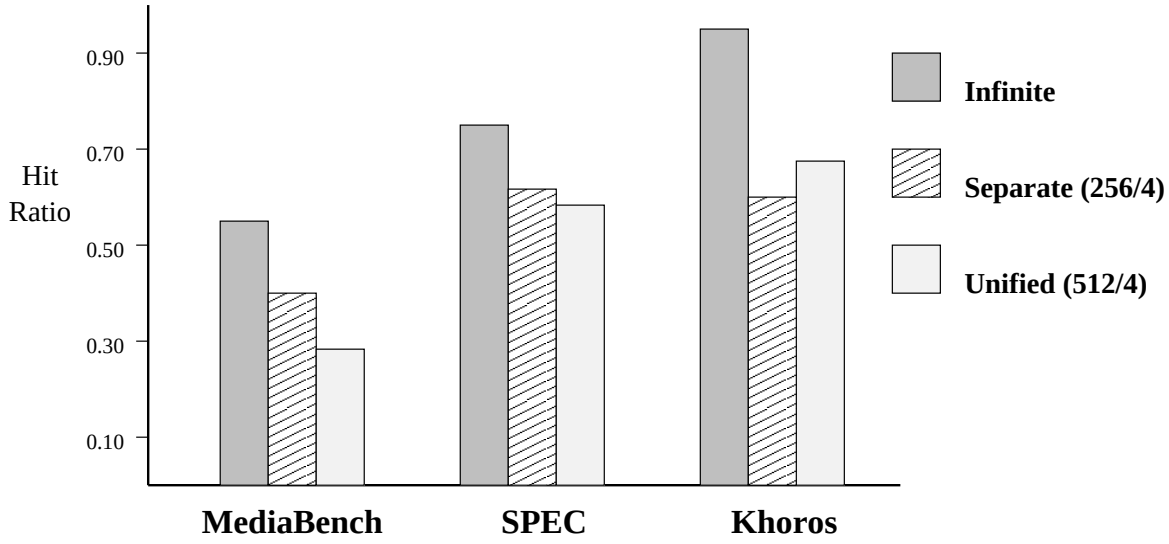


Figure B.2: *Breakdown of hit ratios by suite*

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.

B.2.4 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 (function OVL) and wave5 (functions VAVG, ERF, DENSX, and DENSY). The functions memoized have one or two arguments and one return value. As such they are perfect candidates for memoization in our proposed infrastructure, and the results of memoizing them are encouraging. Table B.6 shows the hit ratios and speedups of memoizing user defined instructions (in addition to the mathematical instructions) compared with only memoizing mathematical functions.

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 instructions avoiding

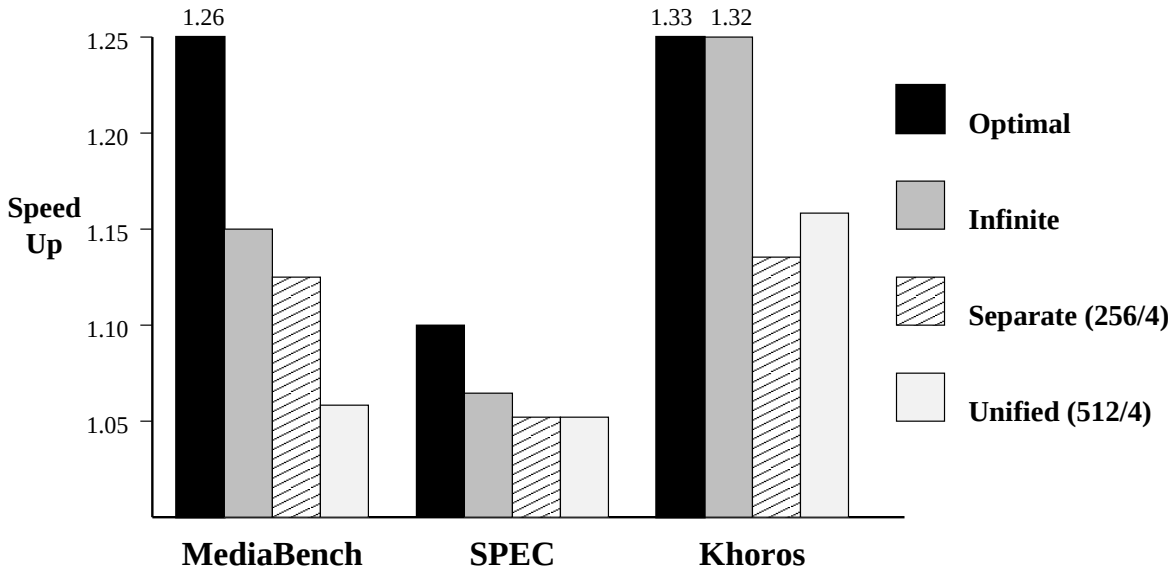


Figure B.3: *Breakdown of speedups by suite*

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

Table B.6: *Memoization of user defined functions (and math + trig functions) compared with memoization of only math and trig functions.*

execution.

B.2.5 Memoization of Functions and Instructions

In this section we will integrate the technique of IM proposed in the previous chapters with the technique of FU introduced in this chapter. We will compare 3 implementations::

1. The implementation explored in this chapter where functions are memoized.
2. Multiple-cycle instructions (without loads/stores) are memoized. Functions are implemented in software and benefit from the memoized instructions.
3. A combined approach where both functions and instructions are memoized.

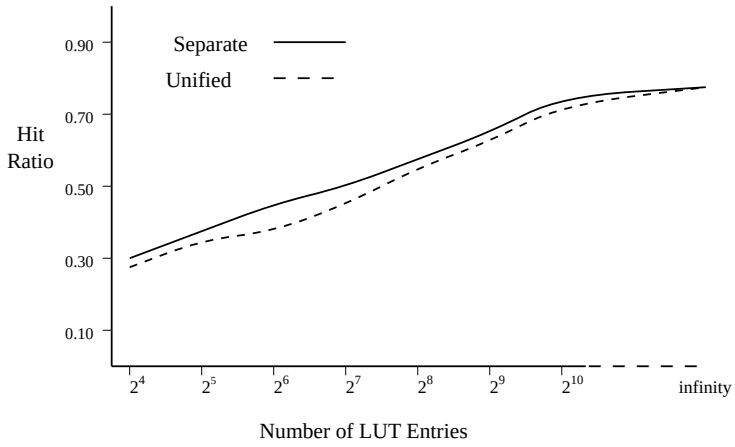


Figure B.4: *Hit ratios as a function of LUT size (set size is 4).* MEMO-TABLE *is unified (dashed line) or separate* MEMO-TABLES *are used (solid line).*

The MEMO-TABLES used to memoize instructions are the tables recommended at the end of chapter 3. Figure B.6 shows the speedups per suite.

The figure 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.

B.2.6 Implementing the Functions in Hardware

In this section we will compare a processor that implements the mathematical and trigonometric functions on chip (like the Pentium family does) to a processor that memoizes these functions. In both cases square-root taking is implemented on chip. In addition we will combine both approaches and memoize the hardware implemented functions. The latencies of the on chip functions are the average

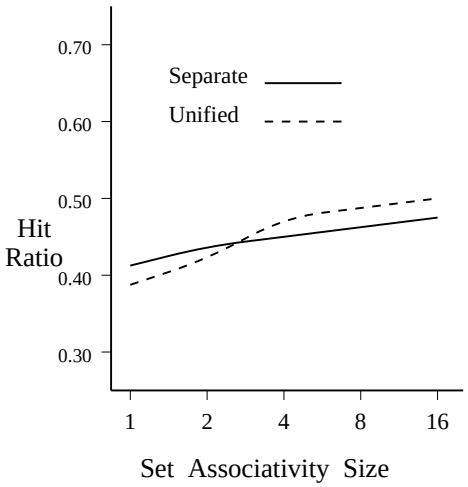


Figure B.5: *Hit ratios as a function of set associativity size.* MEMO-TABLE *is unified (dashed line, 512-entry) or separate* MEMO-TABLES *are used (solid line, 256-entry per* MEMO-TABLE*).*

latencies shown in table B.1.

Figure B.7 compares the approaches. The hardware only approach yields the worst results. For the SPEC benchmarks which use the math & trig functions much less than the Multi-Media applications the hardware approach barely surpasses the base processor. The combined approach is the fastest as it benefits from a lower latency for MEMO-TABLE misses and from a latency of one cycle for MEMO-TABLE hits.

B.3 The Rationale Behind Function Memoization

It is important to understand why the technique works. Why do benchmarks such as vgbox, turb3d and klogexp display such high benchmarks. A look at a simplified excerpt from the source code of vgbox shows (table B.7) that it is computing in a loop the Sine and Cosine of a variable. Profiling showed that this variable doesn't change. 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 doesn't change. Using a MEMO-TABLE solves the problem by saving the previous computations.

The benchmark turb3d contains code that performs a complex Fast Fourier

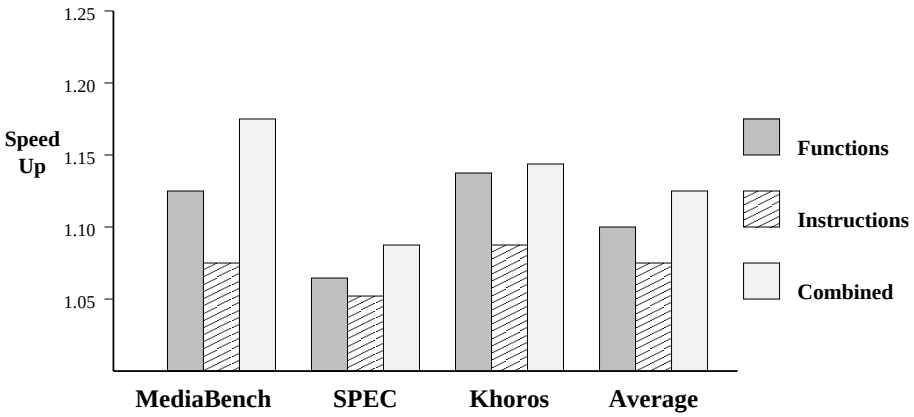


Figure B.6: *Breakdown of function, instruction, and combined memoization speedups by suite.*

Transformation (FFT). The main loop of the fortran subroutine is shown in table B.8. In the inner loop half the values of `TI` are the same as from the previous 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. Section 2.2 has shown that neighboring pixels in an image tend to have the same values leading to a high hit-ratio in the MEMO-TABLE.

```
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 B.7: *Simplified vgbox code.*

B.4 Related Work

Two techniques are comparable to FM. The first is a hardware implementation with extensive software support. The other is a pure software approach.

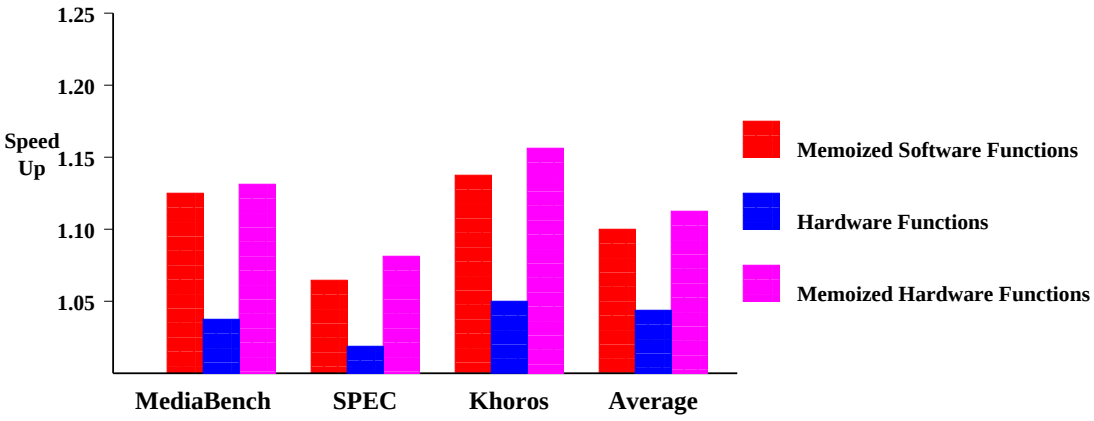


Figure B.7: *Breakdown of software memoization, hardware implemented functions, and combined hardware memoization speedups by suite.*

```
      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 B.8: *FFT routine from turb3d.*

B.4.1 Compiler-Directed Dynamic Computation Reuse

Connors & Hwu [13] propose a general technique for reusing large regions of code which have distinct entry and exit points. They named it: Compiler-Directed Dynamic Computation Reuse. 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. Both works are limited by the number of function arguments and

return values they can support.

B.4.2 Value Profiling

A 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 software miss is greater than the overhead of a hardware miss.

B.5 Comparing Hardware to Software Memoization

A fundamental question about memoization is: "Why can't it be done in software?" At the instruction level it is obvious that software memoization is of no avail. The overhead of a lookup would be tens of instructions. Nevertheless the question is valid in the scope of function memoization.

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 one or two machine cycles as shown in section B.1.2.
- 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.

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

Table B.9 compares hardware based to software based memoization. Shown are the average speedups per suite when using separate MEMO-TABLES for each function. The size of each MEMO-TABLE is 256 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 256/4 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 results were that only 5 out of 17 applications achieve any speedup (with software-memoing & 256-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 favors 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.

suite	*256-entry tables*				*best achieved soft memoization*	
	4-way		*direct*			
	hard	*soft*	*hard*	*soft*	*size/assoc*	*spdp*
MediaBench	1.12	0.86	1.10	0.90	1024/2	0.95
SPEC	1.06	0.93	1.05	0.95	2048/2	1.01
Khoros	1.13	0.88	1.12	0.92	1024/2	1.03
Harmonic mean	**1.10**	**0.89**	**1.08**	**0.93**		**1.01**

Table B.9: *Speedup comparison between hardware based to software based memoization. Hardware* MEMO-TABLES *are separate and of size 256/1 and 256/4.*

B.6 Summary

This chapter 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, 512-entry, 4-way associative MEMO-TABLE's size is 16KBytes) it is possible to attain an average (harmonic mean) 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 one or two 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.

In comparing FM with IM 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. We compared a processor that implements the mathematical and trigonometric functions in hardware on chip, to a processor that memoizes these functions on chip, but executes them in software. The results showed that the latter processor was 7% faster than the former one. Combining both approaches yields a speedup of 15% over the base processor.

Bibliography

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

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

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

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

[5] S. Harbision, "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.

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

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

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

[9] F. Gabbay and A. Mendelson, "Speculative Execution based on Value Prediction", EE Department TR #1080, Technion - Israel Institute of Technology, November 1996.

[10] 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.

[11] M. Lipasti and J. Shen, "Exceeding the Dataflow Limit via Value Prediction", *Proc. of the 29th Int. Symp. on Microarchitecture*, December 1996.

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

[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] 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.

[16] A. Sodani and G. Sohi, "An Empirical Analysis of Instruction Repetition", *Proc. of the 8th Int. Conf. on Architectural Support for Programming Languages and Operationg Systems*, pp. 35–45, October 1998.

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

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

[19] 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.

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

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

[22] http://www.mot.com/SPS/PowerPC/products/semiconductor/cpu/750.html

[23] 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.

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

[25] http://www.mot.com/SPS/PowerPC/products/semiconductor/cpu/604.html

[26] http://www.support.compaq.com/alpha-tools/documentation/current/chip-docs.html

[27] M. Azam, P. Franzon, and W. Liu, "Low Power Data Processing by E limination of Redundant Computations", *7th Int. Symp. on Low Power Electronics and Design*, August 1997.

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

[29] S. Richardson, "Caching Function Results: Faster Arithmetic b y Avoiding Unnecessary Computation", *Sun Microsystems Laboratories*,Techni cal Report TR-92-1, September 1992.

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

[31] F. Gabbay and A. Mendelson, "Can Program Profiling Support Value Prediction?", *Proc. of the 30th Int. Symp. on Microarchitecture*, pp. 138–148, December 1997.

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

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

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