H-0196 October 19, 2003
Computer Science

IBM Research Report

"Look It Up" or "Do the Math": An Energy, Area, and Timing Analysis of Instruction Reuse and Memoization

Daniel Citron
IBM Research Division
Haifa Research Laboratory
Haifa 31905, Israel

Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University of Jerusalem
91904 Jerusalem, Israel

“Look It Up” or “Do The Math”: An Energy, Area, and Timing Analysis of Instruction Reuse and Memoization

Daniel Citron
IBM Haifa Labs
Haifa University Campus
Haifa 31905, Israel
citron@il.ibm.com

Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University of Jerusalem
91904 Jerusalem, Israel
feit@cs.huji.ac.il

Abstract

Instruction reuse and memoization exploit the fact that during a program run there are operations that execute more than once with the same operand values. By saving previous occurrences of instructions (operands and result) in dedicated, on-chip lookup tables, it is possible to avoid re-execution of these instructions. This has been shown to be efficient in a naive model that assumes single-cycle table lookup. We now extend the analysis to consider the energy, area, and timing overheads of maintaining such tables.

We show that reuse opportunities abound in the SPEC CPU2000 benchmark suite, and that by judiciously selecting table configurations it is possible to exploit these opportunities with a minimal penalty. Energy consumption can be further reduced by employing confidence counters, which enable instructions that have a history of failed memoizations to be filtered out. We conclude by identifying those instructions that profit most from memoization, and the conditions under which it is beneficial.

1 Introduction

During program execution there are operations that execute, more than once, with the same operand values. Several papers published in the late 90s proposed exploiting this fact by saving previous occurrences of instruction level operations (operands and result) in dedicated, on-chip, lookup tables. It is then possible to avoid execution of these instructions by matching the current executing instruction’s operands with an entry in the table.

The approach of Sohi and Sodani [16] is to reuse instructions (identifiable by the Program Counter) early in the pipeline by matching their operands or by establishing that their source registers haven’t been overwritten since the instruction’s last invocation. Their technique is called *Instruction Reuse*.

Citron, Feitelson, and Rudolph [4] extend the idea of Richardson [14] and perform the reuse test on the operand values and operation, in parallel to instruction execution. This enables different static instruction instances to reuse each other’s results. This was coined *Instruction Memoization*.

Molina, González, and Tubella [11] combine both approaches: a match is first attempted when indexed by the PC and if that fails the operand values are used as an index. A limited study (multiplication in four applications) by Azam, Franzon, and Liu [1] suggests memoization as a power saving method.

However, these models are naive and outdated. They assume that the latency of the table lookup time is a single cycle, that it can be performed in parallel or ahead of computation without any timing or power penalty, and that a successful lookup will enhance performance. These and other shortcomings have been reviewed by Citron and Feitelson [3].

Table 1 shows that the latencies of most instructions on the current generation of microprocessors are growing in comparison to their predecessors. Instruction Memoization, or IM (this is the term we will use throughout this study), has the potential to reduce these latencies and enhance performance given a model that is adapted to the deep pipelines, short cycles, and tight energy budgets of present and future microprocessors. The contributions of this study are fourfold:

1. Ratify that reuse opportunities still exist in CPU intensive applications. Section 2 presents the reuse rates for the SPEC CPU2000 suite compiled for IBM’s Power4 [19] 64-bit architecture.
2. Explore the organization of the lookup tables in

Processor	*Clock Rate*	*IALU*		*IMUL*		*IDIV*		*FADD*		*FMUL*		*FDIV*	
		l	t	l	t	l	t	l	t	l	t	l	t
Power3-II [13]	450MHz	1	1	5	1	37	37	3	1	3	1	18	18
Power4 [20]	1.5GHz	1	1	7	6	68	67	6	1	6	1	30	30
Pentium III [6]	1.4GHz	1	1	4	1	56	56	3	1	5	2	38	38
Pentium 4 [7]	3.2GHz	.5	.5	15	5	56	23	5	1	7	2	38	38
UltraSPARC II [17]	480MHz	1	1	5	5	36	36	3	1	3	1	22	22
UltraSPARC III [18]	1.2GHz	1	1	6	5	39	38	4	1	4	1	20	17

Table 1. *Latencies and throughputs of instructions on current and previous generations of microprocessors.*

terms of reuse rate, access time, energy consumption, and area. Section 3 performs this analysis with 2^k and full factorial designs.

3. Enhance the reuse process. Section 4 will show how using multiple lookup tables (per instruction class), trivial computation detection, and confidence estimators can raise the reuse rate and reduce the miss penalty.

4. Determine which instructions benefit most from IM. Section 5 compares the physical features of various functional units to lookup tables that store their results.

2 Instruction Memoization and Reuse Potential

This section will reconfirm the potential of IM by measuring the reuse rate of an infinitely large lookup table. A lookup table, which we will coin a MEMO-TABLE, is a cache like structure that is composed of a relatively large tag (opcode + operands) portion and a relatively small data portion (result). Figure 1 shows a MEMO-TABLE designed to contain the opcodes, operands, and results of IBM's Power4 [19] 64-bit instructions. The extensive use of FMADD (Floating point Multiply ADD) instructions necessitates the storage of three 64-bit operands in the table. The opcode field is composed of 6 bits of the basic opcode (OPC) and 10 bits of the extended opcode (XO). All bits that aren't used are zeroed when an instruction is placed in the table. There is no need for a valid bit, an illegal opcode loaded at boot-time will prevent matching and reading invalid data.

The daunting problem of matching 207 bits is one aspect that has been neglected by previous studies that focused on two 32-bit operands. Widespread 64-bit computing and enhanced instruction sets forces us to deal with this problem head-on. Section 3 examines the impact this has on performance, power, area and access time.

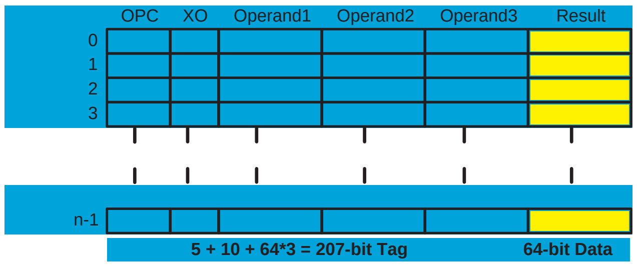


Figure 1. *A generic* MEMO-TABLE *capable of memoizing all Power4 instructions.*

2.1 Simulation Methodology

The infrastructure for all our simulations is Aria [12], an environment for PowerPC microarchitecture exploration. The environment dynamically traces all user and library code (system calls are executed but not traced). Drivers can be written that collect and analyze any subset of instruction types, data values, memory references etc. Specifically, we built drivers to collect memoization statistics for various instruction types.

The data was collected from the SPEC CPU2000 suite using the MinneSPEC [10] input sets. Table 2 shows the exact inputs used and the number of instructions simulated. The C/C++ benchmarks were compiled on a Power4 running AIX version 5.1 using the IBM compiler `xlc` v6.0 with the flags: `-q64 -DSPEC_CPU2000_LP64 -O5`. The Fortran benchmarks were compiled using the `xlf` v8.1 compiler with the flags: `-q64 -O5`.

2.2 Instruction Memoization Potential

In order to gauge the potential of instruction memoization we performed an experiment that measures the reuse rate of most instructions using an "infinite" MEMO-TABLE (1 million entries with 64-way associativity, LRU replacement, and indexed using the XOR

Benchmark	*Input*	*# inst.*
164.gzip	lgred - log	434M
175.vpr	lgred - place	2000M
176.gcc	lgred	2000M
181.mcf	lgred	836M
186.crafty	lgred	838M
197.parser	lgred	2000M
252.eon	lgred - cook	761M
253.perlbmk	lgred	1921M
254.gap	lgred	772M
255.vortex	lgred	1278M
256.bzip2	lgred - source	1759M
300.twolf	lgred	925M
168.wupwise	lgred	2000M
171.swim	lgred	304M
172.mgrid	lgred	94M
173.applu	lgred	66M
177.mesa	lgred	850M
178.galgel	lgred	210M
179.art	lgred	2000M
183.equake	lgred	871M
187.facerec	lgred	356M
188.ammp	lgred	1207M
189.lucas	lgred	212M
191.fma3d	lgred	540M
200.sixtrack	lgred	1329M
301.apsi	lgred	257M

Table 2. *Benchmarks (CINT2000 top half, CFP2000 bottom half), input sets, and instructions executed. The default inputs are the* lgred *sets of MinneSPEC. Benchmarks were terminated after 2 billion instructions.*

of the lower bits of the operands and opcodes). The instructions omitted are of two classes:

- **Branches:** Conditional branches in the Power architecture determine their outcome on precomputed bits in condition registers. Memoizing the instruction to obtain the next PC based on the current PC and condition bits is exactly what the Branch Prediction Unit does, there is no need to duplicate this functionality.

- **Loads/Stores:** Memoization of memory references based on the base address and offset requires storing the effective address and implementing an invalidation mechanism every time data is stored to memory. This reduces the technique to just another level in the memory hierarchy. Nevertheless, the effective address calculation is memoized.

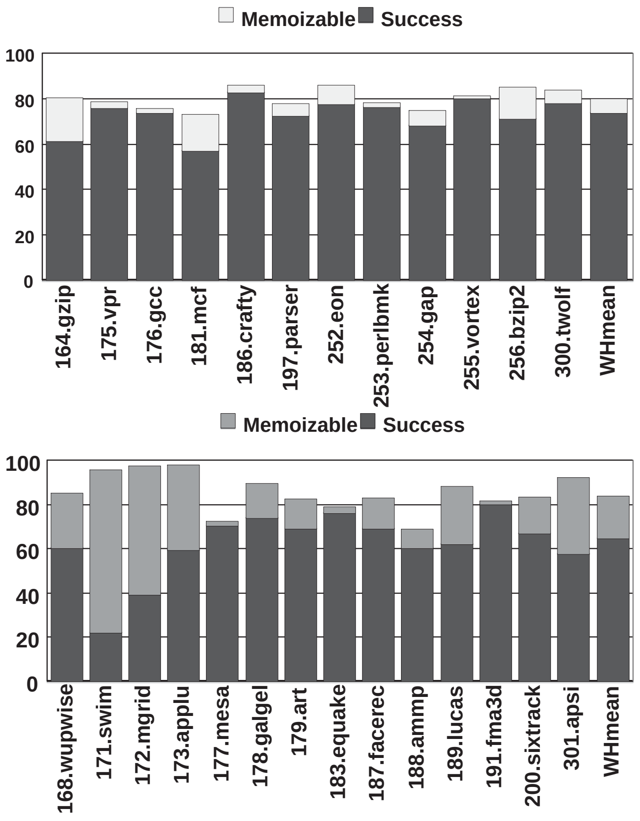


Figure 2. *Percent of all memoizable and successfully memoizable instructions when using a very large* Memo-Table.

Figure 2 shows the percent of dynamic instructions looked up (out of **all** executed instructions, including branches) and the fraction of successful lookups for all 26 applications in the SPEC CPU2000 benchmark. Nearly 75% of all dynamic instructions can be successfully memoized for the CINT2000 suite and 65% for the CFP2000 suite. The weighted harmonic mean is used for all averages in this study. It was chosen for its mathematical properties that are best suited for rates. A study by Yi and Lilja [21] using 7 CPU2000 benchmarks and different metrics concludes that the benchmarks have significant amounts of redundant computations. Our study ratifies these conclusions on the whole suite.

3 Memo-Table Structural Factors

The factors that influence the reuse rate, access time, energy consumption, and area are numerous: size, associativity, indexing, number of ports, etc. A full factorial design would entail hundreds of simulations. In order to reduce this to a manageable size we will first perform a 2^k factorial design using four factors (section 3.1) and then perform a full two-factor design (section 3.2) on the influential factors.

Factor	*Low Level*	*High Level*
size	32 entries	1024 entries
associativity	direct-map	8-way
indexing mode	PC	operands + opcode
replacement mode	random	Least Recently Used

Table 3. MEMO-TABLE *factors and levels used in the 2^k factorial design.*

3.1 2^k Factorial Design

A 2^k Factorial Design [9] is used to determine the effect of k factors, each of which has two levels. When a factor has a continuity of levels two extremes are chosen. The technique computes the allocation of variation contributed to each factor separately and in combination with others. The factors and levels we chose are described in table 3. The metrics measured are:

1. Reuse rates for the CINT and CFP benchmarks measured by dividing the number of successful memoizations by the number of memoizable instructions executed (and then taking the weighted harmonic mean).
2. Energy of an access (nJ).
3. Access time (ns).
4. Total area (mm^2).

Metric	*Allocation of Variation (%)*			
	S	A	M	R
Reuse Rate (CINT)	87	2	9	0
Reuse Rate (CFP)	95	2	2	0
Energy	35	57	0	0
Access Time	54	42	2	0
Area	86	13	0	0

Table 4. *Allocation of variation of the 2^k factorial design (Size, Associativity, Mapping, Replacement).*

One anomaly that arises is the use of the replacement method as a factor in conjunction with a direct mapped table. We chose to keep this level of associativity due to its influence on the access time and energy results. Access time and energy are calculated using CACTI 3.0 [15] modified to accommodate the large tag size[1] and to distinguish between different indexing and replacement modes. One read port, one write port and a technology of 90nm (forecast for next generation technology) are configured. The raw results are in table 5 and the allocation of variation is summed in table 4. From both we can conclude:

1. The size of the MEMO-TABLE is the dominant factor for all metrics except energy. A table with less entries is power efficient, fast, and small, yet it reduces the reuse rate. Further exploration is needed to determine the optimal table size.
2. The associativity of a MEMO-TABLE has a very small effect on the reuse rate yet a large effect on time, energy, and area. Smaller degrees of associativity should be explored.
3. Using the program counter as the index yields poor reuse rates yet hardly effects time, energy, or area. The reduced reuse rate is a result of: (i) all dynamic instances of an instruction are mapped to the same set, reuse is limited to the size of the set; (ii) dynamic instructions of different static instructions (with the same opcode) can't use each others results. Less than half the successful lookups can be attributed to the same static instruction.
4. The replacement method has hardly any effect on any of the metrics.
5. When assuming a clock rate of 2GHz (minimum estimate for future IBM Power implementations) even the fastest configurations take more than one cycle to complete a lookup. This must be: a) minimized. b) compared against the latency of memoized instructions.

After fixing the mapping (operands) and replacement (random) modes, the next step in our study is to perform a full two-factor factorial design using size and associativity.

3.2 Full Two-Factor Factorial Design

In this set of experiments we vary the size of the MEMO-TABLE from 32 to 1024 entries and the degree of associativity from direct-mapped to 16-way and 64-way (which represents fully associative in our model). Indexing is performed using the operands and opcodes and random replacement is implemented. Figure 3 shows the reuse rates[2], energy, access time, and area

[1] Two 64-bit operands were simulated, three operand tables are discussed in section 4.1.

[2] Just the CINT suite, the CFP suite displays similar behavior.

Configuration				*Results*					*Configuration*				*Results*				
S	A	M	R	cint	cfp	*nJ*	*ns*	mm^2	S	A	M	R	cint	cfp	*nJ*	*ns*	mm^2
32	1	pc	rnd	5.1	7.2	0.34	0.65	0.053	1K	1	pc	rnd	36.6	37.0	0.47	1.00	0.583
32	1	pc	lru	5.1	7.2	0.34	0.65	0.054	1K	1	pc	lru	36.6	37.0	0.47	1.00	0.587
32	1	ops	rnd	13.9	11.4	0.35	0.69	0.054	1K	1	ops	rnd	55.8	43.5	0.51	1.04	0.679
32	1	ops	lru	13.9	11.4	0.35	0.69	0.054	1K	1	ops	lru	55.8	43.5	0.51	1.04	0.683
32	8	pc	rnd	8.4	9.4	0.57	1.06	0.245	1K	8	pc	rnd	47.0	45.0	0.98	1.36	0.909
32	8	pc	lru	8.9	9.8	0.57	1.10	0.248	1K	8	pc	lru	49.0	47.0	1.01	1.36	0.917
32	8	ops	rnd	15.1	13.9	0.58	1.06	0.253	1K	8	ops	rnd	64.4	50.4	1.02	1.40	0.919
32	8	ops	lru	15.4	14.4	0.58	1.06	0.255	1K	8	ops	lru	66.4	52.0	1.04	1.40	0.929

Table 5. *Configurations and results of 2^k factorial design. S- size, A - associativity, M- mapping, R - replacement method.*

(Z-axis) as a function of size (Y-axis) and associativity (X-axis).

A surprising result is that fully-associative tables are faster and consume less energy than corresponding (same number of entries) set-associative tables. This is due to the CAM (Contents Addressable Memory) based design of a fully-associative cache in the CACTI model. For small MEMO-TABLES the overhead of tag decode, routing, and comparison out-weights the added delay and energy of the large CAM cells. However, the almost negligible effect that associativity has on the reuse rate indicates that a direct-mapped MEMO-TABLE is a much better choice.

Assuming a clock rate of 2GHz it would be wise to choose a configuration that minimizes the number of cycles it takes to access the MEMO-TABLE. A 512-entry, direct-mapped MEMO-TABLE has an access time of just under 2-cycles (0.94 ns), a reuse rate of 47.7% (37.9% for CFP), an energy consumption of 0.41 nJoules, and a total area of 0.40 mm^2. In section 4 we will show several techniques to reduce the MEMO-TABLE's size yet retain its reuse rate.

3.3 Instruction Reuse

The large overhead attributed to the tag comparison, the fact that a fully-associative MEMO-TABLE is feasible, and the latency of a MEMO-TABLE lookup, leads us to re-examine the Instruction Reuse (IR) scheme of Sohi and Sodani (see [16] for full details). IR has three versions:

S_v The PC is used to index the Reuse Buffer (RB), the operand values are used to verify reuse. This is similar to the configurations in section 3.1 where mapping is performed by the PC. The difference is that the PC must match and the opcodes are omitted.

S_n An entry is mapped by the PC and stores only the operand's register names, which reduces the tag size. Every time a register is overwritten the corresponding entries are invalidated. Reuse is verified by a PC match and valid bit. However, the scheme implies a CAM based design necessary for the invalidations.

S_{n+d} In order to overcome frequent invalidations, consuming instructions are linked to their producers. An entry is valid if its producer is in the table and is the last producer of the register value (an auxiliary table maps each architected register to the RB entry which has its latest result). Thus, a lookup can be composed of up to 3 table reads (instruction and two producers) and two accesses to the auxiliary table.

Two possible optimizations are using time stamps to test if an operand register was overwritten (this simplifies the RB structure), and not invalidating a register that has been overwritten with its current value (reduces the invalidation rate). Table 6 lists all configurations compared and figure 4 displays the results using 1024-entry, 4-way tables and 128-entry, direct-mapped tables.

Although the Snts scheme has better physical metrics than IM, and even assuming the reuse rate of the Snsv scheme, the diminished reuse rate makes it unattractive for future microprocessor enhancements.

4 Improving the Reuse Rate

The results obtained in the previous section, 47.7% (CINT) and 37.9% (CFP), are moderate at best. In this section we will suggest several techniques for enhancing the reuse rate and examine their influence on time, energy, and area.

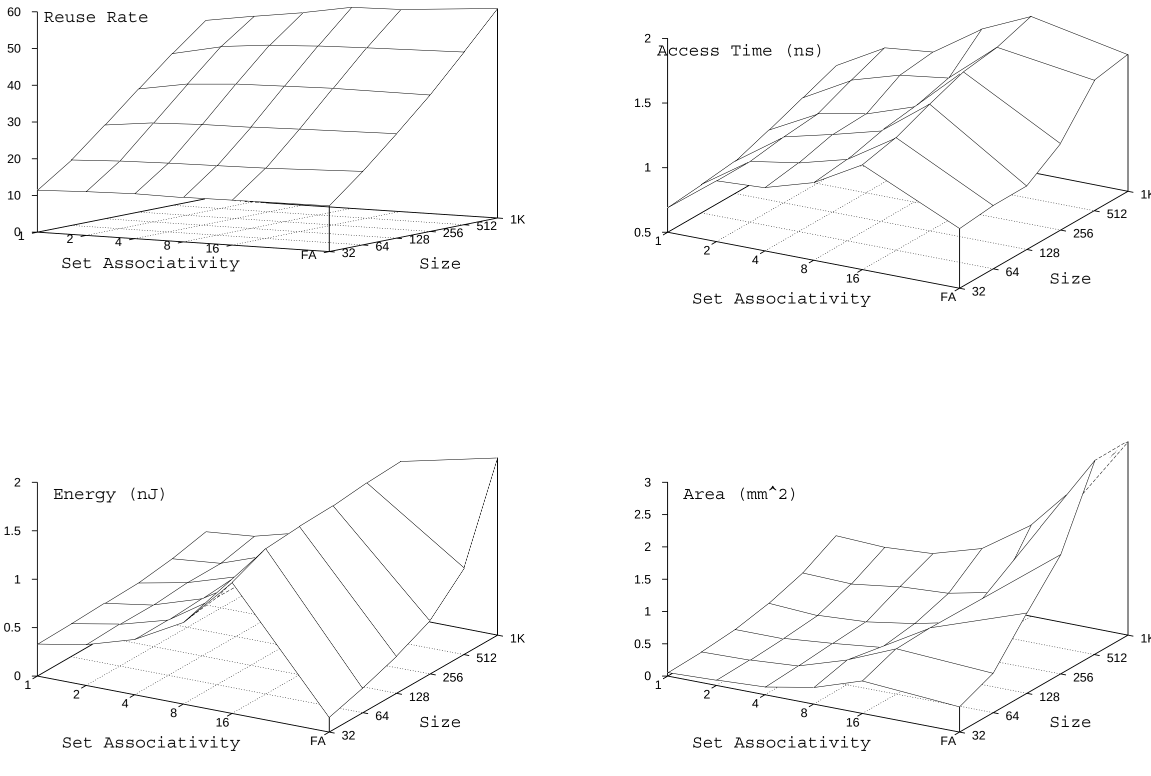


Figure 3. *Reuse rate, access time, energy, and table area as a function of a* Memo-Tableś *size and associativity.*

Name	*Configuration*
IM	Memo-Table described in section 3.2
Sv	S_v scheme described in [16][1]
Sn	S_n scheme of [16], built with CAM cells
Snts	S_n scheme using time stamps
Snsv	S_n scheme, same value doesn't invalidate
Sn+d	S_{n+d} scheme of [16][2]

Table 6. *Instruction Reuse Configurations.*

[1]Similar to Memo-Table mapped by PC.

[2]Incorporates Snts and Snsv techniques.

4.1 Multi Memo-Tables

In the previous experiments all memoized instructions have been "lumped" together into one table. This is unnecessary and even contradictory to the design of the processor's datapath where instructions are dispatched to different queues and/or reservation stations prior to execution. Using this logic we split the Memo-Table into 3 distinct tables: Integer operations, FP operations, and Effective Address (EA) calculation. In order to further enhance reuse chances we mapped the FP Memo-Table using a mix of bits from the exponent and mantissa.

The reuse rates for three 512-entry tables and three 128-entry tables are shown in figure 5. In addition, the combined reuse rate is shown (number of total successes divided by number of total accesses). From a performance perspective it is clearly beneficial to split the global Memo-Table. The combined, Integer, and EA reuse rates all improve. However, the size of the Memo-Tables is now trebled. The reuse rate of three smaller Memo-Tables used to approximate one larger Memo-Table falls slightly lower than the monolithic approach. Nevertheless, the size of the three 512 Memo-Tables is less than half the size of a 32KB on-chip cache, no extra energy is being expended, and

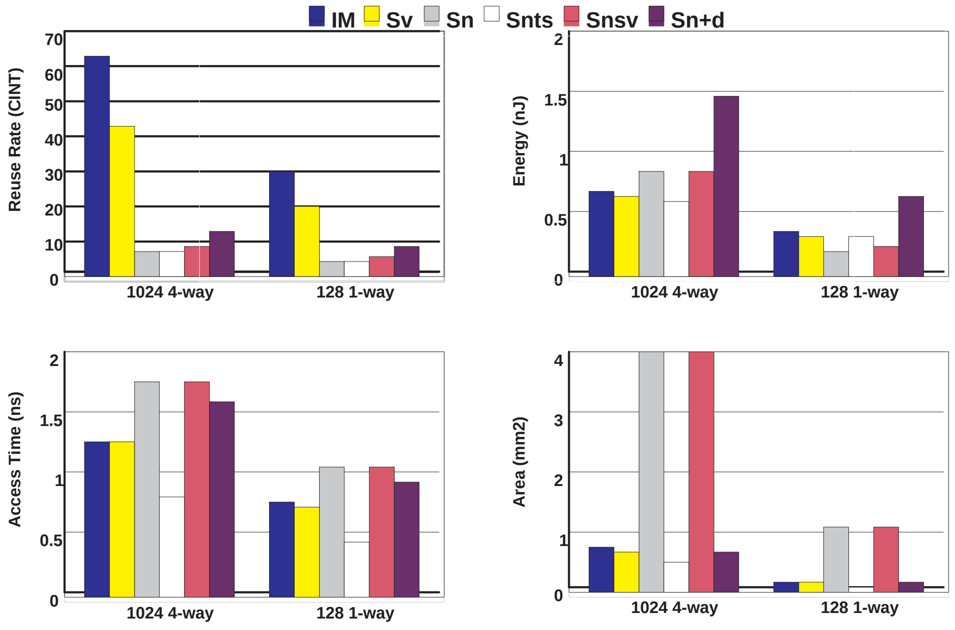


Figure 4. *Reuse rates (CINT), access time, energy, and table area of different memoization and reuse schemes.*

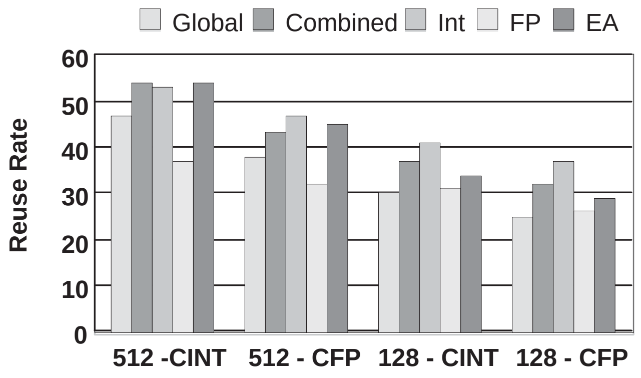


Figure 5. *Reuse rates when the global* MEMO-TABLE *is split into Integer, Float, and EA calculation tables.* MEMO-TABLE *sizes are 512 and 128 entries (direct-mapped).*

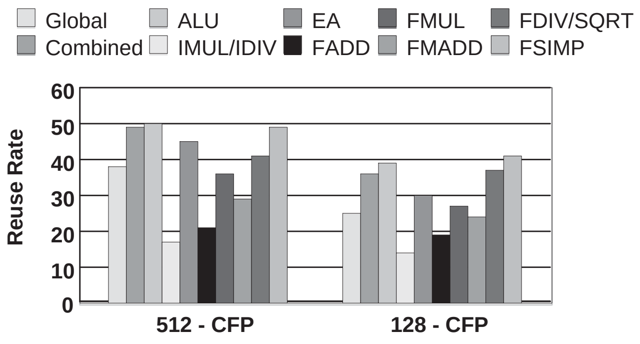


Figure 6. *Reuse rates when the global* MEMO-TABLE *is split into 8 distinct tables.* MEMO-TABLE *sizes are 512 and 128 entries (direct-mapped). Only CFP results are shown.*

the same number of access is being made.

This technique can be further fostered by splitting the Integer MEMO-TABLE into short and long latency instructions (IMUL and IDIV) and by splitting the FP MEMO-TABLE into FADD/FSUB, FMUL, FDIV/FSQRT, FMADD, and all other FP instructions. The rationale is to cluster operations with similar latencies into the same MEMO-TABLE.

Figure 6 shows the reuse rates for this 8-way split (the majority of the MEMO-TABLES store FP calculations so only the CFP results are shown). The results are mixed, the combined reuse rate is the same as for one 512-entry global MEMO-TABLE, and it surpasses three 128-entry MEMO-TABLES (figure 5). However, this is achieved with twice the area.

Reasons to implement such a configuration would be for chip locality: moving the MEMO-TABLE closer to the Functional Unit (FU) it serves can reduce wire delay. This also enables building MEMO-TABLES with different characteristics: A SQRT table needs only one operand while a FMADD table needs three.

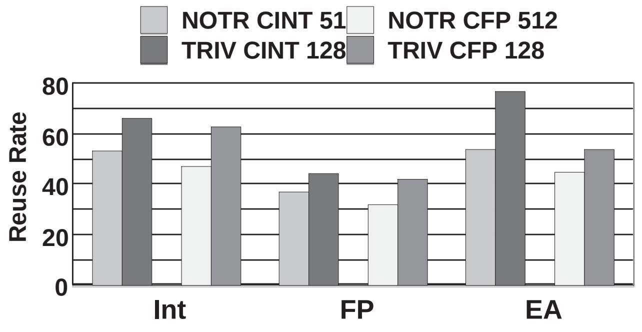


Figure 7. *Comparison of reuse rates of 512-entry* MEMO-TABLES *to 128-entry* MEMO-TABLES *with trivial operation detection.*

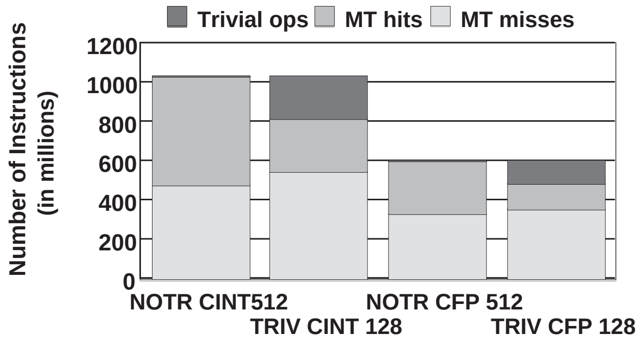


Figure 8. *Comparison of the number of successfully memoized instructions of 512-entry* MEMO-TABLES *and 128-entry* MEMO-TABLES *with trivial operation detection.*

4.2 Trivial Operations

Trivial operation detection has been used in the past as a memoization filter. Both Richardson [14] and Citron et al. [4] have used it in their works. Operations are defined as trivial when they can be matched against constant values (0,1,-1) and their results are straightforward from the operands ($a + 0 = a$, $a \times 1 = a$, $a/1 = a$, ...). The premise is that instead of storing these operations they will be detected by dedicated circuitry before or in parallel to a MEMO-TABLE lookup. In fact, the trivial operation detection can be viewed as an extra degree of associativity.

Figure 7 shows the reuse rate of the three major MEMO-TABLES (Integer, Float, EA) with and without trivial operation detection (in this case a trivial operation detection is considered a successful lookup). At first glance its results are impressive: for both suites all MEMO-TABLES display an enhanced reuse rate. Moreover, a second, closer, look at the column labels shows that we are comparing 512-entry MEMO-TABLES to 128-entry MEMO-TABLES. By using trivial operation detection we have quartered the size of the MEMO-TABLES, and improved the reuse rate. Figure 8 compares the raw number of accesses to MEMO-TABLES in both cases (average number of access per benchmark). When using the smaller tables the number of misses is larger: only non-trivial operations are memoized, but there are less accesses which saves energy (in addition to the smaller table sizes).

Nevertheless, trivial operation detection isn't free. Our calculations show that a 0,1,-1 detector for two 64-bit operands has an energy consumption of 0.00051nJ and an area of $0.0023mm^2$. These are inconsequential when compared to the 0.35nJ and $0.12mm^2$ of a direct-mapped, 128-entry MEMO-TABLE. However, it has an access time of 0.21ns. Accessing it in parallel to the MEMO-TABLE hides this latency yet burns energy. A sequential lookup (first trivial operation then MEMO-TABLE) results in an access time of 0.96ns (0.21 + 0.75) which is comparable to a 512-entry, direct-mapped MEMO-TABLE (0.94ns) and is just under two clock cycles for a 2GHz clock.

Yi and Lilja [22] suggest detecting trivial operations earlier in the pipeline by testing the first operand to arrive, this can solve the delay problem and should be considered in a detailed pipeline model.

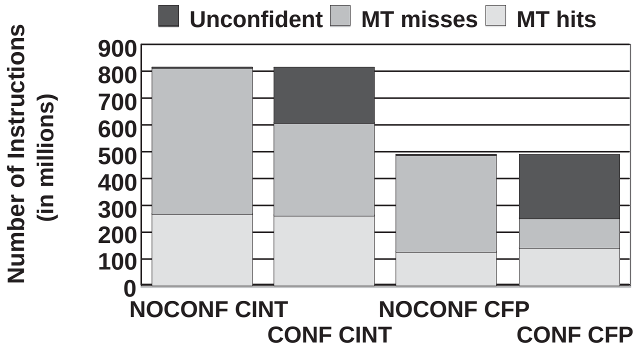


Figure 9. *Comparison of the number of successfully memoized instructions of 128-entry* MEMO-TABLES, *with and without confidence filtering. Trivial operations aren't counted.*

4.3 Confidence Counters

Although trivial operation detection reduces the number of MEMO-TABLE accesses the number of misses is still high. Some benchmarks just aren't amenable to memoization (`171.swim` and `171.mgrid` for instance). A well known technique for filtering these wasteful

MEMO-TABLE accesses is the use of *confidence counters*. They are usually used for branch [8] and value prediction [2].

In our model every instruction fetched is mapped to a Confidence Table (CT) which contains a n-bit saturating counter per entry. When a dynamic instruction instance hits (or is trivial) the counter is decreased, when it misses it is increased. After n misses a static instruction is marked as non-memoizable (although it can still be tested for triviality) and this data is passed on with it to the MEMO-TABLES. After a predetermined number of cycles the CT is flushed in order to give "mis-memoizing" instructions a second chance.

Figure 9 displays the numbers of MEMO-TABLE hits and misses when a 5-bit saturating confidence counter is used per entry (trivial ops aren't counted so we can directly compare MEMO-TABLE misses). The CT contains 1024 entries and is flushed every 131072 cycles and the MEMO-TABLES are direct-mapped with 128 entries and trivial op detection. The results are impressive, the CT manages to filter out many unsuccessful instructions while raising the number the memoized instructions. The "price" is a table that has an energy consumption of 0.011nJ, an area of $0.087mm^2$, and an access time of 0.24ns. The energy savings are huge, every aborted lookup saves $0.35 - 0.011 = 0.339nJ$ and the CT can be accessed way before memoization, hiding the CT's latency. The only complexity is linking the results of the MEMO-TABLE lookups to the CT, this has to be explored in a detailed datapath design.

The CT can be further reduced to 256 entries and 4 bits per counter, while retaining a better reuse rate than not using confidence counters. This shaves off several picoseconds from the access time ($0.24ns - 0.21ns = 30ps$).

5. "Look It Up" or "Do the Math"?

Finally after exploring the range of MEMO-TABLE attributes we must compare the memoization paradigm to the basic computations themselves. Table 7 lists the characteristics of several 64-bit functional units[3] and the reuse[4] (rr), trivial operation (tr), and confidence (cr) rates (instructions that haven't failed memoization) of the 128-entry MEMO-TABLES servicing them.

It is assumed that the MEMO-TABLE lookup is performed in parallel to computation and squashes it upon success. All instructions access the CT and memoized instructions update it as well. To measure the usefulness of memoization we defined two equations not unlike the Average Memory Acess Time (AMAT).

ACT ***A**verage **C**omputation **T**ime* The average time (in cycles) to compute an operation: $ACT = rrMT_t + trTO_t + [1 - (rr + tr)]FU_t$

ACE ***A**verage **C**omputation **E**nergy* The average energy (nJoules) expended when computing an operation. The MEMO-TABLE lookup and update energy are distinct measures ($MT_{lk.e}$, $MT_{up.e}$): $ACE = TO_e + (cr - tr)MT_{lk.e} + [cr - (rr + tr)](MT_{up.e} + FU_e) + (1 - cr)FU_e + (1 + cr)CT_e$

Figure 10 shows the ACT and ACE of the afore listed units compared to the latencies and energies without memoization (the CFP2000 suite is used). The ACT and ACE both show that it is counterproductive to memoize integer addition instructions, it incurs both performance and energy penalties. All other units display moderate performance gains and great energy gains. The gains are proportional to the units latency, the longer the latency the higher the performance potential. The problem is that long latency instructions usually have a low frequency of execution. This must be overcome in future work.

6. Observations and Conclusions

First we will list several key observations noticed during this study and then we will define a basis for future instruction memoization exploration.

- Reuse opportunities are rampant in SPEC CPU2000. 65-75% of all dynamic instructions have been executed with the same operands previously.
- Mapping the MEMO-TABLES using the operand values utilizes the full table and enables dynamic instruction instances of different static instructions to use each others results.
- The associativity of a MEMO-TABLE profoundly affects its access time, energy, and size yet hardly enhances its reuse rate. Direct-mapped is the way to go.
- Directing instructions to several MEMO-TABLES based on instruction classes is more cost effective than a single monolithic table.
- Trivial operation detection can quarter a MEMO-TABLE's size while increasing its reuse rate.
- Confidence counters filter out many unmemoizable instructions without reducing the number of successful MEMO-TABLE lookups.

[3]The data pertaining to the Power4 processor is labeled IBM Confidential at the time this study was compiled. We are working on obtaining publication clearance.The data is from other open sources.

[4]The number of succesful memoizations divided by the **total** number of instructions executed.

Func.	*Unit Features*			*CFP Rates*		
Unit	*latency*	*energy*	*area*	*confidence*	*reuse*	*trivial*
IADD	1	0.05	0.01	28.4	21.4	29.6
IMUL	7	0.34	0.10	4.4	3.8	9.2
IDIV	68	2.53	0.10	19.7	14.8	75.7
FADD	6	0.12	0.28	21.5	14.2	33.4
FMUL	6	0.34	0.69	14.4	6.4	32.9
FMADD	6	0.41	0.69	17.7	6.9	19.4
FDIV	30	1.75	0.72	12.9	8.5	38.8
MEMO-TABLE	2	0.35	0.12	128-entry, 1-way, update 0.17nJ		
TO detector	1	0.00051	0.0023			
CT	0	0.011	0.087	1024-entry, 5-bit counter		

Table 7. *Characteristics of 64-bit functional units and their adjacent* MEMO-TABLES .

- A comparison between direct computation to computation + memoization shows that it is useless to memoize single-cycle instructions.
- Memoization of long latency instructions shows a potential for performance improvement, and due to the use of confidence counters memoization results in energy savings for most units.

This study is the first step in proving that instruction memoization is a viable performance improving technique for modern microprocessors. We have shown that it is possible to obtain high reuse rates combined with low energy penalties and area overhead. Nonetheless, there is still plenty of work ahead, in light of our observations we must perform the following:

1. Choose pipeline stages in which the confidence and trivial operation tests will be performed, and then link their results to the MEMO-TABLES.
2. Supply operands to the MEMO-TABLES early enough to be useful.
3. Integrate dependent instructions into one memoization unit that can be reused together (similar to the S_{n+d} scheme and the Dynamic Computation Reuse scheme of Connors and Hwu [5]).
4. Test the effect of compiler scheduling on memoized instructions.

The bottom line: Fast clock rates are increasing the latency of many complex instructions. Instruction Memoization can reduce these latencies and reduce energy consumption to boot.

References

[1] M. Azam, P. Franzon, and W. Liu. Low Power Data Processing by Elimination of Redundant Computations. In *Proceedings of the 7th International Symposium on Low Power Electronics and Design*, pages 259–264, August 1997.

[2] M. Burtscher and B. G. Zorn. Prediction Outcome History-based Confidence Estimation for Load Value Prediction. *Journal of Instruction-Level Parallelism*, 1, May 1999.

[3] D. Citron and D. Feitelson. Revisiting instruction level reuse. In *Proceedings of the 1st Workshop on Duplicating, Deconstructing, and Debunking*, pages 62–70, May 2002.

[4] D. Citron, D. Feitelson, and L. Rudolph. Accelerating multi-media processing by implementing memoing in multiplication and division units. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operationg Systems*, pages 252–261, October 1998.

[5] D. Connors and W. mei Hwu. Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results. In *Proceedings of the 32nd International Symposium on Microarchitecture*, pages 158–169, November 1999.

[6] Intel Corporation. *Differences in Optimizing for the Pentium 4 Processor vs. the Pentium III Processor.*

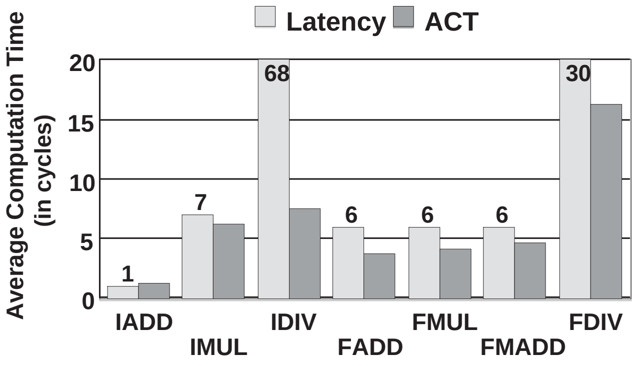
[7] Intel Corporation. *IA-32 Intel Architecture Optimization Reference Manual*, 2003.

[8] E. Jacobsen, E. Rotenberg, and J. Smith. Assigning Confidence to Conditional Branch Predictions. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 142–152, December 1996.

[9] R. Jain. *The Art of Computer Systems Performance Analysis.* Wiley Professional Computing, 1992.

[10] A. KleinOsowski and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *Computer Architecture Letters*, 1, June 2002.

[11] C. Molina, A. González, and J. Tubella. Dynamic Removal of Redundant Computations. In *Proceedings of the 1999 International Conference on Supercomputing*, pages 474–481, June 1999.



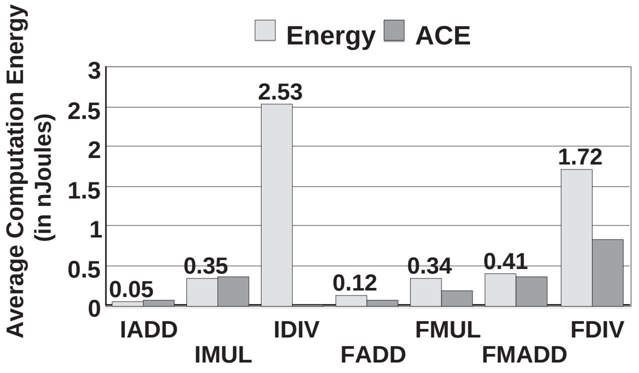


Figure 10. *Average Computation Time (ACT) and Average Computation Energy (ACE) compared to the latencies and energy consumption of seven functional units. The results are for CFP2000.*

[12] M. Moudgill, J. Wellman, and J. Moreno. Environment for PowerPC Microarchitecture Exploration. *IEEE Micro*, 19(3):15–25, May/June 1999.

[13] F. O'Connell and S. White. POWER3: the next generation of PowerPC processors. *IBM Journal of Research and Development*, 44(6):873–884, 2000.

[14] S. Richardson. Exploiting Trivial and Redundant Computation. In *Proceedings of the 11th Symposium on Computer Arithmetic*, pages 220–227, July 1993.

[15] P. Shivakumar and N. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. Technical report, Compaq: Western Research Laboratory, August 2001.

[16] A. Sodani and G. Sohi. Dynamic Instruction Reuse. In *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.

[17] http://www.sun.com/processors/UltraSPARC-II/details.html.

[18] Sun Microsystems. *UltraSPARC III User Manual*, 2.2 edition, May 2003.

[19] J. M. Tendler, J. S. Dodson, J. J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[20] S. Vetter et al. *The POWER4 Processor Introduction and Tuning Guide*. IBM, November 2001.

[21] J. Yi and D. Lilja. An Analysis of the Amount of Global Level Redundant Computation in the SPEC 95 and SPEC 2000 Benchmarks. In *Proceedings of the 4th Annual Workshop on Workload Characterization*, December 2001.

[22] J. Yi and D. Lilja. Improving Processor Performance by Simplifying and Bypassing Trivial Computations. In *Proceedings of the 20th International Conference on Computer Design*, September 2002.