Revisiting Instruction Level Reuse

Daniel Citron
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
citrond@us.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

The concept of instruction reuse states that execution of a dynamic instruction instance that has been executed in the past can be avoided. Data associated with each dynamic instruction (register values, register names, ...) is stored in on-chip dedicated lookup tables and before an instruction is executed it is looked up in the table. If a "match" occurs the result (which must be stored in the table as well) is extracted from the table, instruction execution is avoided, and the instruction is passed directly to the commit buffer.

This paper revisits the concept by repeating and widening the scope of past simulations, in particular the work of Sodani and Sohi on "Dynamic Instruction Reuse" and our previous work (Citron, Feitelson, and Rudolph) on "Instruction Memoization". The former targets all instructions for potential reuse while the latter focuses on long latency instruction only (multiplication and division).

This paper will answer the obvious question: "How can a single-cycle table lookup speedup a single-cycle execution?" In a nutshell the answer is simple: The lookup table is used as an additional functional unit. An instruction that would have met with a structural hazard is "executed" by the LUT.

We will show that adding additional ALUs nullifies a large percentage of the potential speedup. The flip side of this insight is that instructions with multiple-cycle execute latencies almost always benefit from instruction reuse techniques, the reduced execution latency always improves performance.

1 Introduction

The idea of *instruction reuse (IR)* at a granularity of single instructions has been introduced in the late 1990's by several concept papers but has never been thoroughly investigated in the literature. It has quickly been abandoned by researchers for its superset of *region reuse* [3] and the emerging field of *value prediction* [5, 4].

The premise of instruction reuse is simple: dynamic instances of a static instruction are executed with the same operand value(s) many times. If so why re-execute the instruction? Data associated with recently executed dynamic instruction instances is stored in on-chip dedicated lookup tables and before an instruction is executed it is looked up in the table. If a "match" occurs the result (stored in the table as well) is extracted from the table, instruction execution is avoided, and the instruction is passed directly to the commit buffer.

But can a reuse opportunity be detected and exploited faster than just executing the instruction? It has been our personal experience that most peer reviewers don't believe this possible or they refer to the concept papers as definitive and dismiss further examination of the technique. Thus, in this paper we will revisit the proposed techniques, rerun past simulations and run new ones, and finally strive to set the limits of this technique in the context of modern microprocessors.

We will gradually show that several of the assumptions made regarding lookup table access and invalidation, data availability, and reusing results were optimistic. After applying a set of limitations to the reuse techniques many reuse opportunities disappear due to data unavailability, and others are eliminated by adding more Functional Units (FUs).

The rest of this section will describe the processor environment in which IR is to be used. Section 2 will describe the suggested techniques. Section 3 will analyze several of the assumptions made by the reuse techniques. Section 4 will rerun past simulations and run new ones, and section 5 will summarize and discuss the results.

1.1 Processor Environment

The datapath modeled is the one used in Simplescalar [1], an architecturally detailed RISC instruction-level sim-

ulator based upon the MIPS ISA. It supports branch prediction, speculative execution, multiple issue, and Out-Of-Order execution: A maximum of n (issue width) instructions are executed per cycle. They are fetched in program order, possibly executed out-of-order, and finally committed in order. It has five basic stages for all instructions: Fetch, Decode, Issue, Execute, and Commit[1]. The issue rate, number of units, cache sizes, instruction latencies, and branch prediction techniques may vary within the above framework.

The different "flavors" of instruction reuse aim to reduce the occupancy of the reused instructions in the various stages of the pipeline, this is the source of their differences. In the following section we will describe the various proposals and identify the stage(s) they strive to avoid or reduce.

2 Instruction Reuse

This paper revisits the technique called *Instruction Reuse (IR)* introduced by Sodani and Sohi [8] and a subset of it coined *Instruction Memoization (IM)* introduced by Richardson [7] and further expanded on by Citron, Feitelson, and Rudolph [2] [2]. A third hybrid technique named *Instruction Level Reuse* was presented by Molina, Gonzalez, and Tubella [6].

IR focuses on reusing a dynamic instance of an instruction identified by its address, the *Program Counter (PC)* value. On the other hand IM focuses on computation reuse, the operands and operation of the instruction identify it and confirm or deny reuse. ILR proposes to match an instruction by both. In this section we will present the techniques and published results of each of the three techniques. We will start with the simplest IM, continue to IR, and conclude with ILR which integrates both approaches.

2.1 Instruction Memoization (IM)

The main theme in Instruction Memoization (IM) is to perform the lookup in parallel to the execution of th instruction. In the case where the lookup is successful (a "hit") the execution is terminated and the result read from the result cache ([7]) or Memo-Table ([2]), we will use the generic name LookUp Table (LUT) for all future references. If the lookup fails (a "miss") the execution completes and the LUT is updated with the result. Both [7] and [2] add trivial operation detection. Operations such as: $a*0, a*1, a/1, ...$ are detected and the result is either a constant ($a/0 = Inf$) or obtained from the operands ($a*1 = a$). Figure 1 shows a LUT adjacent to a division unit.

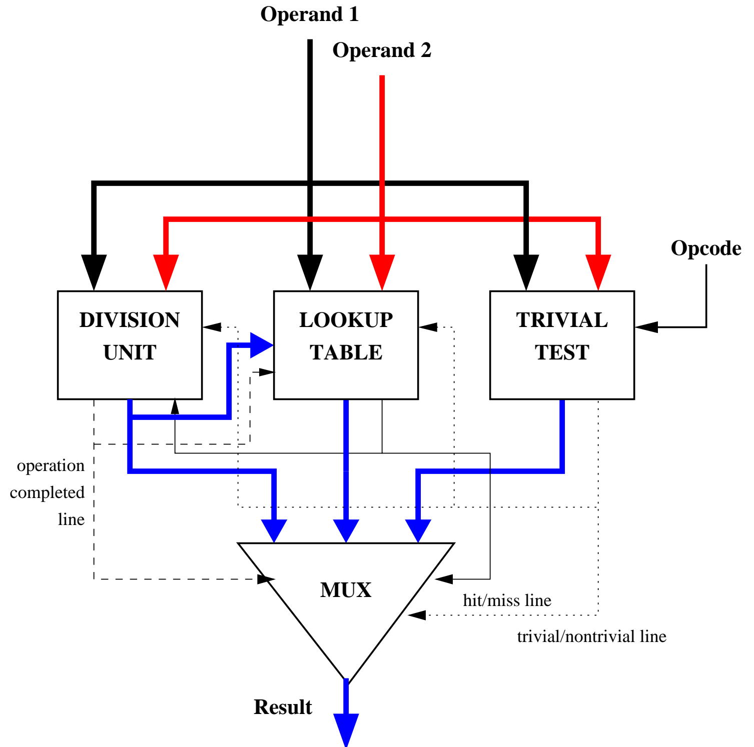

Figure 1. Layout of a LUT and a Trivial Test Unit (TTU) adjacent to a Division Unit.

Instructions are mapped into the LUT by taking a subset of their operand values bits and using them as an index into the LUT. Each entry contains the operands, operation, and result. The model of execution used implies that the lookup is performed in the Execute stage of the pipeline, when a *Functional Unit (FU)* and the operands are ready the instruction is issued to the FU. The authors of the memoization techniques limited themselves to instructions with long latencies, however if the LUT is designed to produce the result in a single cycle any instruction that has a latency of more than one cycle is a candidate for reuse. The basic datapath model can be seen in figure 2. The main limitation of this technique is that its scope is limited in instruction type reused.

As the authors of this paper we are in the position to disregard the results published as they were obtained using outdated simulation techniques inferior to Simplescalar. Results using Simplescalar were obtained after publication and are the basis of the results we will publish here.

[1] This *classic* design may seem outdated in modern processors with pipeline depths of 7 and more (much more in some cases) stages. However, those processors just split up the existing stages and functionally behave the same.

[2] The former two authors are collaborating on this paper.

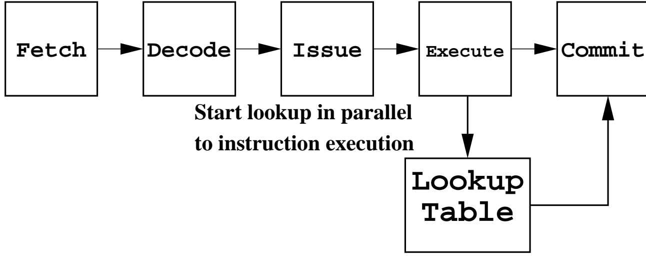

Figure 2. Datapath Integrated with Instruction Memoization (IM).

2.2 Instruction Reuse (IR)

Instruction Reuse (IR) strives to reuse all instructions, they are presented at decode time to a table called the *Reuse Buffer (RB)*. After execution the RB is updated with the current instruction's result. Three reuse schemes are presented:

S_v Each entry contains the PC, operand values, and result of an instruction. The PC is used to index the LUT, and if the 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. Hence, in practice a PC match for a valid entry suffices for reuse.

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 after their registers are written upon if their source instruction is in the RB.

In addition to the above fields all schemes contain an address and memory valid bit fields used by load instructions. Any store to memory must scan the RB and invalidate any entries with the same address.

The first scheme raises the question: "If the operands are known why perform a lookup?" Most instructions have an execution latency of one cycle. 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.

The third scheme 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 is aimed at avoiding the high rate of invalidation that is inherent in scheme S_n.

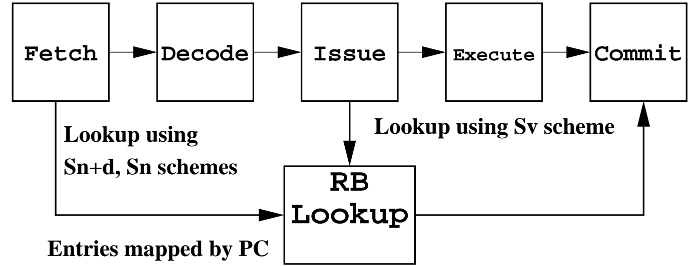

Figure 3. Datapath Integrated with Instruction Reuse (IR).

Figure 3 shows a datapath integrated with IR. The idea behind the technique is to "skip" the issue and execute stage and send the reusable instruction straight to the commit buffer. The technique was validated by simulation on Simplescalar [1] an instruction level MIPS based simulator. Twelve integer benchmarks were simulated, 5 from SPECINT 92 (*gcc, compress, eqntott, espresso, xlisp*), 5 from SPECINT95 (*go, m88ksim, vortex, ijpeg, perl*), *Yacr2*, and *Mpeg*. The microarchitecture simulated is shown in table 1.

The results published in [8] are summarized (harmonic means) in table 2 which show the percent of reuse and percent of speedup for all three techniques with three sizes of fully associative tables.

Scheme	*32 entries*		*128 entries*		*1024 entries*	
	Reuse	Spdp	Reuse	Spdp	Reuse	Spdp
S_v	2.2	3.7	8.5	7.0	25.7	14.9
S_n	3.1	4.3	9.5	5.7	12.5	7.5
S_{n+d}	2.6	3.8	10.4	7.2	20.6	10.5

Table 2. Percentage of reuse and percentage of speedup with table sizes of 32, 128, and 1024 entries. All tables are fully associative.

The table shows a correlation between percentage of reuse and speedup. However, the tables are assumed to be fully associative, Additional data supplied by the authors shows that for scheme S_v a 4-way associative table performs similarly to a fully associative table.

2.3 Instruction Level Reuse (ILR)

Different instructions that perform the same operation are defined as *quasi-common subexpressions*. These cannot

L1 Instruction Cache	16-KBs, 32-Byte blocks, direct-mapped
L1 Data Cache	16-KBs, 32-Byte blocks, 2-way associative
Memory Latencies (cycles)	L1 hit - 1, L1 miss - 6
Branch Prediction	2048-entry bimodal predictor
Registers	32 General Purpose, 32 Floating Point
Function Units	4 IALU, 1 IMULT 4 FADD unit, 1 FMULT, 2 MMU
Instruction Latencies & Throughputs	Integer multiplication: 3,1 Integer division: 20,19 All other integer instructions: 1,1 Floating point multiplication: 4,1 Floating point division: 12,12 Floating point Sqrt: 24,24 All other floating point instructions: 2,1
Pipeline attributes	4-instructions fetched, decoded, issued, and committed per cycle; 32 instructions in instruction queue, out-of-order execution, in-order retirement

Table 1. Microarchitecture of simulated processor.

be detected by the previous approach (IR). Molina, Gonzalez and Tubella [6] conceived an elaborate scheme that links different instructions (with the same opcode) that once produced the same result, their LUT (named the *Redundant Computation Buffer (RCB)*) is accessed by the PC and its entries contain links to other producing instructions. As with IR all instructions are stored[3]. They show that this method results in higher reuse rates and speedups over the S_v scheme of IR.

Nevertheless, they published that a hybrid scheme that uses both the PC and operand values as indices into the RCB yields better results than any of their other schemes. In the Fetch stage the PC is used to index the table and the operands are ready to be read and compared in the Issue stage, if the lookup is unsuccessful the operand values are used to index the RCB in the Issue stage and comparison takes place in the Execute stage. Figure 4 shows the proposed datapath. This scheme will be the representative ILR scheme.

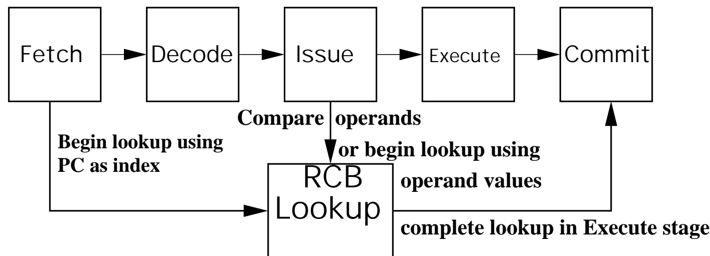

Figure 4. Datapath Integrated with Instruction Level Reuse (ILR).

[3]The loads are stored in a dedicated table.

The technique was tested using 7 benchmarks from the SPEC CINT95 suite (*compress, go, gcc, li, m88ksim, perl, vortex*) and 4 benchmarks from the SPEC CFP95 suite (*applu, mgrid, swim, turb3d*). The benchmarks were compiled for the Alpha version of Simplescalar and run on the same microarchitecture used by [8] (listed in table 1). Each benchmark was run for 125M instructions after its initialization stage[4].

The choice of microarchitecture and CINT95 benchmarks probably wasn't incidental as they match the microarchitecture and CINT95 benchmarks used by [8]. They compare their scheme to IM and the S_v scheme of IR. Each scheme uses the same amount of storage which translates into a different number of entries[5]: For both IR and ILR there are 256 instruction entries, each entry holds a history of the last 4 instances of the instruction. The IM LUT has 1024 entries in sets of 4 (256 sets). Table 3 summarizes the results published in [6] (reuse rates and speedups) and shows that while IM has the highest reuse rate it has the lowest speedup.

It clearly should have a high reuse rate, the LUT can contain many copies of frequently reused instructions and *quasi-common subexpressions* can be detected immediately and without any additional infrastructure. Both IR and ILR can only hold a limited number of entries per instruction, detecting *quasi-common subexpressions* is impossible using IR and is limited using ILR.

The reason why IM has a poor speedup, according to the authors, is due to the latency of a lookup. A lookup can be initiated only when the operands are ready (they are used to

[4]This number isn't disclosed and it isn't clear if it is per benchmark or a general number. The datasets themselves aren't disclosed either.

[5]The following numbers are deduced from the paper, clear cut numbers aren't given.

index an entry). This causes the reuse test (comparing the stored operands to the current operands) to be performed in the Execute stage, instructions with a single cycle of execute latency (*single-cycle instructions*) can't benefit from this scheme, only instructions with a multiple cycle execute latency (*multi-cycle instructions*) can. IR and ILR use the PC early in the pipeline to index the LUT and thus have the stored operands at hand when the current operands are made available, the reuse test can be performed in the Issue stage. A successful lookup "skips" the Execute stage. The next section will analyze the feasibility of this approach.

Scheme	*32KB storage*		*200KB storage*	
	Reuse	Spdp	Reuse	Spdp
IM	28	1.04	32	1.04
IR	18	1.06	26	1.08
ILR	24	1.07	32	1.09

Table 3. Percentage of reuse and speedup with table storage sizes of 32KB and 200KB.

3 Reuse Assumptions

In this section we will examine some of the assumptions made by the authors, and decide if they are reasonable. The valid assumptions (and some dubious ones) will be integrated into a simulation framework and tested for validity in the next section. The following is a list of the most critical assumptions:

Non Load Entry Invalidation The S_n and S_{n+d} schemes of IR assume that an entry is invalidated when any of its source registers are overwritten. However, uncommitted instructions in the pipeline write to *physical registers*, thus it is possible that an instruction is marked valid and reused when it should have been executed with different operand values. Furthermore, even if mapping between logical to physical registers is obtained in time, the invalidation would have to scan every entry in the table as it is indexed by the PC. It is hard to believe that this can be performed in a single cycle[6].

Conclusion: These two schemes aren't significantly better that the S_v scheme, and given the aforementioned problems we consider them impractical.

Load Entry Invalidation Every load would have to scan the table and invalidate entries. Using the effective address as an index just reduces the table to another level of cache in the hierarchy.

Conclusion: We will simulate load instructions assuming invalidations are practical.

Lookup Time It is assumed that N (the associativity or history depth of the LUT) sets of operands can be extracted and compared to in a single cycle. IM assumes that the index can be constructed in the same cycle as well. IR and ILR use the PC so the index is ready at the start of the cycle.

Conclusion: In order to meet these constraints we will assume unpretentious table sizes and associativity.

Lookup Stage IM performs lookup in the Execute stage in parallel to execution. IR talks about lookup in the Decode stage, this is impossible as operands aren't ready at that stage. ILR assumes lookup in the Issue stage. This is tricky as even in the Issue[7] stage operands might not be available. They might be being bypassed to the FU, enabling an instruction to begin execution in the next cycle without the actual operands being in existence.

Conclusion: IR and ILR performs lookup in the Issue stage, IM in the Execute stage. We will simulate operand availability in the Issue stage.

How Execution Time is Reduced Successful IM reduces the latency of long latency instructions to one cycle. IR assumes that a successful reuse test avoids the Execute stage, this is the main assumption we shall investigate. It seems like the LUT is used as an additional unit to "execute" instructions. It isn't clear if the reuse test is counted as an instruction issued nor the number of accesses to the LUT allowed per cycle.

Conclusion: In our simulations we will limit the number of ports to the LUT and count a LUT access as an issue.

4 Simulations

This section will first recreate the previous tests performed by [6] (section 4.1), limit some liberal assumptions (sections 4.2 and 4.3), and finally test different LUT layouts for the various techniques (section 4.4).

4.1 Basic Simulation

We will try to recreate the tests performed by [8] and [6] in the face of several limitations: We don't have access to all the benchmarks (SPEC92), we don't posses an Alpha

[6] A CAM design where each entry contains built in comparison circuitry might do the trick if it weren't limited in size and speed.

[7] Instructions occupy the Issue stage until their operands and executing FU are to be ready in the next cycle. The operands are routed to the FU in the next cycle.

Benchmarks	The 11 used by [6]: *compress, go, gcc, li, m88ksim, perl, vortex, applu, mgrid, swim, turb3d*.
Compiler	gcc 2.6.3 for the PISA version of Simplescalar.
Optimization	`-O3 -finline-functions -funroll-loops`.
Inputs	Reference inputs for each benchmark.
Inst. Count	The first billion instructions per benchmark.
Microarch.	The parameters used by [8] and [6] (listed in table 1).
Memory	A 512-MB, direct mapped, 18 cycle miss latency unified L2 cache. Page faults and context switches aren't simulated.
LUT size	IM - 1024 entries, 256 sets of 4; IR,ILR - 256 entries, each with a depth of 4
Reuse Stage	IM - Execute stage; IR, ILR - Issue stage
Misc.	Access to the LUTs aren't limited and a LUT lookup isn't counted as an instruction issue.

Table 4. Base simulation characteristics.

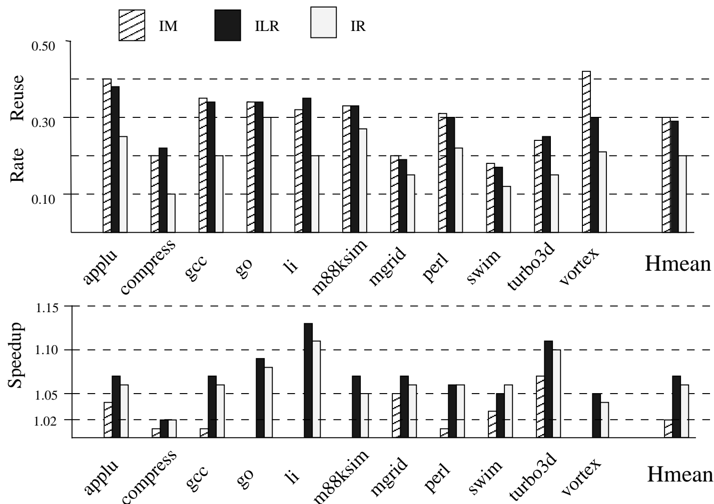

Figure 5. Reuse rates and speedups of basic case.

machine, we don't know the exact inputs and number of instructions used by [6], and it isn't fully clear what size tables and associativity [6] used. Other missing pieces of information are the size of the L2 cache, and the latency of a main memory access, TLB size and page fault latency, and the frequency of context switches. The characteristics of the base simulation are listed in table 4.

Figure 5 displays the reuse rates and speedups for the three techniques. ILR has a slightly lower reuse rate that IM due to several instructions occupying more than one entry, thus shrinking the table. IR has the lowest reuse rate due to its inflexibility in placing instructions and the inability to reuse operations created by different instructions. IM clearly performs the poorest, yielding no speedup for benchmarks with no multi-cycle instructions. There is no advantage in reusing single-cycle instructions in the Execute stage. The results are fairly consistent with the work of [6]. We should point out that IM is severely handicapped, all single-cycle instructions stored in the LUT contribute no speedup and replace multi-cycle instructions that may be reused effectively.

4.2 Operand Availability

In this section we will enforce the limitations on the assumptions we listed in section 3. Limiting the number of accesses to the LUT to two lookups and two updates per cycle hardly affects the results. Given the microarchitecture, an ILP (Instruction Level Parallelism) higher than two is hardly achieved. For the aforementioned reason counting a reuse test as an instruction issue doesn't change the speedups noticeably.

However, when testing for operand availability[8] in the Issue stage the results change dramatically: Less than 20% of the instructions have both operands available at the Issue stage (table 5 shows the per benchmark breakdown). In this case the reuse test is performed in the Execute stage, reducing the IR and ILR schemes to IM with a weakened reuse rate.

The results should have been obvious from the start, an instruction is issued to a FU if its dependencies (operand values) are to be ready during the next cycle and the executing FU is free the next cycle. Given the generous number

[8]Operand availability is defined as having both operands ready during any cycle preceding the first cycle of the Execute stage.

Benchmark	%	*Benchmark*	%
applu	12	mgrid	28
compress	26	perl	7
gcc	15	swim	21
go	18	turb3d	29
li	6	vortex	20
m88ksim	25	**Hmean**	**19**

Table 5. Percentage of instructions that have both operands available during the Issue stage.

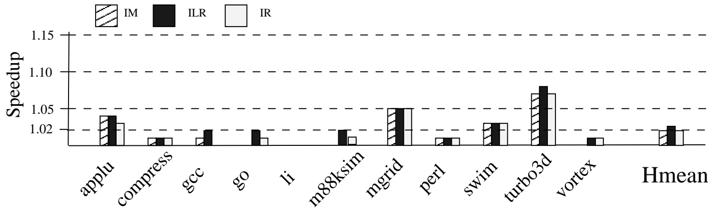

Figure 6. Speedups when operand availability during Issue stage is tested.

of IALUs (4) and MMUs (2) in the simulated microarchitecture the limiting factor is the number of data hazards not structural hazards. Figure 6 shows the new speedups. As expected hardly any benefit is gained by the IR and ILR over IM in this case.

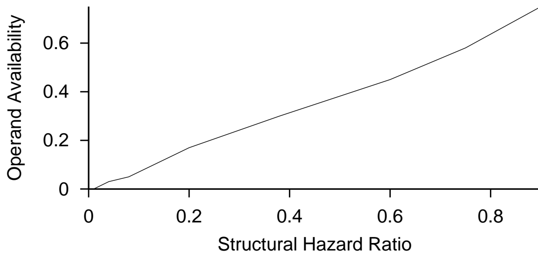

Figure 7. Operand availability as a factor of structural hazard ratio.

The availability of operands in the Issue stage is directly related to the number of FUs. Reducing their number results in more structural hazards per issue attempt, we shall call this the *structural hazard ratio*. Reducing the number of ALUs and FADD units to 2 and allowing just one memory access per cycle raises the rate of availability, the LUT is being used as an FU for instructions that encountered a structural hazard.

On a 16 issue machine, with a 256 instruction queue, perfect memory, and oracle branch prediction, we varied the number of FUs from 1 to 16 per type. Figure 7 shows the linear relation between structural hazard ratio and operand availability. These results might bring us to reconsider the S_n and S_{n+d} schemes of [8], nevertheless we believe them to be impractical due to the reason listed in section 3.

4.3 Lookup Time

A fast lookup time is critical to the success of any reuse technique. Can operand values stored in a LUT be retrieved and compared to the instruction's operands in a single cycle? IR and ILR use the PC as an index enabling a lookup to commence at the beginning of a cycle. IM (and ILR if a PC based lookup has missed) must first construct an index from the operands and then access the LUT.

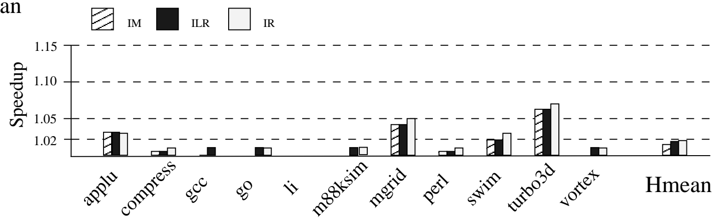

Figure 8. Speedup when IM and operand based ILR lookup is two cycles.

A 1024 entry LUT containing 25 bytes per entry (3 double words and an opcode) is smaller than a 32KB L1 cache available on most microprocessors, thus a single cycle lookup assumption is within reason. Nevertheless, we can't overlook the influence of a two cycle lookup time upon IM and ILR operand indexed lookups. Figure 8 shows the speedups when this is simulated. Both IM and ILR show diminished speedups, with IM suffering the greatest performance loss. A two-cycle lookup time limits IM reuse to multiplication and division instructions only, further curtailing its scope.

4.4 Unleashing the Reuse Techniques

The preceding section showed the futility of blindly trying to reuse all instructions. This section will evaluate the true potential behind reuse by differentiating between single-cycle and multi-cycle reuse. As mentioned in section 4.1 the IM reuse rate is limited by single-cycle instructions that pollute the LUT. Their reuse doesn't improve speedup and they replace reusable multi-cycle instructions.

Each of the three schemes will be enhanced by splitting the LUT into several smaller LUTs for Floating Point (FP)

instructions, loads, multi-cycle integer instructions (multiplication and division) and all other single-cycle instructions (we will forgo this last LUT for IM, which can't benefit from it under any circumstances). Each table will contain 256 entries, securing a single-cycle lookup. We will further enhance ILR by having the multi-cycle tables be indexed by operand values and the single-cycle tables indexed by PC.

Figure 9 shows the reuse rates and speedups for this configuration. Operand availability in the Issue stage is tested and enforced. All schemes benefit from an increased reuse rate, particularly IM which now stores only instructions that may reduce execution time. It even outperforms IR even though it has a 0% reuse rate for the four applications that don't execute multi-cycle instructions (*go, li, m88ksim, vortex*). The speedups show that ILR has a slight edge over IM due to the little reuse obtained from single-cycle instructions.

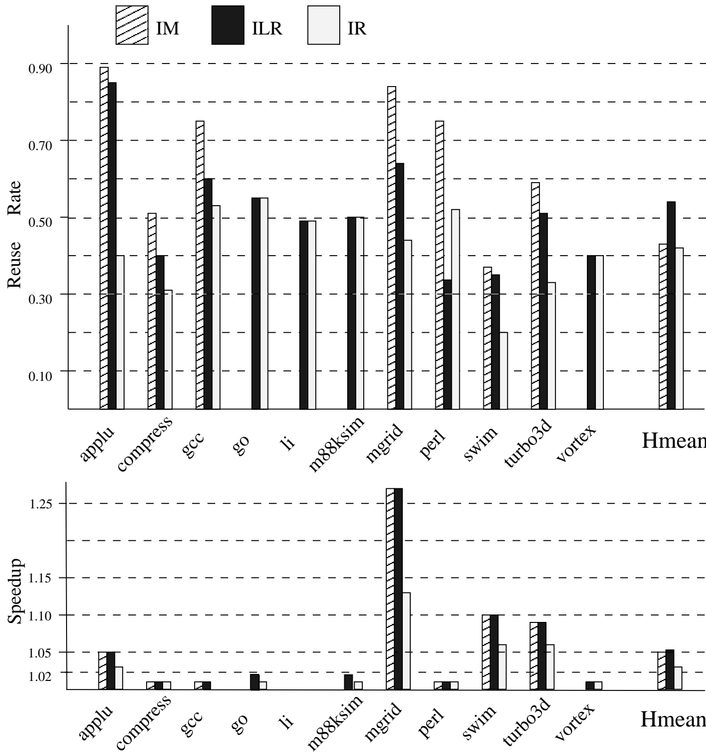

Figure 9. Reuse rates and speedups when instructions are stored in different LUTs.

4.4.1 Full SPEC95 Simulations

The benchmarks used in our experiments were selected in order to reproduce the simulations of [6], who in turn tried to reproduce the simulations of [8]. However, these applications don't represent a full suite. Table 6 summarizes the reuse rates and speedups of all three techniques when the complete SPEC95 suite was simulated using the features listed in section 4.4.

Scheme	*CINT95*		*CFP95*		*CPU95*	
	Reuse	Spdp	Reuse	Spdp	Reuse	Spdp
IM	60	1.00	47	1.10	51	1.05
ILR	55	1.01	49	1.11	54	1.06
IM	38	1.01	28	1.04	34	1.03

Table 6. Percentage of reuse and speedup for SPEC CINT95, CFP95, and CPU95 (INT and FP combined).

The results clearly show that FP intensive applications can benefit from instruction reuse as opposed to integer applications which hardly display any speedup. The high reuse rate of IM for CINT95 is due to the benchmarks *compress, gcc, perl*, the rest of them hardly execute any multi-cycle instructions.

5 Summary and Conclusions

This paper revisits the technique of reusing instruction results proposed by several authors (including us) in the late 1990's. The various techniques proposed (Instruction Memoization (IM) by [2], Instruction Reuse (IR) by [8], and Instruction Level Reuse (ILR) by [6]) are reviewed in section 2 and analyzed in section 3. This detail is necessary in order to understand the various schemes and understand their strengths and limitations.

Section 4 displays the results of four tests only: (i) a reenactment of the original simulation performed by [6] which compared all three techniques; (ii) The same test with limiting factors, the main one being testing for operand availability in the Issue stage;(iii) Assuming a lookup time of two cycles for instructions that are mapped using their operands; (iv) A test of an enhanced configuration where single-cycle and multi-cycle instructions are stored separately; These tests lead to the following conclusions:

Reenactment It is very hard to reenact a simulation based on a published paper. In our case we were limited by the unavailability of an Alpha machine and by several key parameters that were missing such as benchmark inputs and lengths and a full memory hierarchy description.

Reuse Rates Indexing a LUT using operand values is far superior to using the PC. Instances of the same instruction can be distributed equally through the LUT and different instructions (with the same opcode) can supply results for one other (*quasi-common subexpressions*). In this the IM technique is superior.

Lookup Stage A lookup can be performed only when both the instruction's operands are ready. To assume that they will be readable sometime during the Issue stage is mostly a false assumption. Only in 19% is this true, the operands are 81% of the time ready only during the first cycle of the Execute stage. Adding more FUs reduces this percentage even more. It is our conclusion that single-cycle reuse is useless, why try to reuse an instruction with a varying degree of success when adding a FU can execute it with a 100% success rate?

Lookup Time A single-cycle lookup time in the Execute stage is crucial for the success of reuse techniques. A longer lookup time during this critical path limits reuse to multiplication and division instructions

Different Schemes The final test shows that when splitting the instructions into several LUTs according to latency and type (integer, FP), all three schemes are similar. There is no virtually no difference between IM to ILR, and IR performs poorly due to its inferior indexing scheme.

FP Memoization When implementing IM on FP intensive benchmarks (CFP95) an average speedup of 1.10 is achieved. We believe that integrating IM in the FP unit of a processor can greatly accelerate FP intensive application.

Power Considerations At the time of the original papers publication power consumption wasn't an issue. Today it is clear that any innovation must be evaluated in this context, doubly so a technique such as reuse which can result in many LUT accesses and not much improvement, in cases of poor value locality.

In a nutshell we must conclude that reuse at the instruction level is limited to instructions with multiple cycle latencies in the Execute stage and isn't viable as a general, "across the board", enhancement.

References

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

[2] 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 Operating Systems*, pp. 252–261, October 1998.

[3] 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.

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

[5] 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 Operating Systems*, pp. 138–147, October 1996.

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

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

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