Contents

0-8493-0052-5/00/$0.00+$.50

List of Tables

List of Figures

Part I

This is a Part

Chapter 1

Instruction Precomputation: Dynamically Removing Redundant Computations Using Profiling

Joshua J. Yi[1], Resit Sendag[2], and David J. Lilja[3]

[1]*Freescale Semiconductor Inc.*, [2]*University of Rhode Island*, [3]*University of Minnesota at Twin Cities*

Abstract As a program executes, some computations are performed over and over again. These redundant computations increase the program's execution time since they could require multiple cycles to execute and because they consume limited processor resources. To minimize the performance degradation that redundant computations have on the processor, Instruction Precomputation hardware can be used to dynamically remove these redundant computations. Instruction Precomputation profiles the program to determine the highest frequency redundant computations. These computations then are loaded into the Precomputation Table before the program executes. During program execution, the processor accesses the Precomputation Table to determine whether or not an instruction is a redundant computation; instructions that are redundant receive their output value from the Precomputation Table and are removed from the pipeline. The key difference between Instruction Precomputation and Value Reuse – another microarchitectural technique that dynamically removes redundant computations – is that Instruction Precomputation does not dynamically update the Precomputation Table with the most recent redundant computations since it already contains those that occur with the highest frequency. For a 2048-entry Precomputation Table, dynamically removing redundant computations yields an average speedup of 10.53%, while, by comparison, a 2048-entry Value Reuse Table produces an

average speedup of 7.43%.

1.1 Introduction

During the course of a program's execution, a processor executes many redundant computations. A redundant computation is a computation that the processor already performed earlier in the program. Since the actual input operand values may be unknown at compile time – possibly because they depend on the inputs to the program – an optimizing compiler may not be able to remove these redundant computations during the compilation process.

Redundant computations degrade the processor's performance in two ways. First, executing instructions that are redundant computations consumes valuable processor resources, such as functional units, issue slots, and bus bandwidth, which could have been used to execute other instructions. Second, redundant computations that are on the program's critical path can increase the programs overall execution time.

Value Reuse [1] is a microarchitectural technique that improves the processors performance by dynamically removing redundant computations from the pipeline. During the programs execution, the Value Reuse hardware compares the opcode and input operand values of the current instruction against the opcodes and input operand values of all recently executed instructions, which are stored in the Value Reuse Table (VRT). If there is match between the opcodes and input operand values, then the current instruction is a redundant computation and, instead of continuing its execution, the current instruction gets its output value from the result stored in the VRT. On the other hand, if the current instruction's opcode and input operand values do not match those found in the VRT, then the instruction is not a recent redundant computation and it executes normally. After the instruction finishes execution, the Value Reuse hardware stores the opcode, input operand values, and output value for that instruction into the VRT.

While Value Reuse can improve the processors performance, it does not necessarily target the redundant computations that have the most effect on the programs execution time. This shortcoming stems from the fact that the VRT is finite in size. Since the processor constantly updates the VRT, a redundant computation could be stored in the VRT, evicted, re-executed, and stored again. As a result, the VRT could hold redundant computations that have a very low frequency of execution, thus decreasing the effectiveness of this mechanism.

To address this frequency of execution issue, Instruction Precomputation [2] uses profiling to determine the redundant computations with the highest

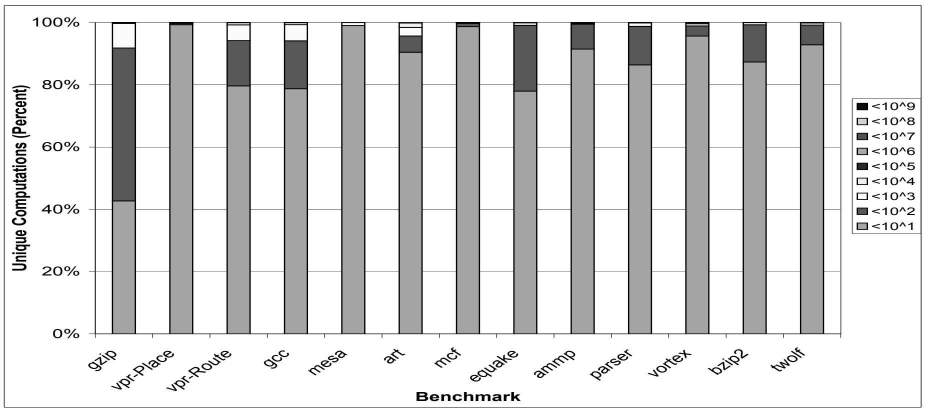

FIGURE 1.1: Frequency distribution of unique computations

frequencies of execution. The opcodes and input operands for these redundant computations then are loaded into the Precomputation Table (PT) before the program executes. During program execution, the PT functions like a VRT, but with two key differences: 1) The PT stores only the highest frequency redundant computations, and 2) The PT does not replace or update any entries. As a result, this approach selectively targets those redundant computations that have the largest impact on the program's performance.

1.2 Redundant Computations

Since Instruction Precomputation does not update the PT with latest redundant computations, the redundant computations that are loaded in to the PT must account for a sufficiently large percentage of the program's total dynamic instruction count or else Instruction Precomputation will not significantly improve the processor's performance. From another point-of-view, if the number of redundant computations needed by the Instruction Precomputation mechanism to improve the processor's performance significantly results in an excessively large PT, then Instruction Precomputation is not a feasible idea. Therefore, the key question is: Is there a small set of high frequency redundant computations that account for a large percentage of the total number of instructions executed?

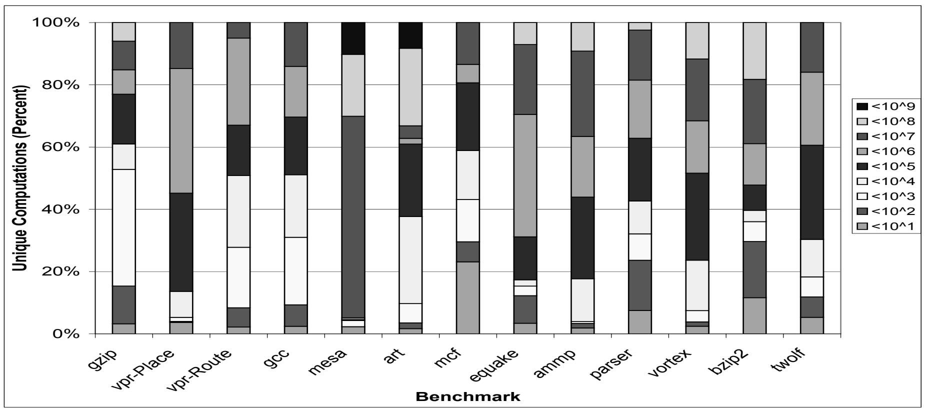

FIGURE 1.2: Percentage of dynamic instructions due to the unique computations in each frequency range

To determine the frequency of redundant computations, the opcodes and input operands (hereafter referred to together as a "unique computation") for all instructions were stored. Accordingly, any unique computation that has a frequency of execution greater than one is a redundant computation, since it was executed more than once in the program. After profiling each benchmark, the unique computations were sorted by their frequency of execution. Figure 1.1 shows the frequency distribution of the unique computations for selected benchmarks from the SPEC CPU 2000 benchmark suite (described in Section 1.4), using logarithmic frequency ranges. Logarithmic ranges were used since they produced the most compact results without affecting the content.

In Figure 1.1, the height of each bar corresponds to the percentage of unique computations that have a frequency of execution within that frequency range. With the exception of *gzip*, almost 80% of all unique computations have execution frequencies less than 10, while over 90% of all unique computations have execution frequencies less than 100. This result shows that most unique computations occur relatively infrequently in a program. Consequently, the performance benefit of caching most of the unique computations is relatively low since they only execute a few times.

A unique computation's frequency of execution corresponds to the number of dynamic instructions that that unique computation represents. Figure 1.2 shows the percentage of the total number of dynamic instructions that are accounted for by the unique computations in each frequency range for the different benchmark programs. For each frequency range, comparing the heights of the bars in Figures 1.1 and 1.2 shows the relationship between the

TABLE 1.1: Characteristics of the 2048 Highest Frequency Unique Computations

Benchmark	% of Total Unique Computations	% of Total Instructions
gzip	0.024	14.68
vpr-Place	0.029	40.57
vpr-Route	0.162	23.44
gcc	0.032	26.25
mesa	0.010	44.49
Art	0.010	20.24
mcf	0.005	19.04
equake	0.017	37.87
ammp	0.079	23.93
parser	0.010	22.86
vortex	0.033	25.24
bzip2	0.002	26.83
twolf	0.026	23.54

unique computations and dynamic instructions. For example, in *vpr-Place*, only 3.66% of all dynamic instructions produce more than 99% of all unique computations.

More than 90% of the unique computations account for only 2.29% (*mesa*) to 29.66% (*bzip2*) of the total number of instructions. In other words, a very large percentage of the unique computations cover a disproportionately small percentage of the total number of instructions. On the other hand, a small set of unique computations accounts for a disproportionately large number of instructions. Therefore, simply storing the highest frequency unique computations will cover a significant percentage of the programs instructions.

Table 1.1 shows the percentage of dynamic instructions that are represented by the highest frequency 2048 unique computations. The top 2048 unique computations, which account for a very small percentage of the total unique computations (0.002% - 0.162%), represent a significant percentage of the total dynamic instructions (14.68% - 44.49%). The conclusion from the results in this section is that a significant percentage of a program's dynamic instructions are due to a small number of high-frequency unique computations.

1.3 Instruction Precomputation

As described above, Instruction Precomputation consists of two main steps: static profiling and dynamic removal of redundant computations. In the profiling step, the compiler executes the program with a representative input set to determine the unique computations that have the highest frequencies

TABLE 1.2: Number of Unique Computations that are Present in Two Sets of the 2048 Highest Frequency Unique Computations from Two Different Input Sets

Benchmark	In Common	Percentage
gzip	2028	99.02
vpr-Place	527	25.73
vpr-Route	1228	59.96
gcc	1951	95.26
mesa	589	28.76
art	1615	78.86
mcf	1675	81.79
equake	1816	88.67
ammp	1862	90.92
parser	1309	63.92
vortex	1298	63.38
bzip2	1198	58.50
twolf	397	19.38

of execution. Alternatively, instead of selecting unique computations based solely on their frequency of execution, the compiler could also factor in the expected execution latency to select unique computations with the highest frequency-latency products (FLP). The FLP is simply the unique computation's frequency of execution multiplied by its execution latency.

Since Instruction Precomputation is based on static profiling using only a single input set, the key question is: Do two input sets have a significant number of high frequency (or FLP) unique computations in common? If the highest frequency computations are simply an artifact of the specific input set that was used, then Instruction Precomputation cannot be used to improve the performance of the processor since the unique computations are not a function of the program. To answer this question, Table 1.2 shows the number of unique computations that are common between the top 2048 highest frequency unique computations for two different input sets. The second column shows the number of unique computations that are present in both sets while the third column shows that number as a percentage of the total number of unique computations (2048) in either set.

The results in Table 1.2 show that with the exceptions of *vpr-Place*, *mesa*, and *twolf*, at least 50% of unique computations in one set are present in the other set. For *gzip*, *gcc*, and *ammp*, over 90% of the unique computations in one set are present in the other. The key conclusion from Table 1.2 is that for most benchmarks, a significant percentage of the same unique computations are present across multiple input sets. Consequently, the conclusion is that the highest frequency unique computations are primarily a function of the

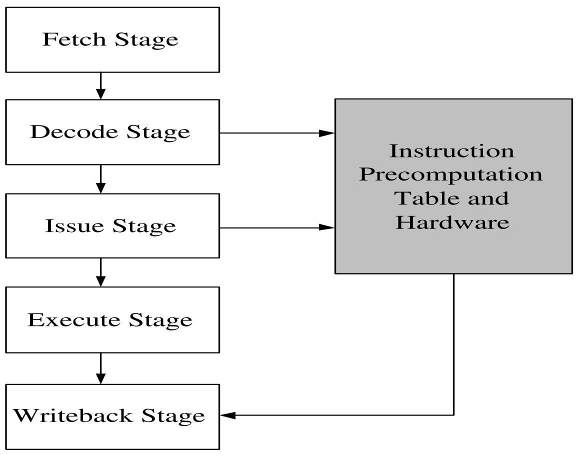

FIGURE 1.3: Operation of a Superscalar Pipeline with Instruction Precomputation

benchmark and less a function of the specific input set. Therefore, Instruction Precomputation can use static profiling to determine the highest frequency unique computations for a program.

After the compiler determines the set of the highest frequency unique computations, they are compiled into the program binary. Therefore, each set is unique only to that program.

The second step for Instruction Precomputation is the removal of redundant computations at run-time. Before the program begins execution, the processor initializes the Precomputation Table with the unique computations that were found in the profiling step. Then, as the program executes, for each instruction, the processor accesses the PT to see if the opcode and input operands match an entry in the PT. If a match is found, then the PT forwards the result of that computation to the instruction and the instruction needs only to commit. If a match is not found, then the instruction continues through the pipeline and executes as normal. Figure 1.3 shows how the PT is integrated into the processor's pipeline.

During the decode and issue stages, the opcode and input operands for each dynamic instruction are sent to the PT, when available. The Instruction Precomputation hardware then determines if there is a match between the current opcode and input operands with the unique computations in the PT. If a match is found, the Instruction Precomputation hardware sends the output value for that instruction to the writeback stage, which commits that value when the instruction is retired.

Finally, unlike Value Reuse, Instruction Precomputation never updates the PT. Rather, the PT is initialized just before the program starts executing. When a unique computation is not found in the PT, the processor executes the instruction as normal. While this approach eliminates the need for hardware to dynamically update the PT – thus decreasing the complexity of and access time to the PT – it also means that the PT cannot be updated with the most current high-frequency unique computations.

1.3.1 A Comparison of Instruction Precomputation and Value Reuse

Overall, Instruction Precomputation and Value Reuse use similar approaches to reduce execution time. Both methods dynamically remove instructions that are redundant computations from the pipeline after forwarding the correct output value from the redundant computation table to that instruction. Both methods define a redundant computation to be one that is currently in the PT or VRT.

The key difference between these two approaches is how a redundant computation gets into the PT or VRT. In Instruction Precomputation, compiler profiling determines the set of redundant computations that are to be put into the PT. Since it is likely that, for that particular input set, the highest frequency unique computations are already in the PT, there is no need for dynamic replacement. In Value Reuse, in contrast, if a unique computation is not found in the VRT, then it is added to the VRT, even if it replaces a higher frequency redundant computation.

The VRT may have a lower access time than the PT, however. Instead of comparing the current instructions opcode and input operands against every unique computation in the VRT, using the current instruction's PC as an index into the VRT can reduce the VRT access time by quickly selecting the matching table entry (although the input operands and the tag still need to be compared). As a result, not only does this approach require fewer comparisons than Instruction Precomputation, it also removes the need to compare opcodes since there can only be one opcode per PC.

1.4 Simulation Methodology

To evaluate the performance potential of Instruction Precomputation compared to conventional Value Reuse, sim-outorder, which is the superscalar processor simulator from the SimpleScalar [3] tool suite, was used. The base processor was configured to be a 4-way issue machine. Table 1.3 shows the values of the key processor and memory parameters that were used for the performance evaluations of both techniques. These parameter values are similar to those found in the Alpha 21264 [4] and the MIPS R10000 [5].

Twelve benchmarks were selected from the SPEC CPU 2000 [6] benchmark

TABLE 1.3: Key Parameters for the Performance Evaluation of Instruction Precomputation

Parameter	Value
Branch Predictor	Combined
Branch History Table Entries	8192
Return Address Stack (RAS) Entries	64
Branch Misprediction Penalty	3 Cycles
Instruction Fetch Queue Entries	32
Reorder Buffer Entries	64
Number of Integer ALUs	2
Number of FP ALUs	2
Number of Integer Multipliers	1
Number of FP Multipliers	1
Load-Store Queue Entries	32
Number of Memory Ports	2
L1 D-Cache Size	32 KB
L1 D-Cache Associativity	2-Way
L1 D-Cache Block Size	32 Bytes
L1 D-Cache Latency	1 Cycle
L1 I-Cache Size	32 KB
L1 I-Cache Associativity	2-Way
L1 I-Cache Block Size	32 Bytes
L1 I-Cache Latency	1 Cycle
L2 Cache Size	256 KB
L2 Cache Associativity	4-Way
L2 Cache Block Size	64 Bytes
L2 Cache Latency	12 Cycles
Memory Latency, First Block	60 cycles
Memory Latency, Following Block	5 Cycles
Memory Bandwidth (Bytes/Cycle)	32
TLB Latency	30 Cycles

suite. These benchmarks were chosen because they were the only ones that had MinneSPEC [7] reduced input sets available at the time. MinneSPEC small, medium, and large reduced input sets and SPEC test and train reduced input sets were used to control the execution time. Benchmarks that use a reduced input set exhibit behavior similar to when the benchmark is executed using the reference input [7]. Since reduced input sets were used, the benchmarks were run to completion without any fast-forwarding. All of the benchmarks that were used in this work were compiled at optimization level -O3 using the SimpleScalar version of *gcc* (version 2.63) for the PISA instruction set, which is a MIPS-like ISA. Table 1.4 shows the benchmarks and the specific input sets that were used. The input set in the second and third columns is arbitrarily named "Input Set A"while the other input set is likewise named "Input Set B".

TABLE 1.4: Selected SPEC CPU 2000 Benchmarks and Input Sets

Benchmark	Input Set A Name	Instr. (M)	Input Set B Name	Instr. (M)
gzip	Small (log)	526.4	Medium (log)	531.4
vpr-Place	Medium	216.9	Small	17.9
vpr-Route	Medium	93.7	Small	5.7
gcc	Medium	451.2	Test	1638.4
mesa	Large	1220.6	Test	3239.6
art	Large	2233.6	Test	4763.4
mcf	Medium	174.7	Small	117.0
equake	Large	715.9	Test	1461.9
ammp	Medium	244.9	Small	68.1
parser	Medium	459.3	Small	215.6
vortex	Medium	380.3	Large	1050.0
bzip2	Large (source)	1553.4	Test	8929.1
twolf	Test	214.6	Large	764.9

1.5 Performance Results for Instruction Precomputation

1.5.1 Upper-Bound - Profile A, Run A

The first set of results presents the upper-bound performance results of Instruction Precomputation; the upper-bound occurs when the same input set is used for both profiling and performance simulation. To determine this upper bound, the benchmark was first profiled with Input Set A to find the highest frequency unique computations. Then the performance of Instruction Precomputation was evaluated with that benchmark by using Input Set A again. The shorthand notation for this experiment is Profile A, Run A.

Figure 1.4 shows the speedup due to Instruction Precomputation for 16 to 2048 PT entries. For comparison, the speedup due to using a L1 D-Cache that is twice as large as the L1 D-Cache of the base processor is included. This result, labeled "Big Cache", represents the alternative of using the chip area for something other than the PT. The total capacity of this cache is 64 KB. These results show that the average upper-bound speedup due to using a 16-entry PT is 4.82% for these 13 benchmarks (counting *vpr-Place* and *vpr-Route* separately) while the average speedup for a 2048-entry PT is 10.87%. Across all benchmarks, the range of speedups for a 2048-entry PT is 0.69% (*art*) to 45.05% (*mesa*). The average speedup results demonstrate that the upper-bound performance improvement due to Instruction Precomputation is fairly good for all table sizes.

Instruction Precomputation is very effective in decreasing the execution time for two benchmarks, *mesa* and *equake*, since the Top 2048 unique computations account for a very large percentage of the total dynamic instructions for these two benchmarks. Table 1.1 shows that the 2048 highest frequency unique computations account for 44.49% and 37.87% of the total dynamic instruction count in *mesa* and equake, respectively.

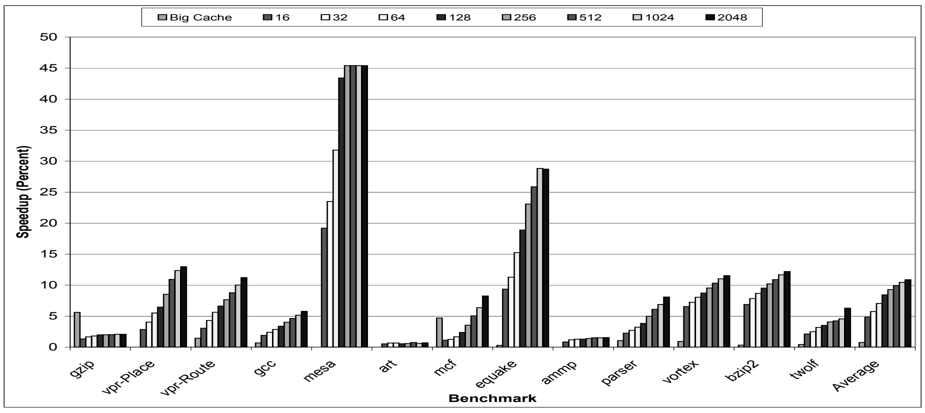

FIGURE 1.4: Instruction Precomputation speedup; profile input Set A, run input set A

The average speedup due to using the larger L1 D-Cache is only 0.74%. By comparison, the upper-bound speedup when using a 2048-entry Precomputation Table averages 10.87%. Therefore, using approximately the same chip area for a Precomputation Table instead of for a larger L1 D-Cache appears to be a much better use of the available chip area.

1.5.2 Different Input Sets - Profile B, Run A

Since it is unlikely that the input set that is used to profile the benchmark will also be the same input set that will be used to run the benchmark, this section evaluates the situation when two different input sets are used for profiling and for execution. This group of results represents the typical case in which Instruction Precomputation is most likely to be used. Figure 1.5 shows the speedup due to Instruction Precomputation when using Input Set B for profiling and Input Set A for execution, i.e. Profile B, Run A.

As shown in Figure 1.5, the average speedup ranges from 4.47% for a 16-entry PT to 10.53% for a 2048-entry PT. By comparison, for the same PT sizes, the speedup for Profile A, Run A ranges from 4.82% to 10.87% for the same table sizes. These results show that the average speedups for Profile B, Run A are very close to the upper bound speedups for the endpoint PT sizes. In addition, with the exception of *mesa*, the speedups for each benchmark are similar.

Although the speedups for a 32-entry, 64-entry, and 128-entry PT are significantly lower than the upper-bound for *mesa*, those differences completely

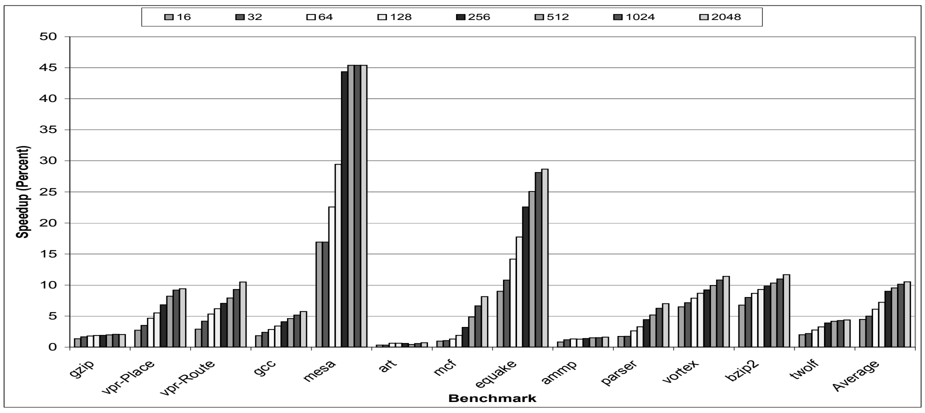

FIGURE 1.5: Instruction Precomputation speedup; profile input set B, run input set A

disappear for PT sizes larger than 256 entries. The reason for the speedup differences and their subsequent disappearance is that the highest frequency unique computations for Input Set B do not have as high a frequency of execution for Input Set A. Therefore, until the highest frequency unique computations for Input Set A are included in the PT (for PT sizes larger than 128), the speedup for Profile B, Run A for *mesa* will be lower than the upper-bound speedup.

In conclusion, the key result of this sub-section is that the performance of Instruction Precomputation is generally not affected by the specific input set since the Profile B, Run A speedups are very close to the upper-bound speedups. This conclusion is not particularly surprising since Table 1.2 showed that a large number of the highest frequency unique computations are common across multiple input sets.

1.5.3 Combination of Input Sets - Profile AB, Run A

While the performance of Instruction Precomputation is generally not affected by the specific input set, the speedup when different input sets are used for profiling and execution affects the speedup for at least one benchmark (*mesa*). Although the difference in speedups disappeared for PT sizes larger than 256 entries, sufficient chip area may not exist to allow for a larger table. One potential solution to this problem is to combine two sets of unique computations – which are the product of two different input sets – to form a single set of unique computations that may be more representative of all input

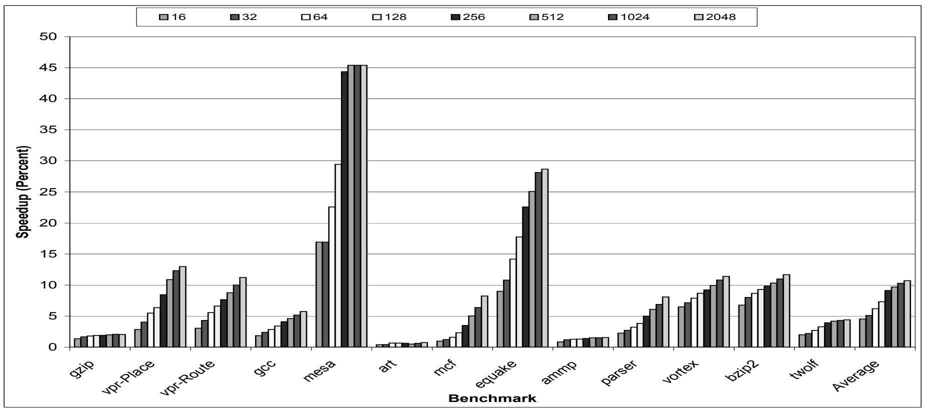

FIGURE 1.6: Instruction Precomputation Speedup; profile input set AB, run input set A

sets. Since two input sets are profiled and combined together, this approach is called Profile AB, Run A.

To form this combined set, unique computations were selected from the 2048 highest frequency unique computations from Input Set A and Input Set B. Excluding duplicates, the unique computations that were chosen for the final set were the ones that accounted for the largest percentage of dynamic instructions for their input set.

Figure 1.6 shows the speedup due to Instruction Precomputation for Profile AB, Run A for 16-entry to 2048-entry PT tables. These results show that the average speedup ranges from 4.53% for a 16-entry PT to 10.71% for a 2048-entry PT. By comparison, the speedup for Profile A, Run A ranges from 4.82% to 10.87% for the same table sizes, while the speedup for Profile B, Run A ranges from 4.47% to 10.53%. Therefore, while the average speedups for Profile AB, Run A are closer to the upper bound speedups, using the combined set of unique computations provides only a slight performance improvement over the Profile B, Run A speedups.

The main reason that the speedups for Profile AB are only slightly higher than the speedups for Profile B is that the highest frequency unique computations from Input Set A are very similar to their counterparts from Input Set B, for most benchmarks. Table 1.2 shows that with the exceptions of *vpr-Place*, *mesa*, and *twolf*, more than half of the highest frequency unique computations are common to both input sets. Therefore, it is not surprising to see that the speedup with this combined profile is only slightly higher.

Finally, although Profile AB yields higher speedups, the downside of this approach is that the compiler needs to profile two input sets. Therefore, from a cost-benefit point-of-view, an additional 0.29% (16 PT entries) to 0.15% (2048 PT entries) average speedup is not likely to offset the cost of profiling two input sets and combining their unique computations together.

1.5.4 Frequency versus Frequency and Latency Product

Although the set of the highest frequency unique computations represents the largest percentage of dynamic instructions, those instructions could have a lower impact on the execution time than their execution frequencies would suggest since many of those dynamic instructions have a single-cycle execution latency. Therefore, instead of choosing unique computations based only on their frequency of execution, choosing the unique computations that have the highest frequency-latency product (FLP) could yield a larger performance gain. The execution latency of a unique computation is strictly determined by its opcode, except for loads. Consequently the FLP for a unique non-load computation can be computed by multiplying the frequency of that unique computation by its execution latency.

Figure 1.7 presents the speedup due to Instruction Precomputation for Profile B, Run A for 16-entry to 2048-entry PT tables. As shown in Figure 1.7, the average speedup ranges from 3.85% for a 16-entry PT to 10.49% for a 2048-entry PT. In most cases, the speedup when using the highest FLP unique computations is slightly lower than the speedups when using the highest frequency unique computations. While this result may seem a little counterintuitive, the reason for this result is because the processor can issue and execute instructions out-of-order. This out-of-order execution allows the processor to hide the latency of high latency instructions by executing other instructions.

While the out-of-order processor is able to tolerate the effects of longer execution latencies, it is somewhat limited by the number of functional units. By using the highest FLP unique computations, fewer instructions are dynamically eliminated (as compared to when using the highest frequency unique computations), thus increasing the number of instructions that require a functional unit. As a result, any performance improvements gained by using the highest FLP unique computations are partially offset by functional unit contention.

1.5.5 Performance of Instruction Precomputation versus Value Reuse

As described above, the key difference between Value Reuse and Instruction Precomputation is that Value Reuse dynamically updates the VRT while the PT is statically managed by the compiler. Since the two approaches are quite similar, this sub-section compares the speedup results of Instruction Precomputation with the speedup results for Value Reuse.

The configuration of the base processor is the same as the base processor

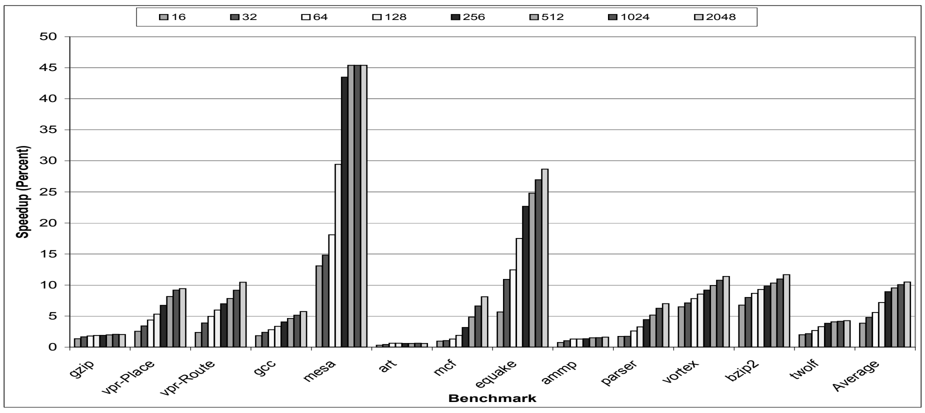

FIGURE 1.7: Instruction Precomputation speedup; profile input set B, run input set A for the highest frequency-latency products

configuration for Instruction Precomputation. The VRT size varies from 16 to 2048 entries. When the program begins execution, all entries of the VRT are invalid. During program execution, the opcode and input operands of each dynamic instruction are compared to the opcodes and input operands in the VRT. As with Instruction Precomputation, when the opcodes and input operands match, the VRT forwards the output value to that instruction and it is removed from the pipeline. Otherwise, the instruction executes normally. Entries in the VRT are replaced only when the VRT is full. In that event, the least-recently used (LRU) entry is replaced.

As shown in Figure 1.8, the average speedup ranges from 1.82% for a 16-entry VRT to 7.43% for a 2048-entry VRT while, by comparison, the speedup for Instruction Precomputation (Profile B, Run A) ranges from 4.47% to 10.53%. For all table sizes, Instruction Precomputation has a higher speedup; this difference is especially noticeable for the 16-entry tables.

Since Value Reuse constantly replaces the LRU entry with the opcode and input operands of the latest dynamic instruction, when the VRT is small, it can easily be filled with low frequency unique computations. By contrast, Instruction Precomputation is most effective when the table size is small since each entry in the PT accounts for a large percentage of dynamic instructions.

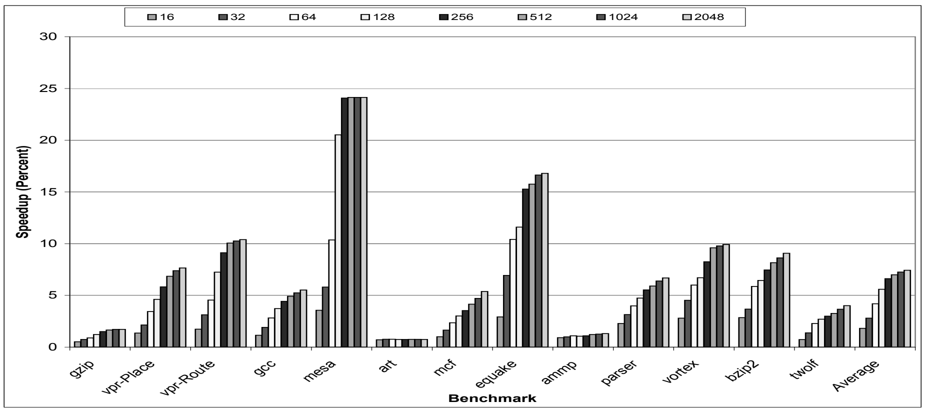

FIGURE 1.8: Speedup Due to Value Reuse; Run A

1.6 An Analytical Evaluation of Instruction Precomputation

While the speedup results in the previous section show that Instruction Precomputation yields significant performance improvements, the key question is: Why does Instruction Precomputation - and, by association, Value Reuse - improve the processor's performance? To answer this question, a Plackett and Burman design [8], as described in [9], was applied to Instruction Precomputation. The Plackett and Burman design is a statistically-based approach that measures the effect that each variable parameter has on the output variable, e.g. execution time. By knowing which processor or memory parameters have the most effect on the execution time, the computer architect can determine which parameters are the largest performance bottlenecks. By comparing the performance bottlenecks that are present in the processor before and after Instruction Precomputation is applied, the effect that Instruction Precomputation has on relieving or exacerbating the processor's bottlenecks can be easily seen.

The advantage that a fractional multi-factorial design, such as the Plackett and Burman design, has over a full multi-factorial design, such as the Analysis of Variance (ANOVA) design, is that the number of test cases required to execute the design is proportionally related to – instead of exponentially related to – the number of variable parameters.

Table 1.5 shows the results of using a Plackett and Burman design to analyze

TABLE 1.5: Processor Performance Bottlenecks, by Average Rank, Before and After Adding Instruction Precomputation

Component	Before	After	Before - After
ROB Entries	2.77	2.77	0.00
L2 Cache Latency	4.00	4.00	0.00
Branch Predictor Accuracy	7.69	7.92	-0.23
Number of Integer ALUs	9.08	10.54	-1.46
L1 D-Cache Latency	10.00	9.62	0.38
L1 I-Cache Size	10.23	10.15	0.08
L2 Cache Size	10.62	10.54	0.08
L1 I-Cache Block Size	11.77	11.38	0.39
Memory Latency, First	12.31	11.62	0.69
LSQ Entries	12.62	13.00	-0.38

the effect that Instruction Precomputation has on the processor's performance bottlenecks. The first column of Table 1.5 lists the ten most significant performance bottlenecks, out of a possible 41, while the second and third columns show the average rank of each component before and after, respectively, Instruction Precomputation is added to the processor. Finally, the fourth column shows the net change in the average rank. Since the components are ranked in descending order of significance, the most significant performance bottleneck is given a rank of 1, while the second most significant parameter is given a rank of 2, and so on.

The results in Table 1.5 show that Instruction Precomputation significantly relieves one performance bottleneck, the number of integer ALUs, since its average rank increases. Therefore, Instruction Precomputation has a similar, but not precisely the same, effect on the processor as adding additional integer ALUs. This result is not particularly surprising since most of the unique computations that are cached in the PT are operations that would have executed on an integer ALU. Therefore, adding Instruction Precomputation reduces the amount of contention for the integer ALU.

On the other hand, adding Instruction Precomputation somewhat exacerbates another performance bottleneck, the memory latency of the first block. The result is expected since, with Instruction Precomputation, the processor core consumes instructions at a faster rate, which puts more stress on the memory hierarchy.

In summary, using a Plackett and Burman design to analyze the effect of Instruction Precomputation shows that Instruction Precomputation improves the processors performance primarily by reducing the amount of pressure on the functional units. However, since the processor's performance is somewhat limited by the memory latency, further performance improvements due to Instruction Precomputation will come only by also diminishing the impact of the memory latency.

1.7 Extending Instruction Precomputation by Incorporating Speculation

Although the speedup results in Section 1.5 show that Instruction Precomputation can significantly improve the processor's performance, two problems limit additional performance gains. The first problem is that Instruction Precomputation is a non-speculative technique. While this characteristic eliminates the need for prediction verification hardware, the associated cost is that the Instruction Precomputation hardware must wait until both input operands are available. Since many of the unique computations in the PT are for operations that would execute on a low latency functional unit, dynamically removing these redundant computations does not dramatically reduce the latency of these operations; in fact, the only pipeline stages that these computations can bypass – and the total number of cycles that can be saved – are between the stage when the PT is accessed and the execute stage. Consequently, the non-speculative nature of Instruction Precomputation, i.e. waiting for both input operand values to become available, limits the performance gain.

The second problem is that the access time of the PT depends on the number of bits in the opcode and input operands. Since complete bit-by-bit comparisons are necessary, increasing the number of bits in the input operands from 32 to 64 bits, for example, dramatically increases the access time to the PT. Furthermore, depending on its specific implementation, the PT may be fully-associative, which further increases the PT access time. However, each additional cycle that is needed to access the PT directly decreases the latency reduction benefit of Instruction Precomputation.

One possible solution to these two problems that can further increase the performance of Instruction Precomputation is to speculatively "reuse"the output value after one input operand becomes available instead of waiting for both to become available. When one of the input operands becomes available, that value, or a hashed version, is used as an index in the PT. Then, depending on its implementation, the PT would either return the output value of the highest frequency unique computation (or multiple high frequency unique computations) that has that value as one of its input operand values. That output value can then be forwarded to any dependent instructions, which can speculatively execute based on that value. Obviously, the output value of the "reused"instruction needs to be checked and dependent instructions need to be re-executed if the speculation was incorrect.

While this approach is essentially a combination of Instruction Precomputation and Value Prediction [10], it has at least one advantage over either approach. As compared to Instruction Precomputation, speculatively reusing instructions requires the Instruction Precomputation hardware to wait for only a single input operand to become available. Also, fewer bits need to be compared. Furthermore, by profiling to determine the highest frequency unique computations and actually using a single input operand value to index into the PT, this approach may yield higher prediction accuracies than value prediction for more difficult-to-predict output value patterns.

1.8 Related Work

Sodani and Sohi [11] analyzed the frequency of unique computations associated with static instructions in the integer benchmarks of the SPEC 95 benchmark suite. Their results showed that 57% to 99% of the dynamic instructions produced the same result as an earlier instance of the instruction. Therefore, in typical programs, a very large percentage of the computations are redundant. Of the static instructions that execute more than once, most of the repetition is due to a small sub-set of the dynamic instructions. More specifically, with the exception of *m88ksim*, less than 20% of the static instructions that execute at least twice are responsible for over 90% of the dynamic instructions that are redundant. For m88ksim, those static instructions are responsible for over 50% of the instruction repetition.

Gonzalez et al. [12] also analyzed the frequency of redundant computations, but in both the integer and floating-point benchmarks of the SPEC 95 benchmark suite. Their results showed that 53% to 99% of the dynamic instructions repeated and that there is not a significant difference in the frequency of redundant computation between the integer and floating-point benchmarks. Overall, their results confirm the key conclusion from [11] that there is a significant amount of redundant computations associated with static instructions.

Sodani and Sohi [1] proposed a dynamic Value Reuse mechanism that exploited the Value Reuse that was associated with each static instruction. Each static instructions PC is used as an index to access the Value Reuse table. This approach produced speedups of 0% to 17%, 2% to 26%, and 6% to 43% for a 32-entry, a 128-entry, and a 1024-entry, respectively, VRT.

By contrast, Molina et al. [13] proposed a dynamic Value Reuse mechanism that exploits the output value that was associated with each static instruction and across all static instructions, i.e. by unique computation. The resultant speedups are correlated to the area used. For instance, when using a 221KB table, the speedups range from 3% to 25%, with an average of 10%. The speedups dropped to 2% to 15% with an average of 7% when the table area decreased to 36KB.

Citron et al. [14] proposed using distributed Value Reuse tables that are accessed in parallel with the functional units. Since this approach reduces the execution latency of the targeted instruction to a single cycle, it is best suited to bypass the execution of long latency arithmetic and logical instructions, e.g. integer and floating-point multiplies, divides, and square roots. As a result, although this mechanism produces speedups up to 20%, it is best suited for benchmarks with a significant percentage of high-latency instructions, such as the MediaBench benchmark suite [16].

Huang and Lilja [15] introduced Basic Block Reuse, which is Value Reuse at the basic block level. This approach uses the compiler to identify basic blocks where the inputs and outputs were semi- invariant. At run-time, the inputs and outputs for that basic block are cached after the first execution of that basic block. Before a subsequent execution, the current inputs of that basic block are compared with cached versions. If a match is found, the register file and memory are updated with the correct results. This approach produced speedups of 1% to 14% with an average of 9%.

Azam et al. [17] proposed using Value Reuse to decrease the processors power consumption. Value Reuse can decrease power consumption by reducing the latency of instructions to a single cycle and by removing the instruction from the pipeline. Their results showed that an eight-entry reuse buffer decreased the power consumption by up to 20% while a 128-entry reuse buffer decreased the power consumption by up to 60%, even after adding a pipeline stage to access the reuse buffer.

1.9 Conclusion

Redundant computations are computations that the processor previously executed. Instruction Precomputation is a new mechanism that can improve the performance of a processor by dynamically eliminating these redundant computations. Instruction Precomputation uses the compiler to determine the highest frequency unique computations, which subsequently are loaded into the Precomputation Table (PT) before the program begins execution. Instead of re-computing the results of a redundant computation, its output value is forwarded from the matching entry in the PT to the dependent instruction and then the redundant instruction is removed from the pipeline.

The results in this chapter show that a small number of unique computations account for a disproportionate number of dynamic instructions. More specifically, less than 0.2% of the total unique computations account for 14.68% to 44.49% of the total dynamic instructions. When using the highest frequency unique computations from Input Set B while running Input Set A (Profile B, Run A), a 2048-entry PT improves the performance of a base 4-way issue superscalar processor by an average of 10.53%. This speedup is very close to the upper-limit speedup of 10.87%. This speedup is higher than the average speedup of 7.43% that Value Reuse yields for the same processor configuration. More importantly, for smaller table sizes (16-entry), Instruction Precomputation outperforms Value Reuse, 4.47% to 1.82%. Finally, the results show that the speedup due to Instruction Precomputation is approximately the same regardless of which input set is used for profiling and regardless of how the unique computations are selected (frequency or frequency/latency product).

Overall, there are two key differences between Instruction Precomputation and Value Reuse. First of all, Instruction Precomputation uses the compiler to profile the program to determine the highest frequency unique computations while Value Reuse does its profiling at run-time. Since the compiler has more time to determine the highest frequency unique computations, the net result is that Instruction Precomputation yields a much higher speedup than Value Reuse for a comparable amount of chip area. Second, although using the compiler to manage the PT eliminates the need for additional hardware to dynamically update the PT, it can dramatically increase the compile time since the compiler must profile the program.

1.10 Acknowledgment

A preliminary version of this work was presented at the 2002 Euro-Par conference [2]. This work was supported in part by National Science Foundation grants EIA-9971666 and CCR-9900605, the IBM Corporation, and the Minnesota Supercomputing Institute.

References

[1] Sodani, A. and Sohi, G., Dynamic Instruction Reuse, *International Symposium on Computer Architecture,* 1997.

[2] Yi, J., Sendag, R., and Lilja, D., Increasing Instruction-Level Parallelism with Instruction Precomputation, *Euro-Par,* 2002.

[3] Burger, D. and Austin, T., The SimpleScalar Tool Set, Version 2.0, *University of Wisconsin-Madison Computer Sciences Department Technical Report #1342,* 1997.

[4] Kessler, R., McLellan, E., and Webb, D., The Alpha 21264 Microprocessor Architecture, *International Conference on Computer Design,* 1998.

[5] Yeager, K., The MIPS R10000 Superscalar Microprocessor, *IEEE Micro,* 16, , 28, 1996.

[6] Henning, J., SPEC CPU2000: Measuring CPU Performance in the New Millennium, *IEEE Computer,* 33, 28, 2000.

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

[8] Plackett, R. and Burman, J., The Design of Optimum Multifactorial Experiments, *Biometrika,* 33, 305, 1946.

[9] Yi, J., Lilja, D., and Hawkins, D., A Statistically Rigorous Approach for Improving Simulation Methodology, *International Conference on High-Performance Computer Architecture,* 2003.

[10] Lipasti, M., Wilkerson, C., and Shen, J., Value Locality and Load Value Prediction, *International Conference on Architectural Support for Programming Languages and Operating Systems,* 1996.

[11] Sodani, A. and Sohi, G., An Empirical Analysis of Instruction Repetition, *International Symposium on Architectural Support for Programming Languages and Operating Systems,* 1998.

[12] Gonzalez, A., Tubella, J., and Molina, C., The Performance Potential of Data Value Reuse, *University of Politecenica of Catalunya Technical Report: UPC-DAC-1998- 23,* 1998.

[13] Molina, C., Gonzalez, A., and Tubella, J., Dynamic Removal of Redundant Computations, *International Conference on Supercomputing,* 1999.

[14] Citron, D. and Feitelson, D., Accelerating Multi-Media processing by Implementing Memoing in Multiplication and Division Units, *International Conference on Architectural Support for Programming Languages and Operating Systems,* 1998.

[15] Huang, J. and Lilja, D., Exploiting Basic Block Locality with Block Reuse, *International Symposium on High Performance Computer Architecture,* 1999.

[16] Lee, C., Potkonjak, M., and , Mangione-Smith, W., MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems, *International Symposium on Microarchitecture,* 1997.

[17] Azam, M., Franzon, P., and Liu, W., Low Power Data Processing by Elimination of Redundant Computations, *International Symposium on Low Power Electronics and Design,* 1997.