Increasing Instruction-Level Parallelism with Instruction Precomputation

Joshua J. Yi, Resit Sendag, and David J. Lilja

Department of Electrical and Computer Engineering
Minnesota Supercomputing Institute
University of Minnesota - Twin Cities
Minneapolis, MN 55455
{jjyi, rsgt, lilja}@ece.umn.edu

Abstract. Value reuse improves a processor's performance by dynamically caching the results of previous instructions into the value reuse table and reusing those results to bypass the execution of future instructions that have the same opcode and input operands. However, replacing the least recently used entries with the results of the current instructions could eventually fill the value reuse table with instructions that are not frequently executed. Furthermore, the complex hardware that replaces entries and updates the table may necessitate an increase in the clock period. We propose *instruction precomputation* to address these issues by profiling programs to determine the opcodes and input operands that have the highest frequencies of execution, or the highest frequency/latency products (F/LP), these instructions then are loaded into the precomputation table before the program executes. During program execution, the precomputation table is used in the same way as the value reuse table, with the exception that the precomputation table does not dynamically replace any entries. For a 2K-entry precomputation table implemented on a 4-way issue machine, this approach produced average speedups of 11.0% (frequency) and 12.0% (F/LP), for a mix of 16 SPECint95 and SPEC 2000 benchmarks. By comparison, a 2K-entry value reuse produced an average speedup of 6.7% on the same benchmarks. We also show that this method is quite independent of the program's input set. Furthermore, due to the profiling step, this method is especially effective for small table sizes – a small (16-entry) precomputation table produces average speedups of 4.1% (frequency) and 4.4% (F/LP) for the same programs. For the same number of table entries, value reuse produces an average speedup of only 1.7%. Therefore, instruction precomputation outperforms value reuse, especially for smaller tables, with the same number of table entries while using less area and having a lower access time.

1 Introduction

During its execution, a program may repeatedly perform the same computations. For example, in a nested pair of FOR loops, an add instruction in the inner FOR loop will repeatedly initialize and increment a loop induction variable. For each iteration of the outer FOR loop, the computations performed by that add instruction are completely identical. These operations typically cannot be removed by an optimizing compiler since the initial value of the induction variable may change each time the loop is executed, for instance.

Value reuse [3, 12, 14] exploits this program characteristic by dynamically caching an instruction's opcode, input operands, and result into a *value reuse table* (VRT). For each instruction, the processor checks if the opcode and input operands for that instruction match an entry in the VRT. If a match is found, then the current instruction is a redundant computation and it can use the result stored in the VRT instead of re-executing the instruction. This reuse increases the amount of instruction-level parallelism (ILP). Since the VRT is accessed before the execute stage in a pipelined processor, value reuse can even reduce the latency of single-cycle instructions such as integer adds.

Since the processor constantly updates the VRT, and since the VRT is limited in size, a redundant computation could be stored in the VRT, evicted, re-executed, and re-stored. Consequently, the VRT could hold redundant computations that have a very low frequency of execution or are not on the critical path, thus decreasing the effectiveness of this method.

To address this frequency of execution issue, *instruction precomputation* uses profiling to determine the redundant computations with the highest frequencies of execution or with the highest frequency/latency products (FL/P). The opcodes and input operands for these redundant computations 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 (FL/P) redundant computations and 2) the PT does not replace or update any entries (except when the program is loaded). As a result, this approach selectively targets those redundant computations that have a large impact on the program.

This paper makes the following contributions:

1. Shows that a large percentage of a program is spent repeatedly executing a handful of redundant computations.
2. Shows that computations with the highest frequencies of execution and the highest F/LPs are independent of the program input set.

3. Describes a novel approach of using profiling to improve the performance of value reuse.
4. Presents a more cost-effective solution (in terms of the area and the cycle time) than value reuse by eliminating the dynamic update and replacement hardware and the associated table fields and by reducing the number of ports needed.

Section 2 describes instruction precomputation in more detail. Section 3 examines the performance of instruction precomputation and the effect of using different input sets. Section 4 compares this approach to some related work, Section 5 lists some items of future work, and Section 6 concludes.

2 Instruction Precomputation

Instruction precomputation consists of two main steps: profiling and execution. In the profiling step, the redundant computations with the highest frequencies, or highest F/LPs, are found. An instruction is a redundant computation if its opcode and input operands match a previously executed instruction's opcode and input operands.

After determining the highest frequency (F/LP) redundant computations, those redundant computations are loaded into the PT before the program executes. In the execution step, the PT is checked to see if there is a match between a PT entry and each instruction's opcode and input operands. If a match is found, then the instruction's result is simply the value in the output field of the matching entry. As a result, that instruction does not need to continue through the remaining stages of the pipeline. If a match is not found, then the instruction continues through the pipeline and executes as normal.

For instruction precomputation to be truly effective, two key questions about this approach need to be answered: 1) While [14] showed that a very high percentage of the dynamic instructions in a program are redundant, what percentage of a program's instructions are due to a limited number of very high frequency (frequency/latency) redundant computations? 2) Secondly, what fraction of these redundant computations are due to the program itself and what fraction are due to the program's input set? Are the same redundant computations present across input sets? In other words, what is the variability of the redundant computations due to the benchmark's input set? The first question is answered later in this section while the second is answered in Section 3.

Since instruction precomputation does not replace any table entries, for it to be effective, the redundant computations that are loaded into the PT must account for a significant percentage of the program's dynamic instruction count. Otherwise, instruction precomputation will produce very little speedup since it will affect only a very small percentage of the dynamic instructions. Similarly, it also will produce very little speedup if a large percentage of the redundant computations are due to the input set or if very few of the same redundant computations are produced when using different input sets. The variability of the redundant computations due to the program's input set directly affects the performance of instruction precomputation since all possible input sets for each benchmark cannot be profiled to find the highest frequency (F/LP) redundant computations.

To answer both of these questions, we profiled the benchmarks shown in Table 1 using two different input sets ("A" and "B") to determine the amount of redundant computation that is present in those benchmarks and to determine the effect of the input set on the variability of the highest frequency (F/LP) redundant computations. For this paper, all benchmarks were compiled using gcc 2.6.3 at optimization level O3 and each benchmark ran to completion.

For 099.go, we used a reduced version of the test input set to reduce execution time. For the same reason, some reduced input sets were also used for the SPEC 2000 benchmarks. Benchmarks that use the reduced input sets exhibit similar behavior as compared to when the benchmark uses the test, train, or reference input sets. For more information on the SPEC 2000 reduced input sets, see [11].

For 134.perl, the test input set is composed of two "sub-input" sets (Jumble.pl and Primes.pl) while the train input set is composed of an additional sub-input set (Scrabbl.pl). For 175.vpr, the reduced input sets are composed of two sub-input sets (Place and Route). To avoid losing any information about each benchmark's behavior with a different sub-input set, the results for sub-input sets were not combined together. Instead, they are presented separately for the remainder of this paper, with the exception of Figure 1.

To determine the amount of redundant computation for each instruction, we stored the instruction's opcode and input operands. Hereafter, the combination of opcode and input operands is referred to as a "unique computation". To reduce the memory requirements of storing this information, the frequency of execution was also stored for each unique computation. Therefore, each unique computation needed to be stored only once. Any unique computation that has a frequency of execution greater than one is a redundant computation. As a result, this profiling method shows not only that a unique computation is redundant, but it also shows the amount of redundancy. Since previous works [3,4,5,8,9,10,12,13,14,15] only show that most unique computations are redundant, this unique computation frequency profiling is our first contribution.

TABLE 1. Selected Characteristics for the Benchmarks Previously Tested

Benchmark	Suite	Type	Input Set A	Input Set B
099.go	SPEC 95	Integer	Train	Reduced Test
124.m88ksim	SPEC 95	Integer	Train	Test
126.gcc	SPEC 95	Integer	Test	Train
129.compress	SPEC 95	Integer	Train	Test
130.li	SPEC 95	Integer	Train	Test
132.ijpeg	SPEC 95	Integer	Test	Train
134.perl	SPEC 95	Integer	Test	Train
164.gzip	SPEC 2000	Integer	Reduced Small	Reduced Medium
175.vpr	SPEC 2000	Integer	Reduced Medium	Reduced Small
177.mesa	SPEC 2000	Floating-Point	Reduced Large	Test
181.mcf	SPEC 2000	Integer	Reduced Medium	Reduced Small
183.equake	SPEC 2000	Floating-Point	Reduced Large	Test
188.ammp	SPEC 2000	Floating-Point	Reduced Medium	Reduced Small
197.parser	SPEC 2000	Integer	Reduced Medium	Reduced Small
255.vortex	SPEC 2000	Integer	Reduced Medium	Reduced Large
300.twolf	SPEC 2000	Integer	Test	Reduced Large

After profiling each benchmark, the unique computations were sorted by their frequency of execution. Figure 1 shows what percentage of the total instructions are due to the top 2048 (by frequency) arithmetic unique computations. (Only arithmetic instructions are shown here because they are the only instructions that we allowed into the PT.)

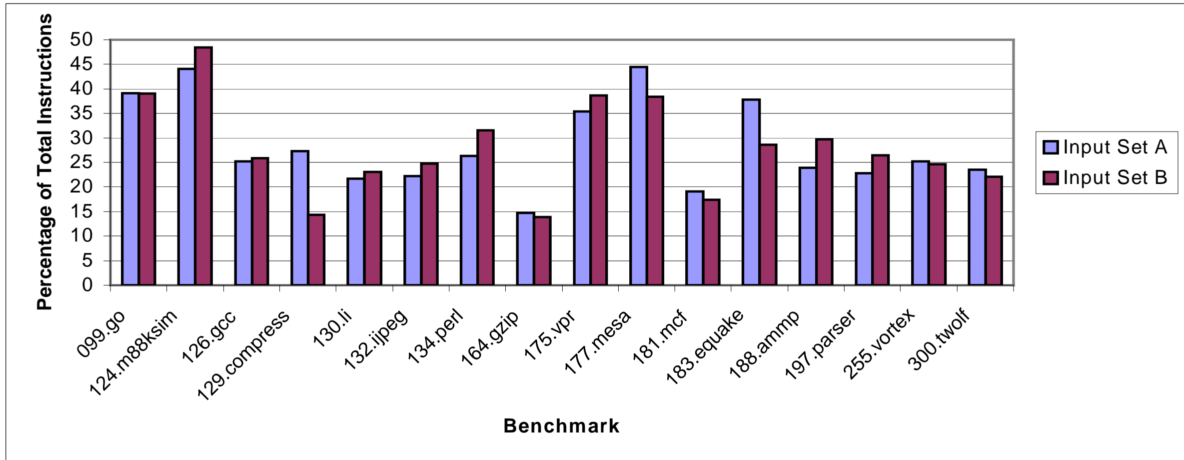

Figure 1. Percentage of Instructions that are Due to the Top 2048 Arithmetic Unique Computations

As can be seen in Figure 1, the top 2048 arithmetic unique computations account for 14.7% to 44.5% (Input Set A) and 13.9% to 48.4% (B) of the total instructions executed by the program. Similar percentages were found for the highest F/LP unique computations when the divisor was the total number of simulated cycles. Furthermore, a small number of unique computations can account for a large percentage of instructions. For instance, the top 16 unique computations account for 3.1% to 19.6% (A) and 2.8% to 16.0% (B). Therefore, profiling a program to determine the highest frequency (F/LP) unique computations and putting them into a PT can significantly improve the processor's performance by reducing the effective latency of each instruction that matches a unique computation in the PT, even for very small tables. For a more complete analysis on the amounts of unique computations in the SPEC 95 and SPEC 2000, please refer to [18].

3 Results and Analysis

3.1 Methodology and Base Machine Parameters

To determine the performance of instruction precomputation, we modified sim-outorder from the Simplescalar tool suite [2] to include a precomputation table. The PT can be accessed in both the dispatch and issue stages.

If a match is found in the dispatch stage, the instruction obtains its result from the PT and is removed from the pipeline (i.e. it waits only for in-order commit to complete its execution). Otherwise, the instruction executes as normal. However, if a match is found in the issue stage, the instruction obtains its result from the PT and is removed from the pipeline only if a free functional unit *cannot* be found. Otherwise, the instruction executes as normal.

The base machine used 2 integer ALUs, 2 floating-point ALUs, 1 integer multiply/divide unit, 1 floating-point multiply/divide unit, a 64 entry RUU, a 32 entry LSQ, and 2 memory ports. The L1 D and I caches were set to 32KB, 32B blocks, 2-way associativity, and a 1 hit cycle latency. The L2 cache was set to 256KB, 64B blocks, 4-way associativity, and a 12 cycle hit latency. The first block from memory took 60 cycles to retrieve while each following block took 5 cycles. The branch predictor was a combined predictor with 8K entries and a 64-entry return address stack.

3.2 Performance and the Effect of the Input Set

This section contains the results for three combinations of input sets used for profiling and execution. To reiterate one key point, the profiling step is used only to determine the highest frequency (F/LP) unique computations. The first combination tested used the same input set for profiling and for execution (i.e. Profile with Input Set A, Run with Input Set A; Profile B, Run B). This represents an approximate upper bound of performance for precomputation for this implementation (Section 5 describes some additional enhancement that can be made to improve the performance). While these combinations represent the approximate upper bound in performance, it is extremely unlikely that the same input set that is used for profiling also will be used during execution. Furthermore, if the input set has a significant effect on the highest frequency (F/LP) unique computations, the performance will be much lower than the upper bound. As a result, we simulate a second combination of input sets – profile the benchmark using one input set, but run the benchmark with another input set (i.e. Profile A, Run B; Profile B, Run A). Finally, to eliminate the peculiarities associated with any single input set, we profile two input sets, combine the unique computations together, and then run the benchmarks with either input set (i.e. Profile and combine A & B; Run A, and Run B). The last two combinations show how the input set affects the highest frequency (F/LP) unique computations and subsequently the performance of instruction precomputation.

3.2.1 Instruction Precomputation Performance Upper Bound

Figure 2 shows the speedup due to instruction precomputation for various numbers of entries in the PT when Input Set A is used for profiling and for execution. As shown in this figure, instruction precomputation improves the performance of all benchmarks by an average of 4.6% (16 entries) to 12.2% (2048 entries). The average is the mean weighted by execution time. The speedups for Profile B, Run B for all benchmarks are similar. Furthermore, the speedups when using the F/LP unique computations are only slightly higher (~0.1%).

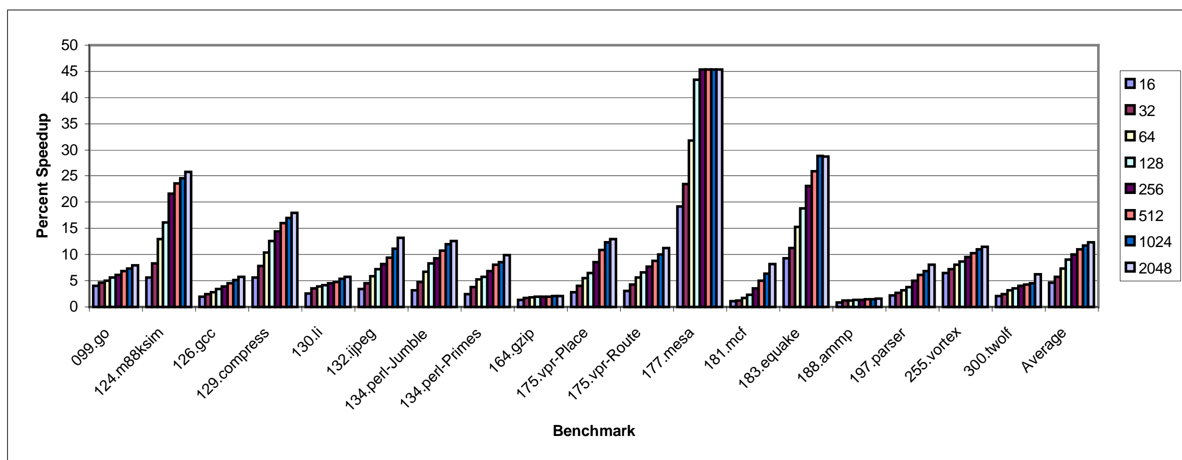

Figure 2. Percent Speedup Due To Instruction Precomputation for Various Table Sizes; Profile Input Set A, Run Input Set A

3.2.2 Input Set Effect

As mentioned above, the differences in the profile input set and the execution input set could have a deleterious effect on the performance of instruction precomputation by substantially affecting which unique computations are put into the PT. Note, however, that the performance is dramatically affected only if the unique

computations are radically different between profiling runs. Conversely, the performance will be relatively unaffected if a large percentage of the same unique computations are present in both input sets. In the latter case, the redundant computations are more a function of the benchmark itself rather than the input set.

Figure 3 shows the speedup when different input sets are used for the profiling and the execution; that is Profile B, Run A. We see that instruction precomputation improves the performance of all benchmarks by an average of 4.1% (16 entries) to 11.0% (2048 entries). The change in the average speedups between this combination and the Profile A, Run A combination (the upper-bound combination) differ by only 0.5% (16 entries) to 1.2% (2048 entries). Furthermore, the speedups between the two combinations are nearly identical for each benchmark and each PT size. Similar results also occur for the Profile A, Run B combination. *These results show that the highest frequency (F/LP) unique computations are common across benchmarks and are not a function of the input set*. This is the second contribution of this work. Finally, the speedups when using the F/LP unique computations are only slightly higher (~0.5%). The difference in speedups between this combination and the Profile A, Run A combination is completely attributable to the slight differences in the unique computations that are loaded into each table.

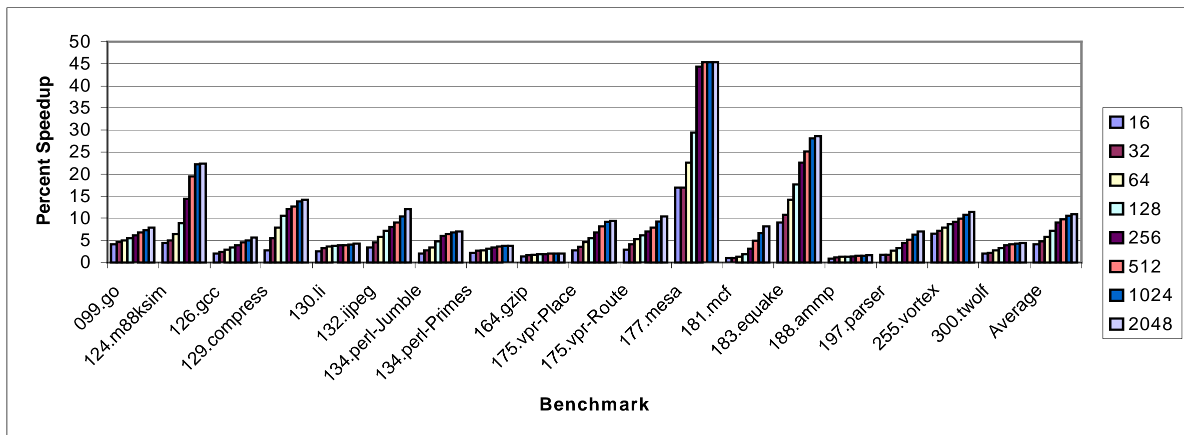

Figure 3. Percent Speedup Due To Instruction Precomputation for Various Table Sizes; Profile Input Set A, Run Input Set B

Although the input set does not dramatically affect the performance, a combination of unique computations from two profiling runs could slightly improve the performance when executing the benchmark using either input set. In other words, the Profile and combine A and B, Run A combination may produce higher speedups than Profile B, Run A. Due to space limitations, the results for this combination are not shown. However, the differences in the speedups are small (less than 1% on average) and are not consistently higher for all PT sizes for this combination.

In conclusion, the input set does NOT affect the whether or not a unique computation is one of the highest frequency (F/LP) unique computations. Consequently, for the same number of unique computations, the speedups using some input sets will be higher than others simply because the unique computations in the table account for a higher percentage of the program. Therefore, the redundant computations are almost completely an artifact of the benchmark and are almost completely unaffected by the benchmark's input set.

3.4 Comparison with Value Reuse

Since instruction precomputation is related to value reuse, it is necessary to compare the speedups of the two techniques. Since our current implementation uses the input operands to index the PT, this implementation of value reuse also uses the input operands to index the VRT.

The following figure compares the speedups of value reuse and instruction precomputation for various table sizes. While value reuse would probably have a longer clock period (due to the need for additional read and write ports and for entry replacement hardware) and would require more bits per entry, comparing the speedup results is as direct a comparison as possible with our simulation environment.

In Figure 4, three table sizes are shown – 32, 256, and 2048 entries. The VR in the legend corresponds to value reuse while the IP corresponds to instruction precomputation. For each benchmark, there are six bars. The leftmost three bars correspond to the speedup for the three different value reuse table sizes. The rightmost three bars correspond to the speedup for the three different instruction PT sizes.

Figure 4 shows two main results. First, instruction precomputation outperforms value reuse for almost all benchmarks and table sizes. When the speedup for value reuse is greater, the maximum difference is about 2%. However, part of this difference in speedup is due to which unique computations were actually loaded into

the table. For instance, several unique computations have the same frequency (F/LP). For those unique computations, there is no tie-breaking policy to choose which unique computation should go into the table; the current policy is first-come. Therefore, if there were two unique computations that had the same frequency (F/LP), for a smaller table, only one of them would make it into the table. However, the other unique computation may be on the critical path more often, thus yielding a higher speedup, than the one in the table. As a result, it is possible to increase the speedup for that table size simply by swapping those two unique computations. Determining a good heuristic to break ties is one of our items of future work. If the unique computations were more carefully chosen for each table size, the speedup for instruction precomputation would be higher than the speedup for value reuse for all benchmarks and all sizes except for 130.li. Even then, for this benchmark, the difference in the speedups is still less than 2%.

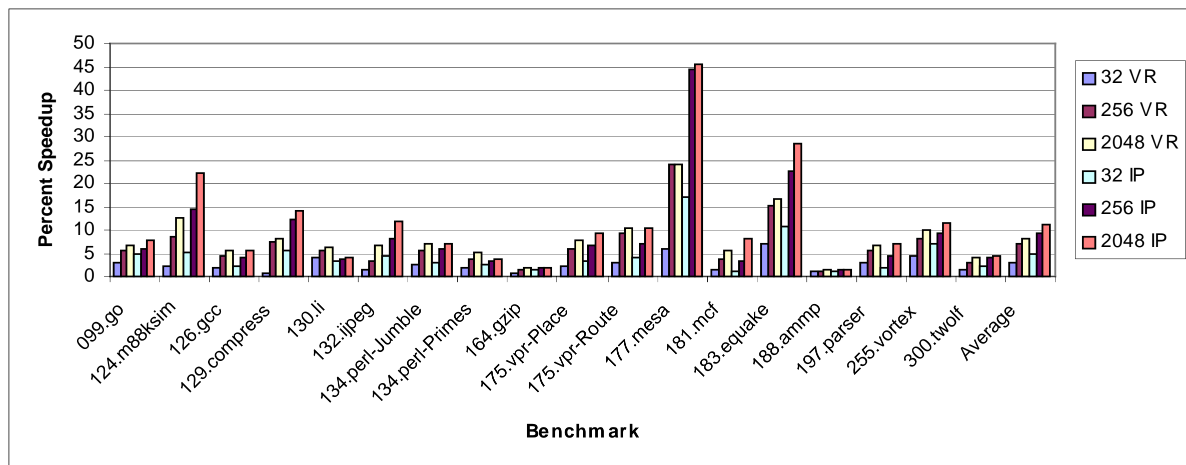

Figure 4. Speedup Comparison Between Value Reuse (VR) and Instruction Precomputation (IP) for Various Table Sizes; Profile Input Set A, Run Input Set B

The second main result is that for smaller table sizes, which are less expensive from an area and cycle time point-of-view, instruction precomputation has non-trivial speedups (4.1% for a 16-entry table) while value reuse has a much smaller speedup (1.7% for a 16-entry table). The reason that instruction precomputation outperforms value reuse for small tables is because instruction precomputation statically determines the unique computations (i.e. the ones with the highest frequency or F/LP) that are likely to have the most impact on the execution time.

Finally, while the speedups for using the highest F/LP unique computations were about the same as using the highest frequency unique computations, this difference can be much greater if the benchmark has large amounts of unique computations with long latencies. Benchmarks that are likely to have these types of unique computations are signal processing and multimedia benchmarks. For these benchmarks, it would probably be very costly from an area and time standpoint for value reuse to implement an effective replacement mechanism that accounted for the instruction's execution latency.

However, the performance of value reuse shows that the instruction precomputation approach of statically capturing the highest frequency (F/LP) unique computations and then using those precomputed results does not *always* yield the best speedup results. This proves the efficacy of dynamic replacement for some benchmarks. But this adds additional ports to the PT and will increase the PT access time, thereby negating an important advantage of instruction precomputation.

While instruction precomputation requires the profiling step to determine the unique computations that should be inserted into PT, this profiling cost is amortized over the ALL the execution runs for this benchmark. Value reuse, however, incurs its cost for each execution run. Furthermore, this profiling shifts some of the design complexity from the hardware to the software.

In conclusion, instruction precomputation produces speedups that are almost always higher than the speedups produced by value reuse for the same table size. Not only does instruction precomputation match or exceed (in most cases) the performance of value reuse, it does so at a lower area cost and with a potentially lower access time. Furthermore, instruction precomputation can easily use the instruction's execution latency to determine the unique computations that could yield the most performance difference. This is likely to be particularly beneficial for multimedia applications. Finally, large tables are not required for non-trivial speedups, especially in the case of 177.mesa.

4 Related Work

Sodani and Sohi in [14] implemented a dynamic value reuse mechanism. Their mechanism produced speedups of 0% to 17%, 2% to 26%, and 6% to 43% for a 32 entry, a 128 entry, and a 1024 entry, VRT, respectively. While the speedups in that paper are comparable to those in this paper, our approach has a smaller area footprint and a potentially lower cycle time. However, their approach does things that our current implementation does not do, including: dynamically replacing entries and reusing non-arithmetic instructions, including loads. Implementing instruction precomputation for load instructions would improve the performance even more.

In [12], Molina et. al. implemented a dynamic value reuse mechanism that exploited value reuse at the both the global (PC-independent) and local-levels (PC-dependent). However, their approach is very area-intensive. As can be expected, their speedups are somewhat dependent on the area used. For instance, their value reuse mechanism produced speedups of 3% to 25% with an average of 10% when using a 221KB table. When the table area is decreased to a more realistic 36KB, the reported speedups dropped to 2% to 15% with an average of 7%. While the speedups between our 2048-entry table and their 200KB version are comparable, our approach uses almost an order of magnitude less area.

Citron and Feitelson [3, 4, 5] proposed using distributed value reuse tables that are accessed in parallel with the functional units. While this approach produced speedups up to 20%, it targets only long latency instructions such as multiply, divide, and square root. In addition to targeting different instructions, using a distributed table, and producing lower speedups, one other difference compared to our approach is that their approach employs dynamic entry replacement while our current implementation does not.

Huang and Lilja introduced basic block reuse in [9, 10], which is value reuse at the basic block level. By reusing basic blocks instead of just instructions, this approach produced speedups of 1% to 14% with an average of 9%. In addition to targeting basic blocks instead of individual instructions (as our approach does) and producing lower speedups, this approach consumes a much larger amount of area. In this approach, each table entry requires at least 60 bytes of area (depending on the tag widths and next block identifier width).

In [17], Weinberg and Nagle proposed using value reuse to reduce the latency of pointer traversals by caching the elements of a pointer chain. This approach reduced the execution latency by up to 11.3%. However, this approach differs with ours in three respects: 1) It only targets pointers, 2) It uses dynamic replacement (which our current implementation does not do), and 3) It consumes a very large amount of area (approximately 600KB).

Azam et. al. In [1] proposed adding a dynamic reuse buffer and an extra pipeline stage (to access the reuse buffer) to decrease the base processor's power consumption. While one of the goals of this approach was to maintain the same performance as the base processor (i.e. the one without the reuse buffer and the extra pipeline stage) while decreasing the power consumption, it was not clear if the performance goal was met.

In summary, all the previous approaches produce comparable or lower speedups while consuming either a little more area or, in some cases, an order of magnitude more area. The other key difference is that the above approaches also target non-arithmetic instructions and use dynamic replacement, while our current version does not require these additions to produce comparable or better performance.

Gabbay and Mendelson in [6,7] presented an approach that used profiling to enhance the performance of value prediction. The value prediction hardware only predicts the values for instructions that have good value locality and a high probability of a correct prediction, as determined by the profiler. While this approach enhances value prediction, it parallels our instruction precomputation approach of profiling and selectively targeting certain instructions.

5 Future Work

This work spawns several other avenues of research. First of all, we plan to investigate different heuristics to guide which unique computations should make be inserted in the table in the case of a tie. Secondly we plan to investigate different replacement methods including: 1) Targeting instructions on the critical path and 2) Filling the PT with program phase specific unique computations. The second replacement method would use temporal based profiling to capture the unique computations that are present in different parts of the code. In addition to improving the performance, this will hopefully also decrease the PT size. Third, we plan to look at different ways of decreasing the area including: 1) Partitioning the table by opcodes and 2) Choosing unique computations that have smaller width input operands, such as immediates and offsets.

We also plan to look at the invariability of unique computations *across* programs. We feel that it is possible to find a set of unique computations that are common to many or all programs. In addition to simplifying the profiling step, this would also solve the side-effect of flushing the PT on a context switch. In addition to this potential solution, we are looking into other means of solving this context switch side effect.

Since the PT is filled with the highest frequency instructions, using instruction precomputation may yield significant speedups while also dramatically decreasing the power consumption. We plan to look into the potential power savings of this approach.

Finally, we are currently investigating using this profiling and precomputation approach to enhance the performance of other mechanisms such as branch prediction, data and instruction prefetching, and basic block reuse.

6 Conclusion

This paper presents a novel approach to value reuse that we call instruction precomputation. This approach uses profiling to determine the unique computations with the highest frequencies of execution or the highest F/LPs. These unique computations are, in all likelihood, the ones that are on the critical path. Those unique computations are then preloaded into the PT before the program begins execution. For each instruction, the opcode and input operands are compared to the opcodes and input operands in the PT. If there is a match, then the instruction is removed from the pipeline. Otherwise, the instruction executes as normal.

For a 2048 entry PT implemented on a 4-way issue machine, this approach produces speedups of 1.6% to 45.4%, with an average speedup of 11.0%. Furthermore, the speedup for instruction precomputation is greater than the speedup for value reuse for almost all benchmarks and table sizes. In addition to the generally superior performance, instruction precomputation also consumes less area and has a lower table access time (which could affect the cycle time) than value reuse.

We find that this technique is almost completely independent of the program's input set. Therefore, profiling a representative benchmark is sufficient to produce significant speedups when *any* input set is used.

This approach exhibits excellent potential and can be further refined by targeting other types of instructions (such as loads), designing a heuristic to carefully select which unique computations are inserted into the table, implementing a replacement policy that fine-tunes the PT entries to store only unique computations that are most likely on the critical path, and customizing the PT entries to specific regions of code. Furthermore, we believe that this technique can be used to improve the performance of hardware-based techniques such as branch prediction, prefetching, and basic block reuse.

Acknowledgements

The authors would like to thank AJ KleinOsowski, Keith Osowski, and Baris Kazar for their helpful comments on previous drafts of this paper.

This work was supported in part by National Science Foundation grant numbers CCR-9900605 and EIA-9971666, and by the Minnesota Supercomputing Institute.

References

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

[2] D. Burger and T. Austin; "The Simplescalar Tool Set, Version 2.0"; University of Wisconsin Computer Sciences Department Technical Report 1342.

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

[4] D. Citron and D. Feitelson; "The Organization of Lookup Tables for Instruction Memoization"; Hebrew University of Jerusalem Technical Report: 2000-4

[5] D. Citron and D. Feitelson; "Hebrew University of Jerusalem Technical Report: 2000-5"; Hebrew University of Jerusalem Technical Report: 2000-5

[6] F. Gabbay and A. Mendelson; "Can Program Profiling Support Value Prediction"; International Symposium on Microarchitecture, 1997

[7] F. Gabbay and A. Mendelson; "Improving achievable ILP through value prediction and program profiling"; Microprocessors and Microsystems, Vol. 22. No. 6, November 30, 1998; Pages 315-332

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

[9] J. Huang and D. Lilja; "Improving Instruction-Level Parallelism by Exploiting Global Value Locality"; University of Minnesota Technical Report: HPPC-98-12, 1998

[10] J. Huang and D. Lilja; "Exploiting Basic Block Locality with Block Reuse"; International Symposium on High Performance Computer Architecture, 1999
[11] A. KleinOsowski, J. Flynn, N. Meares, and D. Lilja; "Adapting the SPEC 2000 Benchmark Suite for Simulation-Based Computer Architecture Research"; Workshop on Workload Characterization, International Conference on Computer Design, 2000.
[12] C. Molina, A. Gonzalez, and J. Tubella; "Dynamic Removal of Redundant Computations"; International Conference on Supercomputing, 1999
[13] S. Richardson; "Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation"; Sun Microsystems Laboratories Technical Report SMLI TR-92-1, 1992
[14] A. Sodani and G. Sohi; "Dynamic Instruction Reuse"; International Symposium on Computer Architecture, 1997
[15] A. Sodani and G. Sohi; "An Empirical Analysis of Instruction Repetition"; International Conference on Architectural Support for Programming Languages and Operating Systems, 1998
[16] A. Sodani and G. Sohi; "Understanding the Differences Between Value Prediction and Instruction Reuse"; International Symposium on Microarchitecture, 1998
[17] N. Weinberg and D. Nagle; "Dynamic Elimination of Pointer-Expressions"; International Conference on Parallel Architectures and Compilation Techniques, 1998
[18] J. Yi and D. Lilja; "An Analysis of the Potential for Global Level Value Reuse in the SPEC95 and SPEC2000 Benchmarks"; University of Minnesota Technical Report: ARCTiC 01-01, 2001