An Analysis of the Amount of Global Level Redundant Computation in the SPEC 95 and SPEC 2000 Benchmarks

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

Abstract

This paper analyzes the amount of global level redundant computation within selected benchmarks of the SPEC 95 and SPEC 2000 benchmark suites. Local level redundant computations are redundant computations that are the result of a single static instruction (i.e. PC dependent) while global level redundant computations are redundant computations that are the result of multiple static instructions (i.e. PC independent). The results show that for all benchmarks more than 90% of the unique computations account for only 1.2% to 31.5% of the total number of instructions. In fact, less than 1000 (0.14%) of the most frequently occurring unique computations accounted for 19.4% - 95.5% of the dynamic instructions. Furthermore, more redundant computation exists at the global level as compared to the traditional local level. For an equal number of unique computations – approximately 100 for each benchmark – at both the global and local levels, the global level unique computations accounted for an additional 1.5% to 12.6% of the total number of dynamic instructions as compared to the local level unique computations. As a result, exploiting redundant computations through value reuse at the global level should yield a significant performance improvement as compared to exploiting redundant computations only at the local level.

1. Introduction

During its execution, a program tends to repeatedly perform the same computations. This is due to the way that programs are written [4]. For example, due to a nested loop, an add instruction in the inner loop may repeatedly initialize and then increment a loop induction variable. For each iteration of the outer loop, the computations performed by that add instruction are completely identical.

In value reuse [5, 4], an on-chip table dynamically caches the results of previous computations. The next time the identical computation appears, the value reuse hardware accesses the table (using the program counter (PC) an index), retrieves the result, and forwards it to dependent instructions. The instruction is then removed from the pipeline since it has finished executing.

Value reuse improves the processor's performance by effectively decreasing the latency of the reused instructions. Decreasing the latency of a reused instruction either directly or indirectly reduces the execution time of the critical path; directly if the reused instruction is on the critical path and indirectly if the reused instruction produces the value of an input operand for an instruction that is on the critical path. Furthermore, since the reused instruction does not pass through all the pipeline stages, the number of resource conflicts (available issue slots, functional units, reservation station entries, etc.) decreases.

Since the PC is used to index the value reuse table, traditional value reuse is based on the computational history of a single static instruction. Consequently, previous computations can only be reused if that particular computation was performed for that static instruction. As a result, while another instruction with the same opcode, but with a different PC, may perform a computation that could be reused by the first instruction, value reuse does not occur because the results of the second instruction cannot be accessed by the first.

This paper refers to PC dependent value reuse as local level or local value reuse and PC independent value reuse as global level or global value reuse. In local level value reuse, the value reuse table is accessed by using the PC. Since the PC is used to access the table, only the value history for that static instruction is accessible. As a result, for value reuse to occur for a dynamic instruction, its static instruction must have previously executed with the same input operands. If not, then the dynamic instruction cannot be reused. However, in global value reuse, the PC is not used to access the value reuse table; instead, the

table is accessed by some combination of the opcode and input operands. As a result, an instruction can reuse the output of any static instruction that previously executed with the same opcode and input operands. In conclusion, since using the PC to access the value reuse table limits the "reusability" to that corresponding instruction, PC dependent value reuse is referred to as local level value reuse. On the other hand, using the opcode and the input operands (i.e. PC independent value reuse) to access the reuse table is called global value reuse.

The contribution of this paper is to show the potential of global value reuse by completely quantifying the amount of redundant computation at the global level and to compare it to the amount of redundant computation at the local level.

This paper is organized as follows: Section 2 describes some related work. Section 3 describes the experimental methodology and setup while Section 4 presents and discusses the results. Section 5 discusses future work and Section 6 concludes.

2. Related Work

[6] analyzed the amount of instruction repetition in the integer benchmarks of the SPEC 95 benchmark suite. Their results showed that 56.9% (129.compress) to 98.8% (124.m88ksim) of the dynamic instructions were repeated. In addition, they also analyzed the causes of instruction repetition. However, these results were only for instruction repetition at the local level.

[2] analyzed the amount of instruction repetition in the integer and floating-point benchmarks of the SPEC 95 benchmark suite. Their results showed that 53% (110.applu) to 99% (104.hydro2d) of the dynamic instructions were repeated. Furthermore, the geometric means of the all the benchmarks, the integer benchmarks only, and the floating-point benchmarks only were 87%, 91%, and 83%, respectively. Therefore, there is not a significant difference in the amount of instruction repetition between the integer and floating-point benchmarks. Like [6], their results were for instruction repetition at only the local level.

[5] implemented a dynamic value reuse mechanism that only exploited local level value reuse and tested it with selected SPEC 92 and 95 benchmarks. Their value reuse mechanism reused 0.2% to 26%, 5% to 27%, and 13% to 27% of the dynamic instructions for a 32 entry, a 128 entry, and a 1024 entry, respectively, value reuse buffer. It 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, value reuse buffer. However, reusing a higher percentage of instructions did not directly translate to greater speedup since some of the reused instructions were not on the critical path.

[4], on the other hand, implemented a dynamic value reuse mechanism that exploited value reuse at the both the global and local levels. To test the performance of their value reuse mechanism, they used selected integer and floating-point benchmarks from the SPEC 95 benchmark suite. Their value reuse mechanism produced speedups of 3% to 25%; on average, it reused about 30% of the instructions, resulting in a 10% speedup. While [4] implemented a global reuse mechanism, it did not determine the potential for global value reuse nor did it analyze which instructions had the highest frequencies of repetition.

3. Experimental Setup

To determine the amount of redundant computation at the global level, the opcode, input operands, and PC for all the dynamic instructions have to be stored. This paper refers to the opcode, input operands, and PC of a dynamic instruction as a “unique computation”. (Note that in some cases, the term “unique computation” only refers to the opcode and input operands.) To reduce the memory requirements for storing this information, for duplicate unique computations (i.e. redundant computations), in addition to storing the unique computation itself, the total number of times that that unique computation was executed was also stored. The instruction output was not stored because it is purely a function of the opcode and input operands.

To determine the amount of global redundant computation, each unique computation’s PC was set to 0. As a result, unique computations that had the same opcode and input operands, but different PCs, mapped to the same unique computation. For the local level, the unique computation’s PC was simply the instruction’s PC.

To gather this data, a modified version of sim-fast from the Simplescalar tool suite [1] was used. Since sim-fast is only a functional simulator, it is optimized for simulation speed. As a result, it does not account for time; only executes instructions serially; and does not model a processor’s pipeline, caches, etc. sim-fast was used as the base simulator instead of sim-outorder for two reasons. The first reason is that since this paper only profiles the instructions, the execution time, cache behavior, etc. are unimportant. Consequently, only a functional simulator is needed. Secondly, since the code that was added to the base simulator accounted for a significant fraction of the simulation time, a fast base simulator was needed to reduce the overall simulation time.

The criteria for selecting which benchmarks to profile was that the benchmark had to be written in C because the Simplescalar tool suite only has a C compiler for PISA. The benchmark input set that was used was the maximum of either: 1) The one that produced the fewest number of dynamic instructions or 2) The one that was closest to 500 million dynamic instructions. Since the unique computation for each dynamic instruction was stored in

Table 1: Benchmark Characteristics

Benchmark	Suite	Type	Instructions (M)	Input Set
099.go	SPEC 95	Integer	548.2	Train
124.m88ksim	SPEC 95	Integer	120.1	Train
126.gcc	SPEC 95	Integer	1273.3	Test
129.compress	SPEC 95	Integer	35.7	Train
130.li	SPEC 95	Integer	183.3	Train
132.ijpeg	SPEC 95	Integer	553.3	Test
134.perl	SPEC 95	Integer	2391.5	Test (Jumble)
147.vortex	SPEC 95	Integer	2520.1	Train
164.gzip	SPEC 2000	Integer	526.4	Reduced Small
175.vpr – Place	SPEC 2000	Integer	216.9	Reduced Medium
175.vpr - Route	SPEC 2000	Integer	93.7	Reduced Medium
177.mesa	SPEC 2000	Floating-Point	1220.9	Reduced Large
181.mcf	SPEC 2000	Integer	174.7	Reduced Medium
183.equake	SPEC 2000	Floating-Point	715.9	Reduced Large
188.ammp	SPEC 2000	Floating-Point	244.9	Reduced Medium
197.parser	SPEC 2000	Integer	459.2	Reduced Medium

memory, the number of instructions for each benchmark was limited to reduce the memory requirements (which needed to be below the machine limit of 50 GB). However, each benchmark ran to completion. All benchmarks were compiled using gcc 2.6.3 at optimization level O3. Table 1 lists the benchmarks profiled in this paper and some selected characteristics:

For the SPEC 2000 benchmarks, reduced input sets were used to reduce the simulation times. 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 reduced input sets for these benchmarks, see [3].

175.vpr is a versatile place and route tool. Executing the benchmark involves first running the place function and then the route function (with the output of the place function as the input). As a result, two separate simulations captured the unique computations for these two functions. Therefore, in this paper, the results for the place and route functions are given separately.

4. Results

The following terms appear in the subsequent subsections: frequency of repetition and occurrences. The frequency of repetition, or frequency, is the number of times that a unique computation occurs (i.e. the number of dynamic instructions with that particular unique computation). Therefore, if one unique computation has a frequency of repetition of 1, it is completely unique, i.e. it is the only dynamic instruction in the entire program with that unique computation.

The number of occurrences is the number of times that a particular frequency is present. See Subsection 4.1 for an example of the number of occurrences.

4.1. Distribution of Occurrences

The first result is the distribution of occurrences for each frequency. For example, consider the following:

0+1, PC = 0, Frequency = 400
0+9, PC = 0, Frequency = 350
1+1, PC = 0, Frequency = 500
1+2, PC = 0, Frequency = 450
1+3, PC = 0, Frequency = 500
1+4, PC = 0, Frequency = 450
1+5, PC = 0, Frequency = 450
1+6, PC = 0, Frequency = 450
1+7, PC = 0, Frequency = 550

Figure 1: Example unique computations

Therefore, 0+9 occurs 350 times in the program; 0+1 400 times; 1+2, 1+4, 1+5, and 1+6 450 times each; 1+1 and

Table 2: Distribution of occurrences for each frequency for the unique computations in fig. 1

Range	Occurrences	Unique Computations
300-349	0	
350-399	1	0+9
400-449	1	0+1
450-499	4	1+2, 1+4, 1+5, 1+6
500-549	2	1+1, 1+3
550-599	1	1+7
600-649	0	

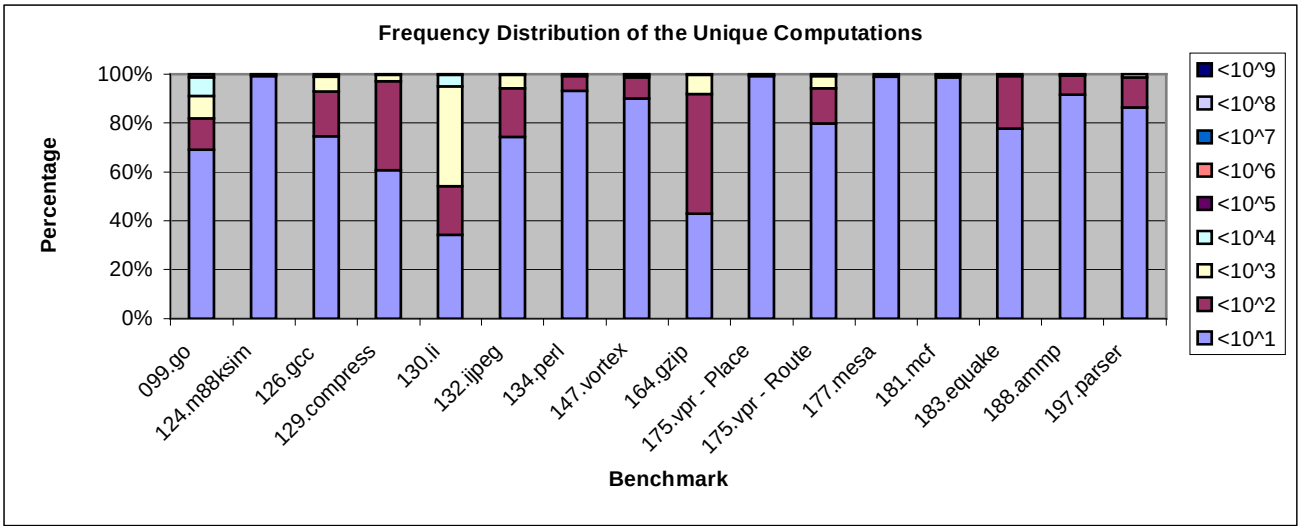

Figure 2: Frequency distribution of unique computations per benchmark, global level, normalized

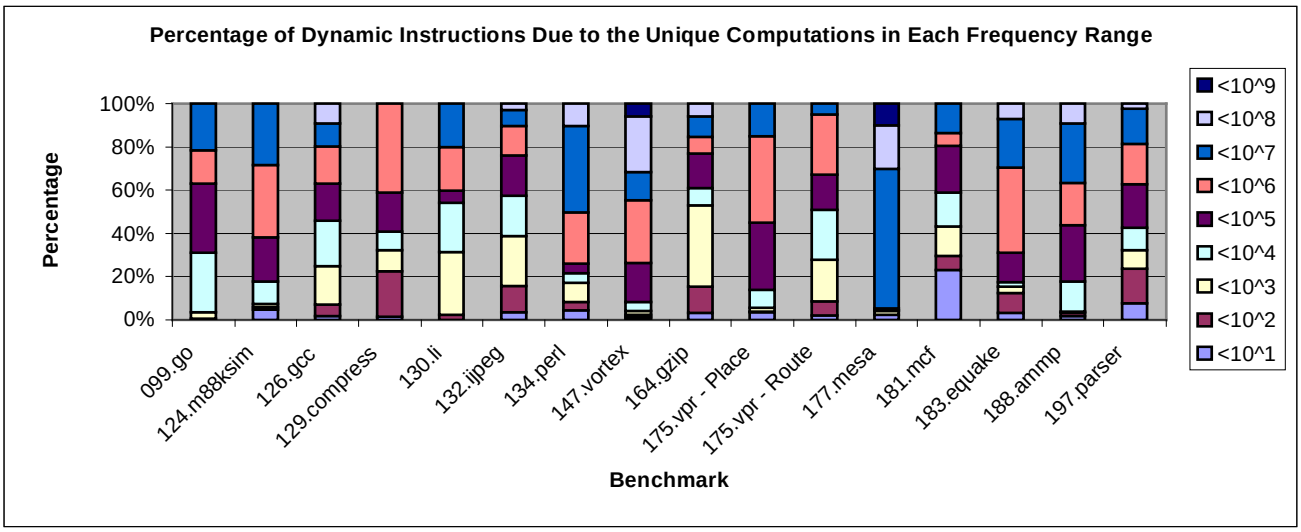

Figure 3: Percentage of dynamic instructions due to the unique computations in each frequency range, global level, normalized

1+3 500 times each; and 1+7 550 times. Table 2 shows the distribution of occurrences for each frequency for the unique computations in Figure 1.

Figure 2 shows the frequency distribution, for logarithmic frequency ranges, of the unique computations for the benchmarks listed in Table 1. After trying several different frequency range sizes, the logarithmic range size was used since it produced the most compact results without affecting the content.

In Figure 2, the height of each bar corresponds to the percentage of unique computations that have a frequency of execution within that frequency range. For example, if the unique computation 10004+11442, PC = 1000 executes 8 times (e.g. frequency of execution = 8), then it falls into the $< 10^1$ frequency range.

As can be seen in Figure 2, for all benchmarks except for 130.li, almost 60% of the unique computations have execution frequencies less than 10 and almost 80% of all unique computations have execution frequencies less than 100. This figure shows that the performance benefit in reusing most of the unique computations is relatively low since most of them are only executed a few times.

4.2. Number of Redundant Instructions

The product of the frequency of execution for a unique computation and the number of occurrences for a particular frequency corresponds to the number of dynamic instructions that those unique computations

represent. For example, if three unique computations each have a frequency of 500,000, then those unique computations are executed a total of 1,500,000 times, which corresponds to 1,500,000 dynamic instructions. Figure 3 shows the percentage of dynamic instructions due to the unique computations in each frequency range.

In Figure 3, the height of each bar corresponds to the percentage of dynamic instructions that have their unique computation in that frequency range. For each frequency range, comparing the heights of the bars in Figures 2 and 3 gives the relationship between the unique computations and dynamic instructions. For instance, in 124.m88ksim, more than 90% of the unique computations represent less than 4.9% of the dynamic instructions. Similar relationships between the percentage of unique computations and the percentage of dynamic instructions hold for the other benchmarks.

In short, more than 90% of the unique computations account for only 1.2% (147.vortex) to 31.5% (130.li) of the total number of instructions. Therefore, from Figures 2 and 3, *a very small percentage of the unique computations account for a disproportionately large percentage of the total number of instructions*. This is one of the key results of this paper.

Building upon this key result and applying it to value reuse, Table 3 shows the percentage of dynamic instructions that are represented by less than 1024 unique computations (the size of a realistically sized reuse buffer).

Table 3 shows that less than 1024 unique computations represent a significant percentage of instructions (19.4% - 95.5%). [7] exploited this characteristic by statically determining the highest frequency unique computations and caching them into a precomputation table (i.e. a value reuse table with no write ports). This produced speedups up to 40.7% when 2048 unique computations were cached.

4.4. Top 100 Unique Computations

The following tables, Table 4, show the characteristics of the top 100 unique computations, by occurrence, for each benchmark. (To be more precise, the tables show the characteristics of the unique computations with the top 100 occurrences. As a result, for some benchmarks, the tables represent the characteristics for more than 100 unique computations if two unique computations have the same number of occurrences.) In Table 4, the second column shows what percentage of the unique computations these Top 100 unique computations represent while the third column does the same for the total number of instructions.

Two conclusions can be drawn from Table 4. First of all, Table 4 confirms the conclusion drawn from Figures 2 and 3 that a very small percentage of the unique computations account for a disproportionately large number of the dynamic instructions.

Although Figures 2 and 3 and Table 4 show that a very small percentage of the unique computations account for a disproportionately large number of the dynamic instructions, not all of those unique computations will yield the same performance gain or will consume the same amount of area in a value reuse table. For instance, while a double floating-point divide takes multiple cycles to execute – and therefore is an ideal candidate for reuse, storing two 64-bit double words (input operands) and one (128-bit) quad word is very expensive in terms of area. Furthermore, comparing two 64-bit numbers could delay the actual reuse of the instruction by a cycle. However, what is not clear for these instructions is whether the large reduction in the execution latency is worth the additional cost in area and a comparatively longer access time. Similarly, reusing the unique computation for a store instruction probably would not increase the performance as much as the frequency of repetition would indicate because stores are probably not on the critical path. However, for load instructions, either the target address for the load can be reused or the data returned by the load can reused (which is essentially last-value prediction). Finally, reusing the unique computations for move instructions (e.g. move to or from the high or low register) does not improve the performance because the instruction does not generate an output. In this case, the instruction only performs an action (a register write), which has to occur in program order. Therefore, the second conclusion from Table 4 is that not all the instructions in the Top 100 occurrences can be reused or will have the same

Table 4: Characteristics of the unique computations for the top 100 occurrences

Benchmark	% of Unique Computations	% of Total Inst.
099.go	0.01214	21.0
124.m88ksim	0.00324	62.6
126.gcc	0.00085	21.9
129.compress	0.02477	51.9
130.li	0.03187	40.7
132.ijpeg	0.00124	19.9
134.perl	0.00023	34.6
147.vortex	0.00078	40.6
164.gzip	0.00145	21.5
175.vpr-Place	0.00163	37.3
175.vpr-Route	0.00958	35.6
177.mesa	0.00110	86.9
181.mcf	0.00030	22.7
183.equake	0.00119	37.4
188.ammp	0.00469	53.6
197.parser	0.00055	25.7

performance gain when reused.

4.5. Comparison of Global Level Redundant Computation in Integer and Floating-Point

After comparing the results of 177.mesa, 183.equake and 188.ammp against the other 13 benchmarks for Figures 2 and 3 and Tables 3 and 4 there does not appear to be any significant differences between the integer and floating-point benchmarks. There is one slight difference, for the floating-point benchmarks, unique computations from floating-point instructions are in the Top 100 list. This is expected since these benchmarks contain a significantly higher percentage of floating-point instructions. However, since only three floating-point benchmarks were profiled, no definite conclusions can be made at this time.

4.6. Comparison to Local

One of the key questions that this paper tries to answer is how much more redundant computation is available and can be exploited at the global level than at the local level? This section compares the global and local level

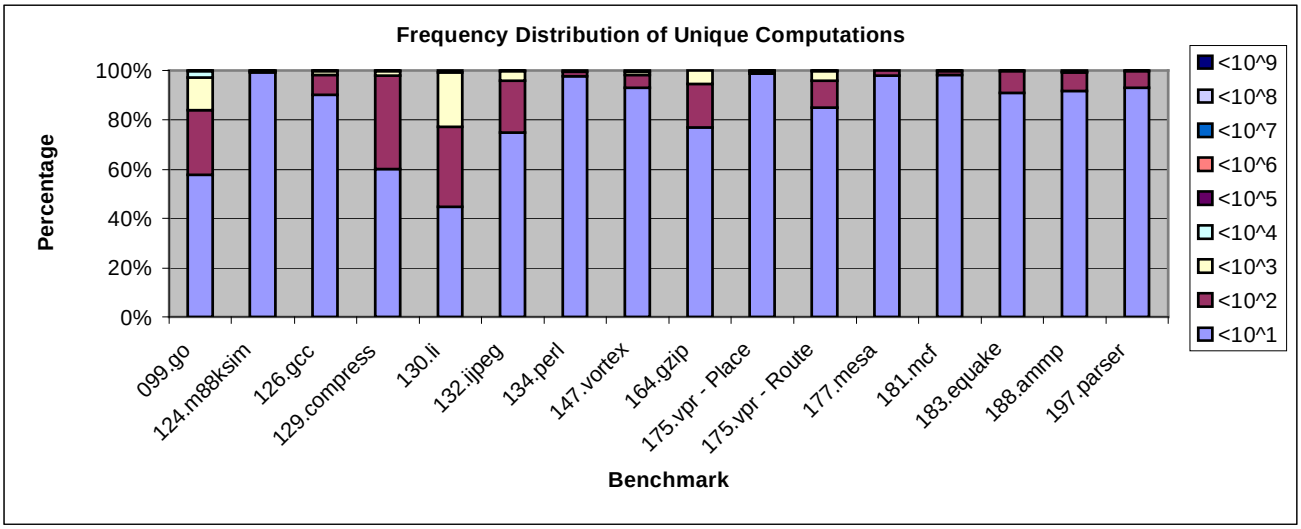

Figure 4A: Frequency distribution of unique computations per benchmark, local level normalized

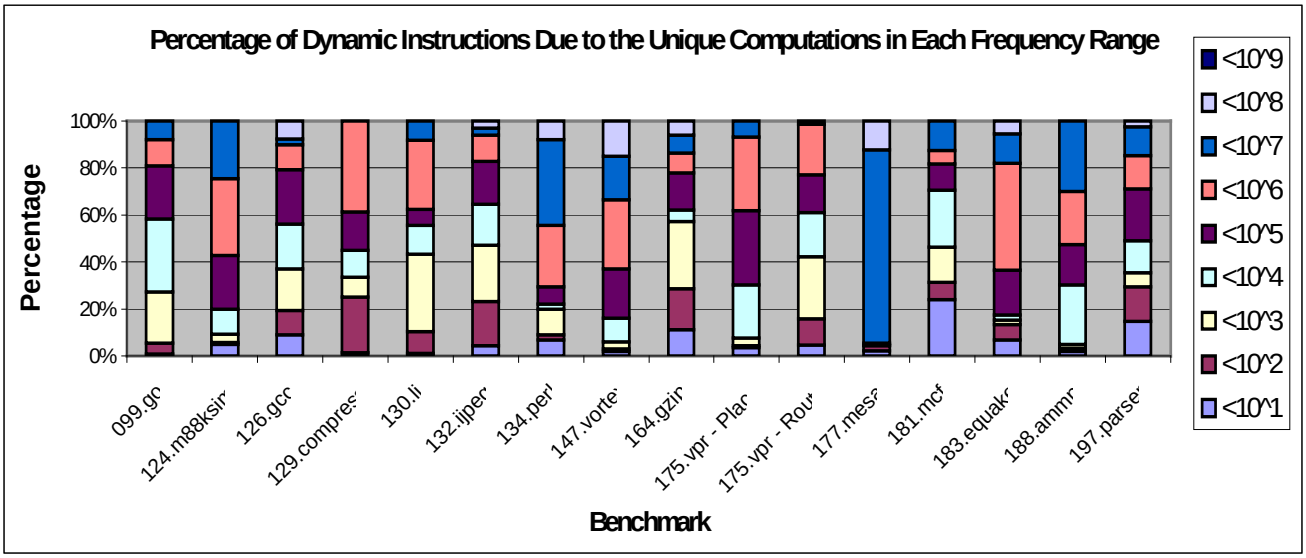

Figure 4B: Percentage of dynamic instructions due to the unique computations in each frequency range, local level, normalized

results for the:

1. Percentage of Instructions in Each Frequency Range
2. Percentage of Instructions Covered by the Unique Computations From the Top 100 Occurrences

4.6.1. Percentage of Instructions in Each Frequency Range

Figure 4A shows the frequency distribution of the unique computations at the local level while Figure 4B shows the percentage of dynamic instructions due to the unique computations in each frequency range at the local level.

For each benchmark, since there are fewer unique computations in the global level case, a direct comparison of Figures 2 and 4A does not make sense. However, since there are the same number of instructions in both the global and local levels (for the same benchmark), a direct comparison can be made.

Comparing those two figures shows that the unique computations in the higher frequency ranges at the global level represent more dynamic instructions as compared to the unique computations in the same frequency ranges at the local level. This is expected since unique computations that have the same opcode and input operands but have different PCs will map to different unique computations at the local level, but to the same unique computation at the global level.

This result could have a significant impact on the performance of value reuse at the local or global levels. First of all, this means that a single global level unique computation accounts for a larger percentage of the benchmark's instructions than does the corresponding local level unique computation. Secondly, at the global level, since there are fewer unique computations, there are fewer replacements in the value reuse table. Therefore, unique computations with very high frequencies of execution should stay in the table longer.

4.6.2. Percentage of instructions covered by unique computations from the top 100 occurrences

Table 5 compares the percentage of instructions due to the unique computations from the Top 100 occurrences for both the global and local levels. The number of unique computations in this table is the same for both the global and local levels and is the number of unique computations in the Top 100 occurrences at the global level.

Table 5: Percentage of instructions due to the unique computations from the top 100 occurrences for both the global and local levels

	Percentage of Total Inst.		
Benchmark	**Global**	**Local**	**Global – Local**
099.go	21.0	14.2	6.8
124.m88ksim	62.6	57.6	5.0
126.gcc	21.9	13.1	8.8
129.compress	51.9	48.0	3.9
130.li	40.7	36.0	4.7
132.ijpeg	19.9	14.4	5.5
134.perl	34.6	27.1	7.5
147.vortex	40.6	28.2	12.4
164.gzip	21.5	19.7	1.8
175.vpr-Place	37.3	25.4	11.9
175.vpr-Route	35.6	26.4	9.2
177.mesa	86.9	76.4	10.5
181.mcf	22.7	21.2	1.5
183.equake	37.4	24.8	12.6
188.ammp	53.6	47.5	6.1
197.parser	25.7	21.7	4.0

As expected, the unique computations for the Top 100 occurrences at the global level cover a higher percentage of instructions for all benchmarks as compared to the unique computations for the Top 100 occurrences at the local level. Therefore, a global value reuse mechanism could reuse an additional 1.5% (181.mcf) to 12.6% (183.equake) of the total number of dynamic instructions as compared to the local level. While these percentages for the global level cases are not dramatically larger than their local level counterparts, it will increase if more occurrences are include (i.e. using a Top 1000 list vs. a Top 100 list).

4.6.3. Global vs. Local Comparison

The conclusion from Subsections 4.6.1 and 4.6.2 is that there is more performance potential for value reuse at the global level as compared to the local level. However, in spite of these results, it is difficult to determine the performance difference between these two approaches. The performance difference primarily depends on the following three factors: 1) The number of reused instructions on the critical path for both approaches, 2) The average number of redundant unique computations in the local value reuse table, and 3) The precise implementation of the global value reuse mechanism.

For the first factor, it is reasonable to assume that distribution of reused instructions that are on the critical path for the two approaches is similar. Therefore, this factor should not be the primary contributor to the performance difference. The second factor is essentially an efficiency metric. If the local value reuse table holds several unique computations that only differ by PC, then some of its entries are wasted – when compared to the global value reuse table. If there are very few unique computations in the local value reuse table, then the

potential performance difference between the two approaches will be greater. Finally, the third factor determines the area of and access time to the global value reuse hardware. For the same area, the global value reuse table may have fewer entries as compared to the local value reuse table if more hardware, such as comparators, is needed. Furthermore, the access time of the global value reuse table could be higher, which would obviously affect the performance.

Therefore, the most accurate way of comparing these two approaches is to implement and then compare them.

However, these two approaches could be complementary, i.e. combining the two approaches could provide better performance than by using either one individually. While the local level does not account for as many instructions as does the global level, due to implementation differences, the local level reuse mechanism could have a lower access time and a lower area cost. Therefore, combining the two approaches could provide better performance than using either one individually.

5. Future Work

These results generate several ideas that warrant further investigation. First of all, how much performance can be gained by only exploiting global level value reuse? What kind of global value reuse mechanism is most efficient in terms of access time and area? Should this global value reuse mechanism only target certain types of instructions?

Secondly, how does that performance gain compare to only exploiting value reuse at the local level? In terms of the area, is that performance gain cost-effective?

Third, can global and local level value reuse mechanisms be combined to produce a more effective (area-wise or performance-wise) value reuse mechanism? Can these two approaches be combined to yield a value reuse mechanism that has the best qualities of each approach?

6. Conclusion

This paper presents an analysis of the potential for global value reuse and compares the amount of redundant computation at the global level to the amount of redundant computation at the local level.

For all benchmarks, less than 10% of the unique computations for account at least 65% of the dynamic instructions. More than 90% of the unique computations account for only 1.2% (147.vortex) to 31.5% (130.li) of the total number of instructions. Furthermore, 19.4% - 95.5% of the dynamic instructions are the results of less than 1000 of the most frequently occurring unique computations. For an equal number of unique computations (approximately 100 for each benchmark) for both the global and local levels, the unique computations for the global level account for an additional 1.5% to 12.6% of the total number of dynamic instructions. This difference will increase as more unique computations are added to both levels.

In conclusion, this paper makes the following key conclusions: 1) All benchmarks have significant amounts of redundant computations, 2) Significantly more redundant computation exists at the global level as compared to the local level, and 3) A very small percentage of unique computations account for a disproportionately large number of dynamic instructions.

7. Bibliography

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

[2] 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, 1998

[3] 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.

[4] C. Molina, A. Gonzalez, and J. Tubella; “Dynamic Removal of Redundant Computations”; International Conference on Supercomputing, 1999

[5] A. Sodani and G. Sohi; “Dynamic Instruction Reuse”; International Symposium on Computer Architecture, 1997.

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

[7] J. Yi and D. Lilja; “Increasing Instruction-Level Parallelism with Instruction Precomputation”; University of Minnesota Electrical and Computer Engineering Technical Report No. ARCTiC 01-03, 2001.

8. Acknowledgements

The authors thank Chris Hescott for implementing a B-tree that stored the unique computation information and dramatically reduced simulation time. The authors would also like to thank Baris Kazar and Keith Osowski for their helpful comment about 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.