An Analysis of the Potential for Global Level Value Reuse in the SPEC 95 and SPEC 2000 Benchmarks

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

Abstract

This paper analyzes the amount of redundant computation at a global level 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, less than 10% of the input sets account more than 65% of the dynamic instructions; an input set is defined as an instruction's opcode, input operands, and PC. In addition, for 8 of the 15 benchmarks profiled in this paper, less than 10% of the input sets accounted for over 90% of the dynamic instructions. Additionally, less than 1000 (0.14%) of the most frequently occurring input sets accounted for 19.4% - 95.5% of the dynamic instructions. Furthermore, more potential for value reuse exists at the global level as compared to the traditional local level. For an equal number of input sets – approximately 100 for each benchmark – at both the global and local levels, the global level input sets accounted for an additional 1.5% to 12.6% of the total number of dynamic instructions as compared to the local level input sets. As a result, exploiting value reuse at the global level should yield a significant performance improvement as compared to exploiting reuse 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 [Molina 99]. 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 [Sodani 97, Molina 99], 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 PC an index), retrieves the result, and forwards the result 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 computation was performed for the instruction associated with that particular PC. As a result, while another instruction of the same type, 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 instruction is accessible. As a result, for value reuse to occur for that dynamic instruction, the 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, the instruction can reuse the output of any previously executed instruction that had the same opcode and input operands. In conclusion, since using the PC to access the value reuse table limits "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 goals of this paper are to determine the potential of global value reuse by 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, analyzes, and discusses the results. Section 5 discusses future work and Section 6 concludes.

2 Related Work

[Sodani 98] 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. However these results were only for instruction repetition at the local level. In addition, they also analyzed the causes of instruction repetition.

[Gonzalez 98] 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 [Sodani 98], their results were for instruction repetition at only the local level.

[Sodani 97] 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.

[Molina 99], 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 simulated 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 that resulted in a 10% speedup. While [Molina 99] 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 dynamic instructions had to be stored. This paper refers to the opcode, input operands, and PC of a dynamic instruction as the "input set" for that instruction. To reduce the memory requirements for storing this information, for duplicate input sets (i.e. redundant computations), in addition to storing the input set itself, the total number of times that that input set was executed was stored. The instruction output was not stored because it is purely a function of the input set.

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

To gather this data, a modified version of sim-fast from the Simplescalar tool suite [Burger 97] 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 is 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. 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 input set for each dynamic instruction was stored in 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:

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
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

Table 1: Benchmark Characteristics

For the SPEC 2000 benchmarks, reduced input sets were used to reduce the simulation times. Benchmarks that used the reduced input sets exhibit similar behavior as compared to when the benchmark used the test, train, or reference input sets. For more information on the reduced input sets for these benchmarks, see [KleinOsowski 00].

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 input sets 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 an input set occurs (i.e. the number of dynamic instructions with that particular input set). Therefore, if one input set has a frequency of repetition of 1, it is completely unique (only one dynamic instruction in the entire program has that input set).

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 for Each Frequency

The first result is the distribution of occurrences for each frequency. For example, consider the following input sets in Figure 1:

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 Input Sets

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

Range	Occurrences	Contributing Input Sets
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	

Table 2: Distribution of Occurrences for Each Frequency for the Input Sets in Figure 1

After sorting the frequencies into several different range sizes, the logarithmic range size produced the most compact results without affecting the content of the results. Table 3 shows

the distribution of occurrences for each frequency for each benchmark for the logarithmic ranges.

	Range								
Benchmark	**< 10^1**	**< 10^2**	**< 10^3**	**< 10^4**	**< 10^5**	**< 10^6**	**< 10^7**	**< 10^8**	**< 10^9**
099.go	580358	107922	75852	64067	11481	618	54	0	0
124.m88ksim	4496289	27822	5499	6434	720	122	16	0	0
126.gcc	8797319	2145525	728418	117313	8122	888	62	6	0
129.compress	413289	249581	17965	1053	289	68	0	0	0
130.li	145432	83172	173744	20682	460	106	20	0	0
132.ijeg	8407217	2219849	619779	36934	3893	297	14	1	0
134.perl	65800155	4277381	527417	57588	3232	2383	388	9	0
147.vortex	13178369	1250703	151615	30943	14640	2663	176	25	1
164.gzip	3718958	4276579	686743	21906	3076	166	19	2	0
175.vpr-Place	7025225	31835	8868	5995	2134	385	15	0	0
175.vpr-Route	1006428	183494	64103	8611	599	87	3	0	0
177.mesa	20582763	2591	196781	2593	0	23	291	11	1
181.mcf	38364541	417713	84431	10012	1399	37	8	0	0
183.equake	9601311	2604542	101950	5299	2778	1007	67	4	0
188.ammp	2361735	206934	3639	7380	2333	142	22	1	0
197.parser	17003501	2431005	223679	14910	4008	404	35	1	0

Table 3: Distribution of Occurrences for Each Frequency

Note that each column is not the cumulative sum of the previous columns, but only the number of occurrences between column headings.

Table 3 shows that most of the input sets have a frequency that is less than 1000, although all the benchmarks, with the exception of 129.compress, have a few input sets that have frequencies over 1,000,000. 147.vortex and 177.mesa have an input set with a frequency of over 100,000,000.

Table 4A shows the distribution of occurrences for each frequency as a percentage of the sum of occurrences while Table 4B shows the cumulative percentage. Therefore, each column is the sum of the occurrences for all the occurrences less than the column label.

	Range								
Benchmark	**< 10^1**	**< 10^2**	**< 10^3**	**< 10^4**	**< 10^5**	**< 10^6**	**< 10^7**	**< 10^8**	**< 10^9**
099.go	69.1	12.8	9.0	7.6	1.4	0.1	0.0	0.0	0.0
124.m88ksim	99.1	0.6	0.1	0.1	0.0	0.0	0.0	0.0	0.0
126.gcc	74.6	18.2	6.2	1.0	0.1	0.0	0.0	0.0	0.0
129.compress	60.6	36.6	2.6	0.2	0.0	0.0	0.0	0.0	0.0
130.li	34.3	19.6	41.0	4.9	0.1	0.0	0.0	0.0	0.0
132.ijeg	74.5	19.7	5.5	0.3	0.0	0.0	0.0	0.0	0.0
134.perl	93.1	6.1	0.7	0.1	0.0	0.0	0.0	0.0	0.0
147.vortex	90.1	8.5	1.0	0.2	0.1	0.0	0.0	0.0	0.0

164.gzip	42.7	49.1	7.9	0.3	0.0	0.0	0.0	0.0	0.0
175.vpr-Place	99.3	0.4	0.1	0.1	0.0	0.0	0.0	0.0	0.0
175.vpr-Route	79.7	14.5	5.1	0.7	0.0	0.0	0.0	0.0	0.0
177.mesa	99.0	0.0	0.9	0.0	0.0	0.0	0.0	0.0	0.0
181.mcf	98.7	1.1	0.2	0.0	0.0	0.0	0.0	0.0	0.0
183.equake	78.0	21.1	0.8	0.0	0.0	0.0	0.0	0.0	0.0
188.ammp	91.5	8.0	0.1	0.3	0.1	0.0	0.0	0.0	0.0
197.parser	86.4	12.4	1.1	0.1	0.0	0.0	0.0	0.0	0.0

Table 4A: Percentage of the Occurrences for Each Frequency

	Range								
Benchmark	**< 10^1**	**< 10^2**	**< 10^3**	**< 10^4**	**< 10^5**	**< 10^6**	**< 10^7**	**< 10^8**	**< 10^9**
099.go	69.1	81.9	90.9	98.6	99.9	100.0	100.0	100.0	100.0
124.m88ksim	99.1	99.7	99.8	100.0	100.0	100.0	100.0	100.0	100.0
126.gcc	74.6	92.8	98.9	99.9	100.0	100.0	100.0	100.0	100.0
129.compress	60.6	97.2	99.8	99.9	100.0	100.0	100.0	100.0	100.0
130.li	34.3	54.0	95.0	99.9	100.0	100.0	100.0	100.0	100.0
132.ijeg	74.5	94.1	99.6	100.0	100.0	100.0	100.0	100.0	100.0
134.perl	93.1	99.2	99.9	100.0	100.0	100.0	100.0	100.0	100.0
147.vortex	90.1	98.6	99.7	99.9	100.0	100.0	100.0	100.0	100.0
164.gzip	42.7	91.8	99.7	100.0	100.0	100.0	100.0	100.0	100.0
175.vpr-Place	99.3	99.8	99.9	100.0	100.0	100.0	100.0	100.0	100.0
175.vpr-Route	79.7	94.2	99.3	99.9	100.0	100.0	100.0	100.0	100.0
177.mesa	99.0	99.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
181.mcf	98.7	99.8	100.0	100.0	100.0	100.0	100.0	100.0	100.0
183.equake	78.0	99.1	99.9	100.0	100.0	100.0	100.0	100.0	100.0
188.ammp	91.5	99.5	99.6	99.9	100.0	100.0	100.0	100.0	100.0
197.parser	86.4	98.8	99.9	100.0	100.0	100.0	100.0	100.0	100.0

Table 4B: Cumulative Percentage of the Occurrences for Each Frequency

For 124.m88ksim, 134.perl, 147.vortex, 175.vpr – Place, 177.mesa, 181.mcf, and 188.ammp; at least 90% of the occurrences have frequencies less than 10. For 126.gcc, 129.compress, 132.ijpeg, 164.gzip, 175.vpr – Route, 183.equake, and 197.parser; at least 90% of the occurrences have frequencies less than 100. Finally, for 099.go and 130.li, at least 90% of the occurrences have frequencies less than 1000. Table 5 summarizes the previous sentence.

Range with 90% Cutoff	**Benchmarks**
< 10	124.m88ksim, 134.perl, 147.vortex, 175.vpr – Place, 177.mesa, 181.mcf, 188.ammp
< 100	126.gcc, 129.compress, 132.ijpeg, 164.gzip, 175.vpr – Route, 183.equake, 197.parser
< 1000	099.go,130.li

Table 5: Frequency Range With At Least 90% of the Cumulative Occurrences

4.2 Number of Redundant Instructions

The product of the occurrences and frequency is the number of dynamic instructions for that input set. For example, if three input sets each have a frequency of 500,000, then those input sets are executed a total of 1,500,000 times, which corresponds to 1,500,000 dynamic instructions. Since Table 3 only shows the number of occurrences for each frequency range, it does not indicate which of the benchmarks could show the largest performance gain if a global reuse mechanism were implemented. Table 6A shows the percentage of the products for each frequency range while Table 6B shows the cumulative percentage of the products.

	Range								
Benchmark	**$< 10^1$**	**$< 10^2$**	**$< 10^3$**	**$< 10^4$**	**$< 10^5$**	**$< 10^6$**	**$< 10^7$**	**$< 10^8$**	**$< 10^9$**
099.go	0.1	0.4	3.1	27.3	**32.0**	15.4	21.6	0.0	0.0
124.m88ksim	4.9	0.6	1.7	10.3	20.6	**33.5**	28.4	0.0	0.0
126.gcc	1.8	5.2	17.7	**21.4**	17.0	17.1	10.5	9.4	0.0
129.compress	1.4	21.0	9.9	8.6	17.8	41.2	0.0	0.0	0.0
130.li	0.1	2.2	**29.1**	22.6	5.6	20.4	19.9	0.0	0.0
132.ijeg	3.6	12.0	23.1	18.5	18.8	13.5	7.6	3.0	0.0
134.perl	4.5	3.4	9.0	4.5	4.6	23.5	**40.1**	10.4	0.0
147.vortex	1.2	1.2	1.7	4.0	18.1	**28.9**	13.2	25.7	6.0
164.gzip	3.2	12.1	**37.5**	8.2	16.0	7.9	9.2	6.0	0.0
175.vpr-Place	3.7	0.3	1.3	8.4	31.5	**40.0**	14.8	0.0	0.0
175.vpr-Route	2.2	6.2	19.4	23.1	16.2	**28.0**	5.0	0.0	0.0
177.mesa	2.3	0.0	2.0	0.2	0.0	0.6	**64.8**	19.9	10.2
181.mcf	**23.1**	6.4	13.6	15.7	21.8	5.9	13.4	0.0	0.0
183.equake	3.4	8.8	3.1	2.0	13.8	**39.3**	22.5	7.1	0.0
188.ammp	1.9	1.5	0.6	13.8	26.2	19.5	**27.4**	9.2	0.0
197.parser	7.5	16.1	8.5	10.5	**20.2**	18.7	16.1	2.4	0.0

Table 6A: Percentage of the Occurrences/Frequency Product (Max. Percentage in Bold)

	Range								
Benchmark	**$< 10^1$**	**$< 10^2$**	**$< 10^3$**	**$< 10^4$**	**$< 10^5$**	**$< 10^6$**	**$< 10^7$**	**$< 10^8$**	**$< 10^9$**
099.go	0.1	0.6	3.7	31.0	63.0	78.4	100.0	100.0	100.0
124.m88ksim	4.9	5.5	7.2	17.5	38.1	71.6	100.0	100.0	100.0
126.gcc	1.8	7.0	24.7	46.1	63.1	80.2	90.6	100.0	100.0
129.compress	1.4	22.4	32.4	41.0	58.8	100.0	100.0	100.0	100.0
130.li	0.1	2.4	31.5	54.1	59.7	80.1	100.0	100.0	100.0
132.ijeg	3.6	15.6	38.7	57.2	76.0	89.5	97.0	100.0	100.0
134.perl	4.5	7.9	16.9	21.4	26.1	49.6	89.6	100.0	100.0
147.vortex	1.2	2.3	4.0	8.1	26.2	55.1	68.4	94.0	100.0
164.gzip	3.2	15.4	52.8	61.0	77.0	84.8	94.0	100.0	100.0
175.vpr-Place	3.7	4.0	5.3	13.6	45.2	85.2	100.0	100.0	100.0
175.vpr-Route	2.2	8.4	27.8	50.9	67.0	95.0	100.0	100.0	100.0

177.mesa	2.3	2.3	4.3	4.5	4.5	5.1	69.9	89.8	100.0
181.mcf	23.1	29.6	43.1	58.9	80.6	86.6	100.0	100.0	100.0
183.equake	3.4	12.2	15.3	17.4	31.2	70.5	92.9	100.0	100.0
188.ammp	1.9	3.3	3.9	17.7	43.9	63.4	90.8	100.0	100.0
197.parser	7.5	23.6	32.1	42.7	62.8	81.5	97.6	100.0	100.0

Table 6B: Cumulative Percentage of the Occurrences/Frequency Product

From Table 5, more than 90% of the input sets (for frequencies less than 10) for 124.m88ksim, 134.perl, 147.vortex, 175.vpr – Place, 177.mesa, 181.mcf, and 188.ammp account for 4.9%, 4.5%, 1.2%, 3.7%, 2.3%, 23.1%, and 1.9%, respectively, of the total number of instructions. More than 90% of the input sets (for frequencies less than 100) for 126.gcc, 129.compress, 132.ijpeg, 164.gzip, 175.vpr – Route, 183.equake, and 197.parser account for 7.0%, 22.4%, 15.6%, 15.4%, 8.4%, 12.2%, and 23.6%, respectively, of the total number of instructions. Finally, more than 90% of the input sets (for frequencies less than 1000) for 099.go and 130.li account for 3.7% and 31.5%, respectively, of the total number of instructions. Therefore, more than 90% of the input sets account for only 1.2% (147.vortex) to 31.5% (130.li) of the total number of instructions. For 7 of the 16 benchmarks, more than 90% of the input sets account for less than 5% of the total number instructions; for 9 of 16 benchmarks, the number is less than 10%; and for 15 of the 16 benchmarks, the number is less than 25%. In other words, with the exception of 130.li, more than 75% of the instructions come from less than 10% of the input sets.

The key point from Tables 4A, 4B, 6A, and 6B is that *a very small percentage of the input sets account for a disproportionately large percentage of the total number of instructions*. Table 7 shows the percentage of the dynamic instructions that are from less than 1024 of the input sets.

Benchmark	Number of Input Set	% of Input Sets	% of Dynamic Instructions
099.go	672	0.07997	37.0
124.m88ksim	858	0.01891	82.5
126.gcc	956	0.00810	36.9
129.compress	357	0.05233	59.0
130.li	586	0.13833	45.9
132.ijeg	312	0.00276	24.0
134.perl	397	0.00056	50.4
147.vortex	202	0.00138	44.9
164.gzip	187	0.00215	23.0
175.vpr-Place	400	0.00565	54.8
175.vpr-Route	689	0.05454	49.1
177.mesa	326	0.00157	95.5
181.mcf	45	0.00012	19.4
183.equake	71	0.00058	29.5
188.ammp	165	0.00639	56.1
197.parser	440	0.00224	37.2

Table 7: Percentage of Dynamic Instructions From Less Than 1024 of the Input Sets

Table 7 shows that with a realistically sized reuse buffer (less than 1024 entries), a significant percentage of instructions can be reused (19.4% - 95.5%).

4.4. Top 100 Input Sets by Occurrence

The following tables, Tables 8A and 8B, show the characteristics of the top 100 input sets, by occurrence, for each benchmark. (To be more precise, the tables show the characteristics of the input sets with the top 100 occurrences. As a result, for some benchmarks, the tables represent the characteristics for more than 100 input sets if two input sets have the same number of occurrences.) In Table 8A, the second and third columns show the minimum and maximum number of occurrences of the input sets in the Top 100 list. The fourth column shows what percentage of the input sets these Top 100 occurrences represent while the rightmost column does the same for the total number of instructions. Table 8B shows the instruction types that are represented in the list.

	Occurrences			
Benchmark	**Minimum**	**Maximum**	**% of Input Sets**	**% of Total Instructions**
099.go	340375	11785454	0.01214	21.0
124.m88ksim	79063	4660428	0.00324	62.6
126.gcc	694953	27584059	0.00085	21.9
129.compress	20600	648118	0.02477	51.9
130.li	84971	4058778	0.03187	40.7
132.ijeg	183032	16339592	0.00124	19.9
134.perl	2282917	60464845	0.00023	34.6
147.vortex	1479010	150761871	0.00078	40.6
164.gzip	167914	19703045	0.00145	21.5
175.vpr-Place	283292	6588115	0.00163	37.3
175.vpr-Route	61965	1862986	0.00958	35.6
177.mesa	1310736	124783523	0.00110	86.9
181.mcf	75142	5471245	0.00030	22.7
183.equake	608828	16323043	0.00119	37.4
188.ammp	190011	22467568	0.00469	53.6
197.parser	291888	11138791	0.00055	25.7

Table 8A: Characteristics of the Input Sets for the Top 100 Occurrences

Benchmark	Instruction Type
099.go	ADDIU, ADDU, BEQ, BNE, JAL, JUMP, LUI, LW, SLL, SLTI, SUBU, SW
124.m88ksim	ADDIU, ADDU, AND, ANDI, BEQ, BNE, DLW, JAL, JR, JUMP, LBU, LUI, LW, MFHI, MFLO, MULTU, OR, ORI, SLL, SLLV, SLTI, SLTIU, SLTU, SRL, SRLV, SUBU, SW
126.gcc	ADDIU, ADDU, AND, ANDI, BEQ, BGEZ, BLEZ, BLTZ, BNE, LB, LHU, LUI, LW, NOR, OR, SLL, SLT, SLTI, SLTIU, SRA
129.compress	ADDIU, ADDU, AND, ANDI, BC1F, BEQ, BGTZ, BLEZ, BLTZ, BNE, JAL, JR, JUMP, LBU, LUI, LW, SB, SLL, SLLV, SLT, SLTI, SLTU, SRA, SUBU, SW

130.li	ADDIU, ADDU, AND, ANDI, BEQ, BNE, JAL, JR, JUMP, LB, LUI, LW, ORI, SLL, SLTI, SLTIU, SLTU, SRA, SW
132.ijeg	ADDIU, ADDU, AND, ANDI, BEQ, BGEZ, BNE, JAL, JR, JUMP, LUI, LW, MFLO, OR, SH, SLL, SLLV, SLT, SLTI, SRA, SUBU, SW
134.perl	ADDIU, ADDU, ANDI, BEQ, BNE, JAL, JALR, JR, JUMP, LB, LH, LUI, LW, SB, SLL, SLT, SLTIU, SLTU, SW
147.vortex	ADDIU, ADDU, ANDI, BEQ, BLEZ, BNE, JAL, JR, LHU, LUI, LW, ORI, SLL, SLTU, SW
164.gzip	ADDIU, ADDU, BEQ, BNE, JAL, JR, JUMP, LBU, LHU, LUI, LW, ORI, SB, SH, SLTI, SLTIU, SLTU, SW
175.vpr-Place	ADDIU, ADDU, BC1F, BC1T, BEQ, BGTZ, BNE, C_LT_D, CVT_D_S, CVT_S_W, JAL, JR, JUMP, L_D, L_S, LUI, LW, MTC1, ORI, SLL, SLT, SLTI, SW
175.vpr-Route	ADDIU, ADDU, BC1F, BC1T, BEQ, BNE, C_EQ_D, C_LT_S, CVT_D_S, JAL, JR, L_D, LH, LUI, LW, MTC1, S_D, SLL, SLT, SW
177.mesa	ADDIU, ADDU, AND, ANDI, BEQ, BGTZ, BLEZ, BNE, JAL, JR, JUMP, LB, LBU, LHU, LUI, LW, NOR, ORI, SB, SLL, SLTI, SLTIU, SLTU, SRA, SRL, SUBU, SW, XORI
181.mcf	ADDIU, ADDU, BEQ, BGEZ, BLEZ, BNE, JAL, JR, JUMP, LUI, LW, SLT, SUBU, SW
183.equake	ADDIU, ADDU, ANDI, BC1F, BEQ, BLEZ, BLTZ, BNE, FADD_D, FMUL_D, JAL, JUMP, L_D, LB, LBU, LUI, LW, MFC1, MTC1, ORI, S_D, SLL, SLT, SLTI, SRL, SW
188.ammp	ADDIU, ADDU, ANDI, BC1F, BEQ, BLTZ, BNE, FMOV_D, FMUL_D, JAL, JR, JUMP, L_S, LBU, LHU, LUI, LW, MFC1, MTC1, ORI, SLL, SLT, SW
197.parser	ADDIU, ADDU, AND, ANDI, BEQ, BNE, JAL, JR, JUMP, LB, LHU, LUI, LW, NOR, ORI, SLL, SLTU, SRA, SUBU, SW

Table 8B: Characteristics of the Input Sets for the Top 100 Occurrences

Two conclusions can be drawn from Tables 8A and 8B. First of all, Table 8A confirms the conclusion drawn from Tables 4A, 4B, 6A, and 6B that a very small percentage of the input sets account for a disproportionately large number of the dynamic instructions.

Although Tables 4A, 4B, 6A, 6B, and 8A show that a very small percentage of the input sets account for a disproportionately large number of the dynamic instructions, not all the input sets in that small percentage will yield the same performance gain or will consume the same amount of area in the 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 input set 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 input sets for move instructions (move to or from the high or low register and to or from the integer or floating-point registers), 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 Tables 8A and 8B is that not all the instructions in the Top 100 occurrences can be reused or will have the same performance gain when reused.

4.5 Comparison of Global Level Value Locality in Integer and Floating-Point

After comparing the results of 177.mesa, 183.equake and 188.ammp against the other 13 benchmarks for Tables 4A, 4B, 6A, 6B, 7, and 8A, there does not appear to be any significant differences between the integer and floating-point benchmarks. One slight difference appears in Table 8B; for the floating-point benchmarks, input sets from floating-point instructions are in the Top 100 occurrences. 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 Traditional (Local) Value Reuse

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

1. Cumulative Percentage of the Occurrences/Frequency Product
2. Percentage of Instructions Covered by the Input Sets From the Top 100 Occurrences

4.6.1 Cumulative Percentage of the Occurrences/Frequency Product

For the cumulative percentage of the occurrences/frequency product, if there is more locality at the global level, then the global cumulative percentages should be lower, for a given frequency range, than that of the local cumulative percentages.

For convenience, Table 6B is restated below:

	Range								
Benchmark	**$< 10^1$**	**$< 10^2$**	**$< 10^3$**	**$< 10^4$**	**$< 10^5$**	**$< 10^6$**	**$< 10^7$**	**$< 10^8$**	**$< 10^9$**
099.go	0.1	0.6	3.7	31.0	63.0	78.4	100.0	100.0	100.0
124.m88ksim	4.9	5.5	7.2	17.5	38.1	71.6	100.0	100.0	100.0
126.gcc	1.8	7.0	24.7	46.1	63.1	80.2	90.6	100.0	100.0
129.compress	1.4	22.4	32.4	41.0	58.8	100.0	100.0	100.0	100.0
130.li	0.1	2.4	31.5	54.1	59.7	80.1	100.0	100.0	100.0
132.ijeg	3.6	15.6	38.7	57.2	76.0	89.5	97.0	100.0	100.0
134.perl	4.5	7.9	16.9	21.4	26.1	49.6	89.6	100.0	100.0
147.vortex	1.2	2.3	4.0	8.1	26.2	55.1	68.4	94.0	100.0
164.gzip	3.2	15.4	52.8	61.0	77.0	84.8	94.0	100.0	100.0
175.vpr-Place	3.7	4.0	5.3	13.6	45.2	85.2	100.0	100.0	100.0
175.vpr-Route	2.2	8.4	27.8	50.9	67.0	95.0	100.0	100.0	100.0
177.mesa	2.3	2.3	4.3	4.5	4.5	5.1	69.9	89.8	100.0

181.mcf	23.1	29.6	43.1	58.9	80.6	86.6	100.0	100.0	100.0
183.equake	3.4	12.2	15.3	17.4	31.2	70.5	92.9	100.0	100.0
188.ammp	1.9	3.3	3.9	17.7	43.9	63.4	90.8	100.0	100.0
197.parser	7.5	23.6	32.1	42.7	62.8	81.5	97.6	100.0	100.0

Table 6B (Restated): Cumulative Percentage of the Occurrences/Frequency Product, Global

Table 9A shows the cumulative percentage of the products for the local level while Table 9B shows the difference in the global and local cumulative percentages for each frequency range.

	Range								
Benchmark	**$< 10^1$**	**$< 10^2$**	**$< 10^3$**	**$< 10^4$**	**$< 10^5$**	**$< 10^6$**	**$< 10^7$**	**$< 10^8$**	**$< 10^9$**
099.go	0.8	5.5	27.3	58.3	81.1	92.2	100.0	100.0	100.0
124.m88ksim	5.2	5.8	9.2	20.0	42.9	75.3	100.0	100.0	100.0
126.gcc	9.0	19.2	37.0	56.2	79.2	89.9	92.7	100.0	100.0
129.compress	1.5	25.1	33.6	45.2	61.6	100.0	100.0	100.0	100.0
130.li	1.0	10.3	43.2	55.7	62.5	91.8	100.0	100.0	100.0
132.ijeg	4.4	23.4	47.1	64.7	83.0	94.1	97.0	100.0	100.0
134.perl	6.7	9.0	20.1	22.3	29.2	55.7	92.0	100.0	100.0
147.vortex	1.7	2.8	5.9	16.1	37.3	66.4	84.9	100.0	100.0
164.gzip	11.0	28.6	57.0	62.2	78.0	86.3	94.0	100.0	100.0
175.vpr-Place	3.7	4.1	7.5	30.5	61.8	93.3	100.0	100.0	100.0
175.vpr-Route	4.7	15.8	42.1	61.0	77.1	98.7	100.0	100.0	100.0
177.mesa	2.3	4.3	4.3	4.5	4.5	5.4	87.9	100.0	100.0
181.mcf	23.8	31.4	46.4	70.8	81.6	87.6	100.0	100.0	100.0
183.equake	6.9	13.2	15.4	17.6	36.5	82.0	94.8	100.0	100.0
188.ammp	1.9	3.3	5.1	30.4	47.6	69.9	100.0	100.0	100.0
197.parser	14.6	29.4	35.3	48.8	71.0	85.4	97.6	100.0	100.0

Table 9A: Cumulative Percentage of the Occurrences/Frequency Product, Local

	Range								
Benchmark	**$< 10^1$**	**$< 10^2$**	**$< 10^3$**	**$< 10^4$**	**$< 10^5$**	**$< 10^6$**	**$< 10^7$**	**$< 10^8$**	**$< 10^9$**
099.go	-0.7	-4.9	-23.6	-27.3	-18.1	-13.8	0.0	0.0	0.0
124.m88ksim	-0.2	-0.2	-2.0	-2.5	-4.8	-3.8	0.0	0.0	0.0
126.gcc	-7.2	-12.2	-12.3	-10.1	-16.0	-9.7	-2.0	0.0	0.0
129.compress	-0.1	-2.7	-1.2	-4.2	-2.8	0.0	0.0	0.0	0.0
130.li	-0.9	-8.0	-11.7	-1.6	-2.7	-11.7	0.0	0.0	0.0
132.ijeg	-0.8	-7.7	-8.4	-7.5	-7.0	-4.6	**0.0**	0.0	0.0
134.perl	-2.2	-1.1	-3.2	-0.9	-3.2	-6.1	-2.3	0.0	0.0
147.vortex	-0.5	-0.4	-1.9	-8.0	-11.1	-11.3	-16.5	-6.0	0.0
164.gzip	-7.8	-13.2	-4.2	-1.2	-1.0	-1.5	**0.0**	0.0	0.0
175.vpr-Place	**0.0**	-0.1	-2.2	-16.8	-16.6	-8.1	0.0	0.0	0.0
175.vpr-Route	-2.5	-7.5	-14.3	-10.1	-10.0	-3.7	0.0	0.0	0.0

177.mesa	**0.0**	-2.0	**0.0**	**0.0**	**0.0**	-0.2	-18.0	-10.2	0.0
181.mcf	-0.7	-1.8	-3.3	-11.9	-1.0	-1.1	0.0	0.0	0.0
183.equake	-3.5	-1.0	**0.0**	-0.2	-5.4	-11.5	-1.8	0.0	0.0
188.ammp	**0.0**	**0.0**	-1.2	-12.7	-3.7	-6.5	-9.2	0.0	0.0
197.parser	-7.1	-5.7	-3.2	-6.1	-8.2	-3.9	**0.0**	0.0	0.0

Table 9B: Difference in the Global and Local (Global – Local) Cumulative Percentages of the Occurrences/Frequency Product; Differences within 0.1% and for Global Cumulative Percentages under 100% are in Bold

As Table 9B clearly shows, the global level has a lower cumulative percentage at each frequency range, for global cumulative percentages less than 100%, for all benchmarks and frequency ranges except for 132.ijpeg: $< 10^7$; 164.gzip: $< 10^7$; 175.vpr – Place: $< 10^1$; 177.mesa: 10^1, 10^3, 10^4, and 10^5; 183.equake: 10^3; 188.ammp: 10^1 and 10^2; 197.parser: 10^7. The difference in cumulative percentages that are within 0.1% are shown in boldface type (only for global cumulative percentages below 100%). Note that the cumulative local level percentages can never be lower than the cumulative global level percentages for any given frequency range.

Therefore, as expected, the global cumulative percentages increase more slowly than the local cumulative percentages and for all benchmarks and frequency ranges, the global cumulative percentage is lower than (for most cases) or equal to the local cumulative percentage. Consequently, significantly more reuse possibilities exist at the global level as compared to the local level because the all input sets have higher frequencies.

4.6.2 Percentage of Instructions Covered by Input Sets From the Top 100 Occurrences

Table 10 compares the percentage of instructions due to the input sets from the Top 100 occurrences for both the global and local levels. The number of input sets in this table is the same for both the global and local levels and is the number of input sets in the Top 100 occurrences at the global level.

		Percentage of Total Instructions		
Benchmark	**Number of Input Sets**	**Global**	**Local**	**Difference (Global – Local)**
099.go	102	21.0	14.2	6.8
124.m88ksim	147	62.6	57.6	5.0
126.gcc	100	21.9	13.1	8.8
129.compress	169	51.9	48.0	3.9
130.li	135	40.7	36.0	4.7
132.ijeg	140	19.9	14.4	5.5
134.perl	161	34.6	27.1	7.5
147.vortex	114	40.6	28.2	12.4
164.gzip	126	21.5	19.7	1.8
175.vpr-Place	115	37.3	25.4	11.9
175.vpr-Route	121	35.6	26.4	9.2
177.mesa	229	86.9	76.4	10.5
181.mcf	117	22.7	21.2	1.5
183.equake	146	37.4	24.8	12.6

188.ammp	121	53.6	47.5	6.1
197.parser	109	25.7	21.7	4.0

Table 10: Percentage of Instructions Due to the Input Sets from the Top 100 Occurrences for Both the Global and Local Levels

As expected, the input sets for the Top 100 occurrences at the global level cover a higher percentage of instructions for all benchmarks as compared to the input sets 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.

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 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 input sets 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 redundant PC-independent input sets, then some of its entries are wasted – when compared to the global value reuse table. If there are very few unique input sets 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 value locality is a subset of the global level value locality (i.e. at the global level, fewer input sets account for all the dynamic instructions), 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, such as the instructions in Table 8B?

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 input sets account at least 65% of the dynamic instructions and for 8 of the 15 benchmarks – when 175.vpr is counted a single benchmark, less than 10% of the input sets account for over 90% of the dynamic instructions. Furthermore, 19.4% - 95.5% of the dynamic instructions are the results of less than 1000 of the most frequently occurring input sets.

For an equal number of input sets (approximately 100 for each benchmark) for both the global and local levels, the input sets for the global level account for an additional 1.5% to 12.6% of the total number of dynamic instructions. Therefore, more opportunity for reuse exists at the global level as compared to the local level.

7 Bibliography

[Burger 97] D. Burger and T. Austin; “The Simplescalar Tool Set, Version 2.0”; University of Wisconsin Computer Sciences Department Technical Report 1342.

[Gonzalez 98] 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

[KleinOsowski 00] 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.

[Molina 99] C. Molina, A. Gonzalez, and J. Tubella; “Dynamic Removal of Redundant Computations”; International Conference on Supercomputing, 1999

[Sodani 97] A. Sodani and G. Sohi; “Dynamic Instruction Reuse”; International Symposium on Computer Architecture, 1997.

[Sodani 98] A. Sodani and G. Sohi; “An Empirical Analysis of Instruction Repetition”; International Symposium on Architectural Support for Programming Languages and Operating Systems, 1998.

8 Acknowledgements

The authors thank Christian Hescott for implementing a B-tree that stored the input set information and dramatically reduced simulation time. This work was supported in part by National Science Foundation grant numbers CCR-9900605 and EIA-9971666, and by the Minnesota Supercomputing Institute.

9 Appendix – Amount of Repetition

Section 5 suggested the idea of selective value reuse, or in other words, only reusing the values for only certain types of instructions. One potential implementation of this idea is to have a separate table for each instruction type to be reused. However, this idea is more effective if there are relatively few input sets for those instruction types. Table 11 shows the number of input sets for a particular instruction type as a fraction of the number of dynamic instructions for ADDIU, ADDU, BEQ, BNE, and LW. These instruction types were the only reusable instruction types that were common to all the benchmarks and that were present in the Top 100 occurrences. The average amount of repetition is also shown. The entries that have a lower amount of repetition than the average (i.e. a higher number than the average) are in boldface type.

		Instruction Type				
Benchmark	**Average**	**ADDIU**	**ADDU**	**BEQ**	**BNE**	**LW**
099.go	0.00085	**0.00094**	**0.00103**	**0.00094**	0.00033	0.00048
124.m88ksim	0.03778	**0.07949**	0.03644	0.00059	0.01432	0.01722
126.gcc	0.00926	0.00739	**0.01907**	**0.01164**	**0.00954**	0.00514
129.compress	0.01912	**0.01946**	0.01614	0.00458	0.00226	0.00361
130.li	0.00231	**0.00421**	0.00095	0.00098	0.00200	0.00146
132.ijeg	0.02040	**0.02103**	**0.02380**	0.00066	0.00060	0.00465
134.perl	0.02955	0.02076	**0.03625**	0.02292	**0.03660**	0.01259
147.vortex	0.00581	**0.00959**	0.00484	0.00027	0.00484	0.00249
164.gzip	0.01654	0.00517	**0.02123**	0.00176	0.00270	0.00982
175.vpr-Place	0.03261	0.02960	0.00062	0.00035	0.00100	0.00048
175.vpr-Route	0.01350	**0.01742**	0.01034	0.00704	0.00681	0.00506
177.mesa	0.01703	0.01403	0.01587	0.00001	0.01474	0.00027
181.mcf	0.22252	**0.33165**	0.02858	0.00852	0.21953	0.01610
183.equake	0.01721	0.00620	**0.03483**	0.01243	0.00845	**0.01892**
188.ammp	0.01055	0.00592	0.00593	0.00213	**0.06983**	0.00036
197.parser	0.04285	0.02932	**0.10837**	0.00829	0.03322	0.02301

Table 11: Amount of Repetition (Input Sets/Dynamic Instructions for that Instruction Type) for Selected Instruction Types. Table Entries Above the Average are in Bold.

Note that from a value reuse perspective, a lower amount of repetition should produce greater performance gains.

Table 11 shows that for ADDIU, 8 of the 16 benchmarks have repetition amounts greater than the average while ADDU, BNE, BEQ, and LW have 7, 2, 3, and 1 benchmarks, respectively. From another point-of-view, 2 benchmarks have 5 instruction types, 9 benchmarks have 4 instruction types, 3 benchmarks have 3 instruction types, and 2 benchmarks have 2 instruction types with a lower repetition amount than the average.

Therefore, given that the repetition amounts for some of the instruction types are higher than the average, partitioning a value reuse table into smaller tables based on the instruction type is probably not a good idea. Since these instruction types account for a significant percentage of

the dynamic instructions, their higher-than-average repetition amounts could lead to a higher replacement rate within the value reuse table, which consequently could reduce the performance.

Table 12 shows the repetition amounts for the top 4, 8, and 16 instruction types, sorted by number of dynamic instructions.

	Instruction Types			
Benchmark	**4**	**8**	**16**	**Instruction Types**
099.go	0.00070	0.00074	0.00083	LW, ADDU, ADDIU, SLL, LUI, SW, BNE, BEQ, SLT, JR, JAL, JUMP, SLTI, SUBU, BLEZ, SLTIU
124.m88ksim	0.06225	0.04507	0.03815	LW, SW, ADDIU, ADDU, BNE, BEQ, LUI, SRL, OR, SLL, ANDI, AND, JR, SLTI, JAL, SLTU
126.gcc	0.00942	0.00901	0.00841	LW, ADDIU, SW, ADDU, BNE, BEQ, SLL, LHU, LUI, JR, JAL, SLTIU, SLT, LB, JUMP, SLTI
129.compress	0.01394	0.01279	0.01311	ADDIU, LW, SW, ADDU, LBU, BNE, BEQ, SB, SLL, LUI, JR, JAL, SLT, SLLV, SLTI, BGTZ
130.li	0.00261	0.00246	0.00230	LW, SW, ADDIU, ADDU, BEQ, BNE, LBU, JR, JUMP, ANDI, JAL, SLL, SB, SLTU, SRA, SLTIU
132.ijeg	0.01662	0.01622	0.01617	ADDU, ADDIU, LW, SLL, LBU, SUBU, BNE, SW, SRA, SB, SLTI, BEQ, MFLO, MULT, JUMP, OR
134.perl	0.02206	0.02415	0.02482	LW, ADDIU, SW, ADDU, BEQ, BNE, JR, LBU, SB, SLL, BGTZ, JAL, JUMP, SLTIU, LB, LUI
147.vortex	0.00549	0.00491	0.00546	LW, SW, ADDU, ADDIU, BEQ, BNE, JR, SLL, JAL, ANDI, SLTU, LHU, LUI, SB, LBU, JUMP
164.gzip	0.00784	0.00991	0.01546	ADDIU, ADDU, LBU, BNE, LW, BEQ, SLTU, SLL, LHU, ANDI, XOR, SW, SH, SRL, JUMP, LUI
175.vpr-Place	0.00466	0.00355	0.00812	LW, ADDU, SLL, ADDIU, SW, BEQ, SLT, BNE, L_S, BGTZ, MTC1, JUMP, FMUL_S, S_S, JR, JAL
175.vpr-Route	0.00733	0.01284	0.01148	LW, ADDU, SW, SLL, ADDIU, L_S, BNE, SLT, C_LT_S, LH, BEQ, JR, JAL, BC1T, BC1F, MTC1
177.mesa	0.00915	0.01324	0.01512	ADDIU, ADDU, BEQ, LW, BNE, LBU, SW, LB, JUMP, SB, ANDI, SLTU, SUBU, SLL, LUI, SRA
181.mcf	0.29958	0.24564	0.22545	LW, SW, ADDIU, ADDU, BNE, BEQ, SLT, SUBU, BGEZ, BLEZ, SLTU, SLL, LUI, JUMP, SLTI, SRA
183.equake	0.01918	0.01789	0.01751	ADDIU, ADDU, LW, L_D, BNE, BEQ, SW, SLL, FMUL_D, FADD_D, S_D, SLTI, L_S, MTC1, LUI, ANDI
188.ammp	0.01360	0.01108	0.01008	LW, BNE, ADDIU, ADDU, SW, JUMP, BEQ, SLT, LB, L_S, ANDI, SLL, LBU, S_S, JR, JAL
197.parser	0.04525	0.03850	0.04145	LW, ADDIU, ADDU, BNE, SW, LB, BEQ, SLL, SLT, LBU, SUBU, JR, JAL, JUMP, AND, ANDI

Table 12: Amount of Repetition for the Top 4, 8, and 16 Instruction Types, Sorted by the Number of Dynamic Instructions; Instruction Types are Given in Descending Order of Number of Dynamic Instructions

For all three numbers of instruction types, the highest amount of repetition (smallest fraction) is for 099.go while the lowest amount of repetition (highest fraction) is for 181.mcf. For all the benchmarks except for 099.go and 181.mcf, the ratio of input sets to dynamic instructions is on the order of 1:100 or 1:1000; for 099.go and 181.mcf, the ratio of input sets to dynamic instructions is on the order of 1:10,000 and 1:1, respectively. Therefore, with the exception of 181.mcf, there is a very high amount of repetition for all the benchmarks.

Furthermore, from Table 12, there does not appear to be any discernable pattern to the amount of repetition as more instruction types are included.