Experiments in Data Colocation

Clark L. Coleman
Jack W. Davidson
University of Virginia Computer Science Department
Research Memorandum RM-98-01

1.0 Background and Purpose

In current compilers, the definitions and uses of objects are determined during the analysis phases of code generation. During optimization, this information is used, for example, in register allocation. If two objects have "live ranges" that do not overlap, they can be assigned to the same register, decreasing register pressure. It seems that a similar thing could be done with global and/or local objects which do not have overlapping live ranges: namely, they could be assigned to the same memory locations, which could decrease certain pressures in the memory hierarchy of the computer system.

2.0 Hypothesis

Colocation of objects should lead to a measurable decrease in memory traffic due to measurable decreases in misses in various levels of the memory hierarchy (L1 cache, L2 cache, TLB, etc.) By carefully choosing the sizes of two arrays with respect to some particular element of the memory hierarchy, we should be able to measure the run-time effect of the benefits of colocating those two arrays as contrasted with assigning them to different memory addresses.

3.0 Experimental Procedure

- The memory hierarchy parameters (L1 cache sizes and replacement policies, ditto for L2 cache and TLB) will be determined for a variety of machines available for public experimentation in the department.
- Synthetic benchmark programs will be devised that will perform writes to each element in two arrays. The two arrays will be of the same size, which will be varied in different runs of the experiment to equal the size of the L1 D-cache, the L2 cache, the amount of memory mapped by the TLB, and perhaps the amount of main memory available on the machine.
- Each version of the benchmark program will be compiled so as to produce assembly language output. A copy of the assembly language output of each version of the benchmark program will be modified to colocate the two arrays. Two executables will then be produced from the two versions of the assembly language.

- Multiple timing runs will be made for each executable, and the Real, User and System times will be recorded and averaged. Comparison of these times for colocated and non-colocated versions of the same code will be made.

NOTE: The benchmark programs will consist primarily of stores into the arrays, not reads of array elements. This is because we are trying to simulate the following real-world situation: A program initializes a large array, then uses it for various computations. After the initialization, the array is in the data caches of the system to the extent that it will fit, and the reads to it are then primarily hits in the cache. Some time later, another array is used, and its use does not overlap the first array in time. This array must be written to before it is read, else the initial reads would be of undefined values. So, writes to the array are performed first, which will be cache misses. In a write-back, write-allocate cache, the write misses will cause write-backs of whichever lines will be replaced in the cache (if they are dirty), and then reads (write-allocates) of the cache lines for the second array. If the second array had been colocated with the first array, then the initialization of the second array would have produced write hits, so no write-backs nor write-allocates will occur (if the first array is still in the cache.)

Thus, the ideal situation for colocation is to have the first array be entirely within the element of the memory hierarchy that we are measuring, and have the references to the second array be 100% hits. Colocation will never decrease read misses if a program does not read undefined data, but it can decrease write misses. We seek to measure the benefits of decreasing write misses through data colocation.

4.0 Expected Results

The run times will show significant decreases in elapsed and CPU times when colocation decreases the write misses in any element of the memory hierarchy, as memory systems are generally slow in comparison to CPUs, and extra misses and extra traffic anywhere in a memory hierarchy should produce measurable results.

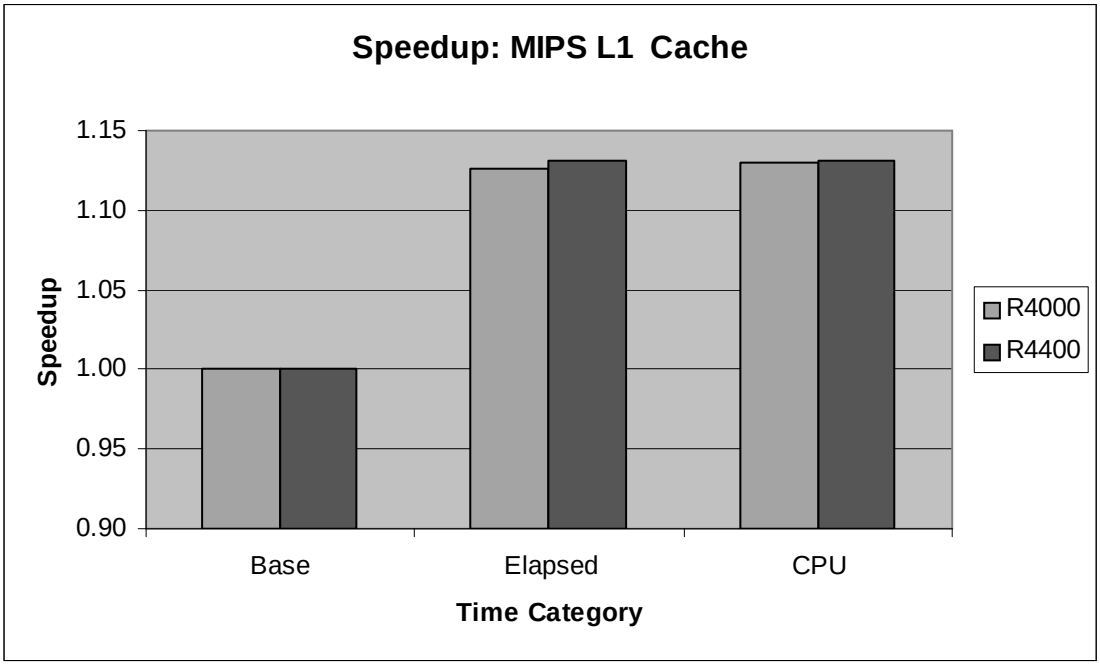
5.0 Results

5.1 L1 Cache Results

The first experiments were done to measure the effects of colocation within the L1 cache of a Sun SuperSparc-I CPU, as found on the system `cobra.cs.virginia.edu`. This CPU has a 16 KB L1 data cache, 4-way set associative, with 32-byte lines, 128 lines (with four sets per line), with a write policy of write-through, no-write-allocate. The L1 instruction cache is a separate 20 KB, 5-way set associative cache. Accordingly, two arrays of size 16 KB were declared in the test program. In the colocated version of the program, the assembly language output from the compiler was modified to colocate these two arrays, while the two arrays were separate declarations in the non-colocated version. No statistically significant differences in elapsed or CPU time were seen between the colocated and non-colocated versions of the program.

The failure to achieve significant results on this experiment were explained by noting that the L1 D-cache write policy of the SuperSparc is write-through, no-write-allocate. Having learned empirically that decreasing write misses would have no measurable effect on such a cache, we confined all further experiments to machines with write-back, write-allocate cache policies.

Our first such machine was `sutherland.cs.virginia.edu`, a 100 MHz MIPS R4000 workstation with an 8KB direct-mapped L1 instruction cache; a separate 8KB direct-mapped data cache, write-back, write-allocate; a 1MB L2 unified cache with the same write policy as the L1 data cache, and 64MB of main memory. Results from this machine can be compared with the very similar results from our second MIPS machine, `dis-ney.cs.virginia.edu`, a 150 MHz R4400 with 16KB split primary caches (and the same as `sutherland` in the other elements of the memory hierarchy.) (See the Appendix for the actual code used in these experiments.) The speedup achieved through colocation for these two machines can be seen in the figure below.



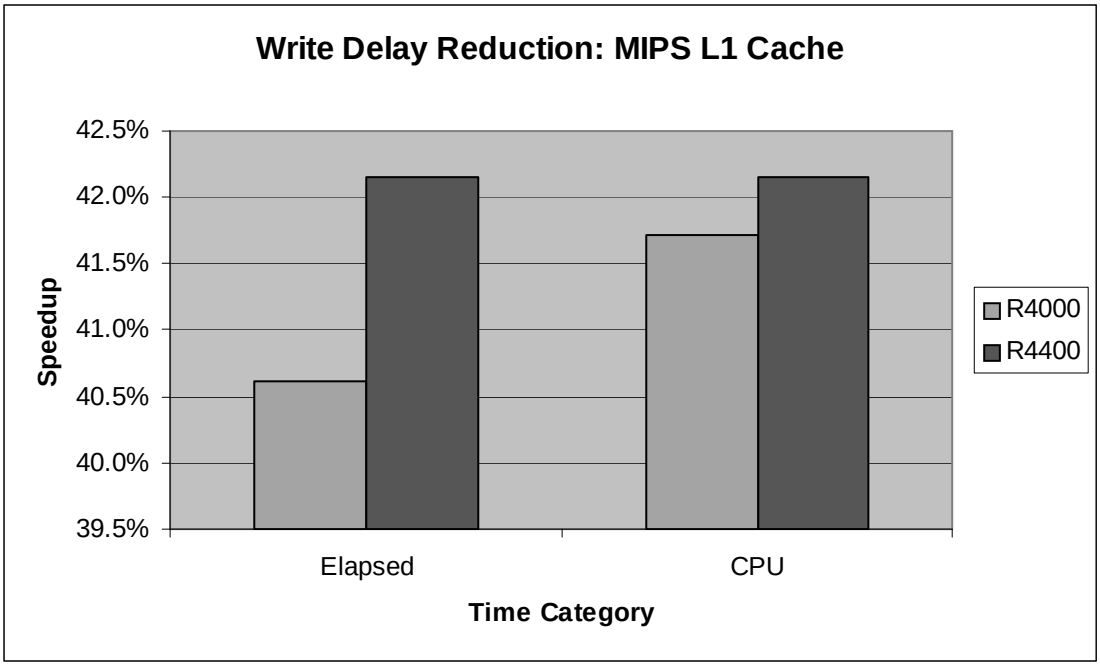
A note on terminology: The term *reduction* refers to the percentage of the non-colocated program's time that has been eliminated by colocation. The term *speedup* refers to the ratio of the non-colocated program's time to the colocated program's time. For example, if a 100 second time is reduced to 80 seconds by the colocation of the arrays, we say that we have a 20% *reduction* and a *speedup* of 1.25. This is the terminology used in the famous Hennessy and Patterson architecture books.

In this experiment, the colocation only reduced the run time of the program by a little more than 11%, giving a speedup figure of 1.13 for each MIPS machine tested. Our expec-

tations were that the reduction of write misses would lead to more significant time reductions than we achieved.

Some serious reflection produced an explanation for the mismatch between our expectations and the results on the MIPS machines. We had expected that the run times of these programs would be dominated by the time required to service cache misses. Our results caused us to suspect that the program run times were not dominated by the numerous writes to the two arrays, but were instead dominated by CPU cycles spent on loop overhead, address arithmetic, etc. To test this conjecture, a copy was made of the colocated assembly code on each MIPS machine. The copy was modified in only two assembly language statements: the store statements to each of the arrays, which were modified to merely store into an unused register. Timing runs were then performed on this CPU-intensive code. These timings revealed that about 75% of our time was being spent in CPU operations, and less than 25% in our memory writes. The high performance of the memory systems in these machines surprised us.

Accordingly, we decided to measure the reduction in the portion of the elapsed time that was attributable to our memory writes. The time reductions we had achieved were divided by the approximately 25% of the elapsed and CPU times that were attributable to memory writes. The reduction of these write delays is shown in the chart below.

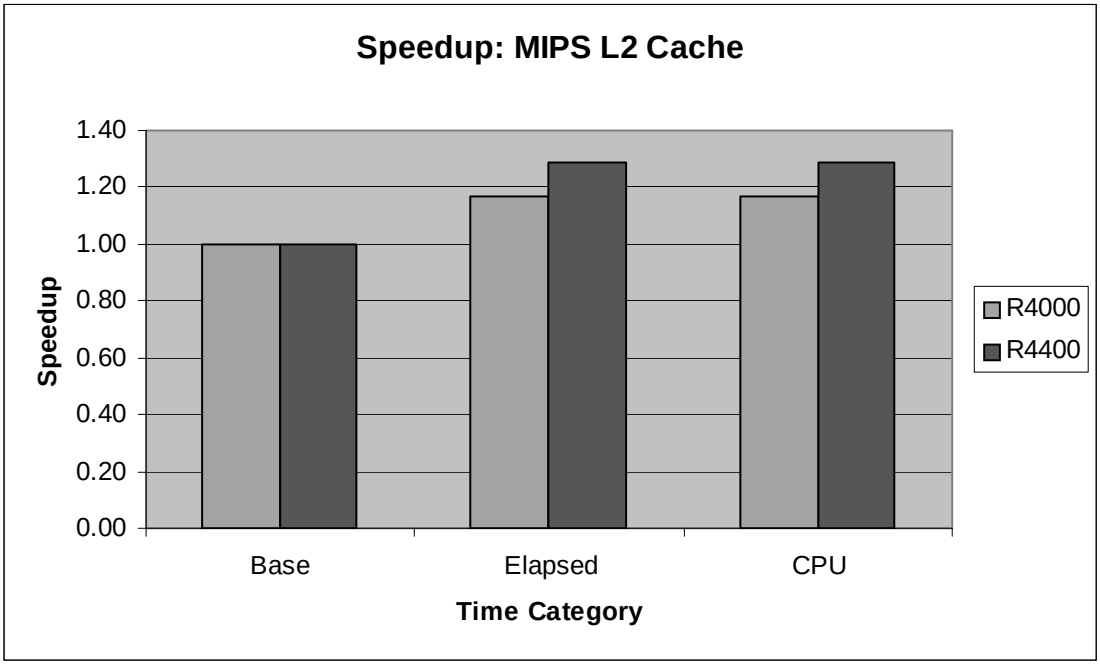


This computation indicates that we reduced more than 40% of the portion of the run time that we had any chance to reduce by colocation. As the two arrays in the non-colocated version of our test program would both reside in the L2 cache without conflict, despite the fact that they conflict in the L1 D-cache, and both arrays can easily fit in the amount of memory that the MIPS TLB can map, there will be no improvement in the portion of the

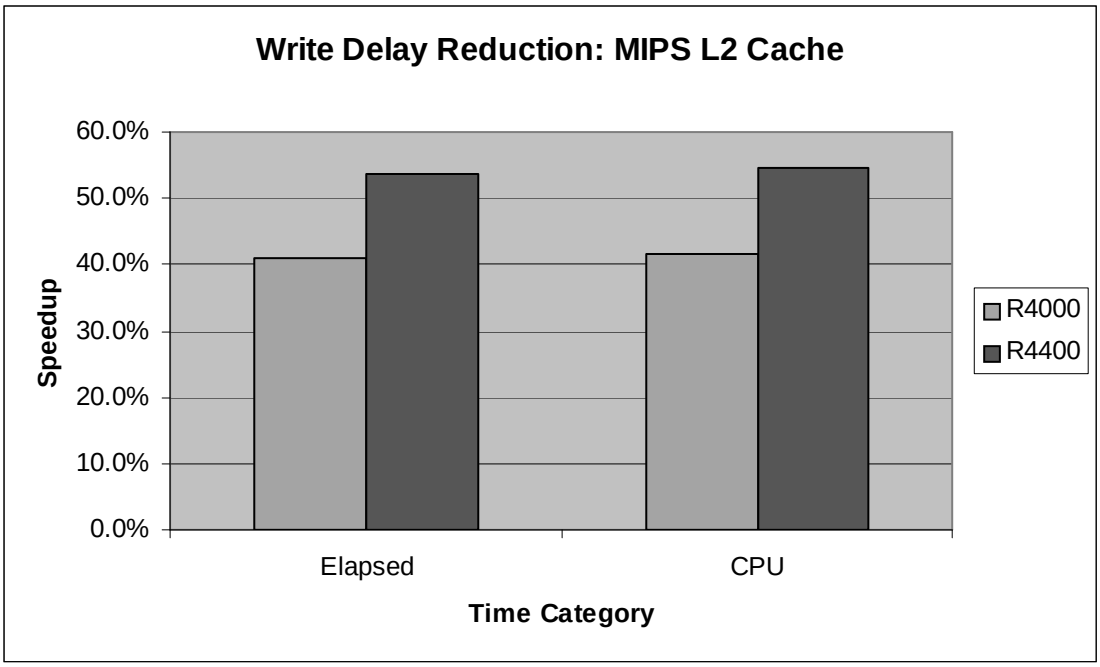
write delay associated with TLB and L2 cache. Thus, not even all of the approximately 25% of the run time attributed to write delays is available for improvement due to colocation.

5.2 L2 Cache Results

Scaling the arrays up to the size of the L2 caches of these machines produces the results shown in the chart below.

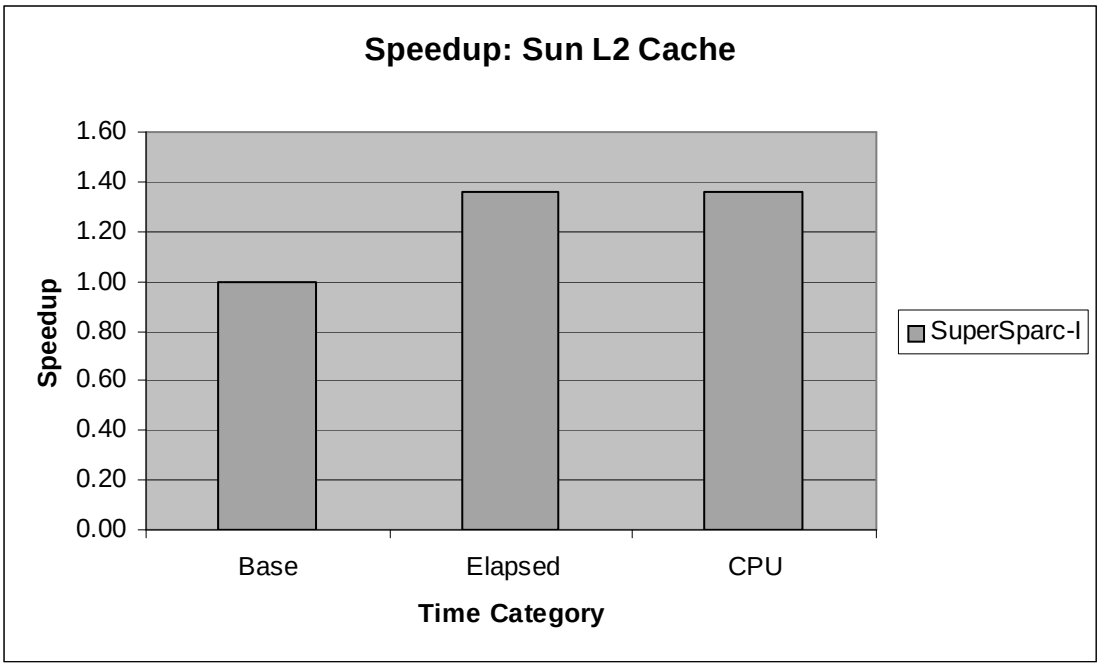


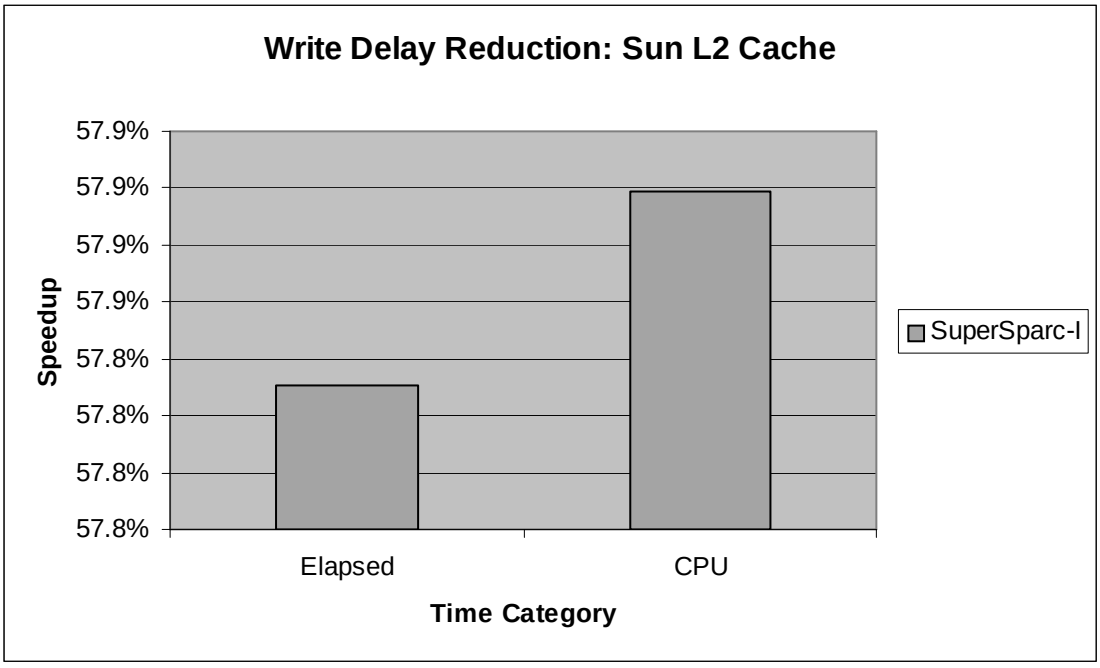
The R4400 speedup is significantly greater than the R4000 speedup. Both numbers are less than our original expectations, for reasons explained in the previous section. The corresponding numbers for reduction in the write delay are given in the chart below.



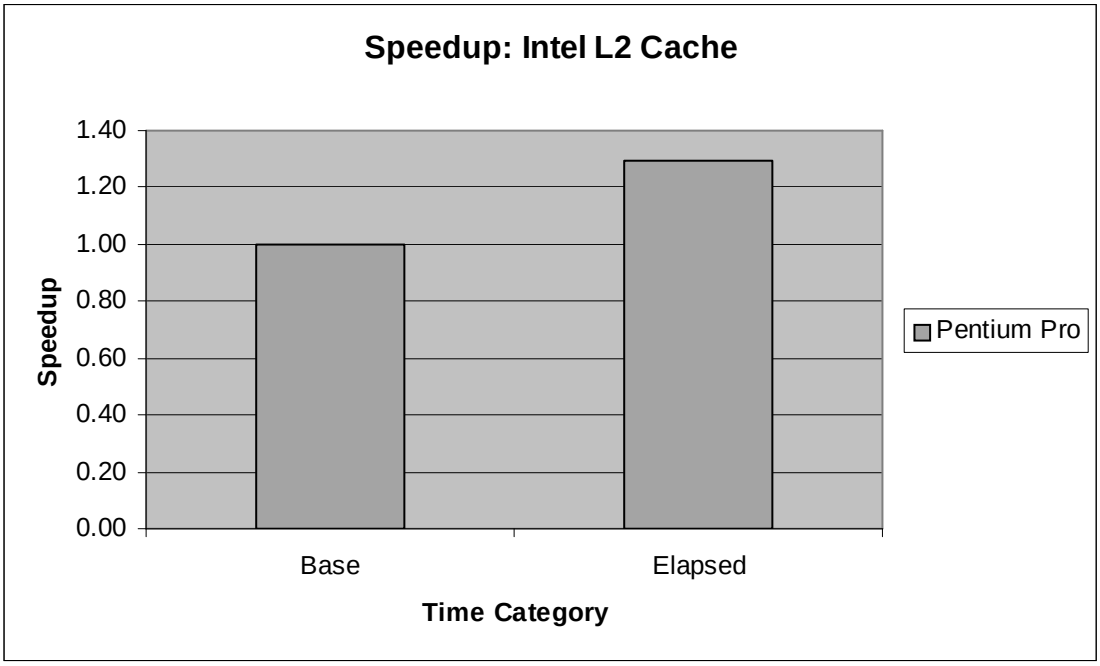
The marked difference in speedups for L2-cache-sized arrays for the two MIPS machines can be explained as follows.The R4400 showed a very significant improvement in speedup compared to its L1 results (1.29 versus 1.13 for both Real and User seconds.) The R4000 had a rather minuscule improvement in speedup (from 1.13 to 1.17.) This can be explained by looking at the ratio of L2 cache size to L1 D-cache size, which is 64 for the R4400 system and 128 for the R4000 system. Thus, cycling through an array that fills the L2 cache will cause twice as many L1 cache misses on the R4000 as it will on the R4400. Colocation of the two L2-sized arrays will not have any effect on L1 misses in either system, as we hit each element of the array sequentially and do not reuse any element. By the time we see a given array element again on the next iteration of the outer loop (see the code in the appendix), we have long since flushed it from the L1 cache. So, the time consumed by either program is about 75% CPU-intensive, and the other 25% is divided among the servicing of references to the L1 cache, TLB, and L2 cache. The L1 cache service time is much greater on the R4000 system than on the R4400 system, leaving proportionately less time in the servicing of L2 misses, and thus leaving less potential for speedup from a colocation optimization that is only reducing L2 misses. (A similar argument will show that TLB misses are identical between the two systems, as the L2 cache is much larger than the amount of memory mapped by the TLB.)

Running the L2 experiments on the Sun machine `cobra.cs.virginia.edu` produced results that were somewhat more impressive than either MIPS system. As noted earlier, this system has split L1 caches (20KB I-cache, 16KB write-through, no-write-allocate D-cache), but has a 1MB unified L2 cache with a write policy of write-back, write-allocate. Despite the fact that it is not suffering write-backs and write-allocates on its L1 D-cache in our experiments, this 50MHz SuperSparc-I system gets a greater speedup from colocating the L2-sized arrays than the R4400 system with identical L1 D-cache and L2 cache sizes.


Speedup: Sun L2 Cache
1.60
1.40
1.20
1.00
0.80
0.60
0.40
0.20
0.00
Speedup
Base
Elapsed
CPU
Time Category
SuperSparc-I


Write Delay Reduction: Sun L2 Cache
57.9%
57.9%
57.9%
57.9%
57.8%
57.8%
57.8%
57.8%
Speedup
Elapsed
CPU
Time Category
SuperSparc-I

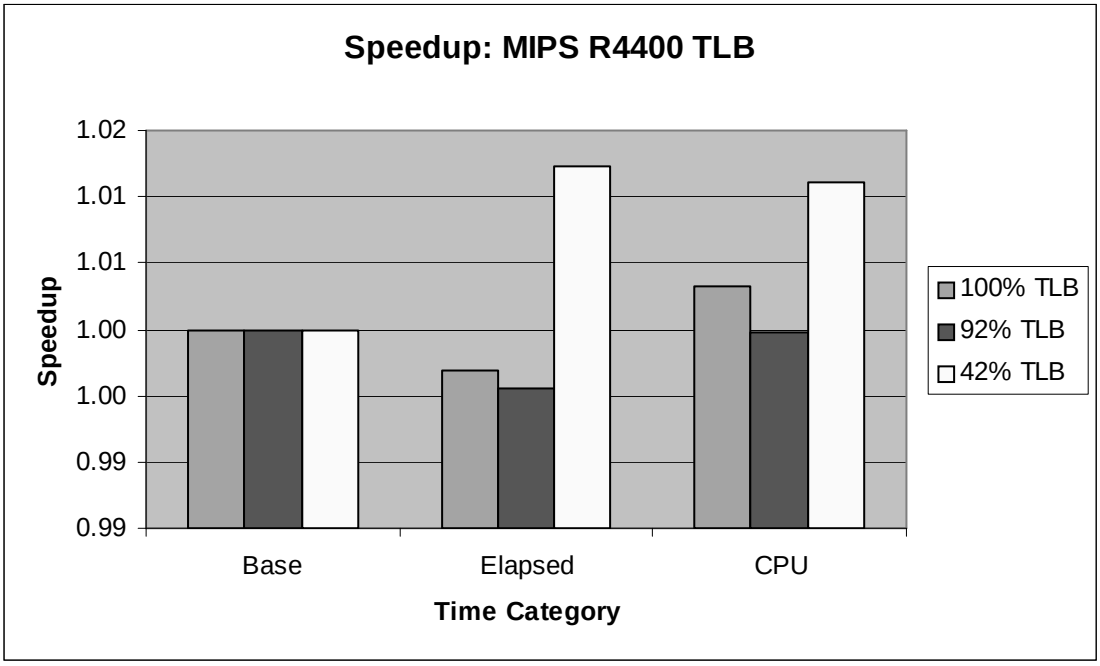
Similarly, the L2 experiments on the 200 MHz Pentium Pro PC produced a speedup of 1.30 versus 1.29 for the R4400, despite having an L2-to-L1 cache size ratio of only 32 (256KB to 8KB) compared to the R4400's ratio of 64. Only elapsed times are available for the PC experiments, due to operating system differences from the Unix environment in which all other data were gathered.



5.3 TLB Results

The MIPS R4000/R4400 documentation describes the TLB as mapping “48 pairs” of virtual addresses. Presumably, this means 96 virtual addresses, with the TLB laid out as 48 lines, with two virtual addresses in each line, per various statements in the documentation. The possibility exists that the second address in each pair is not an independent address, perhaps being used to map the page limit when variable-sized pages are turned on by supervisor code. (Normally, the pages are 4KB each.) In the one case, the TLB maps 384KB of memory (code and data unified); in the other case, it maps 192KB. Several runs of the program tested these limits without revealing any potential for significant speedups by reducing TLB misses, as the chart below indicates.

The first run used 384KB arrays. The second run used 352KB arrays, leaving 32KB of TLB-mapped memory to hold code pages (the test executable is less than 13KB in size, and some operating system code will always be resident.) The third run used 160KB arrays, leaving ample space for code pages if 96 pages are mapped by the TLB, and still leaving 32KB for code if only 48 pages are mapped. The improvements in system time spent servicing TLB misses (the MIPS chips have software TLB handlers, not hardware miss handlers) were trivial and did not cause any measurable improvement in elapsed time.



6.0 Conclusions

- Colocation can reduce write misses and improve performance in systems with suitable caches. L2 caches seem to be able to service L1 cache misses efficiently; only by affecting L2 miss rates can significant performance improvements occur.
- Colocation will not lead to measurable improvements unless the system includes write-back, write-allocate caches at some level of the cache hierarchy.
- TLB performance on programs that access arrays sequentially is a non-factor in overall performance.
- Modern memory systems are surprisingly efficient at masking various latencies throughout the memory hierarchy. Different systems are measurably superior to others in this regard.
- As CPU speeds are increasing at a faster rate over time than memory speeds, cache miss penalties are increasing. Therefore, the potential speedup from colocation will increase over time. (This is also true of the potential for data placement, as discussed in research memorandum RM-98-02.)

7.0 Future Work

- Determine if there are common programs that access arrays non-sequentially and could thrash a TLB so much that significant slowdowns occur in real programs.
- Devise def-use-chain based code for `vpo` that will detect potential colocation possibilities, and gather data from real benchmarks concerning its usefulness.
- Extend arrays to main memory sizes and gather more data.
- Gain access to additional machines (e.g. DEC, HP, IBM) and gather more data.

Appendix A: Code Used in Experiments.

The code below is a typical example of the C code used in all of these experiments. By merely changing the definition of constant `ARRAY_SIZE`, the code can be adapted to different elements in the memory hierarchy. Similarly, by changing the definition of constant `L1_DATA_CACHE_SIZE` or `L2_DATA_CACHE_SIZE`, the code can be adapted to a new system.

```
#define PAGE_SIZE  4096
#define L1_DATA_CACHE_SIZE 8192
#define L2_DATA_CACHE_SIZE 1048576L
#define TLB_MAPPING_SIZE  (48 * PAGE_SIZE)
#define ARRAY_SIZE ((TLB_MAPPING_SIZE - (8 * PAGE_SIZE)) / sizeof(int))
#define LOOP_LIMIT 30000L

static int A[ARRAY_SIZE];
static int B[ARRAY_SIZE];

int main() {
   long i, j;

   for (i = 0L; i < LOOP_LIMIT; ++i) {
      for (j = 0L; j < ARRAY_SIZE; ++j) {
         A[j] = 2;
      }
      for (j = 0L; j < ARRAY_SIZE; ++j) {
         B[j] = 3;
      }
   }

   return 0;

}
```

Appendix B: Raw Timing Data.

R4000 L1 D-cache-sized Arrays; Outer Loop count = 300,000

	Non-colocated				Colocated		
	Real	**User**	**System**		**Real**	**User**	**System**
	239.95	237.70	0.41		213.93	210.39	0.31
	239.76	237.70	0.36		212.09	210.38	0.33
	239.88	237.71	0.38		212.58	210.38	0.35
	240.55	238.04	0.57		213.66	210.40	0.33
					213.00	210.65	0.35
average:	240.04	237.79	0.43		213.05	210.44	0.33
				reduction:	11.2%	11.5%	22.3%
				speedup:	1.13	1.13	1.29
write delay	66.43	65.61	0.19		39.45	38.26	0.09
				reduction:	40.6%	41.7%	51.4%
				speedup:	1.68	1.71	2.06

R4400 L1 D-cache-sized Arrays; Outer Loop count = 300,000

	Non-colocated				Colocated		
	Real	**User**	**System**		**Real**	**User**	**System**
	319.01	316.49	0.65		281.47	279.73	0.45
	318.20	316.35	0.49		281.54	279.75	0.56
	318.08	316.20	0.49		281.74	279.79	0.46
average:	318.43	316.35	0.54		281.58	279.76	0.49
				reduction:	11.6%	11.6%	9.8%
				speedup:	1.13	1.13	1.11
write delay	87.41	86.73	0.19		50.57	50.14	0.14
				reduction:	42.2%	42.2%	27.6%
				speedup:	1.73	1.73	1.38

R4000 L2 D-cache-sized Arrays; Outer Loop count = 3000

	Non-colocated				Colocated		
	Real	**User**	**System**		**Real**	**User**	**System**
	366.93	360.26	1.17		314.63	308.38	1.06
	365.59	360.24	1.16		313.10	308.35	1.05
	365.63	360.25	1.16		312.99	308.31	1.05
average:	366.05	360.25	1.16		313.57	308.35	1.05
				reduction:	14.3%	14.4%	9.5%
				speedup:	1.17	1.17	1.10
write delay	128.17	124.18	0.84		75.70	72.28	0.73
				reduction:	40.9%	41.8%	13.1%
				speedup:	1.69	1.72	1.15

R4400 L2 D-cache-sized Arrays; Outer Loop count = 3000

	Non-colocated				Colocated		
	Real	**User**	**System**		**Real**	**User**	**System**
	270.12	266.15	1.28		211.58	206.62	1.18
	270.71	266.18	1.33		209.79	206.46	1.19
	270.32	266.22	1.30		208.93	206.15	1.18
average:	270.38	266.18	1.30		210.10	206.41	1.18
				reduction:	22.3%	22.5%	9.2%
				speedup:	1.29	1.29	1.10
write delay	112.05	108.76	1.07		51.77	48.99	0.95
				reduction:	53.8%	55.0%	11.3%
				speedup:	2.16	2.22	1.13

SuperSPARC L2 D-cache-sized Arrays; Outer Loop count = 3000

	Non-colocated				Colocated		
	Real	**User**	**System**		**Real**	**User**	**System**
	473.63	470.80	0.29		344.70	342.70	0.23
	472.73	469.92	0.25		347.78	345.71	0.21
	475.00	472.57	0.26		351.41	349.11	0.21
average:	473.79	471.10	0.27		347.96	345.84	0.22
				reduction:	26.6%	26.6%	18.8%
				speedup:	1.36	1.36	1.23
write delay	217.57	216.22	0.21		91.75	90.96	0.16
				reduction:	57.8%	57.9%	24.2%
				speedup:	2.37	2.38	1.32

Pentium Pro L2 D-cache-sized Arrays; Outer Loop count = 3000

	Non-colocated				Colocated		
	Real	**User**	**System**		**Real**	**User**	**System**
	165				111		
	164				118		
	168				131		
	158				135		
	162				136		
average:	163.75				126.20		
				reduction:	22.9%		
				speedup:	1.30		

R4400 TLB-sized Arrays; Outer Loop count = 3000

	Non-colocated				Colocated		
	Real	**User**	**System**		**Real**	**User**	**System**
	78.81	77.54	0.52		78.83	76.46	0.27
	77.48	76.63	0.27		79.39	78.09	0.42
	78.40	77.08	0.44		77.21	76.26	0.24
average:	78.23	77.08	0.41		78.48	76.94	0.31
				reduction:	-0.3%	0.2%	24.4%
				speedup:	1.00	1.00	1.32

R4400 TLB-sized Arrays, 8 code pages reserved; Outer Loop count = 3000

	Non-colocated				Colocated		
	Real	**User**	**System**		**Real**	**User**	**System**
	71.12	70.44	0.23		71.60	70.35	0.35
	72.05	71.01	0.44		71.66	70.61	0.23
	70.85	70.15	0.20		71.73	70.71	0.27
average:	71.34	70.53	0.29		71.66	70.56	0.28
				reduction:	-0.5%	0.0%	2.3%
				speedup:	1.00	1.00	1.02

R4400 50% TLB-sized Arrays; Outer Loop count = 30,000

	Non-colocated				Colocated		
	Real	**User**	**System**		**Real**	**User**	**System**
	323.39	321.30	0.64		321.95	317.41	0.69
	328.84	321.81	2.00		319.90	317.22	0.67
	319.89	317.06	0.58		318.45	316.30	0.52
average:	324.04	320.06	1.07		320.10	316.98	0.63
				reduction:	1.2%	1.0%	41.6%
				speedup:	1.01	1.01	1.71