Experiments in Data Placement

Clark L. Coleman
Jack W. Davidson
University of Virginia Computer Science Department
Research Memorandum RM-98-02

1.0 Background and Purpose

The placement of variables in the data space of a program can affect the program's dynamic memory performance. In particular, a poor placement of data can reduce the spatial locality of the data address references generated by the program. This experiment seeks to measure the range of effects of data placement on overall run-time performance. Since current compilers assign addresses to user-declared variables in the order in which the declarations were encountered in the source program text, it may be possible to improve run-time performance through memory-hierarchy-conscious data placement.

2.0 Hypothesis

Data placement can have a significant, measurable effect on the run-time performance of a program. Variable placements that maximize cache-line reuse and minimize cache conflicts yield programs that run measurably faster than those with variable placements that are suboptimal.

3.0 Proposed Experimental Procedure

- The data cache structure of the target machine, `cobra.cs.virginia.edu`, will be determined from Sun Microsystems documentation. Cache size, associativity, and line size are the key parameters to be determined.
- Using this cache information, a synthetic benchmark will be devised that exhibits high spatial locality. This will be accomplished by using user-declared variables that fill the data cache and exercise it thoroughly while maximizing reuse of loaded cache lines. Small arrays of exactly one cache line in size will be declared, each followed by a larger array that consumes the remaining lines in the first level data cache. This pattern of declarations will be repeated until there are enough small arrays to fill up all of the sets available for a given line. Then some scalar variables, collectively taking up one cache line in size, will be declared. This positions ensuing data placement on the next line after the cache line that has been filled with small arrays. Then more small and large arrays will be declared as in the first set of declarations. The program code will then heavily exercise the small arrays, which can all fit in the first level data cache at the same time. By avoiding references to the larger arrays (which just filled up cache lines)

and the scalars, we will avoid cache conflicts. Timing runs will be made in the evening when the machine is lightly loaded. Mean times will be computed from several timing runs for each form of the experiment.

- A copy of the benchmark program will be made with the variable declarations rearranged to ruin the spatial locality and minimize the reuse of data cache lines. This can be easily accomplished by moving the scalar variable declarations down below the second set of array declarations. This will cause the small arrays of the program to conflict with one another, and only half as many sets are available as would be required to avoid the conflicts. Loaded data cache lines will be flushed out of the cache by a deliberate reference sequence to create cache line conflicts. The total data space consumption of the program will remain the same as in the high-spatial-locality version, as will the total number of computations performed. Timing runs will be made immediately after the timing runs of the original program. (See the code samples in the Appendix for the actual data declarations and reference sequence.)

4.0 Expected Results

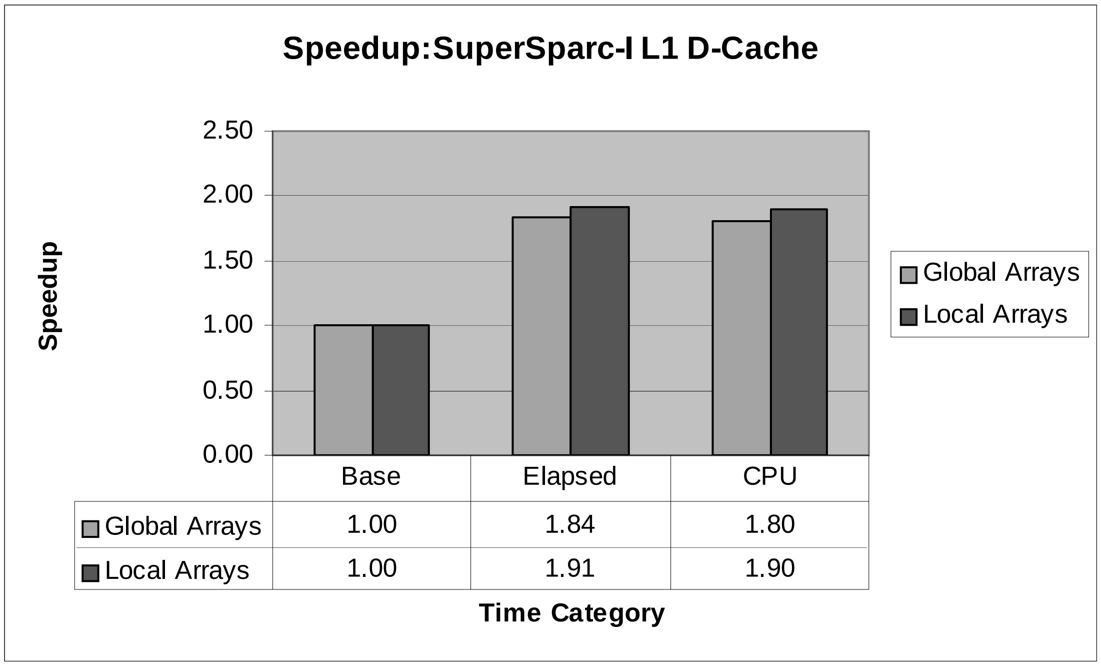
The run times will show the effects of lessened data cache hit rates, with a significant increase in run times for the second form of the program despite a constant number of computations performed. This will confirm the importance of data placement.

5.0 Results

Two versions of the experiment were run. The first version used global variables declared before the `main()` function of the test program. The second version used the same variables, but declared as locals to `main()`. Some tinkering with assembly code, and printing of resulting addresses of variables to confirm their placement, eventually produced test executables that produced similar results for both the global variable runs and the local variable runs. The `gcc` compiler was used for the first run (using globals) and `vpo` was used for the second run (using locals).

Using various sources, it was determined that the target machine, `cobra.cs.virginia.edu`, possesses a level-one data cache (L1 D-cache) of 16 KBytes, 4-way set associative, with 32 byte lines, and hence has 128 lines (128 lines * 4 sets/line * 32 bytes/set = 16KB). The code was written with these parameters in mind. (See the Appendix for code details.)

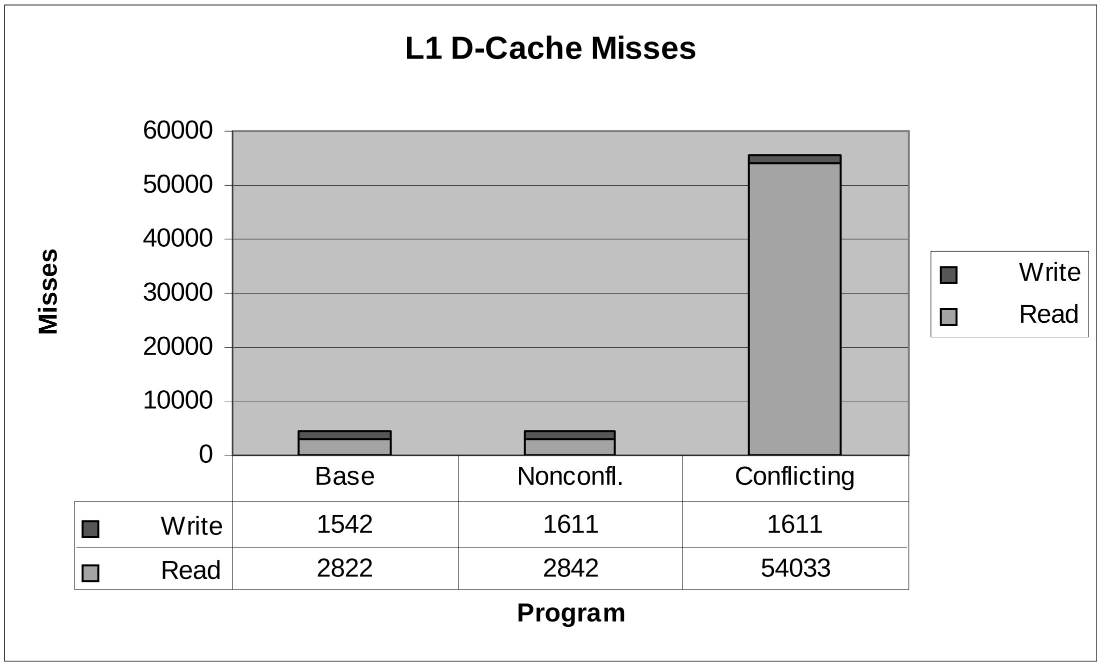
Results for the two runs of the experiment are shown in the tables on the next page, measured in elapsed and CPU seconds. (The Unix *time* command produces three time values for a program run. The *real* value is elapsed time, while the CPU time is split into *user* and *system* values. The CPU time shown in the charts below is the sum of *user* and *system* values, averaged over five runs of the programs.)



	Base	Elapsed	CPU
▨ Global Arrays	1.00	1.84	1.80
■ Local Arrays	1.00	1.91	1.90

Using the Unix *time* command, with an outer loop in the test program running enough iterations (8,000,000) to get valid run times, the first (nonconflicting) form of the program showed 44% to 47% decrease in user time compared to the second (conflicting) form of the program. (Translating this reduction to the *speedup* measure, as used in the Hennessy and Patterson texts, gives the data shown in the bar chart above. Speedup is simply the ratio or original program run time (before an optimization) to the new program run time (after the optimization.) If a program optimization reduces its run time by 50%, for example, it is said to be a speedup of 2.0. If a program change has no effect on run time, it has a speedup of 1.0. A speedup value of less than 1.0 would indicate that the performance actually worsened.)

Reducing the outer loop iteration count to 800 so that a cache simulator could be feasibly run (the *cachesim5* simulator that is part of the *Shade* toolkit,) the pattern of cache hits and misses met expectations. The first form of the program had only a few (20) start-up data cache read misses, while the second form of the program showed the expected increase in data cache read misses (to more than 51,000, or almost 100% misses on data cache reads.) The numbers were obtained by determining the program start-up overhead using a null program (one that entered `main()` and then returned) that was measured using *cachesim5*. These overhead measurements were then subtracted from the measurements for the two forms of the test program, as shown below:



	Base	Nonconfl.	Conflicting
■ Write	1542	1611	1611
□ Read	2822	2842	54033

These numbers for cache misses minus the null (base) program can be multiplied by 10,000 to get the number of extra misses caused by unaligned arrays in the earlier CPU time results. Doing so reveals that more than 50,000,000 extra L1 D-cache read misses will only cause a little over 2 CPU seconds of additional time. This would seem to indicate that L1 D-cache misses that are hits in the L2 cache can be serviced quickly.

6.0 Conclusions

The conclusions from this experiment are listed below:

- Level 1 Data cache conflicts among heavily used arrays can have a significant effect on the run times of programs.
- Such data cache conflicts can be avoided by efficient data placement.
- While such data placement was done manually in this experiment, it appears that a compiler should be able to accomplish the same result.
- The cache effects of data placement can be measured easily with a cache simulator. Care must be taken to factor out the start-up costs of the program, such as copying the values of environment variables into the C language parameter `*argp.`
- The experiment's hypothesis, as stated in section 2.0 above, is confirmed.

7.0 Follow-up Work

- In order to gain experience with the profiling tool *Shade,* a single run of each program can be made with data address tracing enabled in the *Shade* tool. The data address traces can be analyzed to determine the data cache hits and misses for the two programs.
- The *vpo* compilation system can be modified to use the static register dependency graph to optimize data placement. Ignoring graph nodes corresponding to temporary variables that were not declared by the user and which will reside only in registers, and taking advantage of the live-range analysis already performed by the register allocation code in *vpo*, a graph-coloring approach to data placement could be implemented, with cache lines, rather than registers, being the scarce resource that we are allocating. The timing runs could then be repeated, with the expectation that the two programs would have the same performance. Examination of the assembly code produced could determine how closely the two data placements resemble each other.

Appendix: Code Used in Experiments.

Below is the aligned (“good”) version of the code using global variables.

```
#include <stdlib.h>
#include <stdio.h>

  /* This program exercises the SuperSPARC data cache with high spatial
     and temporal locality. It serves as an ideal benchmark. A permutation
     of this code, dpbadcode.c, will rearrange the data declarations to
     damage the spatial locality with respect to the 32-byte data cache
     lines of the SuperSPARC.

     NOTE: The SuperSPARC has 16KB of on-chip, L1 data cache, arranged
     with 4-way set associativity and 32-byte lines, for a total of 512
     lines (128 per set.) Data items with the same least significant 7 bits
     of their (byte) addresses will fall into the same set.
  */

  /* We will name some arrays as follows: arr##?, where ## = cache line
     number from 0 to 127, and ? is the set (arbitrarily called A,B,C,D). */

  float arr00A[8], arrRestA[1016];
  float arr00B[8], arrRestB[1016];
  float arr00C[8], arrRestC[1016];
  float arr00D[8], arrRestD[1016];

  /* Let’s set up some scalar variables that will get us out of the
     cache lines 0/32/64/96 to avoid conflicts. */

  float scalar1, scalar2, scalar3, scalar4;
  float scalar5, scalar6, scalar7, scalar8;

  /* Let’s set up some more arrays that will not cause conflict in
     dpgoodcode.c, but will cause conflict in dpbadcode.c when we
     move the above scalars down below these arrays.  */

  float arr01A[8], arrRest1A[1016];
  float arr01B[8], arrRest1B[1016];
  float arr01C[8], arrRest1C[1016];
  float arr01D[8], arrRest1D[1000]; /* save room for loop counters */

  /* Some loop counters. */
  long k, l;

int main() {

  /* Put some values in both arrays */
  for (k = 0; k < 8; ++k) {
    arr00A[k] = (float) k;
    arr00B[k] = (float) (k + 1);
    arr00C[k] = (float) (k + 2);
    arr00D[k] = (float) (k + 3);
    arr01A[k] = (float) k;
    arr01B[k] = (float) (k + 1);
    arr01C[k] = (float) (k + 2);
    arr01D[k] = (float) (k + 3);
  } /* end for k = 0 ... */
  for (l = 0; l < 800000L; ++l) {
    /* Use both groups of arrays at once. */
    for (k = 0; k < 8; ++k) {
      if ((arr00A[k] < arr00B[k]) &&
          (arr00C[k] < arr00D[k]) &&
          (arr01A[k] < arr01B[k]) &&
          (arr01C[k] > arr01D[k])) {
        arrRestA[k + 32] = 0.0; /* never executed */
      }
    } /* end for k = 0 ... */
  } /* end for l = 0 ... */
 :
 :
  return EXIT_SUCCESS;
}  /* end of main() */
```

To create the local version of this code, we simply move the data declarations for all arrays and scalars inside function `main()`.

To create the unaligned versions of the code, we simply move the declarations of variables `scalar1` through `scalar8` down below the second group of arrays. This causes the second group of arrays to exactly overlay the first group in the L1 data cache, creating conflicts among eight heavily used arrays of size on cache line each: `arr00A[8]`, `arr00B[8]`, through `arr01D[8]`. With only 4 sets, the D-cache cannot handle eight heavily used lines in conflict with one another. In the aligned code, the scalars are used to push the second group of arrays down one line, just as we would expect a data-placement conscious compiler to do.