578770

Efficacy of code optimizations on cache-based processors

Rob F. Van der Wijngaart
MRJ Technology Solutions
NASA Ames Research Center
MS T27A-1
Moffett Field, CA 94035

1 Introduction

The current common wisdom in the U.S. is that the powerful, cost-effective supercomputers of tomorrow will be based on commodity (RISC) micro-processors with cache memories. Already, most distributed systems in the world use such hardware as building blocks. This shift away from vector supercomputers and towards cache-based systems has brought about a change in programming paradigm, even when ignoring issues of parallelism. Vector machines require inner-loop independence and regular, non-pathological memory strides (usually this means: non-power-of-two strides) to allow efficient vectorization of array operations. Cache-based systems require spatial and temporal locality of data, so that data once read from main memory and stored in high-speed cache memory is used optimally before being written back to main memory (see Bailey '93 [1]). This means that the most cache-friendly array operations are those that feature zero or unit stride, so that each unit of data read from main memory (a cache line) contains information for the next iteration in the loop. Moreover, loops ought to be 'fat', meaning that as many operations as possible are performed on cache data—provided instruction caches do not overflow and enough registers are available. If unit stride is not possible, for example because of some data dependency, then care must be taken to avoid pathological strides, just as on vector computers. For cache-based systems the issues are more complex, due to the effects of associativity and of non-unit block (cache line) size (see Bailey '95 [2]). But there is more to the story. Most modern micro-processors are superscalar, which means that they can issue several (arithmetic) instructions per clock cycle, provided that there are enough independent instructions in the loop body. This is another argument for providing fat loop bodies.

With these restrictions, it appears fairly straightforward to produce code that will run efficiently on any cache-based system. It can be argued that although some of the important computational algorithms employed at NASA Ames require different programming styles on vector machines and cache-based machines, respectively, neither architecture class appeared to be favored by particular algorithms *in principle*. Practice tells us that the situation is more complicated. This report presents observations and some analysis of performance tuning for cache-based systems. We point out several counterintuitive results that serve as a cautionary reminder that memory accesses are not the only factors that determine performance, and that within the class of cache-based systems, significant differences exist.

2 Kernel code

The most important flow solver program in use at NASA Ames is OVERFLOW, based on an Alternating-Direction Implicit (ADI) algorithm described by Beam and Warming [3]. The efficient diagonalized form of the algorithm, described by Pulliam and Chaussee [4], is the most commonly used solver kernel of OVERFLOW. Its essence is captured in the Scalar Penta-diagonal (SP) program, part of the suite of source codes that make up the NAS Parallel Benchmarks 2 (NPB 2) [5]. SP constitutes a stress test on the memory system of the computer, since there are fairly few operations per grid point to be executed in any one loop of the code. The most critical part of the code is the line solver, which solves independent systems of linear equations associated with grid lines. Since there are three families of grid lines in three-dimensional space, there are also three different solver routines for the three so-called factors. We use these three routines, with the generic names `x-loop`, `y-loop`, and `z-loop`, respectively, as examples of codes to be run on cache-based systems. They are described in Appendix A. Successive cumulative optimizations attempted in these codes, indicated by suffixes 1 through 5, are as follows.

1. Baseline, which is now contained in NPB 2.
2. Eliminate temporaries for incremented indices, i.e. `i+1, i+2, j+1, j+2, k+1, k+2` instead of `i1, i2, j1, j2, k1, k2`. This enables good compilers to do better register allocation.
3. Unroll inner loops of fixed length (`m=1,2,3`). This reduces loop overhead and register demand.
4. Move the fourth index (`m`) of the arrays `rhs, lhs` to the first position. This improves spatial data locality, since each iteration of the inner loop uses all three elements of `rhs` and all five elements of `lhs` at each grid point.
5. Unroll the first available loop (`j` for `x-loop`, `i` for `y-loop` and `z-loop`) to a level of two. This increases the number of independent computations in the inner loop. Notice the location of the assignment `fac2 = 1.d0/lhs(3,i,j+1,k)`. It is moved to the front of the inner loop, far ahead of the subsequent assignments that make use of `fac2`, to improve possibilities for optimal scheduling by the compiler.

In the cases of `y-loop` and `z-loop` there are two more code variations possible that have to do with the order of the loops. The previous optimizations all employ the so-called *canonical* loop nest orderings for `x-loop`, `y-loop` and `z-loop`, namely running index `k` for the outer loop, `j` for the middle loop, and `i` for the inner loop. But it is most *natural* computationally to finish a whole grid line in the inner loop before moving to the next grid line.

6. Use j and k as the running index for y-loop and z-loop, respectively, while keeping the unrollings described in optimization 5. This causes large strides in the arrays, which is generally bad. But the benefit is that there is some overlap in subsequent iterations, since there is a recursion in the grid line direction in the solver algorithm. The running index for the middle loop in each of the two loop nests is always i.

7. Use the same natural loop order as in the previous optimization, but eliminate unrolling optimization 5.

Each subroutine is executed for four grid sizes, i.e. 16^3, 32^3, 64^3, and 80^3. To avoid bad strides (see optimizations 6 and 7), all array grid dimensions are padded by 1. Tests with different paddings did not produce improvements on the architectures under consideration, and cache simulations (see Section 5) indicate that no cache lines remain unused when padding is set to 1. Moreover, when the m-index is moved into the first position, primeness of its size (3 for rhs and 5 for lhs) effectively guarantees safe strides. An optimization that was attempted but that is not reported quantitatively here concerns transposition of the arrays to align the line solve direction with the second array index (following m). This technique, which can pay off well when many operations are performed at each grid point, such as in FFTs on large grids, obtained very poor results for SP due to the sparseness of computations.

3 Machines

The machines investigated in this report are mainly RISC processors: MIPS R5000, MIPS R8000, MIPS R10000, DEC Alpha EV4, DEC Alpha EV5, IBM RS6000, Sun UltraSparc I. The one CISC architecture is the Intel PentiumPro. A number of these processors are currently used in parallel platforms: MIPS R8000 in the SGI PowerChallenge Davinci cluster at NASA Ames, MIPSR10000 in the SGI Origin2000 cluster at the University of Urbana Champaign, DEC Alpha EV4 in the Cray T3D Cosmos system at the Jet Propulsion Laboratory, DEC Alpha EV5 in the Cray T3E machine at the National Energy Research Scientific Computing Center in Berkeley, IBM RS6000 in the SP Wide Node Babbage system at NASA Ames, and Sun UltraSparc I in the University of Berkeley Network of Workstations (NOW) system.

The PentiumPro is being considered for a new massively parallel system called Whitney, currently under construction at NASA Ames. Machine specifics are summarized in Table 1. They reflect the alterations made to the processors to integrate them in their parallel platforms. In particular, the DEC Alpha EV4 and EV5 modified by Cray Research lost their off-chip second-level and third-level caches, respectively, for numerical processing. One of the most important system parameters, the memory bandwidth, is not listed, because vendor-provided data are generally inconclusive. It is assumed that memory bandwidth is always an active constraint on processor performance.

On all machines we select the highest acceptable level of optimization for the Fortran compiler (generally -O3). Native Fortran compilers are available for all platforms, except the PentiumPro. On the latter system the compiler from the Portland Group is used.

4 Measured performance results

Performances of the processors–and compilers—tested, in MFlop/s (millions of floating point computations per second), are presented in Appendix B. They are obtained by running each of the loop nests in Appendix A a number of times on each system, ignoring the first iteration to eliminate start-up overhead, and determining the minimum of the execution times for the remaining iterations to eliminate noise.

The results for each processor are shown in three different ways, as indicated in the schematic in Figure 1. The first display (Figure a) scales the performance within each set of computations for a certain grid size and a certain factor (x, y, or z). Maximum performance relative to the other optimization strategies for the same grid size and factor is indicated in black, minimum in light gray. This type of display allows the comparison of a particular optimization across all grid sizes and factors; for example, if row 3 is black for the y-factor for a certain processor (as shown in Figure 1a), then unrolling of the `m`-loop, in conjunction with optimizations 2, guarantees optimal performance of that y-factor. If rows 3 for the x-factor and z-factor were black as well, then unrolling the `m`-loop would be a generally optimal strategy for that processor.

The second display (Figure b) scales the performance within each set of computations for a certain factor (x, y, or z). This allows the easy assessment of the influence of grid size (and hence of array size) on performance. For example, if black blocks occur only in the left few columns of figure b (as shown in Figure 1b), then the processor's performance apparently reduces when the problem grows too large to fit completely in the cache.

The third display (Figure c) scales the performance within the whole set of computations for each processor. This facilitates the combined assessment of cache size and stride on the performance. Factor x features the smallest stride, and z the largest, even when the loop ordering is theoretically optimal (outer loop `k`, then `j`, and inner loop `i`). This may be borne out by the performance results if the maximum performance is obtained for the x-factor, as suggested in Figure 1c.

Note that the absolute magnitudes of the performance numbers are the same for corresponding cases in each set of three figures a, b and c; it is only the shading that differs. Some

Table 1: Processor specifications summary

Name	RAM MBytes	L1-cache KBytes	L2-cache KBytes	cache line Bytes	associativity	CPU MHz	Peak MFlop/s
R5000	64	32i+32d	0	32	2	150	300
R8000	4096	16i+16d	4096	512	4	90	360
R10000	4096	32i+32d	4096	128	2	195	390
EV4	8	8i+8d	0	32	1	150	300
EV5	8	8i+8d	96	32	3	300	600
RS6000	128	32i	256	256	4	66	267
PPro	128	8i+8d	256/512	32	4	200	200
Sparc	1024	16i+16d	512	64	1	167	334

interesting observations can be made, based on the performance figures.

MIPS R5000. Figure 2a shows a marked improvement for unrolling the `m`-loop for the x-factor, and, to a lesser extent, for the y- and z-factors. Making `m` the first array index gives a substantial improvement for the y-factor and also some for the x-factor, but reduces performance for the z-factor. And, counterintuitively, the z-factor benefits most from the natural, large-stride loop ordering, as opposed to the canonical ordering.

MIPS R8000. Figure 3a shows that unrolling the `i`-loops for the y- and z-factors greatly deteriorates performance, hinting at a problem with the allocation of registers for the fattened loop body. Figure 3b clearly demonstrates the reduced processor performance for large grids that do not fit completely in cache (sizes 64 and 80). Apparently, the memory bandwidth is not sufficient to fill the cache fast enough from main memory. Finally, Figure 3c reveals that the generally most cache-friendly x-factor code performs more poorly than the y- and z-factors. This may be caused by the interleaved structure of the cache.

MIPS R10000. Figure 4a shows a strong dependence of problem size on the best optimization strategy. For small grids making `m` the first array index greatly improves performance, but this optimization is completely counterproductive for large grid sizes. In either case, the greatest relative performance improvement is obtained by the virtually trivial optimization of unrolling the `m`-loop. Figure 4b again indicates a memory bandwidth problem. Unlike for the MIPS R8000, the x-factor now exhibits the best performance again among all three factors, as evidenced by Figure 4c.

IBM RS6000. Figure 5a shows, somewhat surprisingly, that elimination of the auxiliary variables `i1` and `i2` in the x-factor is beneficial on the RS6000, whereas a similar program change for the y- and z-factors has a negative effect. In addition, we notice that the best performance for the x-factor is obtained for a partially unrolled j-loop, whereas optimal performance for the y- and z-factors is achieved for without unrolling of the i-, j-, and k-loops. Finally, moving the m-index proves positive for the **x**-, neutral for the **y**-, and negative for the **z**-loop. From Figure 5b we conclude that there is a relatively small effect of grid size on performance, indicating that the memory bandwidth is sufficient to feed the cache.

Intel PentiumPro 200 MHz. Figure 6a demonstrates that there is great sensitivity to problem size with respect to the optimizations of the z-factor. But more importantly, Figure 6c shows that, with proper tuning, a performance of about 20 Mflops/s can be maintained for a nontrivial floating point computation on this processor. This places the PentiumPro squarely in the scientific workstation performance range.

DEC Alpha EV4. Evidently, the x-factor performance improves through the unrolling of the `m`-loop, whereas y- and z-factor performances deteriorate under the same code change, as evidenced by Figure 7a. It also shows, in contrast with, for example, the RS6000, that the best performance is obtained for a partially unrolled `j`-loop for the x-factor, but for completely rolled up `i`-loops for the y- and z-factors. We also observe in Figure 7c that the best results overall are obtained for the x-factor with `m` as the first array index, which may be explained by the small cache of the machine—as modified by Cray Research—and the (perceived) good data locality of this code variation.

DEC Alpha EV5. For this DEC chip with a larger (secondary) cache the performance of the y-factor degrades significantly when switching to the natural loop order, whereas for the z-factor the effect is mixed. The largest performance improvement for all factors comes from moving the m-index to the first array position, with unrolling the i- or j-loops having little or no positive effect, as follows from Figure 8a. As for the EV4, the EV5 again shows best overall behavior for the x-factor with m as the first array index (Figure 8c).

SUN ULTRASPARC I. From the fairly smooth distribution of performances across columns in Figure 9a we deduce that the UltraSparc is not very sensitive to code optimizations by the user, except for the very smallest grid size that fits entirely in the cache. Also notice that whereas unrolling the i-loop did not help improve the performance of the x-factor, it was necessary, combined with natural loop ordering, for best performance of the y- and z-loops. A quick glance at Figure 9b shows that performance degrades drastically as soon as the problem no longer fits entirely in cache, again pointing to a lack of memory bandwidth.

There are many ways in which the information in Figures 2a through 9a can be summarized. We choose a simple one for Table 2. For each processor we count the number of grid sizes for which a particular optimization technique is superior over all others. The sum of these numbers over all processors is listed as *total*. The number of processors for which each particular optimization is optimal for all grid sizes is listed as *unanimous*.

Table 2: Summary of processor performance

Name	x-factor optimizations					y-factor optimizations							z-factor optimizations						
	1	2	3	4	5	1	2	3	4	5	6	7	1	2	3	4	5	6	7
R5000	0	0	0	0	4	0	0	0	0	3	1	0	0	0	0	0	0	4	0
R8000	0	0	4	0	0	0	0	2	0	0	0	2	0	0	2	0	0	0	2
R10000	0	0	2	1	1	0	0	2	0	0	2	0	0	0	2	0	0	2	0
RS6000	0	0	0	0	4	0	0	0	0	0	0	4	0	0	0	0	0	1	3
PentiumPro	0	0	0	4	0	0	0	0	4	0	0	0	0	0	0	0	0	0	4
DEC EV4	0	0	0	4	0	0	0	0	0	0	4	0	0	0	0	0	0	4	0
DEC EV5	0	0	0	4	0	0	0	0	0	4	0	0	0	0	0	0	2	0	2
UltraSparc	0	0	0	4	0	0	0	0	0	4	0	0	0	1	0	0	0	3	0
total	0	0	6	17	9	0	0	4	4	11	7	6	0	1	4	0	2	14	11
unanimous	0	0	1	4	2	0	0	0	1	2	1	1	0	0	0	0	0	2	1

Evidently, there is not a single optimization strategy that gives the best performance for all grid sizes for all processors. But even if we restrict the attention to one processor at a time, most still do not allow a single uniform optimization strategy. Only the PentiumPro and the DEC Alpha EV4 chip show consistently best performance (within 1% accuracy) for fixed strategies *x-4, y-4, z-7* and *x-4, y-6, z-6*, respectively. If we rule out unrolling the i, j, or k loops (*y-5,6, z-5,6)*, then the generally most acceptable single optimization strategy is: *x-4, y-7, z-7*, i.e. the canonical loop ordering, with large strides for the y- and z-factors, and m as the first array index. We notice that the scatter in the optimal tuning strategies is substantially larger for the y- and z-factors than for the x-factor.

5 Cache simulations

It follows from the results in the previous section that intuition is a poor guide to performance tuning. More detailed knowledge and insight are needed to design a good optimization strategy. In order to produce code with portable performance characteristics, we limit the type of knowledge to enter into the investigation to relatively easily quantifiable parameters on arbitrary systems. Specifically, we choose to simulate the cache behavior of the same set of processors as before to understand better the observed computational results. The model for the simple cache simulator is as follows. The highest-level cache is assumed to constitute the memory bottleneck; its parameters are used for the simulator. In case of set-associative caches a Least Recently Used (LRU) replacement policy of cache lines is employed. No knowledge about the cache other than line length, number of lines, and associativity, is incorporated; For example, the effects of cache memory interleaving, and of sophisticated strategies such as *early restart* and *requested word first*, are not taken into account. Moreover, we assume a separate data cache, since unified caches require vastly more information for simulation, including the assembler version of the code. We simply transform the assignments in the source code by hand to calls to the cache simulation routines. Only memory loads are modeled. The structure of the computational loops is such that writes are always done to memory locations that are already in cache, which means that no cache lines need to be flushed to accommodate write operations. We ignore the effects of writing through the cache to main memory. This is a potentially serious simplification. Depending on the buffering strategy and the implementation of the cache controller, substantial stalling may be incurred due to write operations.

Results of the simulations are presented in Appendix C. Each figure shows the percentage of memory accesses (fetches) that cause cache misses for the optimizations described in Appendix A (all code fragments exhibit the same total number of array references). Tallying cache hits and misses starts after each loop nest has been executed once. This fills the cache with (potentially) useful data beforehand, which is equivalent to the preconditioning in the actual performance measurements described in Section 4. Note that optimization strategies *x-2, x-3, y-2, y-3, z-2* and *z-3* are not displayed, since these feature exactly the same memory access patterns as *x-1, y-1* and *z-1*, respectively. Shading is applied in the same fashion as in Figures a of Appendix B. The darker the shading, the fewer cache misses. Since we expect the percentage of cache misses to be smaller for better performing code fragments, we call this relation between the two a positive correlation.

MIPS R5000. Figure 10 exhibits a positive correlation between performance and cache misses for the x- and z-factors, but a (mostly) negative one for y. We also notice that there is a large difference in the performance of optimizations `x-1` through `x-3`, although the memory reference patterns are identical. Finally, we observe that the number of cache misses for the z-factor is slashed in three—somewhat unexpectedly—by using the natural loop ordering.

MIPS R8000. Figure 11 features a very small miss rate (less than 1%), and virtually no correlation between performance and cache misses.

MIPS R10000. Again there is very poor correlation between processor performance and miss rate (Figure 12), and in the case of a grid size of 32^3 points the correlation is even negative.

IBM RS6000. As in the case of the MIPS R5000, the number of cache misses for the z-factor is reduced significantly by reverting to the natural loop order for large grids. Figure 13 indicates a fair correlation between performance and cache misses overall.

INTEL PENTIUMPRO 200 MHz. Although there is a clear positive correlation between cache misses and performance for the x-factor, Figure 14 reveals a negative and mixed correlation for the y- and z-factors, respectively. And again, natural loop ordering benefits the memory access pattern of the z-factor.

DEC ALPHA EV4. Figure 15 confirms a positive correlation between cache misses and performance for the z-factor, and negative and mixed correlations for x and y. It is of some interest to note the erratic simulated cache behavior for the y-factor, with *y-7, y-1, y-7* and *y-6* being the optimal strategies for grid sizes 16^3, 32^3, 64^3 and 80^3, respectively.

DEC ALPHA EV5. Figure 16 points out that reductions of cache misses by up to a factor of 3 for the larger grids of the z-factor by switching to the natural loop ordering has no effect or no positive effect on the performance. The y-factor shows widely varying performance figures for identical numbers of cache misses.

SUN ULTRASPARC I. Performance of the UltraSparc for small grids (16^3) varies significantly, especially for the x-factor, although the problem fits entirely in the cache, and no misses occur. For larger grids the number of cache misses varies widely for the z-factor, with little difference in processor performance.

Evidently, the simulated cache behavior for the processors studied correlates poorly with the actually observed performance. It can be argued that this is due to the limitations of the simulator, and that better correlations can be obtained by incorporating more detailed characteristics of the particular machines. But since we are interested in producing *portable* code, introducing even more information into the program construction will make this task virtually impossible. We also notice that there is significant—and nontrivial—dependence of the processor performance on the problem size, which is generally not known at compile time. This makes the task of writing portable codes even harder. Moreover, the cache miss rate of the simple piece of code investigated in this paper is a complex function of loop organization, problem size, and cache structure, even if only very few parameters are used to describe the cache.

6 Conclusions

We conclude that tuning codes for high performance on commercial cache-based processors available today is a process that requires so much knowledge and information about the entire configuration of user program, problem parameters, system software, and hardware organization, that it is practically impossible to produce 'good cache code'. By this we mean that it is not possible to tell whether a piece of numerical software will perform well by using textual inspection and a few high-level parameters of the system under consideration; General optimization strategies designed to take advantage of cache do not alone yield high performance. Other factors, such as software pipelining, need to be considered as well when designing efficient codes, but this will hamper portability of codes between architectures.

Finally, it should be noted that extreme care must be taken to extend the optimization results obtained from the loops in Appendix A to entire application programs in case data lay-outs are changed. For example, it may appear natural to move the `m`-index to the front of the `rhs` and `lhs` arrays to increase cache data reuse. But for loops in which only, say, the first element of `rhs` is used, performance may suffer, due to 'gaps' in the array.

References

[1] D.H. Bailey, *RISC Microprocessors and Scientific Computing*, Proc. Supercomputing '93, IEEE Computer Society, 1993, pp. 645-654.

[2] D.H. Bailey, *Unfavorable Strides in Cache Memory Systems*, Scientific Programming, vol. 4 (1995), pp. 53-58

[3] R.M. Beam, R.F. Warming, *An Implicit Factored Scheme for the Incompressible Navier-Stokes Equations*, AIAA Journal, vol. 16 (1978), pp. 393-401

[4] T.H. Pulliam, D.S. Chaussee, *A diagonal form of an implicit approximate factorization algorithm*, Journal of Computational Physics, Vol. 29, p. 1037, 1975

[5] D.H. Bailey, T. Harris, W.C. Saphir, R.F. Van der Wijngaart,, A. Woo, M. Yarrow, *The NAS Parallel Benchmarks 2.0*, NAS Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, 1995

Appendix A: Codes

Below follow the source listings of the SP solver fragments used in the performance tests; only the forward elimination parts of the penta-diagonal line solvers are used. Diagonalization makes it possible to compute three solution values (`rhs(..,m=1,3)`) per grid point simultaneously from a single penta-diagonal matrix (`lhs`). For each of the three factors to be inverted in the ADI scheme (corresponding to the x-, y-, and z-directions), the same optimizations are performed for corresponding suffixes. For example, *x-3, y-3* and *z-3* all unroll inner loops of length three. *x-1, y-1* and *z-1* are the actual code fragments used in NPB 2. Here we only show *x-1* through *x-5*, *y-6*, and *y-7*. The other code fragments are easily inferred. Italic typeface is used to indicate which parts of the code fragments are affected by the optimizations. The dimensions of the arrays are `rhs(nx,ny,nz,3)` and `lhs(nx,ny,nz,5)`, respectively, when `m` is the last index, and `rhs(3,nx,ny,nz)` and `lhs(5,nx,ny,nz)`, respectively, when `m` is the first index.

x-1: NPB 2.2 code

```
do    k = 1, nz
   do    j = 1, ny
      do    i = 1, nx-2
         i1 = i  + 1
         i2 = i  + 2
         fac1            = 1.d0/lhs(i,j,k,3)
         lhs(i,j,k,4)    = fac1*lhs(i,j,k,4)
         lhs(i,j,k,5)    = fac1*lhs(i,j,k,5)
         do    m = 1, 3
            rhs(i,j,k,m) = fac1*rhs(i,j,k,m)
         end do
         lhs(i1,j,k,3) = lhs(i1,j,k,3) - lhs(i1,j,k,2)*lhs(i,j,k,4)
         lhs(i1,j,k,4) = lhs(i1,j,k,4) - lhs(i1,j,k,2)*lhs(i,j,k,5)
         do    m = 1, 3
            rhs(i1,j,k,m) = rhs(i1,j,k,m) - lhs(i1,j,k,2)*rhs(i,j,k,m)
         end do
         lhs(i2,j,k,2) = lhs(i2,j,k,2) - lhs(i2,j,k,1)*lhs(i,j,k,4)
         lhs(i2,j,k,3) = lhs(i2,j,k,3) - lhs(i2,j,k,1)*lhs(i,j,k,5)
         do    m = 1, 3
            rhs(i2,j,k,m) = rhs(i2,j,k,m) - lhs(i2,j,k,1)*rhs(i,j,k,m)
         end do
      end do
   end do
end do
```

x-2: do not use auxiliary variables for incremented indices

```
do    k = 1, nz
   do    j = 1, ny
      do    i = 1, nx-2
         fac1            = 1.d0/lhs(i,j,k,3)
```

```
            lhs(i,j,k,4)    = fac1*lhs(i,j,k,4)
            lhs(i,j,k,5)    = fac1*lhs(i,j,k,5)
            do    m = 1, 3
               rhs(i,j,k,m) = fac1*rhs(i,j,k,m)
            end do
            lhs(i+1,j,k,3) = lhs(i+1,j,k,3) - lhs(i+1,j,k,2)*lhs(i,j,k,4)
            lhs(i+1,j,k,4) = lhs(i+1,j,k,4) - lhs(i+1,j,k,2)*lhs(i,j,k,5)
            do    m = 1, 3
               rhs(i+1,j,k,m) = rhs(i+1,j,k,m) - lhs(i+1,j,k,2)*rhs(i,j,k,m)
            end do
            lhs(i+2,j,k,2) = lhs(i+2,j,k,2) - lhs(i+2,j,k,1)*lhs(i,j,k,4)
            lhs(i+2,j,k,3) = lhs(i+2,j,k,3) - lhs(i+2,j,k,1)*lhs(i,j,k,5)
            do    m = 1, 3
               rhs(i+2,j,k,m) = rhs(i+2,j,k,m) - lhs(i+2,j,k,1)*rhs(i,j,k,m)
            end do
         end do
      end do
   end do
```

x-3: unroll small inner loops of length 3

```
   do    k = 1, nz
      do    j = 1, ny
         do    i = 1, nx-2
            fac1            = 1.d0/lhs(i,j,k,3)
            lhs(i,j,k,4)    = fac1*lhs(i,j,k,4)
            lhs(i,j,k,5)    = fac1*lhs(i,j,k,5)
            rhs(i,j,k,1)    = fac1*rhs(i,j,k,1)
            rhs(i,j,k,2)    = fac1*rhs(i,j,k,2)
            rhs(i,j,k,3)    = fac1*rhs(i,j,k,3)
            lhs(i+1,j,k,3) = lhs(i+1,j,k,3) - lhs(i+1,j,k,2)*lhs(i,j,k,4)
            lhs(i+1,j,k,4) = lhs(i+1,j,k,4) - lhs(i+1,j,k,2)*lhs(i,j,k,5)
            rhs(i+1,j,k,1) = rhs(i+1,j,k,1) - lhs(i+1,j,k,2)*rhs(i,j,k,1)
            rhs(i+1,j,k,2) = rhs(i+1,j,k,2) - lhs(i+1,j,k,2)*rhs(i,j,k,2)
            rhs(i+1,j,k,3) = rhs(i+1,j,k,3) - lhs(i+1,j,k,2)*rhs(i,j,k,3)
            lhs(i+2,j,k,2) = lhs(i+2,j,k,2) - lhs(i+2,j,k,1)*lhs(i,j,k,4)
            lhs(i+2,j,k,3) = lhs(i+2,j,k,3) - lhs(i+2,j,k,1)*lhs(i,j,k,5)
            rhs(i+2,j,k,1) = rhs(i+2,j,k,1) - lhs(i+2,j,k,1)*rhs(i,j,k,1)
            rhs(i+2,j,k,2) = rhs(i+2,j,k,2) - lhs(i+2,j,k,1)*rhs(i,j,k,2)
            rhs(i+2,j,k,3) = rhs(i+2,j,k,3) - lhs(i+2,j,k,1)*rhs(i,j,k,3)
         end do
      end do
   end do
```

x-4: move last index of rhs and lhs to the front

```
   do    k = 1, nz
      do    j = 1, ny
         do    i = 1, nx-2
         fac1           = 1.d0/lhs(3,i,j,k)
         lhs(4,i,j,k)   = fac1*lhs(4,i,j,k)
         lhs(5,i,j,k)   = fac1*lhs(5,i,j,k)
         rhs(1,i,j,k)   = fac1*rhs(1,i,j,k)
         rhs(2,i,j,k)   = fac1*rhs(2,i,j,k)
         rhs(3,i,j,k)   = fac1*rhs(3,i,j,k)
         lhs(3,i+1,j,k) = lhs(3,i+1,j,k) - lhs(2,i+1,j,k)*lhs(4,i,j,k)
         lhs(4,i+1,j,k) = lhs(4,i+1,j,k) - lhs(2,i+1,j,k)*lhs(5,i,j,k)
         rhs(1,i+1,j,k) = rhs(1,i+1,j,k) - lhs(2,i+1,j,k)*rhs(1,i,j,k)
         rhs(2,i+1,j,k) = rhs(2,i+1,j,k) - lhs(2,i+1,j,k)*rhs(2,i,j,k)
         rhs(3,i+1,j,k) = rhs(3,i+1,j,k) - lhs(2,i+1,j,k)*rhs(3,i,j,k)
         lhs(2,i+2,j,k) = lhs(2,i+2,j,k) - lhs(1,i+2,j,k)*lhs(4,i,j,k)
         lhs(3,i+2,j,k) = lhs(3,i+2,j,k) - lhs(1,i+2,j,k)*lhs(5,i,j,k)
         rhs(1,i+2,j,k) = rhs(1,i+2,j,k) - lhs(1,i+2,j,k)*rhs(1,i,j,k)
         rhs(2,i+2,j,k) = rhs(2,i+2,j,k) - lhs(1,i+2,j,k)*rhs(2,i,j,k)
         rhs(3,i+2,j,k) = rhs(3,i+2,j,k) - lhs(1,i+2,j,k)*rhs(3,i,j,k)
      end do
   end do
end do
```

x-5: unroll j-loop to a level of 2

```
do    k = 1, nz
   do    j = 1, ny, 2
      do    i = 1, nx-2
         fac1             = 1.d0/lhs(3,i,j,k)
         fac2             = 1.d0/lhs(3,i,j+1,k)
         lhs(4,i,j,k)     = fac1*lhs(4,i,j,k)
         lhs(5,i,j,k)     = fac1*lhs(5,i,j,k)
         rhs(1,i,j,k)     = fac1*rhs(1,i,j,k)
         rhs(2,i,j,k)     = fac1*rhs(2,i,j,k)
         rhs(3,i,j,k)     = fac1*rhs(3,i,j,k)
         lhs(3,i+1,j,k)   = lhs(3,i+1,j,k) - lhs(2,i+1,j,k)*lhs(4,i,j,k)
         lhs(4,i+1,j,k)   = lhs(4,i+1,j,k) - lhs(2,i+1,j,k)*lhs(5,i,j,k)
         rhs(1,i+1,j,k)   = rhs(1,i+1,j,k) - lhs(2,i+1,j,k)*rhs(1,i,j,k)
         rhs(2,i+1,j,k)   = rhs(2,i+1,j,k) - lhs(2,i+1,j,k)*rhs(2,i,j,k)
         rhs(3,i+1,j,k)   = rhs(3,i+1,j,k) - lhs(2,i+1,j,k)*rhs(3,i,j,k)
         lhs(2,i+2,j,k)   = lhs(2,i+2,j,k) - lhs(1,i+2,j,k)*lhs(4,i,j,k)
         lhs(3,i+2,j,k)   = lhs(3,i+2,j,k) - lhs(1,i+2,j,k)*lhs(5,i,j,k)
         rhs(1,i+2,j,k)   = rhs(1,i+2,j,k) - lhs(1,i+2,j,k)*rhs(1,i,j,k)
         rhs(2,i+2,j,k)   = rhs(2,i+2,j,k) - lhs(1,i+2,j,k)*rhs(2,i,j,k)
         rhs(3,i+2,j,k)   = rhs(3,i+2,j,k) - lhs(1,i+2,j,k)*rhs(3,i,j,k)
         lhs(4,i,j+1,k)   = fac2*lhs(4,i,j+1,k)
         lhs(5,i,j+1,k)   = fac2*lhs(5,i,j+1,k)
         rhs(1,i,j+1,k)   = fac2*rhs(1,i,j+1,k)
         rhs(2,i,j+1,k)   = fac2*rhs(2,i,j+1,k)
         rhs(3,i,j+1,k)   = fac2*rhs(3,i,j+1,k)
         lhs(3,i+1,j+1,k) = lhs(3,i+1,j+1,k) - lhs(2,i+1,j+1,k)*lhs(4,i,j+1,k)
         lhs(4,i+1,j+1,k) = lhs(4,i+1,j+1,k) - lhs(2,i+1,j+1,k)*lhs(5,i,j+1,k)
         rhs(1,i+1,j+1,k) = rhs(1,i+1,j+1,k) - lhs(2,i+1,j+1,k)*rhs(1,i,j+1,k)
         rhs(2,i+1,j+1,k) = rhs(2,i+1,j+1,k) - lhs(2,i+1,j+1,k)*rhs(2,i,j+1,k)
         rhs(3,i+1,j+1,k) = rhs(3,i+1,j+1,k) - lhs(2,i+1,j+1,k)*rhs(3,i,j+1,k)
         lhs(2,i+2,j+1,k) = lhs(2,i+2,j+1,k) - lhs(1,i+2,j+1,k)*lhs(4,i,j+1,k)
         lhs(3,i+2,j+1,k) = lhs(3,i+2,j+1,k) - lhs(1,i+2,j+1,k)*lhs(5,i,j+1,k)
         rhs(1,i+2,j+1,k) = rhs(1,i+2,j+1,k) - lhs(1,i+2,j+1,k)*rhs(1,i,j+1,k)
         rhs(2,i+2,j+1,k) = rhs(2,i+2,j+1,k) - lhs(1,i+2,j+1,k)*rhs(2,i,j+1,k)
         rhs(3,i+2,j+1,k) = rhs(3,i+2,j+1,k) - lhs(1,i+2,j+1,k)*rhs(3,i,j+1,k)
       end do
   end do
end do
```

y-6: canonical loop order, i-loop unrolled to level 2

```
do    k = 1, nz
   do    i = 1, nx, 2
      do    j = 1, ny-2

         fac1             = 1.d0/lhs(3,i,j,k)
         fac2             = 1.d0/lhs(3,i+1,j,k)
         lhs(4,i,j,k)     = fac1*lhs(4,i,j,k)
         lhs(5,i,j,k)     = fac1*lhs(5,i,j,k)
         rhs(1,i,j,k)     = fac1*rhs(1,i,j,k)
         rhs(2,i,j,k)     = fac1*rhs(2,i,j,k)
         rhs(3,i,j,k)     = fac1*rhs(3,i,j,k)
         lhs(3,i,j+1,k)   = lhs(3,i,j+1,k) - lhs(2,i,j+1,k)*lhs(4,i,j,k)
         lhs(4,i,j+1,k)   = lhs(4,i,j+1,k) - lhs(2,i,j+1,k)*lhs(5,i,j,k)
         rhs(1,i,j+1,k)   = rhs(1,i,j+1,k) - lhs(2,i,j+1,k)*rhs(1,i,j,k)
         rhs(2,i,j+1,k)   = rhs(2,i,j+1,k) - lhs(2,i,j+1,k)*rhs(2,i,j,k)
         rhs(3,i,j+1,k)   = rhs(3,i,j+1,k) - lhs(2,i,j+1,k)*rhs(3,i,j,k)
         lhs(2,i,j+2,k)   = lhs(2,i,j+2,k) - lhs(1,i,j+2,k)*lhs(4,i,j,k)
         lhs(3,i,j+2,k)   = lhs(3,i,j+2,k) - lhs(1,i,j+2,k)*lhs(5,i,j,k)
         rhs(1,i,j+2,k)   = rhs(1,i,j+2,k) - lhs(1,i,j+2,k)*rhs(1,i,j,k)
         rhs(2,i,j+2,k)   = rhs(2,i,j+2,k) - lhs(1,i,j+2,k)*rhs(2,i,j,k)
         rhs(3,i,j+2,k)   = rhs(3,i,j+2,k) - lhs(1,i,j+2,k)*rhs(3,i,j,k)
         lhs(4,i+1,j,k)   = fac2*lhs(4,i+1,j,k)
         lhs(5,i+1,j,k)   = fac2*lhs(5,i+1,j,k)
         rhs(1,i+1,j,k)   = fac2*rhs(1,i+1,j,k)
         rhs(2,i+1,j,k)   = fac2*rhs(2,i+1,j,k)
         rhs(3,i+1,j,k)   = fac2*rhs(3,i+1,j,k)
         lhs(3,i+1,j+1,k) = lhs(3,i+1,j+1,k) - lhs(2,i+1,j+1,k)*lhs(4,i+1,j,k)
         lhs(4,i+1,j+1,k) = lhs(4,i+1,j+1,k) - lhs(2,i+1,j+1,k)*lhs(5,i+1,j,k)
         rhs(1,i+1,j+1,k) = rhs(1,i+1,j+1,k) - lhs(2,i+1,j+1,k)*rhs(1,i+1,j,k)
         rhs(2,i+1,j+1,k) = rhs(2,i+1,j+1,k) - lhs(2,i+1,j+1,k)*rhs(2,i+1,j,k)
         rhs(3,i+1,j+1,k) = rhs(3,i+1,j+1,k) - lhs(2,i+1,j+1,k)*rhs(3,i+1,j,k)
         lhs(2,i+1,j+2,k) = lhs(2,i+1,j+2,k) - lhs(1,i+1,j+2,k)*lhs(4,i+1,j,k)
         lhs(3,i+1,j+2,k) = lhs(3,i+1,j+2,k) - lhs(1,i+1,j+2,k)*lhs(5,i+1,j,k)
```

```
        rhs(1,i+1,j+2,k) = rhs(1,i+1,j+2,k) - lhs(1,i+1,j+2,k)*rhs(1,i+1,j,k)
        rhs(2,i+1,j+2,k) = rhs(2,i+1,j+2,k) - lhs(1,i+1,j+2,k)*rhs(2,i+1,j,k)
        rhs(3,i+1,j+2,k) = rhs(3,i+1,j+2,k) - lhs(1,i+1,j+2,k)*rhs(3,i+1,j,k)

      end do
   end do
end do
```

y-7: canonical loop order, rolled-up i-loop

```
do    k = 1, nz
   do    i = 1, nx
      do    j = 1, ny-2

        fac1            = 1.d0/lhs(3,i,j,k)
        lhs(4,i,j,k)    = fac1*lhs(4,i,j,k)
        lhs(5,i,j,k)    = fac1*lhs(5,i,j,k)
        rhs(1,i,j,k)    = fac1*rhs(1,i,j,k)
        rhs(2,i,j,k)    = fac1*rhs(2,i,j,k)
        rhs(3,i,j,k)    = fac1*rhs(3,i,j,k)
        lhs(3,i,j+1,k) = lhs(3,i,j+1,k) - lhs(2,i,j+1,k)*lhs(4,i,j,k)
        lhs(4,i,j+1,k) = lhs(4,i,j+1,k) - lhs(2,i,j+1,k)*lhs(5,i,j,k)
        rhs(1,i,j+1,k) = rhs(1,i,j+1,k) - lhs(2,i,j+1,k)*rhs(1,i,j,k)
        rhs(2,i,j+1,k) = rhs(2,i,j+1,k) - lhs(2,i,j+1,k)*rhs(2,i,j,k)
        rhs(3,i,j+1,k) = rhs(3,i,j+1,k) - lhs(2,i,j+1,k)*rhs(3,i,j,k)
        lhs(2,i,j+2,k) = lhs(2,i,j+2,k) - lhs(1,i,j+2,k)*lhs(4,i,j,k)
        lhs(3,i,j+2,k) = lhs(3,i,j+2,k) - lhs(1,i,j+2,k)*lhs(5,i,j,k)
        rhs(1,i,j+2,k) = rhs(1,i,j+2,k) - lhs(1,i,j+2,k)*rhs(1,i,j,k)
        rhs(2,i,j+2,k) = rhs(2,i,j+2,k) - lhs(1,i,j+2,k)*rhs(2,i,j,k)
        rhs(3,i,j+2,k) = rhs(3,i,j+2,k) - lhs(1,i,j+2,k)*rhs(3,i,j,k)

      end do
   end do
end do
```

Appendix B: Measured computational performance

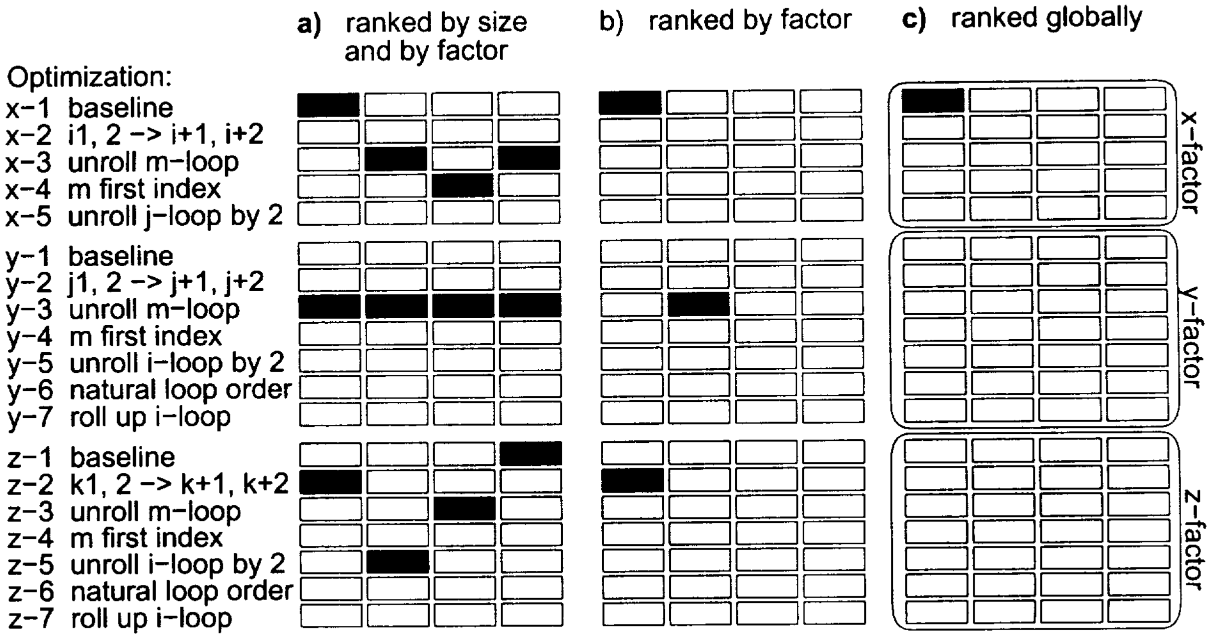
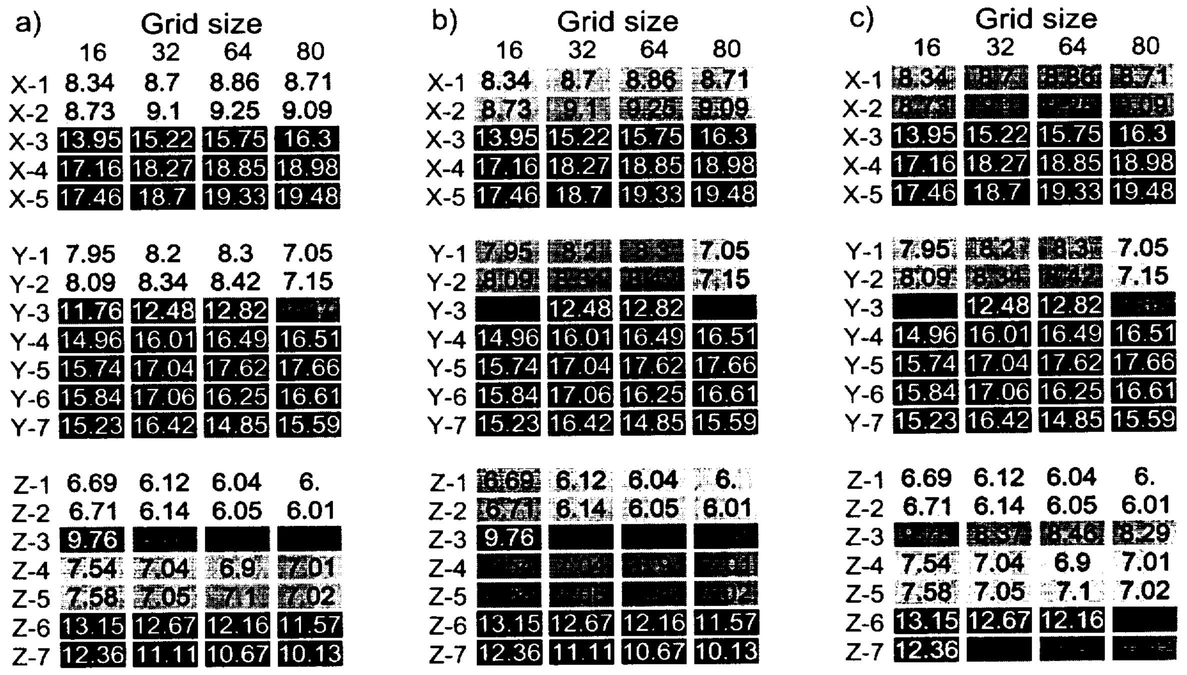


Figure 1: Schematic of processor performance presentation; black box signifies maximum performance within certain range, other gray scales not shown



a)

	Grid size 16	32	64	80
X-1	8.34	8.7	8.86	8.71
X-2	8.73	9.1	9.25	9.09
X-3	13.95	15.22	15.75	16.3
X-4	17.16	18.27	18.85	18.98
X-5	17.46	18.7	19.33	19.48
Y-1	7.95	8.2	8.3	7.05
Y-2	8.09	8.34	8.42	7.15
Y-3	11.76	12.48	12.82	[illegible]
Y-4	14.96	16.01	16.49	16.51
Y-5	15.74	17.04	17.62	17.66
Y-6	15.84	17.06	16.25	16.61
Y-7	15.23	16.42	14.85	15.59
Z-1	6.69	6.12	6.04	6.
Z-2	6.71	6.14	6.05	6.01
Z-3	9.76	[illegible]	[illegible]	[illegible]
Z-4	7.54	7.04	6.9	7.01
Z-5	7.58	7.05	7.1	7.02
Z-6	13.15	12.67	12.16	11.57
Z-7	12.36	11.11	10.67	10.13

b)

	Grid size 16	32	64	80
X-1	8.34	8.7	8.86	8.71
X-2	8.73	9.1	9.25	9.09
X-3	13.95	15.22	15.75	16.3
X-4	17.16	18.27	18.85	18.98
X-5	17.46	18.7	19.33	19.48
Y-1	7.95	8.2	[illegible]	7.05
Y-2	8.09	[illegible]	[illegible]	7.15
Y-3	[illegible]	12.48	12.82	[illegible]
Y-4	14.96	16.01	16.49	16.51
Y-5	15.74	17.04	17.62	17.66
Y-6	15.84	17.06	16.25	16.61
Y-7	15.23	16.42	14.85	15.59
Z-1	6.69	6.12	6.04	6.
Z-2	6.71	6.14	6.05	6.01
Z-3	9.76	[illegible]	[illegible]	[illegible]
Z-4	[illegible]	[illegible]	[illegible]	[illegible]
Z-5	[illegible]	[illegible]	[illegible]	[illegible]
Z-6	13.15	12.67	12.16	11.57
Z-7	12.36	11.11	10.67	10.13

c)

	Grid size 16	32	64	80
X-1	8.34	8.7	8.86	8.71
X-2	8.73	[illegible]	[illegible]	9.09
X-3	13.95	15.22	15.75	16.3
X-4	17.16	18.27	18.85	18.98
X-5	17.46	18.7	19.33	19.48
Y-1	7.95	8.2	8.3	7.05
Y-2	8.09	8.34	8.42	7.15
Y-3	[illegible]	12.48	12.82	[illegible]
Y-4	14.96	16.01	16.49	16.51
Y-5	15.74	17.04	17.62	17.66
Y-6	15.84	17.06	16.25	16.61
Y-7	15.23	16.42	14.85	15.59
Z-1	6.69	6.12	6.04	6.
Z-2	6.71	6.14	6.05	6.01
Z-3	[illegible]	8.37	8.46	8.29
Z-4	7.54	7.04	6.9	7.01
Z-5	7.58	7.05	7.1	7.02
Z-6	13.15	12.67	12.16	[illegible]
Z-7	12.36	[illegible]	[illegible]	[illegible]

Figure 2: Performance (Mflop/s) for MIPS R5000 (SGI Indy)

a)

	Grid size			
	16	32	64	80
X-1	22.41	22.43	17.22	17.27
X-2	22.41	22.43	17.21	17.26
X-3	50.95	52.99	29.27	29.13
X-4	48.81	49.82	28.37	28.67
X-5	47.07	47.9	27.49	27.29
Y-1	[illegible]	62.63	28.92	30.25
Y-2	[illegible]	62.65	28.93	30.24
Y-3	85.41	98.88	38.34	39.35
Y-4	84.99	98.64	38.48	39.36
Y-5	28.76	29.21	20.44	20.63
Y-6	28.76	29.22	20.43	20.63
Y-7	84.99	98.64	38.66	39.41
Z-1	29.31	31.4	22.46	20.96
Z-2	29.3	31.41	22.4	20.95
Z-3	85.81	99.15	37.03	37.07
Z-4	85.18	98.77	37.79	38.81
Z-5	28.81	29.24	20.4	20.51
Z-6	28.81	29.23	20.4	20.51
Z-7	80.25	98.76	39.82	39.47

b)

	Grid size			
	16	32	64	80
X-1	[illegible]	[illegible]	17.22	17.27
X-2	[illegible]	[illegible]	17.21	17.26
X-3	50.95	52.99	[illegible]	[illegible]
X-4	48.81	49.82	[illegible]	[illegible]
X-5	47.07	47.9	[illegible]	[illegible]
Y-1	[illegible]	62.63	[illegible]	[illegible]
Y-2	[illegible]	62.65	[illegible]	[illegible]
Y-3	85.41	98.88	[illegible]	[illegible]
Y-4	84.99	98.64	[illegible]	[illegible]
Y-5	[illegible]	[illegible]	20.44	20.63
Y-6	[illegible]	[illegible]	20.43	20.63
Y-7	84.99	98.64	[illegible]	[illegible]
Z-1	[illegible]	[illegible]	22.46	20.96
Z-2	[illegible]	[illegible]	22.4	20.95
Z-3	85.81	99.15	[illegible]	[illegible]
Z-4	85.18	98.77	[illegible]	[illegible]
Z-5	[illegible]	[illegible]	20.4	20.51
Z-6	[illegible]	[illegible]	20.4	20.51
Z-7	80.25	98.76	[illegible]	[illegible]

c)

	Grid size			
	16	32	64	80
X-1	22.41	22.43	17.22	17.27
X-2	22.41	22.43	17.21	17.26
X-3	[illegible]	[illegible]	29.27	29.13
X-4	[illegible]	[illegible]	28.37	28.67
X-5	[illegible]	[illegible]	27.49	27.29
Y-1	[illegible]	62.63	28.92	30.25
Y-2	[illegible]	62.65	28.93	30.24
Y-3	85.41	98.88	[illegible]	[illegible]
Y-4	84.99	98.64	[illegible]	[illegible]
Y-5	28.76	29.21	20.44	20.63
Y-6	28.76	29.22	20.43	20.63
Y-7	84.99	98.64	[illegible]	[illegible]
Z-1	29.31	[illegible]	22.46	20.96
Z-2	29.3	[illegible]	22.4	20.95
Z-3	85.81	99.15	[illegible]	[illegible]
Z-4	85.18	98.77	[illegible]	[illegible]
Z-5	28.81	29.24	20.4	20.51
Z-6	28.81	29.23	20.4	20.51
Z-7	80.25	98.76	[illegible]	[illegible]

Figure 3: Performance (Mflop/s) for MIPS R8000 (SGI PowerChallenge)

a)

	Grid size			
	16	32	64	80
X-1	62.37	63.37	52.85	45.8
X-2	62.37	63.39	52.85	45.82
X-3	[illegible]	[illegible]	72.36	63.84
X-4	126.6	132.4	[illegible]	[illegible]
X-5	131.7	130.6	64.52	53.98
Y-1	63.82	69.05	52.36	42.78
Y-2	63.65	69.34	52.35	42.89
Y-3	92.35	97.92	78.97	59.97
Y-4	101.7	106.8	[illegible]	[illegible]
Y-5	116.4	117.	[illegible]	[illegible]
Y-6	116.5	117.	[illegible]	[illegible]
Y-7	101.5	106.8	[illegible]	[illegible]
Z-1	61.72	56.84	45.01	40.37
Z-2	61.56	56.68	45.12	40.42
Z-3	[illegible]	[illegible]	64.42	58.84
Z-4	102.6	105.5	56.45	[illegible]
Z-5	117.1	114.1	58.16	[illegible]
Z-6	117.1	114.2	58.26	[illegible]
Z-7	102.8	105.6	56.45	[illegible]

b)

	Grid size			
	16	32	64	80
X-1	[illegible]	[illegible]	52.85	45.8
X-2	[illegible]	[illegible]	52.85	45.82
X-3	84.34	87.51	[illegible]	[illegible]
X-4	126.6	132.4	[illegible]	50.26
X-5	131.7	130.6	[illegible]	53.98
Y-1	[illegible]	[illegible]	52.36	42.78
Y-2	[illegible]	[illegible]	52.35	42.89
Y-3	92.35	97.92	78.97	[illegible]
Y-4	101.7	106.8	[illegible]	49.1
Y-5	116.4	117.	[illegible]	49.79
Y-6	116.5	117.	[illegible]	49.76
Y-7	101.5	106.8	[illegible]	49.07
Z-1	[illegible]	[illegible]	45.01	40.37
Z-2	[illegible]	[illegible]	45.12	40.42
Z-3	83.51	76.54	[illegible]	[illegible]
Z-4	102.6	105.5	[illegible]	46.5
Z-5	117.1	114.1	[illegible]	47.26
Z-6	117.1	114.2	[illegible]	47.32
Z-7	102.8	105.6	[illegible]	46.48

c)

	Grid size			
	16	32	64	80
X-1	[illegible]	[illegible]	52.85	45.8
X-2	[illegible]	[illegible]	52.85	45.82
X-3	84.34	87.51	[illegible]	[illegible]
X-4	126.6	132.4	[illegible]	50.26
X-5	131.7	130.6	[illegible]	53.98
Y-1	[illegible]	[illegible]	52.36	42.78
Y-2	[illegible]	[illegible]	52.35	42.89
Y-3	92.35	97.92	[illegible]	59.97
Y-4	101.7	106.8	[illegible]	49.1
Y-5	116.4	117.	[illegible]	49.79
Y-6	116.5	117.	[illegible]	49.76
Y-7	101.5	106.8	[illegible]	49.07
Z-1	[illegible]	56.84	45.01	40.37
Z-2	[illegible]	56.68	45.12	40.42
Z-3	83.51	[illegible]	[illegible]	58.84
Z-4	102.6	105.5	[illegible]	46.5
Z-5	117.1	114.1	[illegible]	47.26
Z-6	117.1	114.2	58.26	47.32
Z-7	102.8	105.6	56.45	46.48

Figure 4: Performance (Mflop/s) for MIPS R10000 (SGI Origin2000)

a) Grid size

	16	32	64	80
X-1	33.94	33.66	33.63	34.22
X-2	34.71	34.35	34.2	35.08
X-3				
X-4	43.26	44.12	44.51	44.66
X-5	49.58	49.47	49.87	50.95
Y-1			35.74	
Y-2	34.44	34.66	33.22	35.31
Y-3				
Y-4			37.91	
Y-5	41.97	42.16	42.2	42.35
Y-6	40.73	40.1	40.44	40.98
Y-7	43.53	43.24	43.76	43.9
Z-1	38.52	37.27	30.86	35.23
Z-2	37.15	36.32	30.43	34.9
Z-3	[illegible]	37.95	31.63	35.86
Z-4	38.16	37.91	31.66	31.18
Z-5	42.07	42.19	[illegible]	32.97
Z-6	41.56	41.03	40.42	38.55
Z-7	44.89	44.33	43.33	

b) Grid size

	16	32	64	80
X-1	33.94	33.66	33.63	34.22
X-2	34.71	34.35	34.2	35.08
X-3				
X-4	43.26	44.12	44.51	44.66
X-5	49.58	49.47	49.87	50.95
Y-1			[illegible]	
Y-2	34.44	34.66	33.22	[illegible]
Y-3				38.81
Y-4				
Y-5	41.97	42.16	42.2	42.35
Y-6	40.73	40.1	40.44	40.98
Y-7	43.53	43.24	43.76	43.9
Z-1	38.52		30.86	
Z-2			30.43	
Z-3	39.2	37.95	31.63	
Z-4	38.16	37.91	31.66	31.18
Z-5	42.07	42.19		[illegible]
Z-6	41.56	41.03	40.42	38.55
Z-7	44.89	44.33	43.33	

c) Grid size

	16	32	64	80
X-1	33.94	33.66	33.63	34.22
X-2	34.71	34.35	34.2	35.08
X-3	40.35			40.76
X-4	43.26	44.12	44.51	44.66
X-5	49.58	49.47	49.87	50.95
Y-1				
Y-2	34.44	34.66	33.22	35.31
Y-3				
Y-4				
Y-5	41.97	42.16	42.2	42.35
Y-6	40.73	40.1	40.44	40.98
Y-7	43.53	43.24	43.76	43.9
Z-1			30.86	35.23
Z-2			30.43	34.9
Z-3			31.63	35.86
Z-4			31.66	31.18
Z-5	42.07	42.19	33.77	32.97
Z-6	41.56	41.03	40.42	
Z-7	44.89	44.33	43.33	33.95

Figure 5: Performance (Mflop/s) for IBM RS6000 (SP-WN)

a) Grid size

	16	32	64	80
X-1	17.38	15.09	15.62	15.56
X-2	18.12	15.5	16.12	16.22
X-3	23.23	19.06	19.73	20.47
X-4	27.81	20.06	21.12	21.63
X-5	25.67	19.72	20.44	20.73
Y-1	19.95	16.88	17.19	17.07
Y-2	19.86	16.88	17.08	17.3
Y-3	24.07	19.43	19.79	20.28
Y-4	26.47	20.06	21.36	21.56
Y-5	24.58	19.86	20.6	20.93
Y-6	24.39	18.97	19.53	19.25
Y-7	25.81	19.52	19.85	19.42
Z-1	19.33		9.56	9.8
Z-2	19.57		9.53	9.75
Z-3	24.78	14.84	10.05	[illegible]
Z-4	26.24	9.22	7.44	7.29
Z-5	25.6	9.31	7.48	7.38
Z-6	26.32	18.53	18.91	19.2
Z-7	28.32	19.2	19.97	20.09

b) Grid size

	16	32	64	80
X-1	[illegible]	15.09	15.62	15.56
X-2		15.5	16.12	16.22
X-3	23.23			
X-4	27.81			21.63
X-5	25.67			
Y-1		16.88	17.19	17.07
Y-2		16.88	17.08	17.3
Y-3	24.07			
Y-4	26.47			
Y-5	24.58			
Y-6	24.39	[illegible]		
Y-7	25.81			
Z-1	19.33		[illegible]	[illegible]
Z-2	19.57		[illegible]	[illegible]
Z-3	24.78		[illegible]	[illegible]
Z-4	26.24	9.22	7.44	7.29
Z-5	25.6	9.31	7.48	7.38
Z-6	26.32	18.53	18.91	19.2
Z-7	28.32	19.2	19.97	20.09

c) Grid size

	16	32	64	80
X-1	17.38			
X-2	18.12			
X-3	23.23	19.06	19.73	20.47
X-4	27.81	20.06	21.12	21.63
X-5	25.67	19.72	20.44	20.73
Y-1	19.95	16.88	17.19	17.07
Y-2	19.86	16.88	17.08	17.3
Y-3	24.07	19.43	19.79	20.28
Y-4	26.47	20.06	21.36	21.56
Y-5	24.58	19.86	20.6	20.93
Y-6	24.39	18.97	19.53	19.25
Y-7	25.81	19.52	19.85	19.42
Z-1	19.33		9.56	9.8
Z-2	19.57		9.53	9.75
Z-3	24.78		10.05	10.3
Z-4	26.24	9.22	7.44	7.29
Z-5	25.6	9.31	7.48	7.38
Z-6	26.32	18.53	18.91	19.2
Z-7	28.32	19.2	19.97	20.09

Figure 6: Performance (Mflop/s) for Intel PentiumPro 200MHz (Whitney)

a)	Grid size			
	16	32	64	80
X-1	7.46	6.7	6.85	6.73
X-2	7.46	6.71	6.85	6.74
X-3	9.25	8.46	8.41	8.58
X-4	22.67	23.1	23.39	23.06
X-5	20.73	21.11	21.35	21.06
Y-1	[illegible]	[illegible]	[illegible]	6.97
Y-2	[illegible]	[illegible]	[illegible]	6.96
Y-3	7.78	8.04	7.23	6.78
Y-4	14.21	13.	11.25	10.98
Y-5	12.05	[illegible]	[illegible]	[illegible]
Y-6	16.32	16.19	15.76	15.38
Y-7	12.1	12.07	11.91	11.54
Z-1	[illegible]	[illegible]	[illegible]	6.5
Z-2	[illegible]	[illegible]	[illegible]	6.5
Z-3	7.31	6.42	6.52	6.48
Z-4	[illegible]	[illegible]	[illegible]	[illegible]
Z-5	[illegible]	[illegible]	[illegible]	[illegible]
Z-6	14.07	13.4	12.59	12.7
Z-7	11.02	10.63	9.99	10.01

b)	Grid size			
	16	32	64	80
X-1	[illegible]	6.7	6.85	6.73
X-2	[illegible]	6.71	6.85	6.74
X-3	[illegible]	[illegible]	[illegible]	[illegible]
X-4	22.67	23.1	23.39	23.06
X-5	20.73	21.11	21.35	21.06
Y-1	[illegible]	[illegible]	[illegible]	6.97
Y-2	[illegible]	[illegible]	[illegible]	6.96
Y-3	[illegible]	[illegible]	7.23	6.78
Y-4	14.21	13.	[illegible]	[illegible]
Y-5	12.05	[illegible]	[illegible]	[illegible]
Y-6	16.32	16.19	15.76	15.38
Y-7	12.1	12.07	11.91	[illegible]
Z-1	[illegible]	[illegible]	[illegible]	6.5
Z-2	[illegible]	[illegible]	[illegible]	6.5
Z-3	[illegible]	6.42	6.52	6.48
Z-4	[illegible]	[illegible]	[illegible]	[illegible]
Z-5	[illegible]	[illegible]	[illegible]	[illegible]
Z-6	14.07	13.4	12.59	12.7
Z-7	11.02	10.63	[illegible]	[illegible]

c)	Grid size			
	16	32	64	80
X-1	7.46	6.7	6.85	6.73
X-2	7.46	6.71	6.85	6.74
X-3	9.25	8.46	8.41	8.58
X-4	22.67	23.1	23.39	23.06
X-5	20.73	21.11	21.35	21.06
Y-1	[illegible]	[illegible]	8.96	6.97
Y-2	[illegible]	[illegible]	8.95	6.96
Y-3	7.78	8.04	7.23	6.78
Y-4	14.21	[illegible]	[illegible]	[illegible]
Y-5	[illegible]	[illegible]	[illegible]	[illegible]
Y-6	16.32	16.19	15.76	15.38
Y-7	[illegible]	[illegible]	[illegible]	[illegible]
Z-1	8.53	8.45	7.9	6.5
Z-2	8.54	8.46	7.9	6.5
Z-3	7.31	6.42	6.52	6.48
Z-4	8.49	8.06	7.77	7.65
Z-5	8.24	7.86	7.61	7.59
Z-6	14.07	[illegible]	[illegible]	[illegible]
Z-7	[illegible]	10.63	9.99	10.01

Figure 7: Performance (Mflop/s) for DEC Alpha EV4 (Cray T3D)

a)	Grid size			
	16	32	64	80
X-1	51.79	52.5	54.31	54.87
X-2	51.48	52.58	54.32	54.88
X-3	50.8	51.89	53.63	54.15
X-4	88.21	98.32	102.3	103.1
X-5	69.01	77.36	82.67	83.61
Y-1	37.02	[illegible]	[illegible]	37.91
Y-2	37.1	[illegible]	[illegible]	37.9
Y-3	36.43	[illegible]	[illegible]	36.67
Y-4	67.23	70.28	69.53	68.34
Y-5	69.31	72.61	71.82	70.56
Y-6	36.85	34.96	32.66	35.16
Y-7	37.66	36.57	32.11	35.83
Z-1	37.14	24.61	20.72	21.12
Z-2	37.49	24.54	20.75	21.16
Z-3	37.05	24.01	21.35	21.43
Z-4	53.01	35.97	34.26	34.78
Z-5	54.01	36.15	34.48	34.98
Z-6	36.5	34.69	35.18	28.42
Z-7	39.22	36.66	36.23	[illegible]

b)	Grid size			
	16	32	64	80
X-1	51.79	52.5	[illegible]	[illegible]
X-2	51.48	[illegible]	[illegible]	[illegible]
X-3	50.8	51.89	[illegible]	[illegible]
X-4	88.21	98.32	102.3	103.1
X-5	[illegible]	77.36	82.67	83.61
Y-1	[illegible]	[illegible]	[illegible]	[illegible]
Y-2	[illegible]	[illegible]	[illegible]	[illegible]
Y-3	[illegible]	[illegible]	[illegible]	[illegible]
Y-4	67.23	70.28	69.53	68.34
Y-5	69.31	72.61	71.82	70.56
Y-6	[illegible]	34.96	32.66	35.16
Y-7	[illegible]	[illegible]	32.11	35.83
Z-1	[illegible]	[illegible]	20.72	21.12
Z-2	37.49	[illegible]	20.75	21.16
Z-3	[illegible]	[illegible]	[illegible]	21.43
Z-4	53.01	[illegible]	[illegible]	[illegible]
Z-5	54.01	[illegible]	[illegible]	[illegible]
Z-6	[illegible]	[illegible]	[illegible]	[illegible]
Z-7	39.22	[illegible]	[illegible]	[illegible]

c)	Grid size			
	16	32	64	80
X-1	[illegible]	[illegible]	[illegible]	[illegible]
X-2	[illegible]	[illegible]	[illegible]	[illegible]
X-3	[illegible]	[illegible]	[illegible]	[illegible]
X-4	88.21	98.32	102.3	103.1
X-5	69.01	77.36	82.67	83.61
Y-1	[illegible]	[illegible]	[illegible]	37.91
Y-2	[illegible]	[illegible]	[illegible]	37.9
Y-3	[illegible]	[illegible]	[illegible]	[illegible]
Y-4	67.23	70.28	69.53	68.34
Y-5	69.31	72.61	71.82	70.56
Y-6	[illegible]	34.96	32.66	35.16
Y-7	[illegible]	36.57	32.11	35.83
Z-1	[illegible]	24.61	20.72	21.12
Z-2	[illegible]	24.54	20.75	21.16
Z-3	[illegible]	24.01	21.35	21.43
Z-4	[illegible]	[illegible]	[illegible]	[illegible]
Z-5	[illegible]	[illegible]	[illegible]	[illegible]
Z-6	[illegible]	[illegible]	[illegible]	28.42
Z-7	[illegible]	36.66	36.23	24.62

Figure 8: Performance (Mflop/s) for DEC Alpha EV5 (Cray T3E)

a)	Grid size			
	16	32	64	80
X-1	31.87	24.52	24.95	25.05
X-2	33.61	25.28	25.73	25.83
X-3	32.	24.54	25.05	25.16
X-4	45.68	31.02	31.61	31.72
X-5	41.99	29.4	29.98	30.08
Y-1	28.18	24.47	25.04	22.3
Y-2	28.83	[illegible]	25.27	22.37
Y-3	28.7	[illegible]	25.08	21.86
Y-4	32.46	24.09	23.74	[illegible]
Y-5	34.51	25.31	25.	24.65
Y-6	34.48	25.89	26.36	26.42
Y-7	33.02	[illegible]	25.19	25.22
Z-1	27.95	21.85	19.69	19.73
Z-2	28.72	22.02	19.74	19.75
Z-3	27.75	21.43	19.29	19.3
Z-4	27.49	[illegible]	16.23	15.29
Z-5	28.78	20.88	[illegible]	16.14
Z-6	34.54	21.9	19.84	19.86
Z-7	33.62	19.11	15.5	15.5

b)	Grid size			
	16	32	64	80
X-1	[illegible]	24.52	24.95	25.05
X-2	[illegible]	25.28	25.73	25.83
X-3	[illegible]	24.54	25.05	25.16
X-4	45.68	[illegible]	[illegible]	[illegible]
X-5	41.99	[illegible]	[illegible]	[illegible]
Y-1	[illegible]	[illegible]	[illegible]	22.3
Y-2	28.83	[illegible]	[illegible]	22.37
Y-3	28.7	[illegible]	[illegible]	21.86
Y-4	32.46	[illegible]	[illegible]	[illegible]
Y-5	34.51	[illegible]	[illegible]	[illegible]
Y-6	34.48	[illegible]	[illegible]	[illegible]
Y-7	33.02	[illegible]	[illegible]	[illegible]
Z-1	27.95	[illegible]	[illegible]	[illegible]
Z-2	28.72	[illegible]	[illegible]	[illegible]
Z-3	27.75	[illegible]	[illegible]	[illegible]
Z-4	27.49	[illegible]	16.23	15.29
Z-5	28.78	[illegible]	17.1	16.14
Z-6	34.54	[illegible]	[illegible]	[illegible]
Z-7	33.62	[illegible]	15.5	15.5

c)	Grid size			
	16	32	64	80
X-1	31.87	[illegible]	[illegible]	[illegible]
X-2	33.61	[illegible]	[illegible]	[illegible]
X-3	32.	[illegible]	[illegible]	[illegible]
X-4	45.68	31.02	31.61	31.72
X-5	41.99	29.4	29.98	30.08
Y-1	[illegible]	[illegible]	[illegible]	22.3
Y-2	28.83	[illegible]	[illegible]	22.37
Y-3	[illegible]	[illegible]	[illegible]	21.86
Y-4	32.46	[illegible]	[illegible]	[illegible]
Y-5	34.51	[illegible]	[illegible]	[illegible]
Y-6	34.48	[illegible]	[illegible]	[illegible]
Y-7	33.02	[illegible]	[illegible]	[illegible]
Z-1	[illegible]	21.85	19.69	19.73
Z-2	[illegible]	22.02	19.74	19.75
Z-3	[illegible]	21.43	19.29	19.3
Z-4	[illegible]	19.95	16.23	15.29
Z-5	[illegible]	20.88	17.1	16.14
Z-6	34.54	21.9	19.84	19.86
Z-7	33.62	19.11	15.5	15.5

Figure 9: Performance (Mflop/s) for Sun UltraSparc (Berkeley NOW)

Appendix C: Simulated cache performance

	Grid size			
	16	32	64	80
X-1	6.78	6.12	5.83	5.77
X-4	6.45	5.97	5.76	5.71
X-5	6.45	5.97	5.76	5.71
Y-1	6.48	5.98	5.76	9.89
Y-4	6.63	6.06	5.8	5.75
Y-5	6.63	6.06	5.8	5.75
Y-6	6.63	6.06	■	6.08
Y-7	6.63	6.06	6.88	6.18
Z-1	10.2	■	■	■
Z-4	15.6	16.9	16.8	16.8
Z-5	15.6	16.9	16.8	16.8
Z-6	6.63	6.06	5.84	5.75
Z-7	6.63	6.06	5.8	5.75

Figure 10: Cache misses (%), MIPS R5000

	Grid size			
	16	32	64	80
X-1	0.	0.	0.369	0.369
X-4	0.	0.	0.369	0.359
X-5	0.	0.	0.369	0.359
Y-1	0.	0.	0.369	0.359
Y-4	0.	0.	0.369	0.359
Y-5	0.	0.	0.369	0.359
Y-6	0.	0.	0.369	0.359
Y-7	0.	0.	0.369	0.359
Z-1	0.	0.	0.369	0.359
Z-4	0.	0.	0.369	0.359
Z-5	0.	0.	0.369	0.359
Z-6	0.	0.	0.369	0.359
Z-7	0.	0.	0.369	0.359

Figure 11: Cache misses (%), MIPS R8000

	Grid size			
	16	32	64	80
X-1	0.	0.06	1.46	1.45
X-4	0.	0.229	1.46	1.44
X-5	0.	0.229	1.46	1.44
Y-1	0.	0.05	1.45	1.43
Y-4	0.	0.229	1.46	1.44
Y-5	0.	0.229	1.46	1.44
Y-6	0.	0.229	1.46	1.44
Y-7	0.	0.229	1.46	1.44
Z-1	0.	0.06	1.45	1.43
Z-4	0.	0.229	1.46	1.44
Z-5	0.	0.229	1.46	1.44
Z-6	0.	0.11	1.46	1.44
Z-7	0.	0.11	1.46	1.44

Figure 12: Cache misses (%), MIPS R10000

	Grid size			
	16	32	64	80
X-1	0.636	0.784	0.735	0.725
X-4	0.587	0.764	0.725	0.725
X-5	0.587	0.764	0.725	0.725
Y-1	0.537	0.764	0.725	0.715
Y-4	0.587	0.764	0.725	0.725
Y-5	0.577	0.764	0.725	0.725
Y-6	0.547	0.764	0.754	0.784
Y-7	0.547	0.764	0.754	0.784
Z-1	0.507	0.764	■	■
Z-4	■	0.764	2.12	2.11
Z-5	■	0.764	2.12	2.11
Z-6	0.587	0.764	0.725	0.725
Z-7	0.587	0.764	0.725	0.725

Figure 13: Cache misses (%), IBM RS6000

	Grid size			
	16	32	64	80
X-1	3.98	6.12	5.83	5.77
X-4	3.22	5.97	5.76	5.71
X-5	3.22	5.97	5.76	5.71
Y-1	2.84	5.98	5.76	5.71
Y-4	3.72	6.06	5.8	5.75
Y-5	3.72	6.06	5.8	5.75
Y-6	3.73	6.06	5.8	5.75
Y-7	3.73	6.06	5.8	5.75
Z-1	3.18	5.98	[illegible]	[illegible]
Z-4	[illegible]	6.06	16.8	16.8
Z-5	[illegible]	6.06	16.8	16.8
Z-6	3.74	6.06	5.8	5.75
Z-7	3.74	6.06	5.8	5.75

Figure 14: Cache misses (%), PentiumPro

	Grid size			
	16	32	64	80
X-1	6.78	6.12	5.83	5.77
X-4	6.86	[illegible]	[illegible]	[illegible]
X-5	7.05	6.53	6.31	6.26
Y-1	10.6	5.98	9.93	[illegible]
Y-4	8.94	10.3	14.	15.2
Y-5	8.98	10.4	14.	15.2
Y-6	7.6	7.4	7.76	8.28
Y-7	7.56	7.46	8.62	8.98
Z-1	[illegible]	12.9	12.7	12.7
Z-4	17.6	17.3	17.2	17.2
Z-5	17.6	17.4	17.2	17.2
Z-6	7.83	7.66	8.19	7.94
Z-7	7.56	8.15	8.36	8.36

Figure 15: Cache misses (%), DEC EV4

	Grid size			
	16	32	64	80
X-1	6.78	6.12	5.83	5.77
X-4	6.45	5.97	5.76	5.71
X-5	6.45	5.97	5.76	5.71
Y-1	6.48	5.98	5.76	5.71
Y-4	6.63	6.06	5.8	5.75
Y-5	6.63	6.06	5.8	5.75
Y-6	6.63	6.06	5.89	5.75
Y-7	6.63	6.06	5.91	5.75
Z-1	6.48	[illegible]	[illegible]	[illegible]
Z-4	6.63	16.9	16.8	16.8
Z-5	6.63	16.9	16.8	16.8
Z-6	6.63	6.06	5.8	5.75
Z-7	6.63	6.06	5.8	5.75

Figure 16: Cache misses (%), DEC EV5

	Grid size			
	16	32	64	80
X-1	0.	3.07	2.91	2.88
X-4	0.	3.04	2.91	2.88
X-5	0.	3.04	2.91	2.88
Y-1	0.	3.	2.88	2.86
Y-4	0.	3.1	2.99	2.98
Y-5	0.	3.1	2.99	2.98
Y-6	0.	3.09	2.97	2.96
Y-7	0.	3.1	2.99	2.98
Z-1	0.	3.	4.93	5.61
Z-4	0.	4.11	6.89	8.42
Z-5	0.	4.11	6.89	8.42
Z-6	0.	3.09	2.97	2.95
Z-7	0.	3.1	2.98	2.97

Figure 17: Cache misses (%), UltraSparc