ELSEVIER

Advances in Engineering Software 31 (2000) 649–654

ADVANCES IN ENGINEERING SOFTWARE

www.elsevier.com/locate/advengsoft

An experimental study of improving performance by scalar replacement

M.E. Hayder

Center for Research on High Performance Software, Rice University, Houston, TX 77005-1892, USA

Abstract

We consider two computational fluid dynamics application codes known as HELIX and MAGI to study optimization by scalar replacement, i.e. loading array element values, those are used repeatedly, into scalars to enable them to be register allocated by the back-end compiler. This technique lowers the execution time by reducing the number of load/store instructions. HELIX computations are on structured grid and are regular, while MAGI computations are irregular. Our experiments compare three versions of HELIX—the original code, a version in which scalar replacement was performed by hand, and version in which scalar replacement was performed automatically by Memoria—a tool for performing scalar replacement developed at Rice University and Michigan Technological University. Our experiments show that scalar replacement improved performance of HELIX by 4–7% over and above performance obtained by using the highest level of optimization with vendor-supplied compilers on an SGI Origin, an SGI O2 workstation, an IBM SP2, and a Cray T3E. We also compare the original and a hand modified version of MAGI on a O2 workstation using three different data sets. Scalar replacement reduced execution time up to about 9%.

Keywords: Scalar replacement; Single processor optimization; Performance improvement; Computational fluid dynamics

1. Introduction

Architectures of modern supercomputers are continuously changing. An optimal code for traditional vector machines is highly unlikely to be optimal for newer machines that are based on deep memory hierarchies. A developer of an application has knowledge of the algorithm and dependencies among variables and sections of the code. With some knowledge of optimization techniques, he may be able to improve the performance of the application. Compilers on the other hand have to discover all optimization opportunities from the code, which at times may be difficult, if not impossible. Sometimes compiler based program transformations can aid the developer in achieving high performance by restructuring the code at the source level so that a back-end compiler can generate efficient code. In this study we concentrate on a particular type of optimization, namely scalar replacement. Motivation for our present work comes from the code migration group at Major Shared Resource Center (MSRC) operated by the US Army Engineer Research and Development Center (ERDC), who indicated that scalar replacement could be used to significantly improve performance of some of their codes. We undertook this study to examine two computational fluid dynamics (CFD) codes of interest to ERDC MSRC. We performed our experiments on an SGI origin, an SGI O2 workstation, Cray T3E and an IBM SP2. All computations were performed on a single node. We present brief descriptions of two codes in the next section, followed by a description of our optimization technique. Simulation results and conclusions are presented in following sections.

2. Applications

The first application code we consider in this study is known as the HELIX [1,2] code. HELIX is used for flow computation of interest to helicopter designers. The second code we study is known as MAGI [3] which is a shock and material response code. The computational domain for HELIX is three-dimensional (3D). It can be 2D or 3D for MAGI. Computations in HELIX are on structured grid and, therefore, have regular pattern; on the other hand computations in MAGI are irregular and adapt throughout execution. We chose these two applications from roughly opposite ends of computational pattern spectrum in a hope that some conclusions drawn from these codes might be applicable to typical application codes. HELIX code solves the unsteady full potential equation in three dimensions on an Eulerian grid with embedded vortical velocity that models the influence of wake. This is a moderate size FORTRAN code with about 6500 source lines. MAGI uses Smoothed Particle Hydrodynamics (SPH) technique for inviscid hydrodynamics with additional terms to handle dynamic material response of solids and works in a pure Lagrangian frame work. This code is suitable for computations to track

0965-9978/00/$ - see front matter
PII: S0965-9978(00)00011-9

debris clouds produced by hypervelocity impacts. A significant portion of computation time is spent in building a list of neighboring particle which is updated every time step. MAGI is a moderate size FORTRAN code of about 7500 source lines.

3. Optimization

3.1. Techniques

There are various optimization techniques which one may use to improve performance of a code. Our present study focuses on a specific type of optimization, namely scalar replacement. We provide a brief introduction to this optimization. Other types of optimizations, such as elimination of common sub expressions, loop unrolling, unit stride access, etc. may also improve the performance of a particular code significantly. Amount of improvements to a code will depend on the algorithm and how it is implemented.

In modern computers there are different types of storage with different access times. If a variable resides in the register, it can be accessed in a clock cycle without generating memory traffic. When an array is accessed in some regular pattern, say in a typical finite difference simulation, some variables used for computations at a grid point may be reused in computations at neighboring grid points. In many cases, load store traffic can be reduced for some of these references by exploiting temporal locality and storing appropriate variables in registers. One way to hint to a back-end compiler that an array reference should be stored in a register is to replace references to arrays with scalars. In the following code

```
DO I = 2, N
  A(I) = C1*A(I) + C2*(A(I + 1) - A(I
  - 1))
ENDDO
```

A(I − 1) is generated in the previous iteration, which may be saved in a register for reuse by the next iteration without loading the value from memory. Such optimization opportunities should be discovered by a good compiler. However, some compiler may miss such opportunities. For these compiler some restructuring of the code as shown below will make may it easier to generate efficient executables.

```
T = A(1)
DO I = 2, N
  T = C1*A(I) + C2*(A(I + 1) - T)
  A(I) = T
ENDDO
```

Additional discussion on scalar replacement may be found elsewhere [4–6]. Discussions on other optimization techniques may also be found in these references.

3.2. Tools

Some shortcomings in vendor compilers may be overcome by using a preprocessor which will modify the code so that that the compiler will be able to easily find temporal localities. One such tool is Memoria which uses optimization techniques discussed in Refs. [5,6] to generate code with optimal scalar replacement. This tool was initially developed at Rice University and has been further enhanced at Michigan Technological University by Steve Carr and his coworkers. Input to Memoria consists of a FORTRAN source code along with information about the target machine architecture. Memoria performs dependence analysis to generate an improved code. Since scalar replacement is already done in the code, the compiler does not need to worry about this particular optimization.

3.3. Modifications to codes

We experimented with three versions of the HELIX code in this study. They are

- *The original code:* This version of the code is written with many array references.
- *Memoria modified code:* Memoria performed scalar replacement in the YSWEEP routine of the original code. Preprocessing was kept limited to the YSWEEP routine to get a reasonable comparison between hand tuned code and Memoria modified code.
- *Hand tuned code:* We performed scalar replacement in two key loops (Do loops 32 and 105) in YSWEEP routine by hand. Loop 32 within the YSWEEP routine computes metric terms of computational cells, derivative terms for the potential and velocities for interior cells. Computations in loop 105 are similar to those in loop 32, except they are for boundary points whereas computations in loop 32 are for interior points.

In Figs. 1–3 we give portions of loop 32 in our three versions.

We experimented with two versions of the MAGI code in this study. They are

- *The original code:* A version of the code from Philips Laboratory.
- *Hand tuned code:* A hand tuned code where we performed scalar replacement in a few important routines, mainly in the STRSTEMP routine.

We found that optimization techniques such as elimination of common subexpressions were applicable on the hand tuned scalar version of the code and improved performance. However, in this study we restricted our optimization to scalar replacement.

4. Results

We used SGI's Perfex tool (see SpeedShop User's Guide

(a)

```
  DO 15 I = 1, NX+1
   X(I)    = X3D(I,J,K)
   XM(I)   = X3D(I,J+1,K)
   XR(I)   = X3D(I,J,K+1)
   XRM(I)  = X3D(I,J+1,K+1)
   ******
   ******

15   CONTINUE
C
  DO 32 I=1,NX
   XX   = XR(I+1)-XR(I)+XRM(I+1)-XRM(I)
   XXS  = X(I+1) -X(I) +XM(I+1) -XM(I)
   XX   = XX +XXS
   XY   = XR(I+1)+XR(I)-XRM(I+1)-XRM(I)
   XYS  = X(I+1) +X(I) -XM(I+1) -XM(I)
   XY   = XY +XYS
   *******
   *******
32 continue
```

Fig. 1. Part of original loops 15 and 32.

[7] for description of this utility) on an SGI origin and an SGI O2 workstation to gather performance statistics. Tests on the SGI origin were done by running the job on a single processor in the batch mode while other users were also using the system. The O2 workstation was not shared with any other user for our measurements. We made two passes over the entire code (by wrapping a simple DO loop around the entire solver) and report timings of the second pass. It was done in an attempt to minimize "cold start" memory allocation and I/O effects in our experiments. All statistics were gathered multiple times and we report the averages. Measurements that showed noticeable variations from the mean were discarded. We used -O0 (no optimization) and -O3 (aggressive optimization) compiler options to

(b)

```
   DO 32 I=1,NX
    XSUBI     = X3D(I,J,K)
    XSUBIP1   = X3D(I+1,J,K)
    XMSUBI    = X3D(I,J+1,K)
    XMSUBIP1  = X3D(I+1,J+1,K)
    XRSUBI    = X3D(I,J,K+1)
    XRSUBIP1  = X3D(I+1,J,K+1)
    XRMSUBI   = X3D(I,J+1,K+1)
    XRMSUBIP1 = X3D(I+1,J+1,K+1)
    XX        = XRSUBIP1 -XRSUBI
  &            +XRMSUBIP1 -XRMSUBI
  &            +(XSUBIP1  -XSUBI
  &            +XMSUBIP1  -XMSUBI)
    XY        = XRSUBIP1 +XRSUBI
  &            -XRMSUBIP1 -XRMSUBI
  &            +(XSUBIP1 +XSUBI
  &            -XMSUBIP1  -XMSUBI)
    *******
    *******
 32 continue
```

Fig. 2. Hand modified code of section shown in Fig. 1.

(c)

```
  do i = 1, nx + 1
   x(i) = x3d(i, j, k)
   xm(i) = x3d(i, j + 1, k)
   xr(i) = x3d(i, j, k + 1)
   xrm(i) = x3d(i, j + 1, k + 1)
   ******
  enddo

  do i = 1, nx
  xr$0$0 = xr(i + 1)
  xr$1$0 = xr(i)
  xrm$2$0 = xrm(i + 1)
  xrm$3$0 = xrm(i)
  xx =xr$0$0 -xr$1$0 +xrm$2$0-xrm$3$0
  x$4$0 = x(i + 1)
  x$5$0 = x(i)
  xm$6$0 = xm(i + 1)
  xm$7$0 = xm(i)
  xxs =x$4$0 -x$5$0 +xm$6$0 -xm$7$0
  xx = xx + xxs
  xy =xr$0$0 +xr$1$0 -xrm$2$0-xrm$3$0
  xys =x$4$0 +x$5$0 -xm$6$0 -xm$7$0
  xy = xy + xys
  *******
  *******
  enddo
```

Fig. 3. Memoria modified code of section shown in Fig. 1.

generate executables. When aggressive compiler option (-O3) is used, the compiler goes through detailed analysis of the code and performs a significant amount of optimizations. It may also perform scalar replacement optimization, which we performed on the source code by hand and Memoria performed automatically. Comparisons for cases when no optimization, i.e. -O0 flag is used show the impact of scalar replacement optimization performed by Memoria and by hand. On the other hand, results when aggressive optimization is used compare optimization done by the compiler (which may include scalar optimization) with those done by Memoria and by hand. Later comparisons give indication of the amount of performance improvements a typical user may get when scalar replacement is done on the original source by hand or by a tool like Memoria. Therefore, we present results for both no optimization and aggressive optimization compilations.

We first consider computations with the HELIX code. All such computations in this study were for 10 iterations. We chose to limit our tests to only 10 iterations to keep computation times reasonable while maintaining a fairly good representation of computational characteristics. Results of computations for different compilation flag using a $101 \times 59 \times 29$ grid on a single node SGI origin are shown in Table 1. Items presented in this table are execution time in seconds, percentages of primary and secondary data cache misses, TLB misses, number of graduated load and store in billions, and percentages of total number of graduated load and stores compared to the original code (this is shown as % LS in the table). Profiling the original code shows that most (over 90%) of the computation time is spent in the YSWEEP

Table 1
Computation of HELIX on a single node SGI origin

Observation	Orig.	Memoria	Hand
(a) No optimization compilation			
Time (s)	6326.3	6107.1	5670.4
% pr. miss	0.786	0.788	0.815
% sec. miss	3.808	3.797	4.421
% TLB miss	0.001	0.001	0.001
Load	568.4	557.8	462.7
Store	68.1	81.7	70.9
% LS	100.0	100.5	83.8
(b) Aggressive optimization compilation			
Time (s)	1226.4	1242.7	1175.9
% pr. miss	8.145	7.909	8.096
% sec. miss	3.021	3.064	3.295
% TLB miss	0.005	0.004	0.005
Load	89.5	95.0	83.5
Store	35.7	31.6	30.8
% LS	100.0	101.1	91.3

routine. Total number of graduated loads and stores reduced by about 80% when aggressive compilation (-O3 option) was used over no optimization compilation on the original code. Effect of scalar replacement optimizations are noticeable in Table 1(a). The hand tuned version reduced total number of graduated load and store by about 16% and was about 10% faster than the original version for no optimization case. The reduction in number of load and store, and execution times dropped by about 9 and 4%, respectively for aggressive optimization case. For no optimization case, Memoria modified code made selected replacements and had about the same total number of graduated load and stores compared to the original code (in the Memoria modified code number of loads is lower, but number of stores is higher) which reduced computation time by about 3.5%. When aggressive compilation option was used, total number of load and stores was slightly higher for Memoria modified code, and the execution time increased slightly.

Table 2
Computation of HELIX on O2 workstation

Observation	Orig.	Memoria	Hand
(a) No optimization compilation			
Time (s)	7603.7	7287.0	6712.5
% pr. miss	0.857	0.935	0.852
% sec. miss	11.056	10.682	11.530
% TLB miss	0.018	0.020	0.018
Load	544.5	488.4	465.8
Store	68.6	82.4	70.9
% LS	100.0	93.1	87.5
(b) Aggressive optimization compilation			
Time (s)	1497.9	1500.4	1420.9
% pr. miss	6.106	5.953	5.918
% sec. miss	9.502	9.501	10.435
% TLB miss	0.080	0.079	0.071
Load	47.3	45.8	46.0
Store	32.3	36.0	27.0
% LS	100.0	102.8	91.7

Table 3
Cache sizes on our test platforms

Cache type	Origin	O2 workstation
Primary	32 KB	32 KB
Secondary	4 MB	1 MB
TLB	2 MB	512 KB

There were less than 1% primary data cache misses when no optimization (-O0) flag was turned on, while it increased to approximately 8% for -O3 option. The primary data cache hit ratio is one measure of optimization, but it is not the only measure. For example, say a variable x is stored in a register instead of keeping it in cache and the program accesses x repeatedly. Then there will be only one cache miss initially and x will be loaded to a register. Since x now resides in a register, it will be used without any need to access it from cache, i.e. no cache hit. On the other hand, if x was not loaded on a register, there would be a cache hit for subsequent uses of x. If all other data access patterns remain unchanged, then the program with x assigned to a register will run faster although it will have higher percentage of cache misses. A close examination of loop 32 shows that many variables are accesses along a line while their structures are 3D. Many data elements (field variables) are accessed repeatedly and used only once in different phases of computations. Due to the limited size of cache, these elements are evicted (from cache) between their uses.

We repeated our tests on the O2 workstation. Both of these machines use the MIPS R10000 processor. Tests on the workstation were done because of our easy access to this machine and they also provided additional data on similar platform for verification of data in our experiments. The workstation was not shared with any other user during our tests, and there were only very small variations in our measurements. Results on the O2 workstation are shown in Table 2 and they are in the same format as in Table 1. While the total number of graduated load and store did not change significantly on the SGI origin for Memoria modified code compiled with -O0 compiler option, it reduced by about 7% on the O2 workstation. Secondary data cache and TLB misses on the O2 workstation were much higher than those on the origin node. This was due to smaller size secondary cache and page size on the O2 workstation. Cache sizes for origin and O2 workstation are given in Table 3. Other results in Tables 1 and 2 show the same general trend.

In Table 4 we show the result for computations on a $149 \times 59 \times 29$ grid on the O2 workstation. For no optimization case, the hand tuned version reduced total number of graduated load and store by about 12% and was about 12% faster than the original version. The reduction in number of load and store, and execution times dropped by about 5 and 6%, respectively for aggressive optimization

Table 4
Computation of HELIX for a larger grid on O2 workstation

Observation	Orig.	Memoria	Hand
(a) No optimization compilation			
Time (sec)	10414.0	9938.2	9138.3
% pr. miss	1.026	1.094	1.033
% sec. miss	9.594	9.336	9.304
% TLB miss	0.016	0.016	0.015
Load	799.9	719.0	685.7
Store	101.2	121.4	105.4
% LS	100.0	93.3	87.8
(b) Aggressive optimization compilation			
Time (s)	2239.5	2249.3	2113.9
% pr. miss	8.021	7.993	7.962
% sec. miss	6.387	6.435	6.727
% TLB miss	0.047	0.034	0.052
Load	120.6	124.5	116.6
Store	52.9	46.7	47.4
% LS	100.0	98.7	94.5

case. For no optimization case, Memoria modified code reduced total number graduated load and stores by 7% and execution time by 5%. Results of the Memoria modified and original codes were comparable when aggressive compilation option was used.

Our study also included computations on a Cray T3E and an IBM SP2 using the $101 \times 59 \times 29$ grid. Execution times (in seconds) for the HELIX code on a single node T3E are shown in Table 5. Memoria modified and hand tuned versions were about 6 and 12%, respectively faster than the original code when compiled with -O0 compiler flag. Hand tuned version was about 6% faster on T3E and about 7% faster on SP2 than the original code when aggressive compilation option was used.

We next consider the MAGI code on our O2 workstation. Through profiling we identified four subroutines which takes about 70% of computing time. We concentrated on those routines for tuning (i.e. scalar replacement) by hand. Results are shown in Table 6. They are in the same format as in Table 1. We used three different data sets (described in Table 6(a))—one small set for 2D computations and two larger sets for 3D computations. Number of particles in a real application usually is much larger. We chose these data sets to keep computation time reasonable and study problems with different irregular data pattern. Only a few particles were active at the beginning of computations. Other particles became active during computations as objects underwent disintegration due to impacts. Results for computations with no optimization compilation and aggressive optimization compilations are shown in Table 6(b) and (c), respectively. Secondary data

Table 5
Computation time (s) of HELIX on a single node T3E

Compiler flag	Orig.	Memoria	Hand
-O0	12563	11814	11080
-O3	2038	2094	1920

Table 6
Computation of MAGI on O2 workstation

Set	Domain	Particles	Objects
(a) Test cases			
S1	3D	35,088	2
S2	3D	28,000	5
S3	2D	8,528	2
(b) No optimization compilation			
Observation	S1 Orig.	S2 Orig.	S3 Orig.
Time (s)	5139.1	2935.6	1675.1
% pr. miss	1.282	1.544	0.592
% sec. miss	25.235	32.105	3.349
% TLB miss	0.522	0.697	0.168
Load	599.2	298.3	306.0
Store	105.5	52.1	31.6
% LS	100.0	100.0	100.0
Observation	S1 Scal	S2 Scal	S3 Scal
Time (s)	4881.2	2760.8	1673.7
% pr. miss	1.364	1.645	0.600
% sec. miss	25.118	31.477	4.312
% TLB miss	0.570	0.756	0.171
Load	536.0	267.7	300.2
Store	123.6	61.0	33.6
% LS	93.6	93.8	98.9
(c) Aggressive optimization compilation			
Observation	S1 Orig.	S2 Orig.	S3 Orig.
Time (s)	2548.6	1629.9	567.9
% pr. miss	6.510	7.181	4.308
% sec. miss	19.275	27.090	3.312
% TLB miss	2.165	2.749	0.806
Load	125.9	67.1	47.5
Store	37.2	18.0	13.8
% LS	100.0	100.0	100.0
Observation	S1 Scal	S2 Scal	S3 Scal
Time (s)	2347.8	1489.0	565.8
% pr. miss	6.694	7.105	4.334
% sec. miss	18.958	27.120	3.302
% TLB miss	2.188	2.674	0.811
Load	124.9	67.0	47.1
Store	37.5	18.2	13.8
% LS	99.5	100.1	99.3

cache and TLB misses were significantly higher in MAGI computations than those in HELIX computations. This is because of irregular data accesses in MAGI (while data access pattern in HELIX was regular). Total number of graduated load and store reduced by about 6% for Sets S1 and S2, and by only 1% for S3 when -O0 (no optimization) compiler flag was used. Computation times reduced by 5–6% for S1 and S2, but it remained essentially the same for S3. When aggressive compilation was used, total number of graduated load and store dropped by only 1% or less. Execution times dropped by 8–9% for S1 and S2, and change in S3 was only slight.

5. Conclusions

In summary, we find the same basic trend for HELIX

computations on all three platforms, namely hand tuning gave best results, which reduced computation time of the original code by up to 12% when no optimization compiler option, and up to 7% when aggressive compiler option was used. Memoria performed selective scalar replacement based on the architecture information and reduced execution time by 3.5–7% when no optimization flag was used. For the MAGI code, reduction in execution time varied from almost none to about 9%. Effects of hand coded scalar replacement or scalar replacement by Memoria were more pronounced when no optimization flag was turned on during compilation. Hand tuning, Memoria or a similar tool can help achieve high performance specially when the compiler is not extremely adept in optimization. However, there will be less benefit if the compiler can generate efficient code. We expect the general trend in our study will also be true for a typical CFD or similar class of codes.

Acknowledgements

Some preliminary results were presented at the DoD HPCMP UGC99 [8]. Contributions of John Mellor-Crummey for that study are highly leveraged in this paper. We are grateful to Steve Carr for providing us with the Memoria tool and many advices via e-mail. We are also grateful to David Medina for giving us access to the MAGI code. Thanks to Dave Whalley for many helpful discussions. Marty Moulton and Steve Bova provided us with grid files for the HELIX code. This work was supported in part by a grant of computing time from the DoD High Performance Computing Modernization Program.

References

[1] Ramachandran K, Tung C, Caradonna C. Rotor hover performance prediction using a free-wake computational fluid dynamics method. Journal of Aircraft 1989;26(12):1105–10.
[2] Steinhoff J, Ramachandran K. Free-wake analysis of compressible rotor flows. AIAA Journal 1990;28(3):426–31.
[3] Libersky LD, Petschek AG, Carney TC, Hipp JR, Allahdadi FA. High starin Lagrangian hydrodynamics—a three-dimensional SPH code for dynamic material response. Journal of Computational Physics 1993;109(1):67–75.
[4] Callahan D, Cocke J, Kennedy K. Estimating interlock and improving balance for pipelined machines. Journal of Parallel and Distributed Computing 1988;5:334–58.
[5] Carr S, Kennedy K. Improving the ratio of memory operations to floating-point operations in loops. ACM Transactions on Programming Languages and Systems 1994;16(6):1768–810.
[6] Carr S, Kennedy K. Scalar replacement in the presence of conditional control flow. Software—practice and experience 1994;24(1):51–77.
[7] SpeedShop User's Guide, Document Number 007-3311-003, Silicon Graphics, Inc.
[8] Hayder ME, Mellor-Crummey, J. Improving performance by scalar replacement, Presented in the DoD High Performance Computing Modernization Program Users Group Conference, June 1999.