MHAOTEU: Memory Hierarchy Analysis and Optimization Tools for the End−User

REPORT M3.D3

Task 3.4 Process for Optimizing an Application

Christine Eisenbeis, Andry Randrianatoavina, François Thomasset, Sid Ahmed Ali Touati, INRIA.
Olivier Temam, Gregory Watts, LRI, Paris South University (previously in Versailles).
Jaume Abella, Carles Ciuraneta, Josep M. Codina, Antonio Gonzalez, Josep Llosa, Xavier Vera, UPC.
Mark Bull and Michael O'Boyle, University of Edinburgh.
Philippe Guillen, ONERA.

Contents

In this document, we present the experience of several partners in the process of analyzing and optimizing an application. This text describes shortly the effort of each partner performed for the MHAOTEU demo workshop held in Barcelona in September 1999. Each partner looked for an application effectively used by academic or industrial end−users (as defined in the MHAOTEU workprogramme), and attempted to optimize that application. This work should not be considered as a description of a *Process for Optimizing an Application* which is a planned deliverable for the third year. It should be viewed as preliminary investigations on that topic.

The text is split in 5 sections, each corresponding to an application. For each application, we provide a short description of the application, performance analysis of the original program and the different steps of the analysis/optimization process.

I. UVSQ Application

1. General presentation of the application

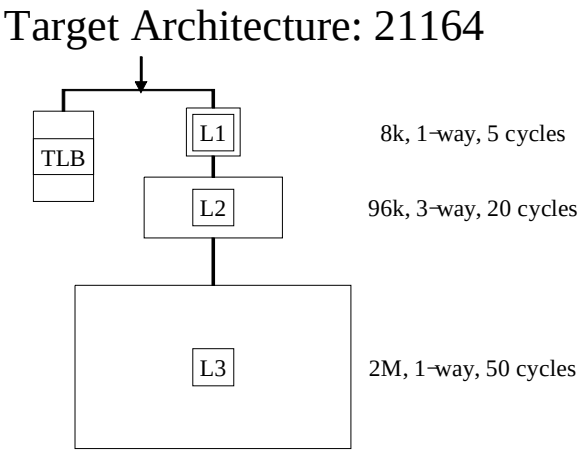
The application provided by Matra−BAe defense division is an electromagnetism code. For confidentiality reasons, only selected routines have been provided that fulfill the following criteria: the routines correspond to the largest share of the execution time and the program spends a significant interval time within the set of provided routines. This latter point is important as data reuse among routines can be critical. If we focus on a routine that corresponds to a large share of the execution time but that is called a large number of times, it is possible the bottleneck is not the routine itself but the transfer of data between this routine and other ones in the program.

However, the present case proved fairly simple as more than 90% of the execution time is spent in one LU decomposition routine which core loops are shown below. Therefore it is a well−known and concise example that is suitable for introducing the analysis/optimisation problems of applications. Several programs discussed in latter sections are more complex. The code was provided with a matrix dimension of N=550 but Matra indicated problem sizes vary between 100 and 4000.

```
DO 2001 i1 = 1,NARETES−1
        PIVINV = 1 / ZMAT(i1,i1)
        DO 2003 i2 = i1+1,NARETES
           TEMP        = ZMAT(i2,i1) * PIVINV
           ZMAT(i2,i1) = TEMP
           DO 2004 i3 = i1+1,NARETES
              ZMAT(i2,i3)= ZMAT(I2,i3)−TEMP*ZMAT(i1,i3)
2004       ENDDO
2003    ENDDO
2001 ENDDO
```

We have performed most of the optimizations with N=550 but we have also evaluated performance improvements for N=2000, and they proved similar. The target architecture for our analysis is an Alpha 21164 500MHz workstation with 512Mo memory. Matra indicated they use an Origin 2000 to run most of their codes but they could not allow external access to their server. However the code optimized for the Alpha could be briefly tested on an Origin 2000 for the sake of comparison, but one must keep in mind that transformations were targeted at the Alpha 21164 which memory hierarchy differs significantly from that of the Origin. It is possible better performance improvements can be achieved on the Origin with targeted optimizations.

Target architecture. We briefly recall the main characteristics of the target architecture. The Alpha processor has a 3−level memory hierarchy with an 8k−byte 1−way L0 cache (closest to the process) with a penalty of 5 cycles to the L1; the L1 is a 96−kbyte 3−way cache (shared by data and instructions) and it is on−chip with a penalty of 20 cycles to the L2. The L2 is a 2−Mbyte 1−way cache off−chip (shared) with approximately 50−cycle penalty to the memory. The L2 cache size and penalty can vary depending on workstations configurations. Finally, the TLB is a 64−entry fully−



associative cache for page translations (4−kbyte pages).

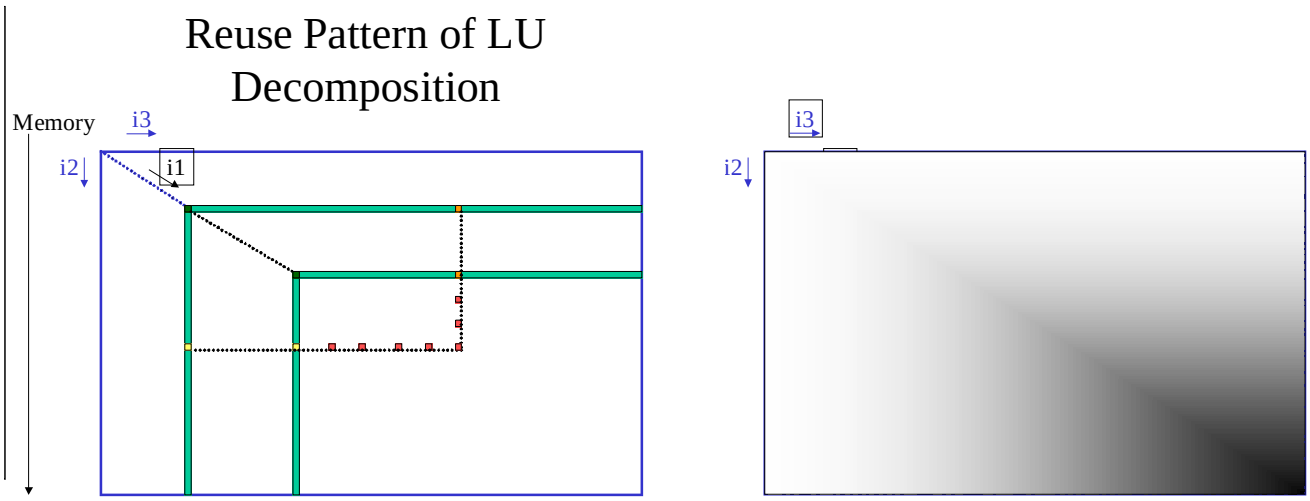
2. Performance analysis of the original application

The execution time of the original application for N=550 on the Alpha is 24,01 seconds. Using the *Profiler* tool we have measured misses on the different cache levels and the TLB, as well as the nature of the misses. The breakdown of misses among cache levels is shown below.

Cache level	Num. of accesses	Misses (ratio)	Load misses	Write misses	Num. of conflicts	Num. of capacity
0	168,949,274	110,127,754 - **65.18%**	109,481,884 - **99.41%**	645,870 - **00.58%**	306,177 - **00.27%**	109,821,577 - **99.72%**
1	165,547,234	24,910,904 - **15.04%**	24,685,024 - **99.09%**	225,880 - **00.90%**	8,490,708 - **34.08%**	16,420,196 - **65.91%**
2	44,302,066	8,968,771 - **20.24%**	8,816,457 - **98.30%**	152,314 - **01.69%**	476,863 - **05.31%**	8,491,908 - **94.68%**
TLB	168,949,274	60,566,603 - **35.84%**	59,895,811 - **98.89%**	670,792 - **01.10%**	0 - **00.0%**	0 - **00.0%**

Obviously, the L0 cache and the TLB experience significant miss ratios and most misses are capacity misses. Besides, even spatial loality is not properly exploited as the L0 miss ratio is greater than 0,25 which corresponds to the case where only one miss per cache line occurs and all words of the line are used. We can therefore assume that the L0 cache, and most likely other caches, are heavily flushed. While the nature of cache misses (capacity) already suggests optimizations (blocking), it is important to get a better understanding of the reuse pattern of the routine.

For each value of *i1*, each matrix element in the lower left rectangle defined by row *i1* and column *i1* is used once. Besides, each matrix element in row *i1* and column *i1* are used *i1* times. Therefore, the lower left matrix elements are more heavily used than the upper right elements, see figure below. On the other hand, as *i1* increases, the reuse distance between two uses of a matrix element in the lower left rectangle tends to

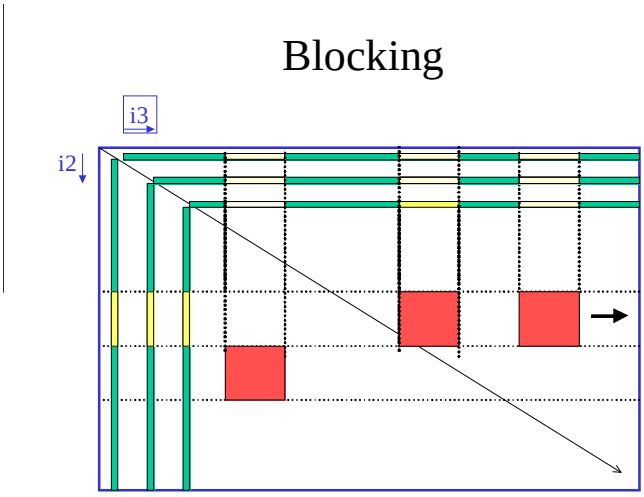


decrease and therefore the reuse is more likely to be naturally exploited by the cache without requiring any program transformation. Conversely, the reuse distance for matrix elements located in the upper right part of the matrix tends to be high though the number of reuses is smaller. The figure above provides an intuitive representation of reuse distribution where darker means more reuse.

As many matrix elements are reused several times, there is a significant amount of potential temporal locality that we can attempt to exploit using standard blocking techniques. We present the first step of the optimization in the next section.

3. Step 1: Blocking to remove capacity misses.

The effect of blocking is to decrease the reuse distance for the reuses of a block of matrix elements. All *i1* computations are performed for a block, then the program moves to another block and so on. The corresponding program is shown as well as a graphical representation of blocking. While the above blocking transformation can be performed with the




```
      do 2000 ii2 = 1, naretes, B
        do 2000 ii3 = 1, naretes, B
          do 2000 i1 = 1, NARETES-1
             pivinv = 1 / zmat(i1,i1)
             do 2000 i2 = max(i1 + 1,ii2), min(naretes, ii2+B-1)
               temp = zmat(i2,i1) * pivinv
               zmat(i2,i1) = temp
               do 2000 i3 = max(i1 + 1,ii3), min(naretes, ii3+B-1)
                 zmat(i2,i3) = zmat(i2,i3) - temp * zmat(i1,i3)
2000         continue
```

Optimization Tool, the LU decomposition presents a special case that requires the insertion of *guards*; for the moment, this transformation is perfomed manually. As shown above, the blocking transformation is not legal as data dependences are crossed. In the algorithm, each *i1* column is divided by a coefficient only once. In the blocked algorithm this division will take place several times because loops have been interchanged. To prevent multiple column updates, the following guard must be inserted:

```
      do 2000 ii2 = 1, naretes, B
        do 2000 ii3 = 1, naretes, B
          do 2000 i1 = 1, NARETES-1
             pivinv = 1 / zmat(i1,i1)
             do 2000 i2 = max(i1 + 1,ii2), min(naretes, ii2+B-1)
               if (((ii3) .LE. (i1+1)) .AND. ((i1+1) .LE. (ii3+B-1))) then
               temp = zmat(i2,i1) * pivinv
               else
               temp = zmat(i2,i1)
               endif
               zmat(i2,i1) = temp
               do 2000 i3 = max(i1 + 1,ii3), min(naretes, ii3+B-1)
                 zmat(i2,i3) = zmat(i2,i3) - temp * zmat(i1,i3)
2000         continue
```

The breakdown of misses is now the following:

Cache level	Num. of accesses	Misses (ratio)	Load misses	Write misses	Num. of conflicts	Num. of capacity
0	172,569,050	37,340,403 - **21.63%**	36,694,533 - **98.27%**	645,870 - **01.72%**	9,181,709 - **24.58%**	28,158,694 - **75.41%**
1	94,536,315	1,763,595 - **01.86%**	1,532,316 - **86.88%**	231,279 - **13.11%**	265,545 - **15.05%**	1,498,050 - **84.94%**
2	2,645,173	715,729 - **27.05%**	563,280 - **78.70%**	152,449 - **21.29%**	33,095 - **04.62%**	682,634 - **95.37%**
TLB	172,569,050	1,668,200 - **00.96%**	1,063,197 - **63.73%**	605,003 - **36.26%**	0 - **00.0%**	0 - **00.0%**

L0 misses are down to 21,63% and spatial locality is now exploited. The execution time is now equal to 2,67s, i.e., a speedup of 9,00. Note that the number of conflict misses has increased, both because blocking can have introduced some additional conflicts and because the removed capacity misses could have been hiding conflict misses.

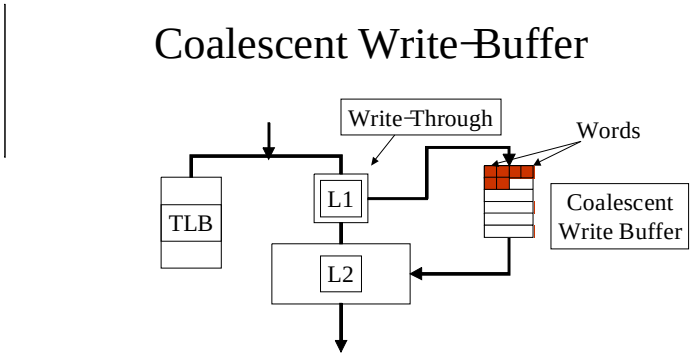
Blocking has globally reduced capacity misses because it has the effect of reducing reuse distances. But it is actually a tradeoff as some other references can see an increase of reuse distances. For instance, the reuse of the data in a *i1* row or column are increased. For that reason and because of conflict misses, blocking is very sensitive to the block size. In this case, the block size has been computed using techniques by Coleman and McKinley. Consider the above code with block sizes B=57, 58 and 59. The corresponding execution times are shown below. As can be observed, a block size variation of 2 can result in a 57% increase of the execution time.

Block Size	Execution Time (seconds)
57	2,67
58	3,28
59	4,19

4. Step 2: Exploiting a coalescent write−buffer

As we have achieved significant miss reductions, we now focus on other memory hierarchy bottlenecks, namely bus conflicts. In the above loop nests, we can notice that the innermost loop nest (*i3*) is scanning the *zmat* array row−wise, i.e., not memory−wise. In terms of misses, the impact is small as the block defined by *i2* and *i3* fits in the different cache levels. Therefore, interchanging *i2* and *i3* does not improve significantly the number of misses. On the other hand, the L0 cache of the 21164 is write−through, and more important, the write−buffer is *coalescent*. In a normal write−buffer, each time a word is sent back to memory for writing, it is stored in an entry and when the bus is available, a request is issued. The higher the number of requests, the more often the bus is used, the more likely a load request will be delayed by a write request inducing

processor stalls. To alleviate that problem, coalescent write−buffers are used. The principle is to have a tag for each write−buffer entry, and an entry corresponds to cache line, not a word. When the cache sends a data to the write−buffer, it checks whether the corresponding cache line already exists in the write−buffer, and in that case, only a word of that cache line is updated and no new entry is added to the write−buffer. Naturally, this mechanism can potentially reduce the number of write requests. It works particularly well in the presence of strong spatial locality, i.e., numerous write requests to consecutive memory addresses.



Because *i2* and *i3* do not scan *zmat* memory−wise, each write request *zmat(i2,i3)* corresponds to a new cache line and therefore the fact the write−buffer is coalescent is not exploited. Consequently, when the write−buffer is filled, additional write requests are stalled until the write−buffer can access the bus. By simply interchanging *i2* and *i3*, this property is exploited and the new execution time is 1,47s, i.e., a speedup of 1,81.

Because of the code structure, the two innermost loops need actually to be split before being interchanged and the resulting code is shown below.

```
      do 2000 ii3 = 1, naretes, B
        ub3 = min(naretes,ii3+B−1)
        do 2000 ii2 = 1, naretes, B
           ub2 = min(naretes, ii2+B−1)
           ub1 = min(naretes − 1, min(ub2, ub3))
           do 2000 i1 = 1,ub1
              if (((ii3) .LE. (i1+1)) .AND. ((i1+1) .LE. (ii3+B−1))) then
                 pivinv = 1 / zmat(i1,i1)
                 do 2001 i2 = max(i1 + 1,ii2), ub2
                    zmat(i2,i1) = zmat(i2,i1) * pivinv
2001               continue
                 do 2002 i3 = max(i1 + 1,ii3), ub3
                    temp = zmat(i1,i3)
                    do 2002 i2 = max(i1 + 1,ii2), ub2
                       zmat(i2,i3) = zmat(i2,i3) − temp * zmat(i2,i1)
2002               continue
              else
                 do 2003 i3 = max(i1 + 1,ii3), ub3
                    temp = zmat(i1,i3)
                    do 2003 i2 = max(i1 + 1,ii2), ub2
```

```
                        zmat(i2,i3) = zmat(i2,i3) - temp * zmat(i2,i1)
2003                 continue
                     endif
2000              continue
```

5. Final results and conclusions

To a certain extent, reducing the number of misses is a very long process as new paths of improvements always exist, even after drastic improvements. For instance, in the present case, taking into account the triangular nature of the algorithm could lead to further improvements. But optimizing an application is an uncertain and time−consuming task so it is important to have a rough guess of the potential expected benefits. Therefore, we lack a tool to evaluate the potential benefits of additional optimizations, a task initially planned in MHAOTEU, but which has been removed at the last review meeting.

Besides the determination of the potential optimal performance, our current approach is also limited by the accuracy of our memory hierarchy model. We take into account all components but we use a simplified view of the architecture where mostly miss issues are highlighted. Timing issues (conflicts between miss requests to the bus…) and more detailed characteristics of the memory hierarchy components are ignored. We have shown that this approximation can hide significant potential improvements, so providing a more detailed architectural model could be an important add−on to the project.

II. EDINBURGH APPLICATION

1. Application

The application from Edinburgh is GAUGE, a lattice quantum−chromodynamics application developed at EPCC in collaboration with the Physics Department at the University of Edinburgh. A parallel version of this code, with some recent enhancements, consumes a significant proportion of the CPU time on the 344 processor T3E at Edinburgh. The code consists of about 10,000 lines of Fortran77, and makes significant use of include files and cpp macros.

The code performs computations on a regular 4−D lattice, and features high dimensional arrays, predominantly linear array accesses, and deep loop nests, with some loops having short trip counts. As this code is so heavily used on some of the world's most expensive machines, it has been extensively tuned for both cache based and vector architectures.

Due to this extensive tuning, any performance gained from the cache optimised version, would be a significant gain, as all obvious performance problems should have been already eliminated.

2. Performance Analysis of the Original Application

The key early decision was to decide on the size of system to investigate. The algorithm has O(n**4 x s) time complexity and O(n**4) space complexity, where n is the size of the lattice and s is the number of steps taking to converge. As the number of steps can typically be thousands and the code run for weeks, it was important that it was scaled to an appropriate size without effecting the validity of the analysis. Fortunately we could examine a realistic value n=8 over a short number of time steps s=20, as the behaviour of each iteration after the first iteration is identical in terms of control path and memory access. Once the data size was selected, this was fed into the pre−processing scripts and a 8 x 8 x 8 x 16 Fortran code was generated.

2.1 Analysis

The principal tools used were dynamic analysis and the memory profiler. After an initial profiling we found that the total execution time was 132.24 seconds. Further investigation showed that the two most expensive routines were: make_random_su3 and uni which took 23% and 17% of the execution time respectively.

A profile of the code showed that the most expensive routine, make_random_su3 was not one where significant floating point computation was taking place. The dynamic analysis tool also showed that it was also not responsible for high numbers of cache misses. On closer inspection, it was seen that make_random_su3 contains mainly nested if conditions and it was connected that branch delay penalties are the main cost − an issue beyond the scope of the project.

After this disappointing start, we turned our attention to the next dominant routine uni where dynamic analysis tool showed that this routine was responsible for a very high numbers of cache misses. On closer inspection it was seen that it had poor stride access, within its inner loop nest, causing high numbers of L1 conflict misses.

```
      DO 23, l=0,Ncolour−1
         DO 22, k=0,Ncolour−2
            DO 21 j=0,Ncomplex−1
               index = j + ( Ncomplex*( k + (Ncolour−1)*l ) )
               DO 20 site=0,Max_body−1
                  su3_matrix(j,k,l,site,par) = rn(index,site)
20             CONTINUE
21          CONTINUE
22       CONTINUE
23    CONTINUE
```

It appears that due to a a programmer oversight, there is poor spatial locality within this loop nest. This was probably due to it being tuned mainly on Sparc based processors where its cache behaviour is insignificant relative to the overall execution time.

3. Optimisation

The main problem here is that rn is a linearised array. What we would like to do, is delinearised it and then apply loop interchange so that we have perfect stride access. Such a data transformation is only possible, if we can propagate the transformation to all other instances. In deliverable M2.D2, we showed how this may be achieved. As rn was a local array within uni , this was easily achieved.

The first transformation required to correct this was data tiling on the local array (expanding the array from 2 dimensions to 4). As this is not currently supported by the transformation engine, data tiling was performed by hand:

```
DO 23, l=0,Ncolour-1
   DO 22, k=0,Ncolour-2
      DO 21 j=0,Ncomplex-1
         index = j + ( Ncomplex*( k + (Ncolour-1)*l ) )
         DO 20 site=0,Max_body-1
            su3_matrix(j,k,l,site,par) = rn(j,k,l,site)
         END DO
      END DO
   END DO
END DO
```

This was followed by a series of loop interchanges, already successfully implemented with the loop transformation tool. This gave the following code:

```
DO site=0,Max_body-1
   DO l=0,Ncolour-1
      DO k=0,Ncolour-2
         DO j=0,Ncomplex-1
            su3_matrix(j,k,l,site,par) = rn(j,k,l,site)
         END DO
      END DO
   END DO
END DO
```

Following this, we performed a new dynamic analysis−−−this showed that the L1 capacity misses had been greatly reduced in this routine. The new execution time was 117.48secs − a 16% improvement in the code. Thus as far as this one routine is concerned we have over a 90% reduction in execution time.

Furthermore, dynamic analysis showed that the remainder of the code exhibited very good locality properties, and confirmed that further efforts to tune the code in terms of memory hierarchy exploitation were highly unlikely to be beneficial. This was further confirmed by running the memory debugger on one of the critical routines. These routines are at the computational core of the lattice gauge algorithm and are frequently hand−written in assembler code. It was unlikely that they would have poor memory behaviour.

However, this highlights a very important point, often ignored in performance tuning. This dynamic analysis and memory profiling provided evidence to suggest that future optimisation was unlikely to provide any further benefit. This type of information is invaluable to those who must balance the expense of programmer time against possible performance gains. As the porting and maintenance of the QCD code is a a significant limit on the amount of new science implemented in each new version of the code at EPCC, this was considered the most valuable result of the application study.

4. Conclusions

We found that the type of analysis provided by the MHAOTEU toolset to be very useful for analysing the GAUGE application. As this was a such highly tuned application in the first place, it was thought unlikely that any significan performance improvement was possible. However, we were able to improve the running time by 16% − a significant improvement. This update has now been incorporated into the UK QCD benchmark. Furthermore, we have confirmed that other routines are not memory−bound and that further memory optimisations is unlikely to provide any further improvement.

III. UPC APPLICATION

1. General presentation of the application

This chapter presents the detailed analysis and optimization process of a nuclear reactor

application called VMEC (Variational Moments Equilibrium Code). VMEC is used in some nuclear reactors in Spain (for instance: Tokamak in the CIEMAT (Madrid) and a Stellarator in the Max Planck Institute).

2. Initial performance analysis of the application

We have optimized this application for a Digital machine with an Alpha 21164 processor with the following cache characteristics:

	Size	bytes/line	Associativity
1st level cache:	8Kb	32	direct mapped
2nd level cache:	96Kb	64	3 way set−associative
3rd level cache:	2Mb	64	direct mapped

The tools that we have used to obtain information about the application have been: Looptiming (MHAOTEU), SPLAT(MHAOTEU), Alpha 21164 hardware counters and F77 time profiling.

LoopTiming statistics:

As figure 1 shows, the LoopTiming tool gives accurate statistics (compared to the hardware counters statistics) about what the most important routines are with a slowdown smaller than 1.

figure 1

SPLAT statitstics:

By means of the SPLAT tool we have observed that most of the misses produced in the most important routines are capacity misses. It has also been used to identify which are the memory references and loops responsible for the biggest number of misses of each routine, so we have applied some transformations oriented to reduce the volume of each iteration (for instance: blocking, loop distributing,...) getting a significant reduction in the L1 miss ratio as the figure 2a shows.

3. Analysis and optimization process

Using these tools we got some statistics which were used to analyze the application and to perform some transformations on the source code based on their information. After every set of transformations we used the tools to get new information about how much improvement we got in every one of the applied transformations.

Some different applied transformations are shown in the following lines.

LOOP DISTRIBUTING + BLOCKING

Here we show an example of how the number of L1 cache misses in the routine Tomnsp was decreased reducing the volume of each iteration by means of loop distributing and blocking.

Original code

```
      do n = 0,nmax
         do k = 1,nzeta
            do js= jmin2(m),jmax
               frcc(js,n,m) = frcc(js,n,m)
>                   + work3(js,k,01)*cosnv (k,n)
>                   + work3(js,k,02)*sinnvn(k,n)
               frss(js,n,m) = frss(js,n,m)
>                   + work3(js,k,03)*sinnv (k,n)
>                   + work3(js,k,04)*cosnvn(k,n)
               fzcs(js,n,m) = fzcs(js,n,m)
>                   + work3(js,k,05)*sinnv (k,n)
>                   + work3(js,k,06)*cosnvn(k,n)
               fzsc(js,n,m) = fzsc(js,n,m)
>                   + work3(js,k,07)*cosnv (k,n)
>                   + work3(js,k,08)*sinnvn(k,n)
            enddo
            do js= jlam(m),ns
               flcs(js,n,m) = flcs(js,n,m)
>                   + work3(js,k,09)*sinnv (k,n)
>                   + work3(js,k,10)*cosnvn(k,n)
               flsc(js,n,m) = flsc(js,n,m)
>                   + work3(js,k,11)*cosnv (k,n)
>                   + work3(js,k,12)*sinnvn(k,n)
            enddo
         enddo
      enddo
```

Transformed code

```
      do k=1, nzeta, 4
       do js=jmin2(m),jmax,8
         do n=0, nmax
          do k_s=0,min(3,nzeta-k)
            do js_s=0,min(7,jmax-js)
               frcc(js+js_s,n,m) = frcc(js+js_s,n,m)
>                   + work3(js+js_s,k+k_s,01)*cosnv (k+k_s,n)
>                   + work3(js+js_s,k+k_s,02)*sinnvn(k+k_s,n)
            enddo
          enddo
         enddo
        enddo
       enddo
       do k=1, nzeta, 4
        do js=jmin2(m),jmax,8
         do n=0, nmax
          do k_s=0,min(3,nzeta-k)
            do js_s=0,min(7,jmax-js)
               frss(js+js_s,n,m) = frss(js+js_s,n,m)
>                   + work3(js+js_s,k+k_s,03)*sinnv (k+k_s,n)
>                   + work3(js+js_s,k+k_s,04)*cosnvn(k+k_s,n)
            enddo
          enddo
         enddo
```

```
  enddo
 enddo
 do k=1, nzeta, 4
  do js=jmin2(m),jmax,8
   do n=0, nmax
    do k_s=0,min(3,nzeta-k)
     do js_s=0,min(7,jmax-js)
              fzcs(js+js_s,n,m) = fzcs(js+js_s,n,m)
>                  + work3(js+js_s,k+k_s,05)*sinnv (k+k_s,n)
>                  + work3(js+js_s,k+k_s,06)*cosnvn(k+k_s,n)
      enddo
     enddo
    enddo
  enddo
 enddo
 do k=1, nzeta, 4
  do js=jmin2(m),jmax,8
   do n=0, nmax
    do k_s=0,min(3,nzeta-k)
     do js_s=0,min(7,jmax-js)
              fzsc(js+js_s,n,m) = fzsc(js+js_s,n,m)
>                  + work3(js+js_s,k+k_s,07)*cosnv (k+k_s,n)
>                  + work3(js+js_s,k+k_s,08)*sinnvn(k+k_s,n)
      enddo
     enddo
    enddo
  enddo
 enddo
 do k=1, nzeta, 4
  do js=jlam(m),ns,8
   do n=0, nmax
    do k_s=0,min(3,nzeta-k)
     do js_s=0,min(7,ns-js)
              flcs(js+js_s,n,m) = flcs(js+js_s,n,m)
>                  + work3(js+js_s,k+k_s,09)*sinnv (k+k_s,n)
>                  + work3(js+js_s,k+k_s,10)*cosnvn(k+k_s,n)
      enddo
     enddo
    enddo
  enddo
 enddo
 do k=1, nzeta, 4
  do js=jlam(m),ns,8
   do n=0, nmax
    do k_s=0,min(3,nzeta-k)
     do js_s=0,min(7,ns-js)
              flsc(js+js_s,n,m) = flsc(js+js_s,n,m)
>                  + work3(js+js_s,k+k_s,11)*cosnv (k+k_s,n)
>                  + work3(js+js_s,k+k_s,12)*sinnvn(k+k_s,n)
      enddo
     enddo
    enddo
  enddo
 enddo
```

LOOP DISTRIBUTION + LOOP INTERCHANGE

We performed loop distribution and loop interchange in the routine Loplab in order to reduce the number of capacity misses and L1 cache accesses because it has a high locality which was not exploited.

Original code

```
        do 30 i=1,3
         do 40 j=1,3
          do 50 n=1,nloops
50         db(i,j,n)=0.
          do 60 k=1,3
           do 70 l=1,3
            do 80 n=1,nloops
             db(i,j,n)=db(i,j,n)+amat(k,i,n)*amat(l,j,n)*dbp(k,l,n)
80          continue
70         continue
60        continue
40       continue
30      continue
```

Transformed code

```
        do n=1,nloops
         do j=1,3
          do i=1,3
           db(i,j,n)=0.
          enddo
         enddo
        enddo
        do n=1,nloops
         do j=1,3
          do l=1,3
           do i=1,3
            do k=1,3
             db(i,j,n)=db(i,j,n)+amat(k,i,n)*amat(l,j,n)*dbp(k,l,n)
            enddo
           enddo
          enddo
         enddo
        enddo
```

GLOBAL INDEX REORDERING

In the following code we show a part of the Analyt routine code which had no exploited locality in the arrays conu, sinu, conv and sinv. As the code shows there was not many possible transformations to do to improve it, but it was seen that this optimization was able to reduce the number of L1 cache misses in Analyt and some other routines (for instance: surface, precal,...).

Original code

```
     common /precal2/  sinu(0:mf,nuv),conu(0:mf,nuv)
    $     ,sinv(-nf:nf,nuv),conv(-nf:nf,nuv)

     do 150 l = 0,mf+nf
     ...
     do 120 n = 0,nf
     do 120 m = 0,mf
     ...
     do  50 ip=nsta,nend
        va(ip)    =  -(zm(ip)+zp(ip))*conu(m,ip)*de
```

```
      vb(ip)    =   (zm(ip)+zp(ip))*sinu(m,ip)*de
      vc(ip)    = (tm2(ip)+tp2(ip))*bexn(ip)*sinu(m,ip)
 50 continue
   ...
   do 70 ip=nsta,nend
      va(ip)    = -(zm(ip)+zp(ip))*conv(n,ip)
      vb(ip)    =   (zm(ip)+zp(ip))*sinv(n,ip)
      vc(ip)    = (tm2(ip)+tp2(ip))*sinv(n,ip)*bexn(ip)
 70 continue
   ...
   do 90 ip=nsta,nend
      cw     = conu(m,ip)*conv(n,ip)-sinu(m,ip)*sinv(n,ip)
      sw     = sinu(m,ip)*conv(n,ip)+conu(m,ip)*sinv(n,ip)
      va(ip  ) =   - zp(ip)*cw
      vb(ip)   =     zp(ip)*sw
      vc(ip)   = tp2(ip)*sw*bexn(ip)
 90 continue
   ...
   do 91 ip=nsta,nend
      cwm    = conu(m,ip)*conv(n,ip)+sinu(m,ip)*sinv(n,ip)
      swm    = sinu(m,ip)*conv(n,ip)-conu(m,ip)*sinv(n,ip)
      va(ip)    =   -zm(ip)*cwm
      vb(ip)    =    zm(ip)*swm
      vc(ip)    =  tm2(ip)*swm*bexn(ip)
 91 continue
   ...
120 continue
   ...
150 continue
```

Transformed code

```
   common /precal2/  sinu(nuv,0:mf),conu(nuv,0:mf)
  $     ,sinv(nuv,-nf:nf),conv(nuv,-nf:nf)

   do 150 l = 0,mf+nf
   ...
   do 120 n = 0,nf
   do 120 m = 0,mf
   ...
   do  50 ip=nsta,nend
      va(ip)    =  -(zm(ip)+zp(ip))*conu(ip,m)*de
      vb(ip)    =   (zm(ip)+zp(ip))*sinu(ip,m)*de
      vc(ip)    = (tm2(ip)+tp2(ip))*bexn(ip)*sinu(ip,m)
 50 continue
   ...
   do 70 ip=nsta,nend
      va(ip)    = -(zm(ip)+zp(ip))*conv(ip,n)
      vb(ip)    =   (zm(ip)+zp(ip))*sinv(ip,n)
      vc(ip)    = (tm2(ip)+tp2(ip))*sinv(ip,n)*bexn(ip)
 70 continue
   ...
   do 90 ip=nsta,nend
      cw     = conu(ip,m)*conv(ip,n)-sinu(ip,m)*sinv(ip,n)
      sw     = sinu(ip,m)*conv(ip,n)+conu(ip,m)*sinv(ip,n)
      va(ip  ) =   - zp(ip)*cw
      vb(ip)   =     zp(ip)*sw
      vc(ip)   = tp2(ip)*sw*bexn(ip)
 90 continue
   ...
   do 91 ip=nsta,nend
      cwm    = conu(ip,m)*conv(ip,n)+sinu(ip,m)*sinv(ip,n)
      swm    = sinu(ip,m)*conv(ip,n)-conu(ip,m)*sinv(ip,n)
      va(ip)    =   -zm(ip)*cwm
      vb(ip)    =    zm(ip)*swm
      vc(ip)    =  tm2(ip)*swm*bexn(ip)
```

```
 91 continue
    ...
120 continue
    ...
150 continue
```

LOOP INTERCHANGE + SCALAR VECTORIZATION

In order to exploit some Getiota's loops locality we tried to perform loop interchange but it was necessary to perform scalar vectorization in the scalars bot and top.

Original code

```
      do js = 2,ns
         top = jv(js)
         bot = czero
         do lk = 1,nznt
            top = top - wint(js,lk)*(guu(js,lk)*lv(js,lk)
     >           +                    guv(js,lk)*lu(js,lk))
            bot = bot + wint(js,lk)*phipog(js,lk)*guu(js,lk)
         enddo
         iotas(js) = top/bot
      enddo
```

Transformed code

```
      real top(nsd), bot(nsd)

      do js = 2,ns
         top(js) = jv(js)
         bot(js) = czero
      enddo

      do lk = 1,nznt
         do js = 2,ns
            top(js) = top(js) - wint(js,lk)*(guu(js,lk)*lv(js,lk)
     >           +                    guv(js,lk)*lu(js,lk))
            bot(js) = bot(js) + wint(js,lk)*phipog(js,lk)*guu(js,lk)
            if (lk.eq.nznt) iotas(js) = top(js)/bot(js)
         enddo
      enddo
```

LOOP MERGING + LOOP INTERCHANGE

As the following code shows, the routine precondn had some no exploited locality in both nested loops (arrays ptau, wint,...) so we had to merge these loops to perform loop interchange with the outermost loop.

Original code

```
      do js = 2,ns
       do lk = 1,nznt
        ptau(lk) = r12(js,lk)**2*(bsq(js,lk)-pres(js))
```

```
>           * wint(js,lk)/gsqrt(js,lk)
   t1 = xu12(js,lk)*ohs
   t2 = cp25*(xue(js,lk)/shalf(js) + xuo(js,lk))/shalf(js)
   t3 = cp25*(xue(js-1,lk)/shalf(js) + xuo(js-1,lk))/shalf(js)
   ax(js,1) = ax(js,1) + ptau(lk)*t1*t1
   ax(js,2) = ax(js,2) + ptau(lk)*(-t1+t3)*(t1+t2)
   ax(js,3) = ax(js,3) + ptau(lk)*(t1+t2)**2
   ax(js,4) = ax(js,4) + ptau(lk)*(-t1+t3)**2
  enddo
  do lk=1,nznt
   t1 = cp5*(xs(js,lk) + cp5*xodd(js,lk)/shalf(js))
   t2 = cp5*(xs(js,lk) + cp5*xodd(js-1,lk)/shalf(js))
   bx(js,1) = bx(js,1) + ptau(lk)*t1*t2
   bx(js,2) = bx(js,2) + ptau(lk)*t1**2
   bx(js,3) = bx(js,3) + ptau(lk)*t2**2
   cx(js) = cx(js) + cp25*lu(js,lk)**2*gsqrt(js,lk)*wint(js,lk)
  enddo
 enddo
```

Transformed code

```
 do lk = 1,nznt
  do js = 2,ns
   ptau(lk) = r12(js,lk)**2*(bsq(js,lk)-pres(js))
>           * wint(js,lk)/gsqrt(js,lk)
   t1 = xu12(js,lk)*ohs
   t2 = cp25*(xue(js,lk)/shalf(js) + xuo(js,lk))/shalf(js)
   t3 = cp25*(xue(js-1,lk)/shalf(js) + xuo(js-1,lk))/shalf(js)
   ax(js,1) = ax(js,1) + ptau(lk)*t1*t1
   ax(js,2) = ax(js,2) + ptau(lk)*(-t1+t3)*(t1+t2)
   ax(js,3) = ax(js,3) + ptau(lk)*(t1+t2)**2
   ax(js,4) = ax(js,4) + ptau(lk)*(-t1+t3)**2
   t1 = cp5*(xs(js,lk) + cp5*xodd(js,lk)/shalf(js))
   t2 = cp5*(xs(js,lk) + cp5*xodd(js-1,lk)/shalf(js))
   bx(js,1) = bx(js,1) + ptau(lk)*t1*t2
   bx(js,2) = bx(js,2) + ptau(lk)*t1**2
   bx(js,3) = bx(js,3) + ptau(lk)*t2**2
   cx(js) = cx(js) + cp25*lu(js,lk)**2*gsqrt(js,lk)*wint(js,lk)
  enddo
 enddo
```

4. Final results and conclusions

VMEC has a particular characteristic that allowed us to perform the analysis faster: it is an iterative method so it can be run for 5100 iterations (normal execution taking more than 4.5 hours) or for less iterations, reducing proportionally the execution time in all the most important routines. For commodity we have performed all our analysis (except hardware counters statistics) running the application for 200 iterations (it takes 10 minutes).

Figure 2a shows misses and time statistics before and after optimizations using hardware counters and running VMEC for 5100 iterations.

figure 2a

figure 2b

As figure 2 (a and b) shows, all the optimized routines except Forces have a reduction in the number of L1 cache misses and their execution time. The routine Forces increased the L1 miss ratio but we have analyzed why it spends less time than before the optimizations and we have seen that it is due to a lower number of L2 cache misses.

Global information:

Total exec. Time before optimization:	4:38:29
Total exec. Time after optimization:	3:43:48
Total l1 cache misses ratio before optimization:	20,00%

Total l1 cache misses ratio after optimization:	12,90%

The process we have used has consisted of:

- Obtaining the statistics (SPLAT, LoopTiming, hardware counters,…)
- Analyzing statistics to find where the main bottlenecks are.
- Perform the best transformation for every kind of problem.

It is possible to get more improvement optimizing other routines, but because of the characteristics of these routines it has been impossible to reduce their execution time. It seems that the only way to get more improvement in these routines is using data prefetching, but by the way, this tool is being developed.

IV. INRIA APPLICATION

1 General presentation of the application

The code *Osiris* has been provided by Michel Kern, from project *Estime* in INRIA (Rocquencourt). It is the two dimensional version of a code developped under a contract of the Estime group with Gaz de France (GDF, the national french company for prospectionand production of natural gas).

The purpose of the application is *''Ground Penetrating Radar''*, for identifying objects under ground[1] , with many applications: petrol or gas prospection, exploration of polluted sites, checking of structures such as bridges or dams, analysis of urban underground...

The physical apparatus consists of one box emitting impulses downwards, and an antenna collecting the echos of these impulses reflected by the ground. By moving the first box along axis x, one collects corresponding echos into a trace. From this trace (data in domain (x,y,t), the problem is to compute properties of the ground, i.e. data in domain (x,y)). So this enters in the class of inverse problems, usually considered as difficult problems.

The physical system being modelled by Maxwell equations, the problem amounts to identifying the coefficients of Maxwell equations, knowing an observed trace. For this purpose, one tries to minimize the integrated sum of the difference between the observation and the estimated solution. To find the optimum, the algorithm employed is m1qn3, from the *modulopt* library developped at INRIA[2] , which is a gradient−like method (quasi−Newton with BFGS strategy).

Finally the chief components of the application are[3] :

- solver of discretized 2D Maxwell equations (discretization using the finite differences schema of Yee);
- solver of adjoint equations, for computing the gradient;
- m1qn3 routine for optimization.

For our experiments, a numerical trace is generated beforehand.

2 Initial performance of the application

We decided to measure 1 iteration of the algorithm for computing the optimum, after checking that the time for 10 iterations is exactly ten times the time for one iteration. Initially we measured that the application needed 7 seconds per iteration on the DEC−alpha processor. Using the LoopTiming tool, we could find the files where most of the execution time was spent:

- yee.f: 35 % (solve Maxwell equations)
- yee_adj.f: 8 % (compute the adjoint)
- my_treeverse.f: 10 %
- my_cal_Adj.f: 7 % (compute the adjoint, driver for yee_adj)
- bdfaccla.f: 6 % (boundary conditions)

Also we found that the following loop was consuming 23 % of the time passed in yee.f.

```
C 320
      DO j=2,nz−1
           Hz(1,j) = Hz(1,j) − dtdx * ( Ey(2,j) − Ey(1,j) )
           DO 12 i=2,nx−1
              Hz(i,j) = Hz(i,j) − dtdx * ( Ey(i+1,j) − Ey(i,j) )
              Hx(i,j) = Hx(i,j) + dtdz * ( Ey(i,j+1) − Ey(i,j) )
 12     continue
      ENDDO
C 376
      DO j=2,nz−1
         DO 11 i=2,nx−1
```

```
            Ey(i,j) = eps(i,j)*Ey(i,j)
     *           + ( ddz*(Hx(i,j)-Hx(i,j-1))
     *           -   ddx*(Hz(i,j)-Hz(i-1,j))  ) * sigma(i,j)
 11      continue
       ENDDO
```

Most of the other time consuming loops have a similar pattern.

Use of GRW (UltraSparc)

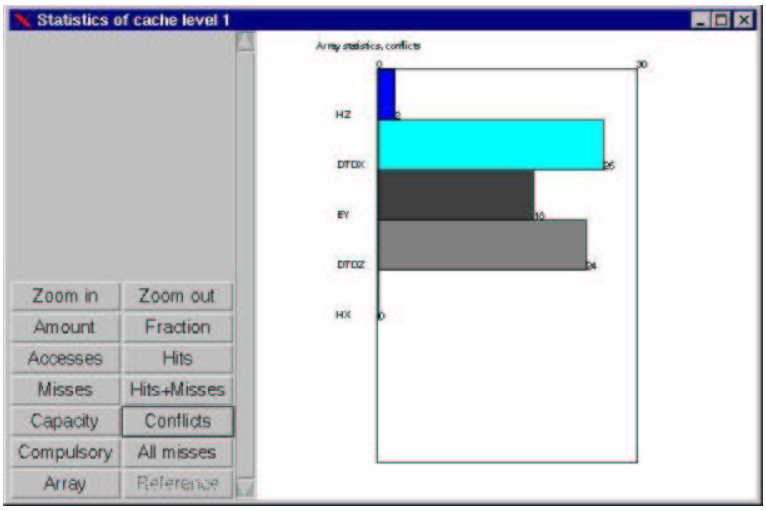
We know that if the size of the GRW is greater than the associativity, then we are sure that the cache behavior will exhibit conflict misses in this loop. In the case of the last loop (# 376), we find that the size of the window is 2 using our GRW tool. So, if the associativity is 1 (direct mapped cache), conflict misses will certainly occur.

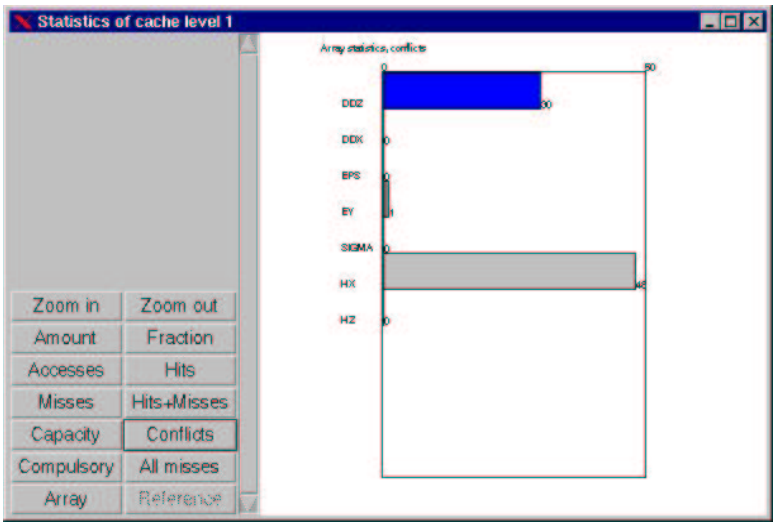
Also, after computing the size of GRW on the former loop (# 320), we find that the window has size 2 if we assume that the scalars (*dtdx* and *dtdz*) are in memory (implying conflicts when the associativity is 1), and only size 1 if we assume they are in registers. So depending on the location on these scalars, the cache behavior would vary.

Use of CVT (UltraSparc)

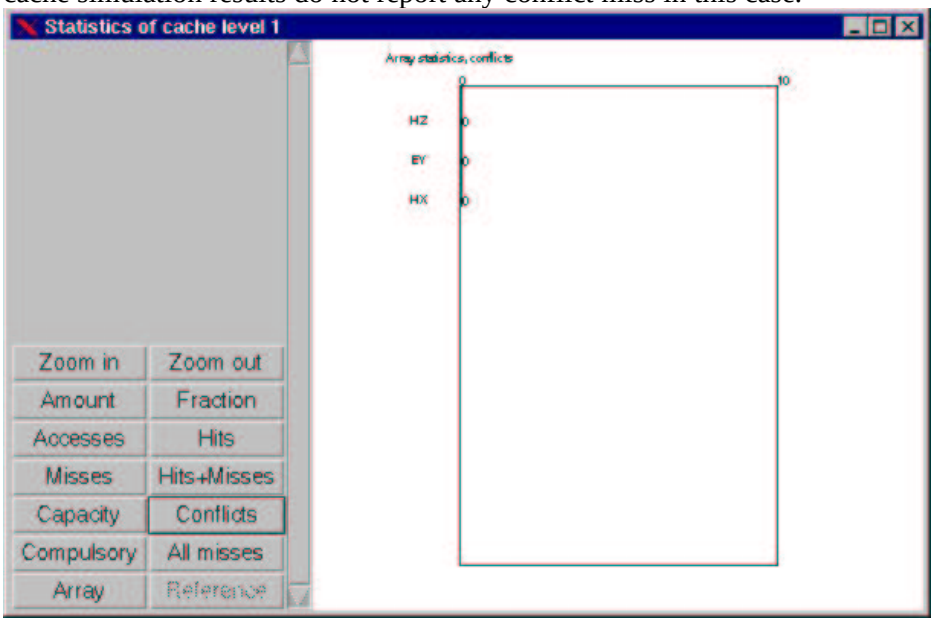
We have performed cache simulation using CVT to verify our theorical deductions with GRW analysis. The CVT tool allows us to have exact results provided by simulating a direct mapped cache (the L1 cache of the 21164 Alpha processor). The simulation confirmed our theorical results:

- when the size of the GRW is strictly greater than 1 (loop # 376, and loop # 320 when assuming scalar in memory), theoretically there should be conflict misses. The cache simulation results show us that there is a significant amount of conflict misses between the scalars *ddz* and the vector *Hx* in the latter loop and between the scalars (*dtdx*, *dtdz*) and the vector *Ey* in the former loop (the two figures below).





- in the case of the second loop, when we assume that scalars are in memory, the size of the GRW is equal to 1, so theoretically there should not be conflict misses. As a matter of fact, cache simulation results do not report any conflict miss in this case.



Using GRW guided us to detect conflicts in the two most important loops without simulation. It also allowed us to see in certain case that scalars if located in memory could be a significant source of conflicts.

3 Steps of the optimization (Dec−alpha)

We performed manually a number of optimizations, decreasing the execution time from 7 s to 4.75 s.

Step 1: loop fusion

We applied loop fusion in several places, as in the example above, getting the following piece of code[4] . Of course we had manually checked that the transformation is valid.

```
DO j=2,nz−1
           Hz(1,j) =  Hz(1,j) −
     &           dtdx * ( Ey(2,j) − Ey(1,j) )
      ENDDO
      DO j=2,nz−1
           DO 12 i=2,nx−1
           Hz(i,j) =  Hz(i,j) −
     &           dtdx * ( Ey(i+1,j) − Ey(i,j) )
           Hx(i,j) = Hx(i,j) +
     &           dtdz * ( Ey(i,j+1) − Ey(i,j) )
           Ey(i,j) = eps(i,j)*Ey(i,j)
     *           + ( ddz*(Hx(i,j)−Hx(i,j−1))
     *           −   ddx*(Hz(i,j)−Hz(i−1,j))  )*sigma(i,j)
 12      continue
      ENDDO
```

This gives a performance of 6.8 s.

Step 2: unroll and jam fusion

In yee_par, we unrolled−and−jam these loops with different factors and with/witout fusion. In the fusion version, there is no significant difference between the different factors (6.2 s for 2, 4 and 8). Without fusion, and by trying different pairs of unrolling factors (2−2, 2−1, 1−4, 1−8, we did not get better than 6.5 s).
Now we started the same process of optimization in the yee_adj procedure. Already by unroll−and−jamming the two loops twice, we got 5.8 s. But found very fast that it is better not to unroll the first one (5.5 s for 1−8 – no unrolling for the first one and unrolling 8 times for the second one). It happened also that unrolling with a prime factor gives very good results: 5.3s when unrolling the second loop or 7 or 15 or 17 times. Hence we got back to the yee_par procedure and tried to unroll the fusionned loops 7 times, resulting in 5.26 s for 1−15 in yee_adj, and then **5.19 s** when trying 5−5 in yee_adj.
Best performance: 5.19 s.

Step 3: scalar replacement

One of the loops in file bdfaccla.f had the following form:

```
DO L=1,NL
      FA1  =  C(L)
      FA2  =  C(L   +   NL)
      ...
      FA4  =  C(L   + 6*NL)
```

```
      FA8  =  C(L   + 7*NL)
      FA9  = 4*FA4
      DO I1=1, N1-1
       F0 = FA9*F(I1,   L, T1P)
     S   + (FA1 * F(I1, L, T1P)
     S   + ...
     S   +  FA4*( E(I1,1) - 2 * F(I1, NL+1, T1P)
     S                    +     F(I1, NL+1, T1M) )
       F(I1,L, T1M) = F0
       E(I1,3) = E(I1,3) + FA8*( F0 - F(I1,L, T1P) )
      ENDDO
      ENDDO
```

The key observation here is that variables `T1P` and `T1M` always take values different from each other[5] , and therefore write accesses to F are independent from reads.

After scalar replacement we got:

```
DO L=1,NL
      DO I1=1, N1-1
       F(I1,L, T1M) = 4*C(L + 6*NL ) * F(I1, L, T1P)
     S   + (C(L) * F(I1, L, T1P)
     S   + ...
     S   +  C(L + 6*NL )*( E(I1,1)
     S                   - 2* F(I1, NL+1, T1P)
     S                   +    F(I1, NL+1, T1M) )
       E(I1,3) = E(I1,3) + C(L + 7*NL) * ( F(I1,L, T1M) - F(I1,L, T1P)
)
      ENDDO
      ENDDO
```

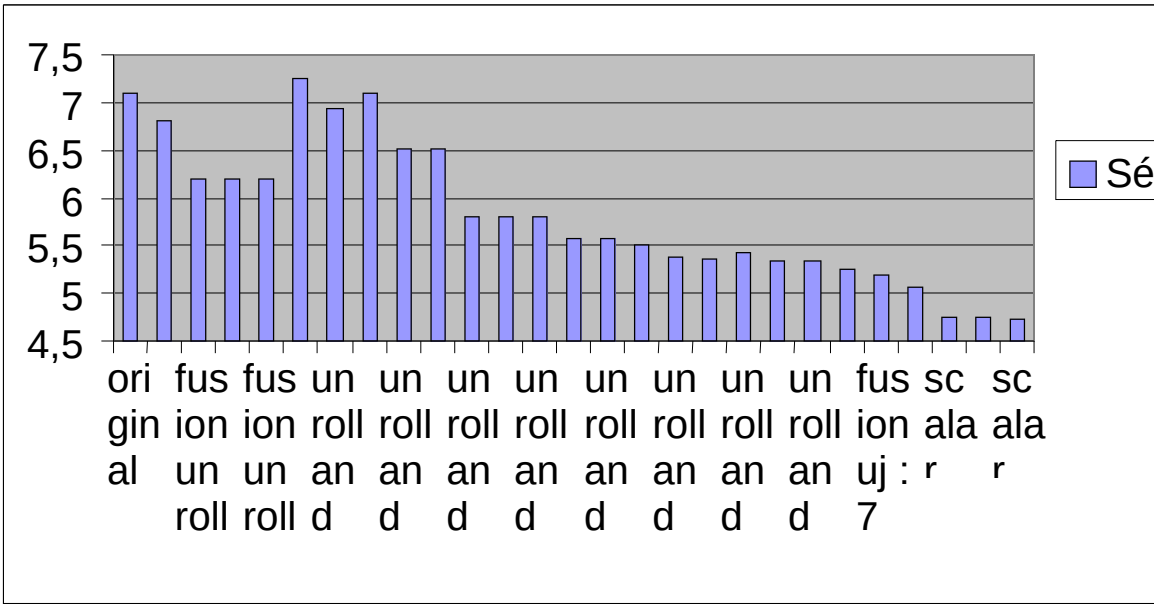
The figure for 1 iteration was now 5.06 s.

Step 4: unroll and jam

By performing unroll and jam on previous loop, the time became 4.75s.

Hence the overall improvement is from 7 s to 4.75 s, that is about 32%. This is interesting since this code is intended to run a very large number of iterations (in a very simple 2D case today one is used to run hundreds of iterations. This optimization may therefore save minutes per run. One should also be aware that the final (very long term) goal is to have this model run in 3D and in real time …

The performances obtained throughout this optimization process are summarized in the next graphic.



4 Final results and conclusion

We observed that several optimizations were able to significantly improve the performance. The process for optimizing is not linear and we had to “try and test” a quite large number of optimizations. We found that doing this manually is tedious and error−prone and suggest that the unroll−and−jam transformation is implemented very soon (this is actually a combination of loop fusion, loop unrolling and loop interchange)..

We used several tools from MHAOTEU: GRW, CVT, LoopTiming, and verified they were useful.

---

1 D.J. Gunton, D.J. Daniels and H.F. Scott. ‘‘Introduction to subsurface radar’’. IEE Proceedings, 135F(4) :278−320, 1988.

2 J.F. Bonnans, J.C. Gilbert, C. Lemarechal, C. Sagastizabal. ‘‘Optimisation numérique : aspects théoriques et pratiques’’. Springer, 1997.

3 For more details on the numerical scheme, see the INRIA report: ‘‘Imagerie du proche sous−sol par un radar géologique’’, by Guillaume Vigo and Michel Kern. INRIA Research report RR−3255, Septembre 1997. `ftp://ftp.inria.fr/INRIA/publication/RR/RR-3255.ps.gz`

4 Note that we first have to perform loop splitting in order to obtain perfectly nested loops.

5 This fact would not be so obvious to detect from an automatic tool. `T1P` and `T1M` are updated in another routine through permutation between integers 1, 2, 3, 4; being initialized at different values, they remain distinct.

V. ONERA APPLICATION

The first thing we would like to underline is the quality and quantity of work achieved by the different development partners. As end−users we fully agree with the commission experts for the completion of a unique tool as we hope to be the first beneficiaries. But as members of a research department (not in computer science but in fluid mechanics) we have to admit that the amount of engineering work exhibited by partners who are, we must recall, researchers first is really quite impressive, even with the efficient help of EPC.
During the demo workshop ONERA could test most of the MHAOTEU tools using the FLU3M software. However, even if the different parsers could be successfully used on the code or on the most important routines we could not yet achieved some first static and dynamic analysis, nor apply efficiently the transformation tool. This is due to the two following main reasons which has lead to a new set of specifications to be included in the new version of the MHAOTEU tool to be released in early December.

First, ONERA code is different from the other pieces of software by its rather important size, more than 200000 lines, and 1200 routines which allows to test the limits of the static and dynamic analysis tools. As it is rather usual for this kind of code just a few number of routines need to be optimized. Most of the tools tested, considered that the software to be optimized was in a single file, what leads to unrealistic times to instrument the code with most of the instrumentation without real interest, and unrealistic times to run the dynamic analysis.

Second, most of the CPU intensive loops in the code does not have constant bounds, which at the moment of the demo−workshop was compulsory for applying most of the transformation.

As a consequence, it was decided that the different tools will have to work with an input set made of a directory with the different routines of the program split and that they will be able to work on a limited subset of the routines. It was also decided to extend the transformation tool to non−constant bounds loops.