Optimizing Cache Access: A Tool for Source-to-Source Transformations and Real-Life Compiler Tests

Ralph Müller-Pfefferkorn, Wolfgang E. Nagel, and Bernd Trenkler

Center for High Performance Computing (ZHR)
Dresden University of Technology
D-01062 Dresden, Germany
{mueller-pfefferkorn,nagel,trenkler}@zhr.tu-dresden.de

Abstract. Loop transformations are well known to be a very useful tool for performance improvements by optimizing cache access. Nevertheless, the automatic application is a complex and challenging task especially for parallel codes. Since the end of the 1980's it has been promised by most compiler vendors that these features will be implemented - in the next release. We tested current FORTRAN90 compilers (on IBM, Intel and SGI hardware) for their capabilities in this field. This paper shows the results of our analysis. Motivated by this experience we have developed the optimization environment *Goofi* to assist programmers in applying loop transformations to their code thus gaining better performance for parallel codes even today.

1 Introduction

Usually, a developer focuses on implementing a correct program which solves the problem underneath. Applications which do not take into account the cache hierarchy of modern microprocessors, most times achieve only a small fraction of the theoretical peak performance. Tuning a program for better cache utilization has become an expensive and time consuming part of the development cycle.

The EP-CACHE project[1] [1, 2] is developing new methods and tools to improve the analysis and the optimization of programs for cache architectures. The work presented here is part of this research activity and focuses on the optimization of the source code.

One way to optimize the cache usage of applications are source-to-source transformations of loops. There are a number of transformations known that improve data locality and therefore the reuse of the data in the cache, like loop interchange, blocking, unrolling etc. (see Fig. 1).

Modern compilers claim to use loop transformations in code optimization. We have tested three FORTRAN90 compilers (Intel ifc 7.1 [3], SGI MIPSpro 7.3 [4], and IBM xlf for AIX V8.1.1 [5]) for loop transformations. On one hand, an

[1] Funded by the German Federal Ministry of Education and Research (BMBF) under contract 01IRB04.

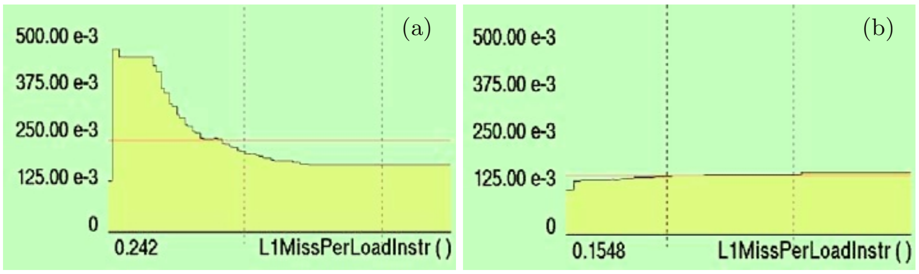


Fig. 1. Visualization of the measured L1 misses per load instruction as a function of time in Vampir [6] of a nested loop (a) before and (b) after the "unrolling" transformation.

example program was compiled with optimizations switched on. On the other hand, the example source code was optimized manually. In Sect. 2 the tests and their results are outlined in more detail.

Restructuring source code (e.g. applying loop transformations) by hand is a complicated and error-prone task. Therefore, we developed a tool to assist developers in optimizing their FORTRAN applications: loop transformations are done automatically on user request. Sect. 3 gives and overview of this tool. Finally, in Sect. 4 our future work and intentions are pointed out.

2 Current Compilers and Their Optimization Capabilities

2.1 Compiler Tests

In order to test the optimization capabilities of current compilers in terms of cache access, we compared the code generated by compilers with code optimized by hand. An example program was written in Fortran, which solves a system of linear equations based on the Gaussian algorithm. Step by step the original code of the algorithm was optimized manually and executed, both sequential and in parallel. OpenMP was used for the realization of the parallel processing.

The original source code did not utilize any optimizations. The elements of the coefficient matrix are accessed line by line. As this is contrary to the internal storage order of the matrix elements, a large number of cache-misses are produced. In the compilation process the maximum optimization level (-O5 for IBM's xlf, -O3 for the other compilers) was used with all tested compilers.

A first manual optimization was realized by implementing a loop interchange. This adapts the data access to the storage order on FORTRAN, increasing temporal locality in the cache and therefore decreasing the cache miss rate.

Further optimizations were tested, both for the sequential and parallel case. The following changes were applied: loop fission; replacing of multiple divisions with one division and multiple multiplications; loading of loop invariant matrix and vector elements into temporary variables; use of different names for loop indices.

2.2 Results

Figure 2 shows the results on SGI Origin 3800 with MIPS R12000 processors (400 MHz) using the MIPSpro 7.3 compiler. With optimizations switched on, the compiler recognizes the cache access problems and optimizes the sequential code ("sequential: -O3" in Fig. 2). Further manual transformations provide only very few improvements in the runtime of the sequential code ("sequential: loop interchange" and "sequential: additional optimizations").

The parallel programs are not optimized by the compiler automatically ("4 threads: -O3"). This can be concluded from the fact, that the manually optimized code yields much better runtimes ("4 threads: loop interchange" and "4 threads: additional optimizations"). The runtime even increases in parallel mode without manual changes compared to the sequential case, which is probably caused by the very good optimization results of the sequential code.

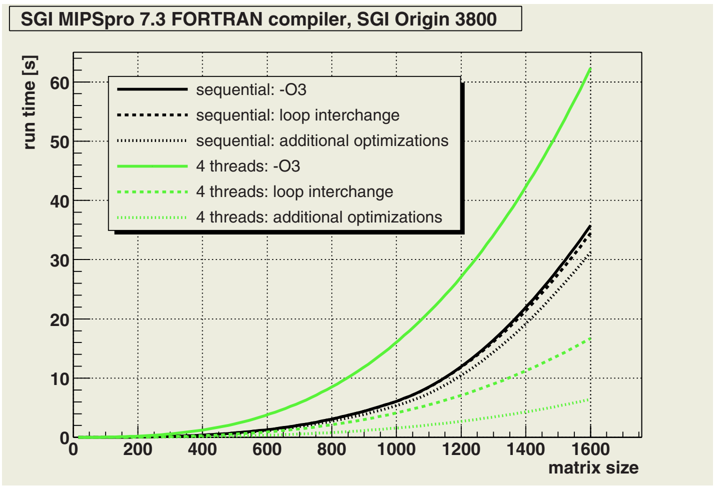


Fig. 2. Runtime as function of the matrix dimension on the SGI Origin 3800 with the MIPSpro 7.3 FORTRAN compiler.

In Figs. 3 and 4 the results of the measurements on Fujitsu Siemens Celsius 670 and IBM Regatta p690 are illustrated. The Celsius machine consists of two Intel Xeon 2.8 GHz processors running in hyperthreading mode (thus having only 4 logical CPUs in contrast to the other machines with 4 physical CPUs). As compiler Intel's ifc Version 7.1 was used. On the Regatta with its Power4 1.7GHz processors, the code was compiled with IBM's xlf FORTRAN compiler for AIX V8.1.1. For either compiler and both in sequential and parallel processing the findings are similar: the improvement in runtime due to manual optimizations is significant and larger than on the SGI Origin. The speedup is in the order of about 10. In contrast to the MIPSpro7 compiler on the Origin, the appliance of the additional optimizations does not result in any improvement. This implies, that comparable optimizations were already done by the compilers.

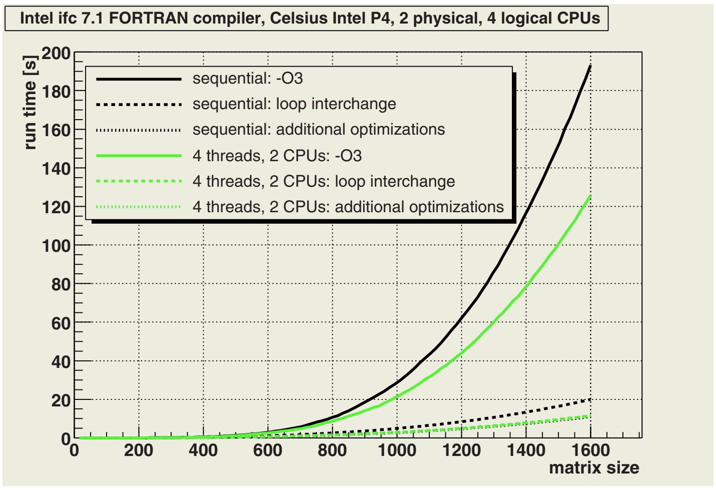


Fig. 3. Runtime as function of the matrix dimension on a Fujitsu Siemens Celsius 670 (2 CPUs in hyperthreading mode) with Intel's FORTRAN compiler 7.1; the measurement curves of the two manually optimized parallel codes and of the code with additional optimizations are on top of each other.

As an example the runtimes for a matrix size of 1600 are listed in Table 1.

Table 1. Runtime in seconds for all tested compilers and cases (matrix size: 1600) Remark: In the parallel case the Intel machine uses Hyperthreading mode. This means that the 4 threads run on 2 physical CPUs only.

	SGI MIPSpro 7.3	Intel ifc 7.1	IBM xlf V8.1.1
sequential			
-03	35.8	190.5	82.4
loop interchange	34.5	19.9	8.7
additional optimizations	31.2	11.4	7.8
parallel, 4 threads			
-03	62.4	125.7	28.5
loop interchange	16.7	11.2	2.6
additional optimizations	6.4	11.2	3.0

2.3 First Summary

Our measurements demonstrate, that the capabilities of the three tested FORTRAN compilers to optimize cache behaviour vary. Only MIPSpro7 is able to automatically optimize sequential code in such a way, that the resulting speedup is comparable with a manual optimization.

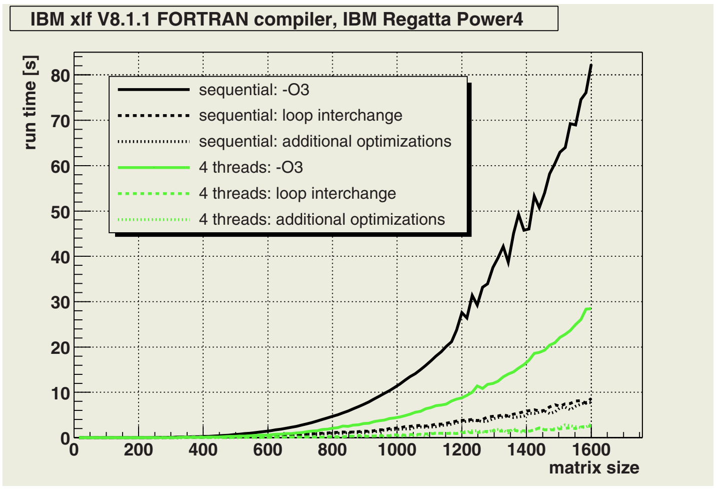


Fig. 4. Runtime as function of the matrix dimension on a IBM Regatta p690 system with IBM's xlf for AIX FORTRAN V8.1.1 compiler; the measurement curves of the two manually optimized parallel codes are on top of each other.

In the case of parallel OpenMP processing, none of the compilers can improve the original source code. Currently, the only way to improve cache access problems in FORTRAN programs seem to be the manual optimizations like loop transformations.

3 Assisting the Developer: *Goofi*

3.1 Goals

There are three drawbacks in a manual optimization of source code: it is time consuming, error-prone and can become quite complicated. Therefore, we developed the tool *Goofi* (Graphical Optimization Of Fortran Implementations) to support cache optimizations for FORTRAN applications. The goals are to provide the user with

- a graphical interface,
- an easy way to request and specify transformations,
- automatic transformations by one mouse click and
- the possibility to easily compare original and transformed source code.

3.2 Doing Loop Transformations with *Goofi*

Making loop transformations with *Goofi* is done in two steps: Insert one or a chain of transformation request directives (called "trafo directives") and secondly, ask *Goofi* to carry out the transformations. A "trafo directive" is composed of the name of the transformation requested and their parameters. It is inserted as a FORTRAN comment beginning with a special character sequence. This allows the user to save the directives with the source code for later reuse without interfering with compilation.

The following directive e.g. requests a blocking of size 8 of the loop immediately following the directive (`I` loop) with the loop nested one level below (`J` loop):

```
                                 DO I_1=1,N,8
                                 I_UPPER_1 = N
!TRA$ BLOCKING 8 1                 DO J_1=1,N,8
DO I=1,N                 ⟹        J_UPPER_1 = N
  DO J=1,N                         DO I=I_1,MIN(I_1+7,I_UPPER_1),1
                                     DO J=J_1,MIN(J_1+7,J_UPPER_1),1
```

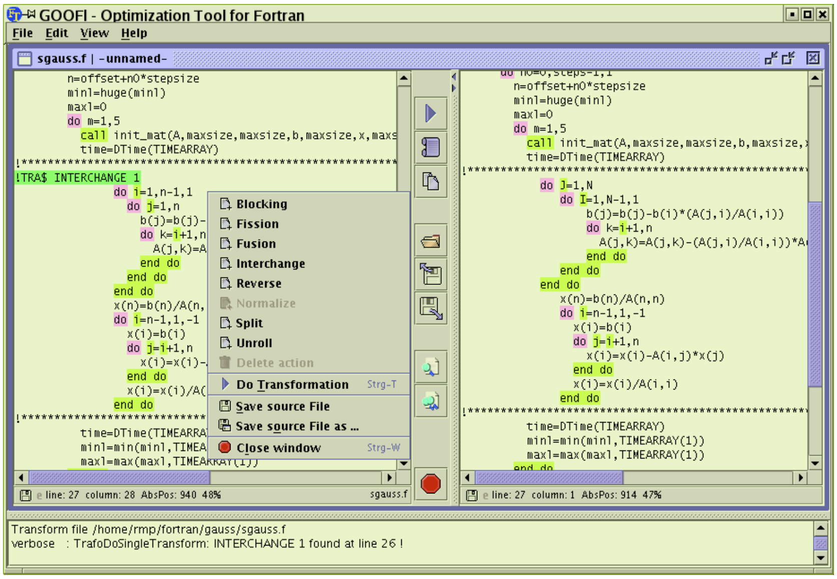
It is also possible to specify a chain of directives at once like “normalize a loop and then merge it with another one”.

Directives can be inserted either by hand or in a more comfortable way by using *Goofi*. For the latter, the user loads the FORTRAN file he/she wants to optimize into the *Goofi* environment. The original source code will appear on the left side of a window splitted into two parts (see Fig. 5). Now, the user can simply insert directives by right clicking into the source code at the places where he/she wants to apply them. After selecting a special transformation, a popup window will appear, where the transformation parameters can be filled in or adjusted.

Finally, the user requests *Goofi* to do the transformations either by a button click or by a selection from a menu. The resulting transformed source file will show up in the right window of the split screen, making direct visual comparison easily possible. It is also possible to edit the source code directly in *Goofi*, which is supported by syntax highlighting.

Currently, the following loop transformations are implemented:

– optimizing loop transformations
  - Index Set Splitting: split a loop into several loops
  - Fission/Distribution: split a loop into several loops distributing the loop body → increase temporal locality by reusing cached data that were overwritten before
  - Fusion: merge two loops → increase temporal locality by reusing data distributed in several loops before
  - Interchange: exchange two nested loops → increases temporal locality
  - Blocking: decompose loops over arrays into blocks → improves cache line usage and data locality
  - Unrolling: replicate the body of a loop → minimizes loop overhead and increases register locality
  - Outer Loop Unrolling: replicate the body of a outer loop in a loop nest → minimizes loop overhead and increases register locality



```
      n=offset+n0*stepsize
      minl=huge(minl)
      maxl=0
      do m=1,5
        call init_mat(A,maxsize,maxsize,b,maxsize,x,maxs
        time=DTime(TIMEARRAY)
!*************************************************************
!TRA$ INTERCHANGE 1
              do i=1,n-1,1
                do j=1,n
                  b(j)=b(j)-
                  do k=i+1,n
                    A(j,k)=A
                  end do
                end do
              end do
              x(n)=b(n)/A(n,
              do i=n-1,1,-1
                x(i)=b(i)
                do j=i+1,n
                  x(i)=x(i)-
                end do
                x(i)=x(i)/A(
              end do
!****************************
        time=DTime(TIMEARRA
        minl=min(minl,TIMEA
        maxl=max(maxl,TIMEARRAY(1))
```

line: 27 column: 28 AbsPos: 940 48% sgauss.f

```
      do n0=0,steps-1,1
        n=offset+n0*stepsize
        minl=huge(minl)
        maxl=0
        do m=1,5
          call init_mat(A,maxsize,maxsize,b,maxsize,
          time=DTime(TIMEARRAY)
!*****************************************************
              do J=1,N
                 do I=1,N-1,1
                     b(j)=b(j)-b(i)*(A(j,i)/A(i,i))
                     do k=i+1,n
                       A(j,k)=A(j,k)-(A(j,i)/A(i,i))*A
                     end do
                 end do
              end do
                 x(n)=b(n)/A(n,n)
                 do i=n-1,1,-1
                   x(i)=b(i)
                   do j=i+1,n
                     x(i)=x(i)-A(i,j)*x(j)
                   end do
                   x(i)=x(i)/A(i,i)
                 end do
!*****************************************************
          time=DTime(TIMEARRAY)
          minl=min(minl,TIMEARRAY(1))
          maxl=max(maxl,TIMEARRAY(1))
        end do
```

line: 27 column: 1 AbsPos: 914 47%

```
Transform file /home/rmp/fortran/gauss/sgauss.f
verbose   : TrafoDoSingleTransform: INTERCHANGE 1 found at line 26 !
```

Fig. 5. Screenshot of *Goofi* with original and transformed source files and the transformation selection window.

- preparing loop transformations
  - Normalization: bring the loop in the normalized form
  - Reversal: reverse the iteration order

The last group of transformations does not directly improve cache utilization. However, it can be used to prepare loops for further transformations. E.g. sometimes a loop can be fused with another one after the normalization step only.

When using OpenMP, *Goofi* is capable to take into account changes needed in OpenMP contructs that are affected by a transformation. As an example, the change of the variable in the "private" clause of the outer parallized loop after an interchange transformation can be mentioned.

Applying transformations can improve the runtime behaviour of applications as cache access due to a prior cache miss is costly. Though *Goofi* is not directly aimed to provide improvements especially for parallel codes the above statement is true both for sequential and parallel applications. A distributed loop still experiences pretty much the same problems, bottlenecks or advantages in the cache behaviour as a sequential one.

3.3 Basics of *Goofi*

Goofi is based on a client-server architecture (see Fig. 6). The clients graphical user interface is written in Java to make it portable to different computing

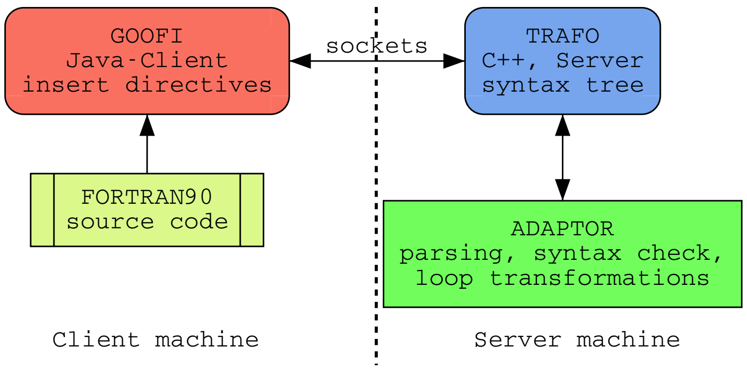


Fig. 6. Basic architectural principles of *Goofi*. The client is just a user interface, the server analyses the FORTRAN code and applies the transformations.

platforms. This allows to bring the tool to the users source code on the platform the code runs on. *Goofi* is dynamically configurable with XML-files. This includes the GUI (like menu structures) and the language preferences. Currently, the english and german language preferences are implemented. There is a strict separation between the server and the client regarding their knowledge on FORTRAN. The client is only the GUI and editor sending the FORTRAN source code to the server for analysis and transformation. Server and client communicate via sockets and use their own protocol. The server is implemented in C++ and C using the front end of the ADAPTOR [7] compilation system to analyse the FORTRAN code. After scanning and parsing the source code, definition and syntax check, a syntax tree is constructed (see Fig. 7). The “trafo directives” requested by the user are inserted into this tree. The transformations are done on the tree first. This includes restructuring the loops and statements (e.g. in an interchange transformation) or inserting new syntax elements (e.g. new lines with upcounted indices in an unrolling transformation). Finally, the transformed source code is generated by unparsing the tree and is sent back to the client.

Additional care has to be taken if there are dependencies in the code. For the syntax analysis, all elements (variables, functions etc.) have to be known. In FORTRAN, there are two possibilities to introduce such dependencies: include files and the use of modules. Dependencies due to include-files are resolved by simply inserting the include-files during parsing. Thus, in the client the user has to specify the directories where *Goofi* can look for them. Then, they have to be sent to the server.

For the usage of modules their syntax trees have to be created before opening files depending on such a module. These syntax trees are saved in files on the server side. If a “USE” statement appears the server can load the syntax tree from file. On the client side, the user only has to specify the directories containing the source code of the modules. By one further click the modules are transfered to the server and their syntax trees are created.

```
Elem              =  ACF_DO
   Line              = 10
   DO_ID             =  LOOP_VAR
     LOOP_VARNAME      =  VAR_OBJ
       Pos               = 10
       Ident             = I
   DO_RANGE          =  SLICE_EXP
     FIRST             =  CONST_EXP
       C                 =  INT_CONSTANT
         value             = 1
         kind              = 4
     STOP              =  VAR_EXP
       V                 =  USED_VAR
         VARNAME           =  VAR_OBJ
           Pos               = 10
           Ident             = N
     INC               =  ...
   DO_BODY           =  ...
   TRANSFO           =  TINTERCHANGE_STMT
       depth             = 1
   ...
```

Fig. 7. Example of the syntax tree of a DO-loop. The TRANSFO element marks the transformation to be performed on this loop.

4 Future Work

Our next steps are to evaluate and validate the performance improvements possible with *Goofi* on large applications. Two representative parallel simulation applications will be used: the local model of the DWD (Deutscher Wetterdienst, German National Meteorological Service) and the geophysical package GeoFEM of RIST, Tokio.

Having a tool like *Goofi* to assist a developer in applying optimizations is just one part in an optimization process, actually it is even the second step. The first major part is the identification and the understanding of bottlenecks in a program due to cache access problems. General information about cache misses are not very useful, as they give the user only the information that something goes wrong, but not where and why.

The EP-CACHE project [2] is also intended to overcome this problem. We are working on the exploration of hardware monitoring and monitor control techniques (TU München, [8–10]) that can help the user to gain more precise and detailed information about the cache behaviour of a program. Combined with the useful performance visualization VAMPIR [6,11,12] and further optimization tools (SCAI Fraunhofer Gesellschaft, Sankt Augustin) the user will be enabled to easily speedup his or her application.

Acknowledgement

First of all, we want to thank Thomas Brandes (SCAI Fraunhofer Gesellschaft, St. Augustin) for providing the compiler tool ADAPTOR and for all his help. We thank the students of the Java practical 2002 at the TU Dresden for their initial work. By providing access to their IBM p690 (JUMP) the John-von-Neuman Institute at the Research Center Juelich made a part of the measurements possible.

References

1. EP-CACHE: Tools for Efficient Parallel Programming of Cache Architectures. WWW Documentation (2002) http://www.scai.fhg.de/292.0.html?&L=1.
2. Brandes, T., et al.: Werkzeuge für die effiziente parallele Programmierung von Cache-Architekturen. In: 19.PARS-Workshop, Basel (2003)
3. Intel Corporation: Intel®Fortran Compiler for Linux Systems. (2003)
4. Silicon Graphics Inc.: MIPSpro Fortran 90. (2003)
5. IBM: XL Fortran for AIX V8.1.1. (2003)
6. Brunst, H., Nagel, W.E., Hoppe, H.C.: Group Based Performance Analysis for Multithreaded SMP Cluster Applications. In: Proceedings of Euro-Par2001. Volume 2150 of Lecture Notes in Computer Science., Manchester, UK, Springer-Verlag Berlin Heidelberg New York (2001) 148ff
7. Brandes, T.: ADAPTOR - High Performance Fortran Compilation System. Institute for Algorithms and Scientific Computing (SCAI FhG), Sankt Augustin. (2000) http://www.scai.fhg.de/index.php?id=291&L=1.
8. Schulz, M., Tao, J., Jeitner, J., Karl, W.: A Proposal for a New Hardware Cache Monitoring Architecture. In: Proceedings of the ACM/SIGPLAN workshop on Memory System Performance, Berlin, Germany (2002) 76–85
9. Tao, J., Karl, W., Schulz, M.: Using Simulation to Understand the Data Layout of Programs. In: Proceedings of the IASTED International Conference on Applied Simulation and Modeling (ASM 2001), Marbella, Spain (2001) 349–354
10. Tao, J., Brandes, T., Gerndt, M.: A Cache Simulation Environment for OpenMP. In: Proceedings of the Fifth European Workshop on OpenMP (EWOMP '03), Aachen, Germany (2003) 137–146
11. Brunst, H., Hoppe, H.C., Nagel, W.E., Winkler, M.: Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach. In: Proceedings of ICCS2001. Volume 2074 of Lecture Notes in Computer Science., San Francisco, USA, Springer-Verlag Berlin Heidelberg New York (2001) 751ff
12. Brunst, H., Nagel, W.E., Malony, A.D.: A distributed performance analysis architecture for clusters. In: IEEE International Conference on Cluster Computing, Cluster 2003, Hong Kong, China, IEEE Computer Society (2003) 73–81