Interaction with Programers

Phillipe D'Anfray, Christine Eisenbeis, Eric Garnier, Philippe Guillen, Michael O'Boyle, Olivier Temam, Sid Touati, Gregory Watts

REPORT M3.D3

INTERACTION WITH PROGRAMMERS

Philippe d'Anfray, Christine Eisenbeis, Eric Garnier, Philippe Guillen, Michael O'Boyle, Olivier Temam, Sid Ahmed Ali Touati, Gregory Watts

ADDING A TOOL

Alan Anderson
EPC

This document is issued in order to explain how additional tools may be added to the existing component parts of the Mhaoteu project and be integrated as part of a coherent software development. Refer to M1.D3 for an overview of the Mhaoteu architecture and look at the other documents on the website for more information.

The Global Software Architecture

The Mhaoteu system presents an integrated appearance to the user through a unified client server style of tool invocation presented to the user through a single user interface. It is not a monolithic program. The individual tools are wrapped and invoked in the style of components.

Much of the information required by the end user such as statistics and marked up lines of source is effectively presented in the form of HTML text.

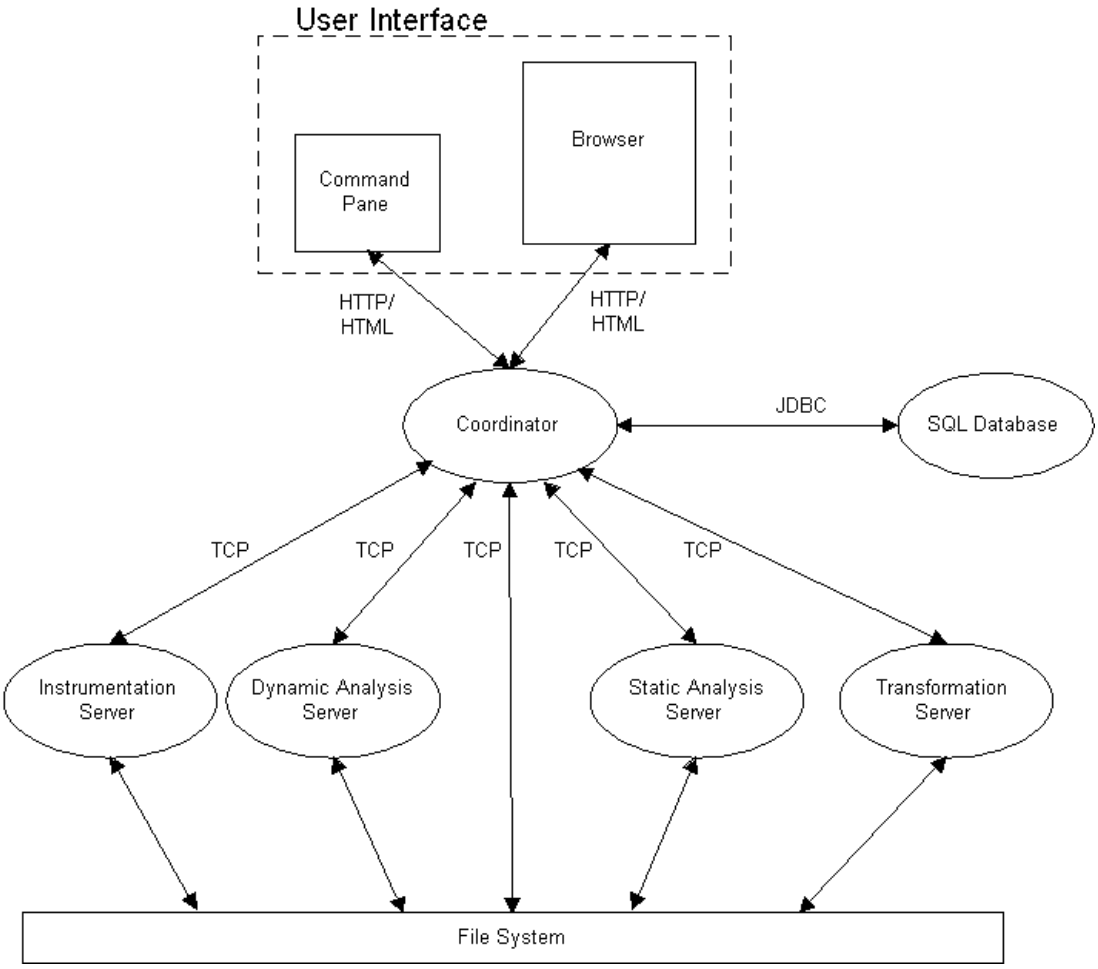
As we plan to progressively augment the tool with new components, and as user feedback may suggest changes, the software architecture is deliberately flexible.

The principles are:

(1) Keep several tool components separate but to hide the disjoint structure to the end-user.

(2) Provide interaction with the end-user through a unique interface composed of a browser and a command panel. Data is displayed in HTML format within the browser rather than graphics.

The Software architecture is shown below:


User Interface
Command Pane
Browser
HTTP/ HTML
HTTP/ HTML
Coordinator
JDBC
SQL Database
TCP
TCP
TCP
TCP
TCP
Instrumentation Server
Dynamic Analysis Server
Static Analysis Server
Transformation Server
File System

Brief notes on the design.

The Mhaoteu system currently comprises the following components:

(A) The instrumentation server - UPC
(B) The dynamic analysis server - Versailles
(C) The static analysis server - UPC
(D) The GRW static analysis server - INRIA
(E) The code transformation server - Edinburgh University

Each of these servers supports one or more tools which carry out analysis or optimisation tasks on behalf of the Mhaoteu system.

(F) The relational database. - controlled by Versailles.
(G) The Coordinator program which knits all these together and presents the user interface. - EPC

As listed above, each component (A) to (G) is clearly assigned as the responsibility of a particular partner.

There are three modes of communication implied by the system diagram

(1) There is a shared filestore such as NFS. The coordinator program manages a directory structure within which the partner tools may collect and deposit files. They create new versions of the user program and communicate with each other through these files.

(3) The coordinator program communicates with the tools over TCP/IP. At the moment this communication is via sockets routing through a small number of "servers", each controlling a number of partner tools on a single platform

(3) A graphical user interface is presented to the user. The browser pane and the command pane are assumed to be generated in HTTP/HTML. In theory this allows for remote operation of Mhaoteu across the Internet.

User Interface: Displaying HTML statistics within a browser allows us:

(1) to add new statistics and evaluate their usefulness quickly,
(2) to browse back and forth through a hierarchy of statistics,
(3) to get detailed information instead of the overview provided by graphics.

Consequently, the browser is the main interface to the end-user. Using a browser, information can be passed and displayed progressively instead of as a whole for a graphics utility. This is particularly important for large codes. Moreover, we can

progressively and easily add new links between statistics as we identify the most useful and logical connections between statistics.

For the analysis part of the tool, the browser is sufficient as we essentially need to display statistics, sort them in different manners or visualize the source code. For the optimization part, we also need to send commands to the different tool components. For that purpose, we have a command panel.

Data Base For large industrial codes, the statistics provided by the profiler or the static analysis module run in the Megabytes range. To efficiently manipulate such large amounts of statistics, we store them in a database. The database tables are built to reflect the way statistics are accessed (tables for procedures, nests, loops, statements and references). The table structure allows us to easily add new statistics for each code construct. Thanks to the database, we can also sort/filter statistics in many different ways depending on the end-user requests.

The different versions of the source code do not presently reside in the database. They are in the normal file system. The database is used to store the transformation sequences, through which particular versions of the source can be recreated from the baseline source.

Implementation: We are using the freeware database *Postgresql*[1]. All database requests are simple SQL requests and provided we have the corresponding Java driver, we could replace *Postgresql* by any other SQL database.

Coordinator. The coordinator is the program that insulates the end-user from the different tool component, provides support for the user interface and assists the user in using the Mhaoteu system. Requests for statistics or transformations are all passed through the coordinator.

Browsing Statistics: Requests for statistics result in requests to the database. The information is passed through the coordinator to the browser. Depending on the request, some transformations can be performed on the statistics on the fly. Therefore, only raw statistics need to be stored in the database though many different statistics can be presented to the end user.

Command Requests: Requests for analysis and optimization, generally issued by the command panel, are passed to the corresponding tool component, which then retrieves the corresponding program version, performs the transformation and dumps the result in the file system again. The end-user can either visualize an HTML version of the result, or request to run or simulate the resulting code for extracting new statistics.

[1] See http://www.postgresql.org/ for more information.

(1) All action requests go through the Coordinator, so that at all times it knows what is going on and can represent to the user an accurate view of the system state.

(2) For similar reasons, all database access requests go through the coordinator. To write information into the database the tools will write a SQL file in the shared filestore. The coordinator will action this file to the database. As described later, the coordinator will avoid detailed knowledge of database formats, but it will know what data is current, being updated or obsolete.

(3) The Cordinator is a multi-threaded program. Although prototypes may well operate on a single user basis, the design should be towards a facility capable of supporting multiple users simultaneously. This supports both commercialisation - through multiple user licensing, and research - through allowing more than one group access to a limited number of platforms.

Implementation: We are implementing the coordinator in Java. Thanks to Java, most of the operating system complexity is hidden (communications with a browser, network communication with different tools, with the database…) so that development time and consequently programming overhead is minimal.

Tool Components Each tool receives commands for performing transformations, extracting statistics or any other functionality. We plan to add more tool components or augment the current ones in the next two years.

Implementation: Tool components use different languages (C, C++, Java) and they have all been augmented with a simple interface to communicate with the coordinator. We have developed a unified command syntax to communicate with the tools.

Distributed Environment. As the Coordinator communicates with tool components and the browser using HTTP or other network protocols, the different tool components need not reside on the same host. A distributed software architecture is not our primary goal, but it is of considerable benefit to the ongoing work as it allows for the seamless interaction of the tools despite there being no adequate provision of common hardware and software at the partner sites. A common target of Sparc Solaris has been agreed for the construction of self-contained single machine versions of Mhaoteu, **but there is no prospect of this being the main research platform at every site.**

The tool components still share a file system where the program source code and the active version reside.

Steps involved in adding a tool to Mhaoteu

(1) Port your tool to Sparc Solaris. This is a temporary matter of practicality. While the architecture allows for multiple platforms, Solaris Sparc is the only single common platform and has thus become the most convenient place for integration work and demonstrations of the Mhaoteu system.

(2) Either add the tool to an existing Mhaoteu tool server program or take a copy of the Edinburgh socket library and set up your own server program. Consult with EU about the socket library and for a sample implementation.

(3) Integrate your tools invocation and operation to the style of the Mhaoteu command language. Your server program (2) is the bridge. Consult with EPC to extend the command language if your tool has unique requirements. Be aware that this stage can involve you in multiple operations. For instance if your tool requires to modify a source program, compile it, link it, run it – and check that it all worked, then this whole activity must be scripted and orchestrated by you in response to a command instruction to run your tool.

(4) If your tool uses or produces statistics, then you need to be aware of the current definitions of the Mhaoteu database tables. In consultation with Paris South, these can be extended to accommodate your tool. It is very much the style of the Mhaoteu project, that your tool should re-use existing statistics and itself produce statistics that may be of use to other tools, so consider this stage carefully.

(5) If your tool creates or accesses bulk data, then this is handled as a file in the shared file store. Consult with Paris South over the format of the data.

(6) Think about the information that a user must supply to your tool in order to invoke it and consult with EPC over the creation of an interactive dialog or form through which this information will be gathered.

(7) If your tool produces some graphical output, then consider if this can be produced in Java so as to fit in with the machine independent architecture style of Mhaoteu.

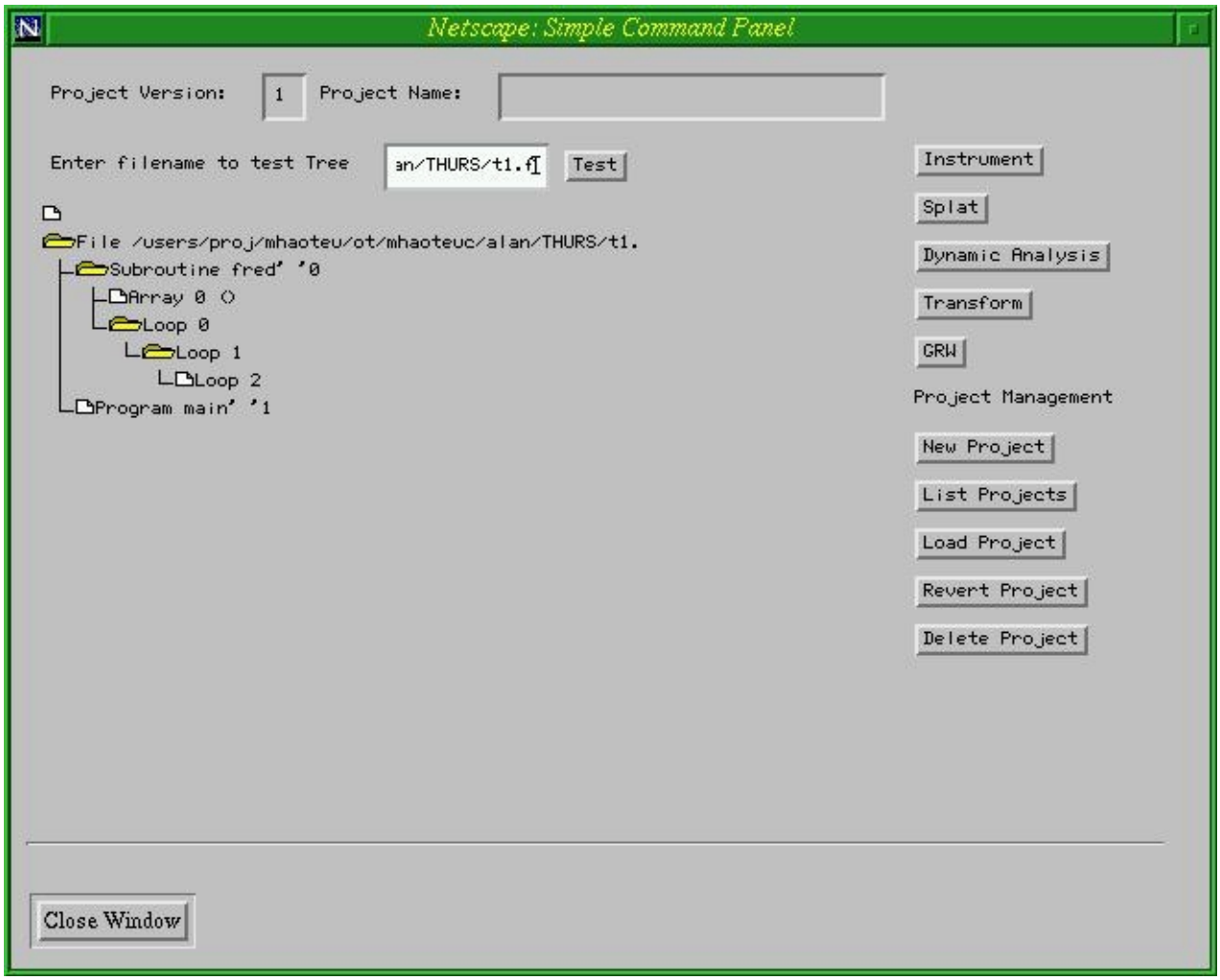
(8) If your tool requires longer to function that a user will want to wait in an interactive session, then build in the functionality of producing a results file with a pre-determined name in order to signal completion asynchronously.

(9) Reply to problems with informative error messages piped back to the user through the command processor.

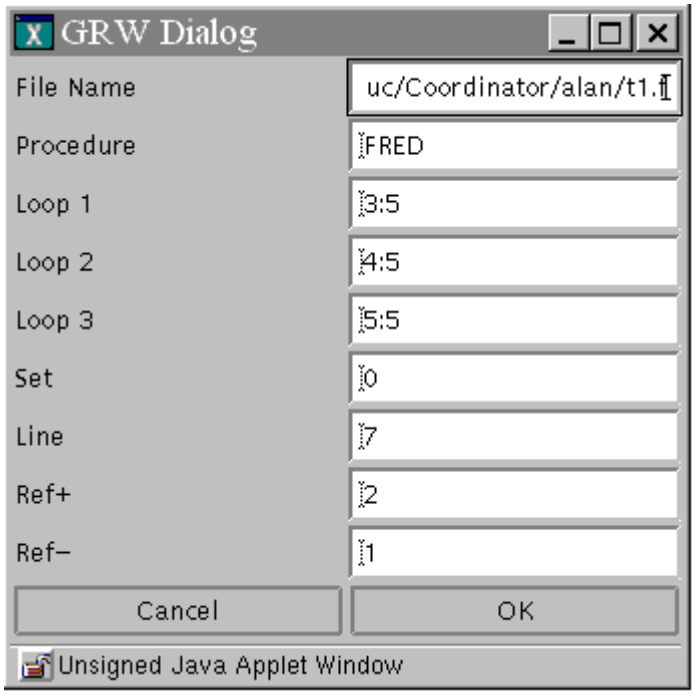
(10) If your tool wishes to provide views of information in the database via the HTML data Visualiser then you should liaise with Paris South over the construction of a database parameter file which will control this activity.

(11) Make your program thread safe in order to support multiple simultaneous users from a single platform.

The first thing that the user will see is the command panel:



This command pane will be enhanced by EPC to have a button to activate the new tool dialog.



This is an example of a Dialog. The Dialog is like a form, by which details are supplied by the user to control the tool invocation. You have to work out what information is required and EPC will construct the Java Dialog.

When working with the coordinator remember that:

(1) All action requests go through the Coordinator, so that at all times it knows what is going on and can represent to the user an accurate view of the system state.

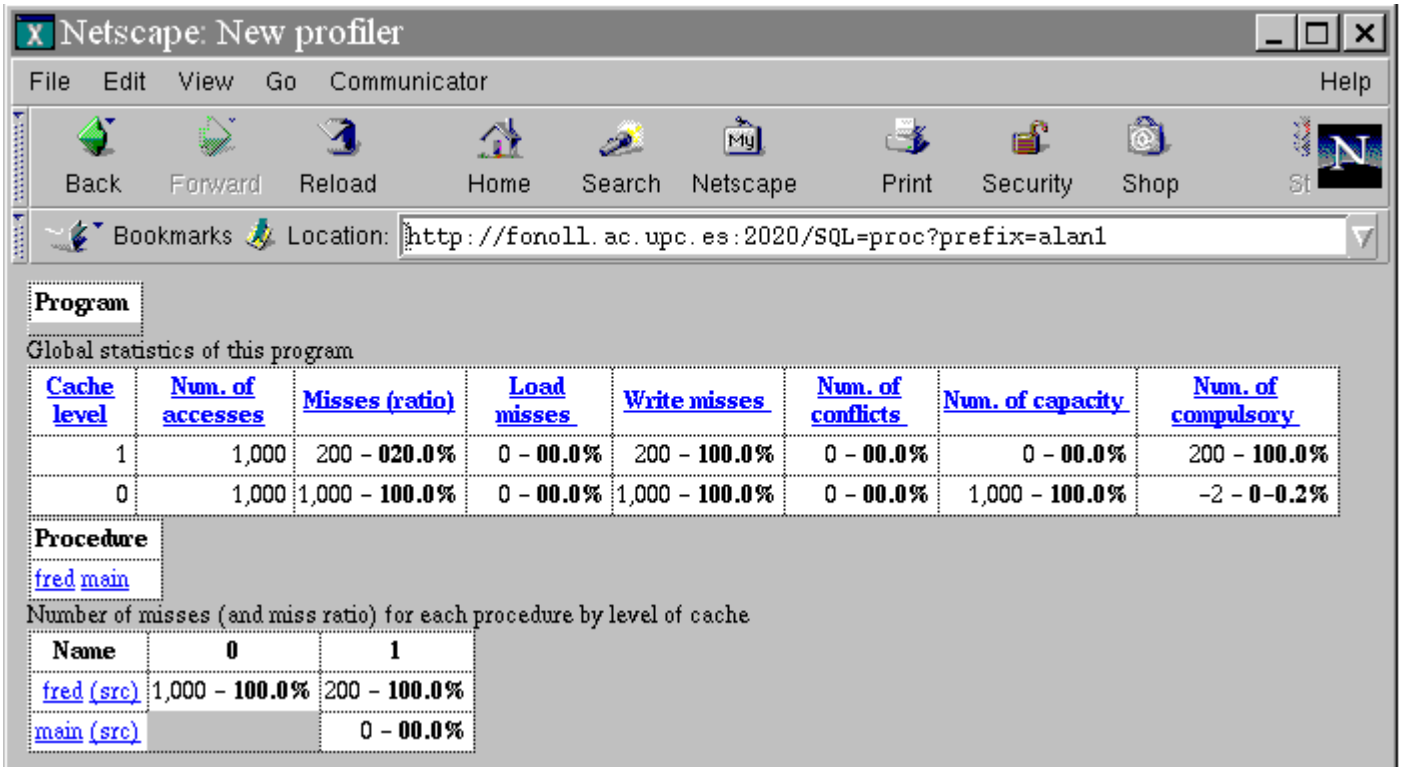
(2) For similar reasons, all database access requests go through the coordinator. To write information into the database the tools will write a SQL file in the shared filestore. The coordinator will action this file to the database

(4) The Cordinator is a multi-threaded program. Although prototypes may well operate on a single user basis, the design should be towards a facility capable of supporting multiple users simultaneously. This supports both commercialisation - through multiple user licensing, and research - through allowing more than one group access to a limited number of platforms.

(5) Each of the tools comprising Mhaoteu is activated and controlled from the Coordinator module. As far as possible, a common tool API is imposed on these diverse tools. This simplifies the design and increases reliability. The cost of adding new tools to the project is also reduced by having a generic framework.

See M1.D3 for more details.

If your tool produces data tables, the user will see an HTML view like that below. It is up to you to decide on the most effective fields to display and in what order. Paris South will assist you in arranging the display and all other database related matters.



Please feel free to contact the author at alan@epc.co.uk for more information about integrating your tool with Mhaoteu.

A PROCESS FOR DRIVING PROGRAM OPTIMIZATION

Olivier Temam, Gregory Watts
Paris South University

Mike O'Boyle
University of Edinburgh

In this deliverable, we present the guiding principles and a first draft of a process for optimizing an application. We then illustrate program optimization by hand with a SpecFp95 program.

Building a Process for Optimizing an Application

There is a very large number of studies on compiler optimization techniques. Most of these studies target very restricted code sections, i.e., loops or loop nests. To our knowledge, very few studies address the issue of defining a global strategy for optimizing programs. Traditionally, within a compiler, locality optimizations are applied either in preprocessors (program transformations, e.g., like in KAP, the preprocessor from Kuck and Associates) or in the backend (e.g., software pipelining, prefetching), and in both cases their program scope is fairly restricted. In this deliverable, we attempt to define a strategy for addressing the problem of optimizing the memory performance of a whole program.

Decision tree. Our goal is not to define a compiler algorithm though we expect that ultimately much of this work will find its way in a production compiler, once analysis techniques perform well enough. Therefore, we focus on program optimization by hand. We initially consider the whole program and we progressively narrow the program constructs and memory component size. With respect to the architecture, a bottom-up approach is used: we first consider the lower levels of the memory hierarchy (the lowest cache level and especially the TLB which performs virtual to physical translations) and we progressively rise along the memory hierarchy.
Optimizing for the L1 cache first can bring significant performance improvements, but it is then difficult to consider optimizations for lower level caches or the TLB once the optimizations for the L1 (or upper-level) cache have been performed. For instance, a blocking technique applied for the L2 cache can be complemented with another optimization applied to the block that fits in the L1 cache and that does not affect the first optimization. Conversely, after optimizing for the L1 cache, a subsequent optimization for the L2 cache is likely to affect the L1 cache optimization.

In summary, we start with the coarser program constructs and the coarser memory hierarchy components, and then progressively narrow the scope of the analysis.
To assist an end-user in the task of optimizing his/her code, we have represented the different steps of the analysis/optimization process as a graph where each node represents either an analysis step, a decision action step or an action step. The end-user applies the analysis step (sometimes a time-consuming dynamic analysis) which is usually followed by a decision step specifying which action or which further analysis must be conducted. The graph can sometimes iterate on certain program constructs (like the loop nests in a procedure, or array references within a loop body).
This decision tree is entirely based on our practical experience with optimizing programs by hand. We have identified the transformations that bring most performance improvements in most cases, and it is important to stress that this set of transformations (e.g., forward substitution, loop merging) is fairly different from the set of transformations largely discussed in the research literature (e.g., loop tiling). Besides identifying the most important individual steps, we have organized these steps according to the coarser-to-finer grain approach mentioned above.
The main steps of the process are the following:

1. We first determine whether the code *is* memory-bound. This may seem like a trivial step, but in an industrial environment, it is important that the end-user quickly focuses on the right problem. There are two reasons why a code can be memory-bound: because some of the caches/TLB behave poorly, or because the ratio memory accesses over computations is high. See *Figure 1 Decision graph (1st part).*

2. If the code is memory-bound because the fraction of memory accesses is high, then we attempt to forward substitute several arrays to eliminate the correspond memory accesses. This has the effect of trading computations for memory accesses. While this is not a natural choice for a programmer, i.e., increasing the number of computations, it can be very profitable performance-wise if functional units are often idle because of the restricted memory bandwidth. See *Figure 2 Decision graph (2nd part).*

3. We then make a combined analysis of the control flow and the data flow and if we find that many consecutive nests share many data then they are potential targets for merging. This is measured by the amount of inter-nest temporal locality, see deliverable M3.D3 (locality analysis). See *Figure 3 Decision graph (3rd part).*

4. We then focus on nests and build classes of references depending on their access patterns, i.e., the reference groups mentioned in M3.D3 (locality analysis). If there is only a single group, loop interchange is considered if applicable, otherwise tiling transformations are considered to minimize the impact of conflicting reference patterns on the TLB. See *Figure 4 Decision graph (4th part).*

5. The last step is more similar to traditional loop optimization techniques. We first focus on spatial conflicts which have the strongest impact on performance,

and only then we attempt to exploit temporal locality within loops, using tiling techniques. See *Figure 5 Decision graph (5th part).*

The graph is represented in the different figures below; analysis steps are represented in dark grey rectangles, decision steps in light grey parallelograms and action steps in white rectangles; the light grey rectangles are “comment” or “label” nodes.

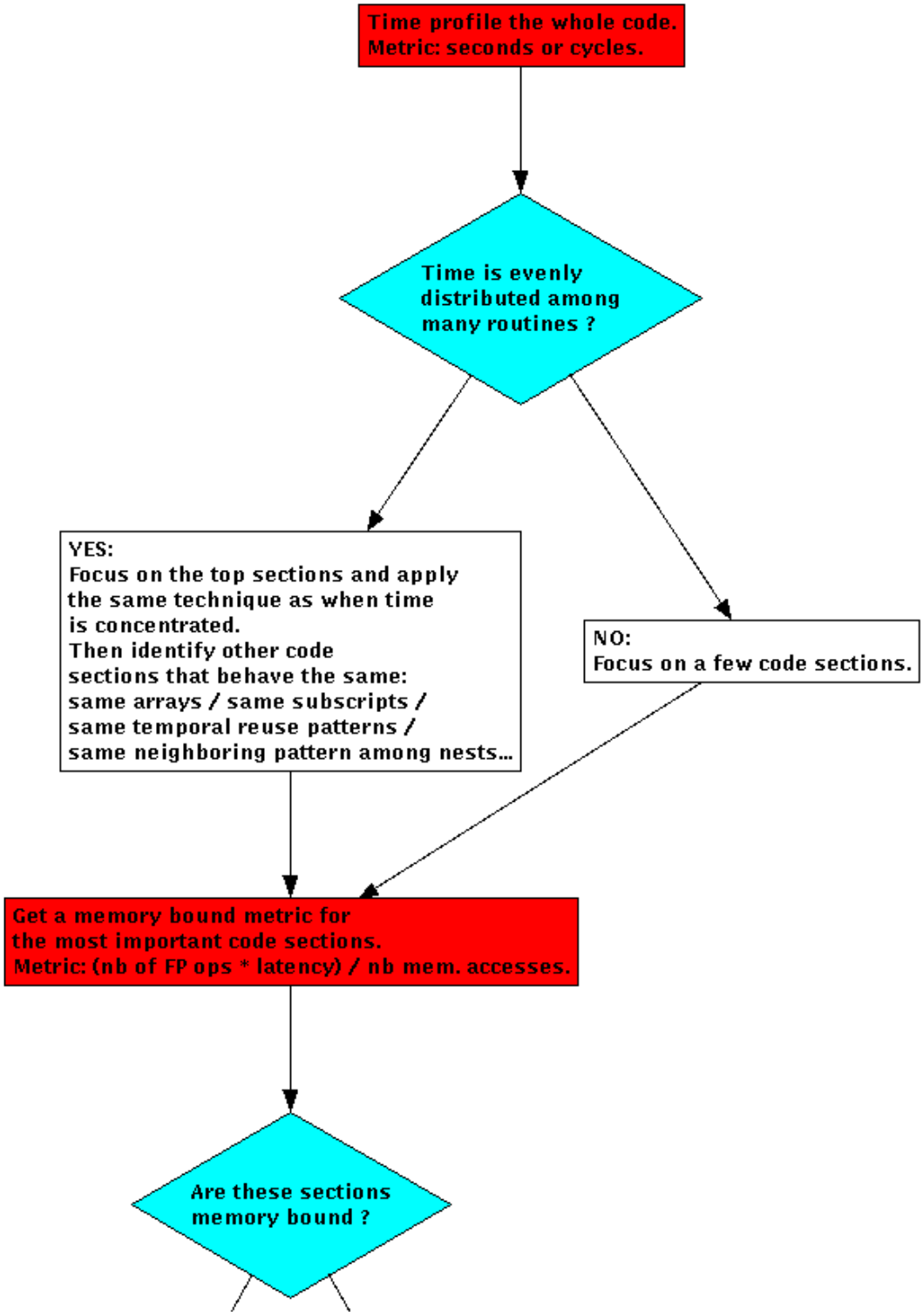


Figure 1 Decision graph (1st part)

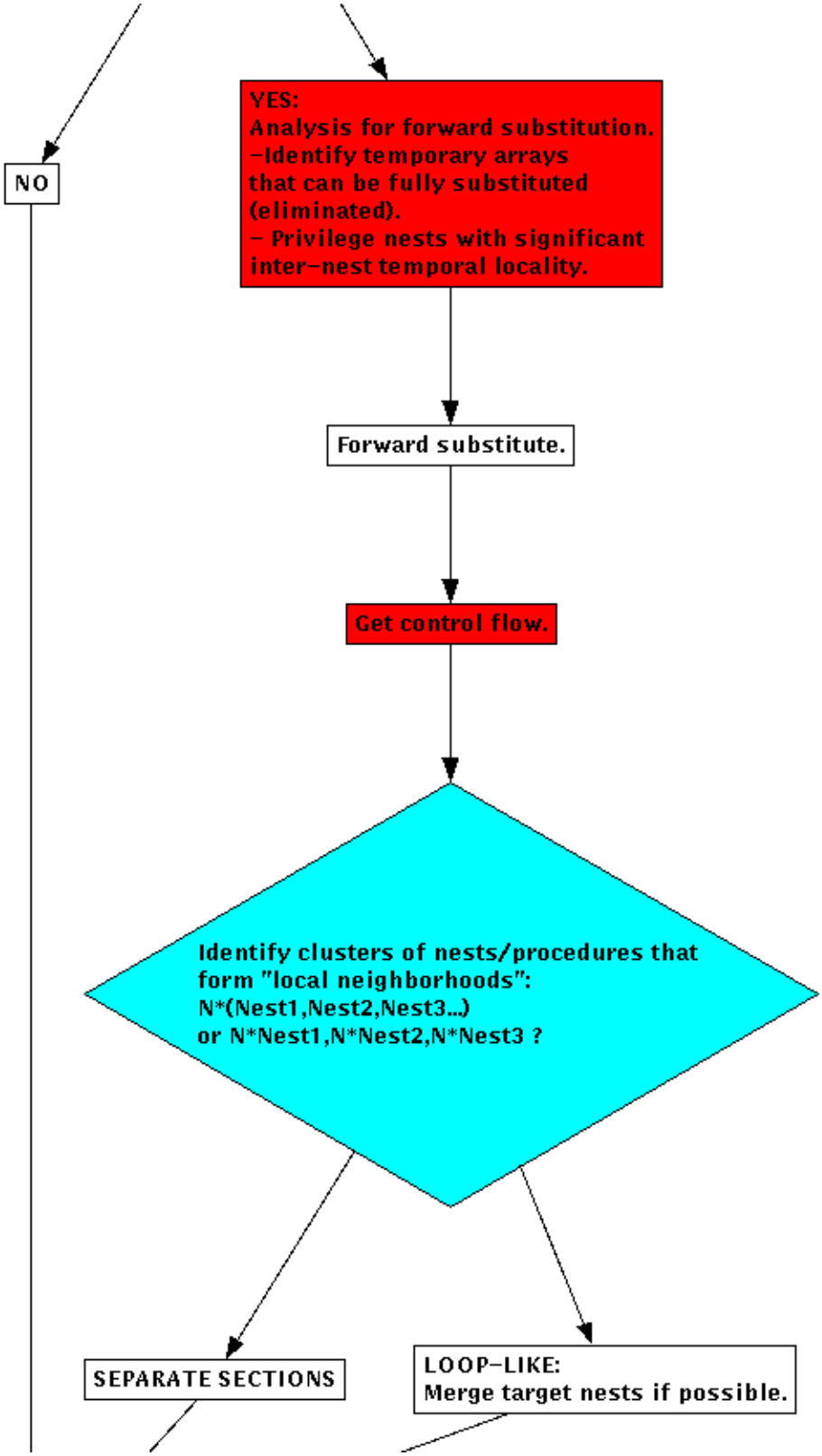


Figure 2 Decision graph (2nd part)

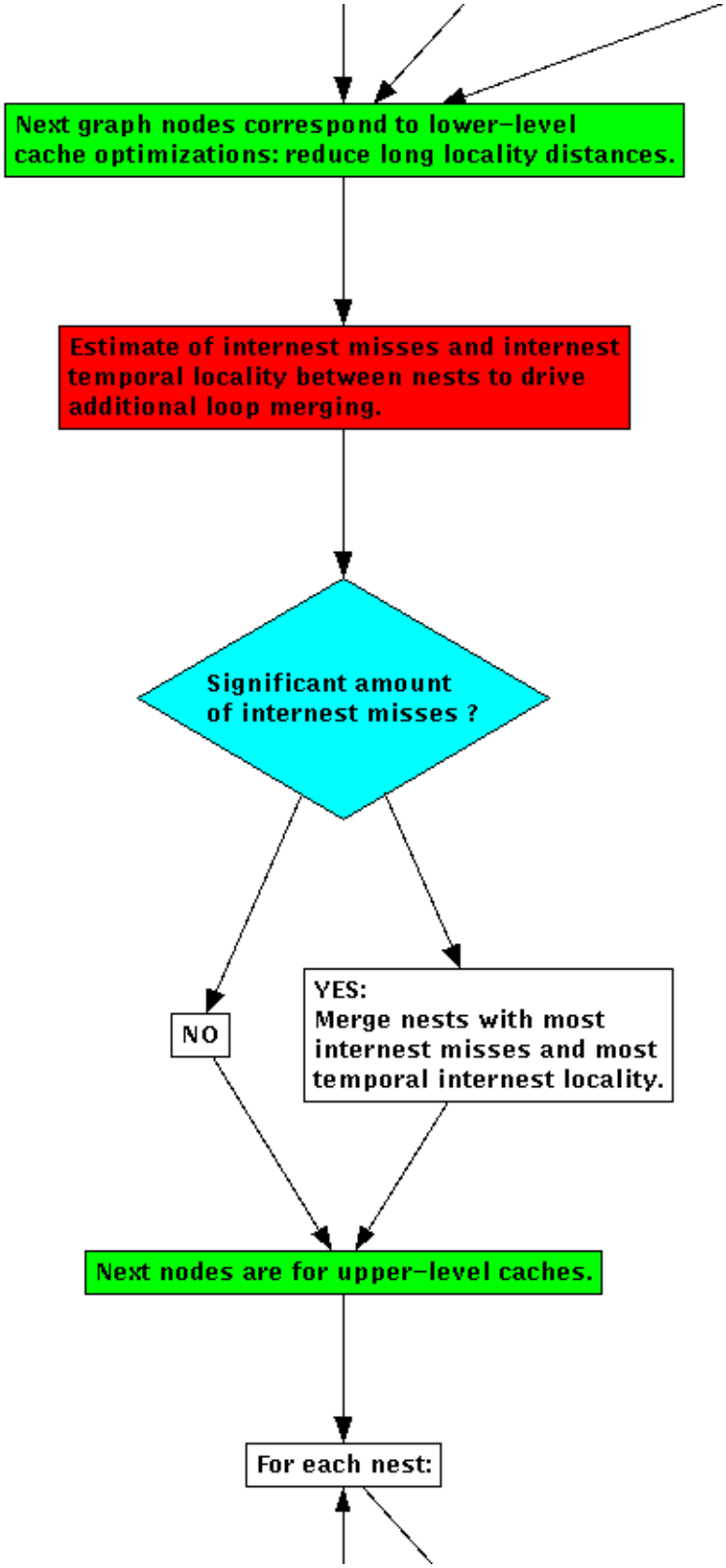


Figure 3 Decision graph (3rd part)

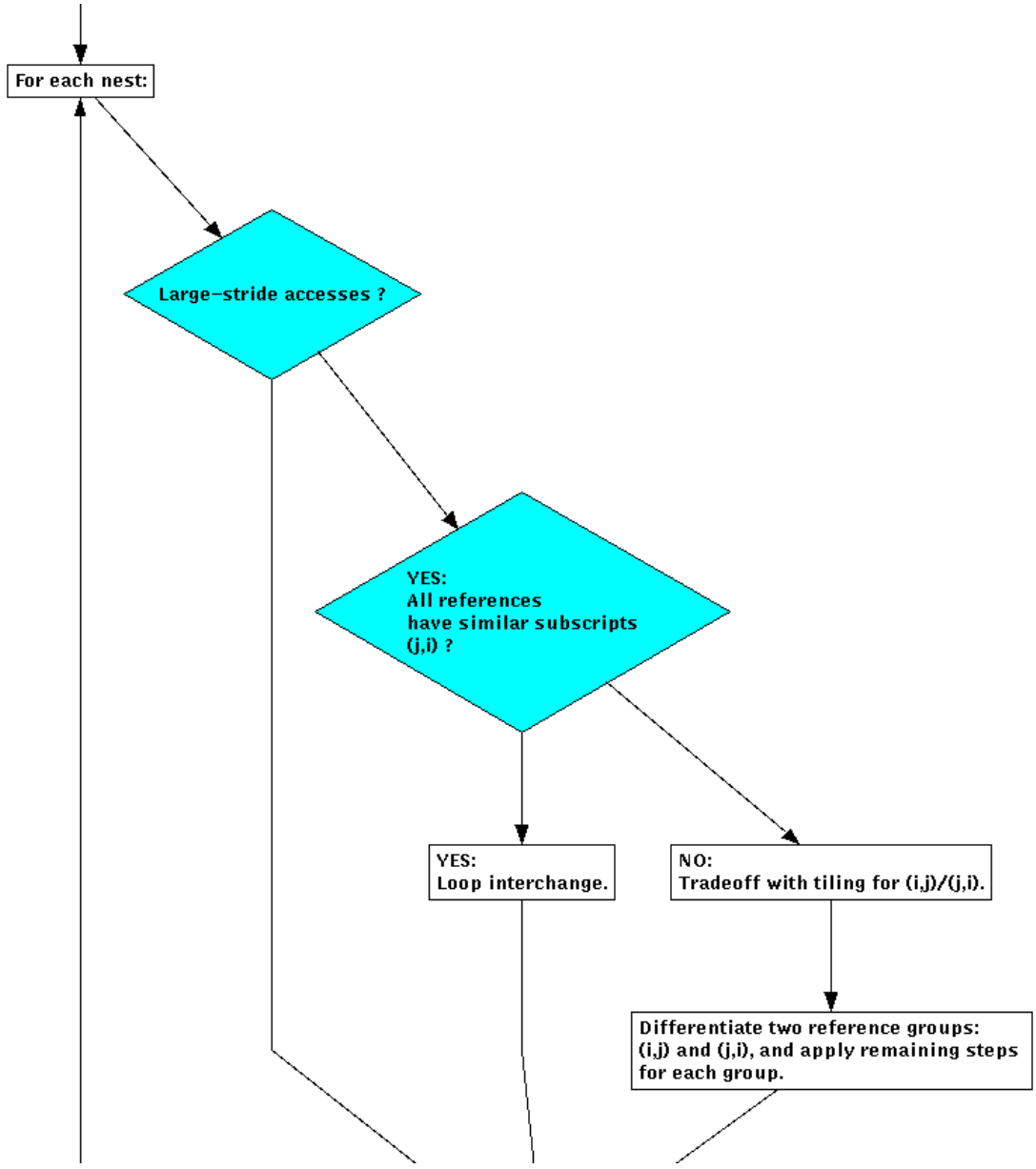


Figure 4 Decision graph (4th part)

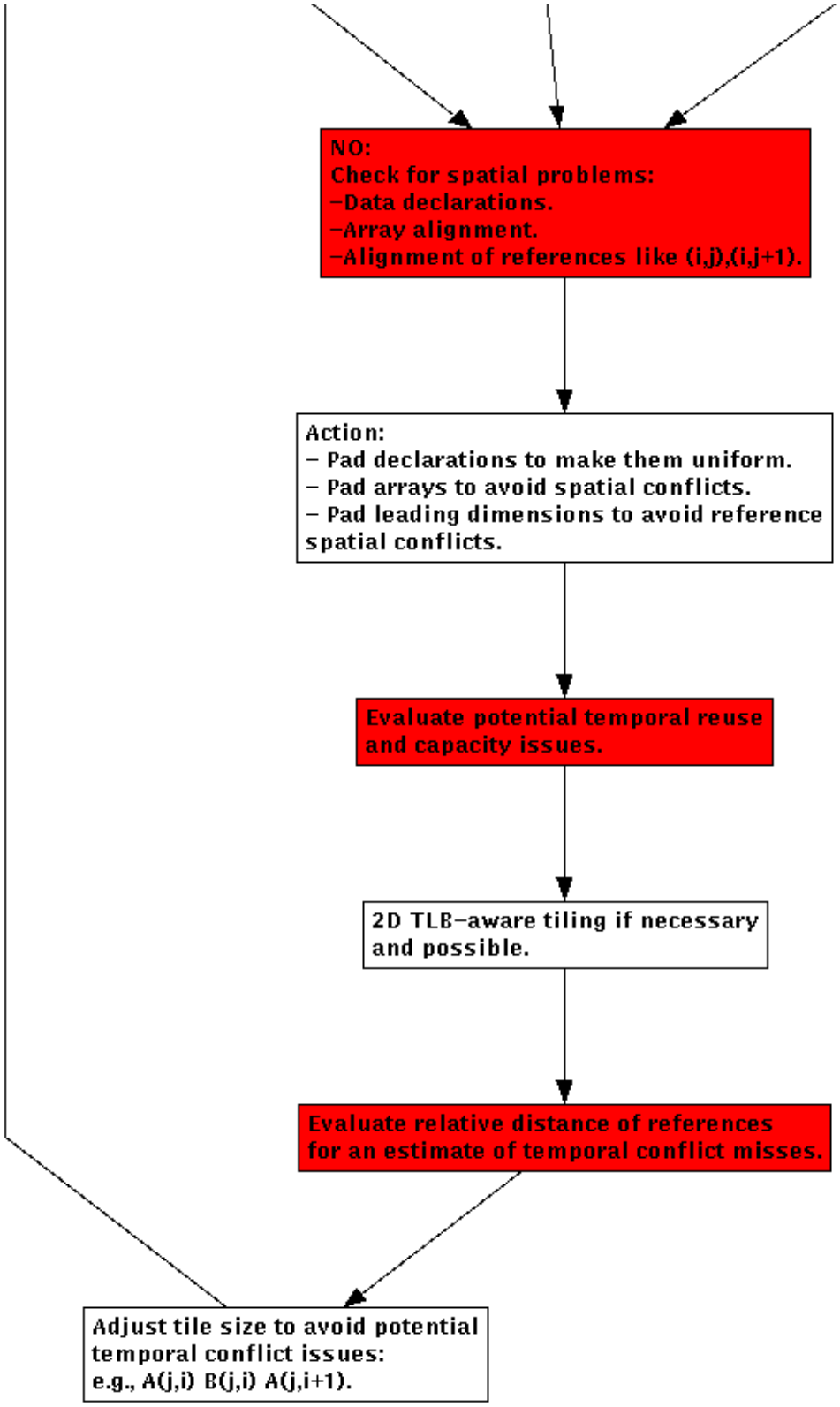


Figure 5 Decision graph (5th part)

An Example of Program Optimization

In this section, we illustrate the process for optimizing an application code, and we show that a precise understanding of codes cache behavior is both necessary and helpful in improving programs' memory performance. We also want to illustrate that traditional program optimization techniques which consider the upper levels of the memory hierarchy first, instead of the TLB for instance, can only achieve part of the potential overall performance improvement.

For that purpose, we pick a SpecFP95 code, SWIM, and we perform a detailed analysis of its cache behavior. We show that classic cache optimizations can strongly benefit from such cache behavior analysis and we achieve 57% execution time reduction on an Alpha 21164 workstation.
SWIM is a weather prediction program based on finite-differences. It is an iterative program composed of three main routines each containing one main loop and several complementary loops dealing with domain borders. These three main routines make the global loop body. The Compaq workstation we used for the evaluation has a 3-level cache hierarchy with an 8-kbyte 1-way first-level cache (L1), a 96-kbyte 3-way shared second-level cache (L2) and a 2-Mbyte 1-way shared third-level cache (L3). The TLB is fully-associative. The Alpha 21164 is a 4-issue in-order superscalar processor. The original program execution time on this workstation using `-O4` optimization level is 228s. The optimizations described below have been conducted with `-O4` instead of `-O5` because software pipelining disrupts locality analysis. However we timed the original and optimized versions with both `-O4` and `-O5`. We achieve a 57% execution time reduction with `-O4` and 45% with `-O5` using the same optimizations, and both optimized versions run in 97s, i.e., the reduction is smaller with `-O5` but we obtain the same execution time. In the paragraphs below, we briefly explain how we have analyzed and optimized the program. All improvements are expressed in percentages of execution time with respect to the original execution time, i.e., 228s.

Spatial Intra-Nest Conflict Misses. We start with a simple time profile and then we use the cache profiler to highlight program critical sections. We first focus on the most time-consuming routine name `calc2`. The main loop of `calc2` is shown below (`N=512, M=512`).

```
PARAMETER (N1=513, N2=513)
COMMON  U(N1,N2), V(N1,N2), P(N1,N2),
UNEW(N1,N2), VNEW(N1,N2),

1        PNEW(N1,N2), UOLD(N1,N2),
VOLD(N1,N2), POLD(N1,N2),
2        CU(N1,N2), CV(N1,N2),
Z(N1,N2), H(N1,N2), PSI(N1,N2)

C......
DO 200 J=1,N
DO 200 I=1,M
UNEW(I+1,J) = UOLD(I+1,J)+
     1
TDTS8*(Z(I+1,J+1)+Z(I+1,J))*(CV(I+1,J+1)+CV(I,J+1)+CV(I,J)
```

```
     2       +CV(I+1,J))-TDTSDX*(H(I+1,J)-H(I,J))
VNEW(I,J+1) = VOLD(I,J+1)-TDTS8*(Z(I+1,J+1)+Z(I,J+1))
     1       *(CU(I+1,J+1)+CU(I,J+1)+CU(I,J)+CU(I+1,J))
     2       -TDTSDY*(H(I,J+1)-H(I,J))
PNEW(I,J) = POLD(I,J)-TDTSDX*(CU(I+1,J)-CU(I,J))
1       -TDTSDY*(CV(I,J+1)-CV(I,J))
200 CONTINUE
C......
```

Example 1 Main loop of the `calc2` subroutine.

The cache profiler indicates that most misses are L1 misses, and that the L1 miss ratio is high, see *Table 1 Performance of `SWIM` after each optimization step.* Conflict misses are responsible for a very large share of the loop nest misses. Individual load/store miss ratios rangefrom 12.5% to 100%. As all array references have the same leading dimension, the linearized subscripts are all identical except for a constant. Therefore, they belong to the same *translation group*. Since there are no self-interference misses, we focus on internal cross-interference misses. Evaluating such misses amounts to studying the relative cache distance of the different array references. The cache debugger/visualizer can be used to quickly monitor the relative cache mapping of the different arrays. *Figure 6 Cache mapping of `calc2` array references (1st iteration).* is a screen dump of the cache visualizer at the first iteration of the main `calc2` loop.

Program version	Cache L1	Cache L2	Cache L3	TLB	Temps (seconds)	
	Miss ratio	Miss ratio	Miss ratio	Miss ratio	O4	O5
original	0,57	0,05	0,58	0,0019	228	172
padding	0,08	0,10	0,57	0,0019	138	129
padding + blocking 1-loop	0,08	0,12	0,53	0,0030	171	149
padding + blocking 2-loops	0,08	0,10	0,57	0,0019	136	127
merging + padding	0,08	0,09	0,60	0,0019	127	121
merging + padding + blocking	0,07	0,09	0,58	0,0019	132	124
forward substitution + merg. + pad. + block.	0,09	0,06	0,54	0,0021	97	95

Table 1 Performance of `SWIM` after each optimization step.

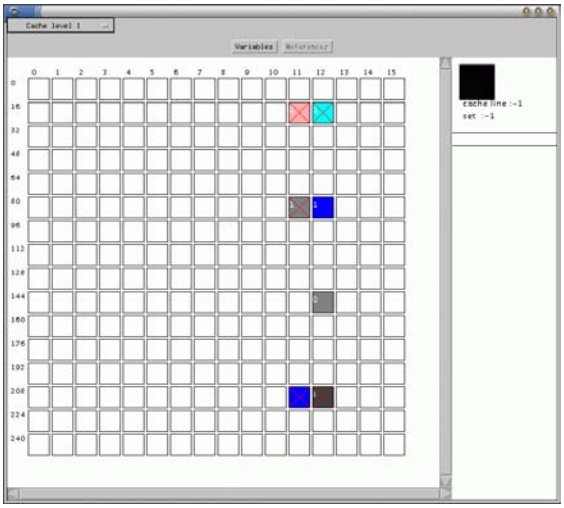


Figure 6 Cache mapping of `calc2` array references (1st iteration).

Array	Element(0,0)	Element(0,1)
u	0	64
v	128	192
p	0	64
unew	128	193
vnew	1	65
pnew	129	193
uold	1	65
vold	129	193
pold	1	65
cu	129	193
cv	1	65
z	129	194
h	2	66
psi	130	194

Table 2 Original arrays mapping in L1 cache.

The precise cache mappings of the different references are indicated in *Table 2 Original arrays mapping in L1 cache.* Several groups of arrays map to the same cache line (129,1,65,193,…) resulting in spatial internal cross-interferences within the translation group. To remove these misses we introduce *padding* arrays `PAD1(PA), PAD2(PA)` between each array declaration and we find the `PA` value that removes all misses (in this case, `PA=8`). An example of array declaration padding is shown below:

```
COMMON U(N1,N2), V(N1,N2),…

C… After Padding
```

```
COMMON U(N1,N2), PAD1(71), V(N1,N2),…
```

Example 2 Padding.

We then achieve a 39% reduction of execution time and a strong reduction of the L1 miss ratio, as shown in *Table 1 Performance of `SWIM` after each optimization step.* The cache debugger/visualizer confirms that array references are then much better distributed within the cache.

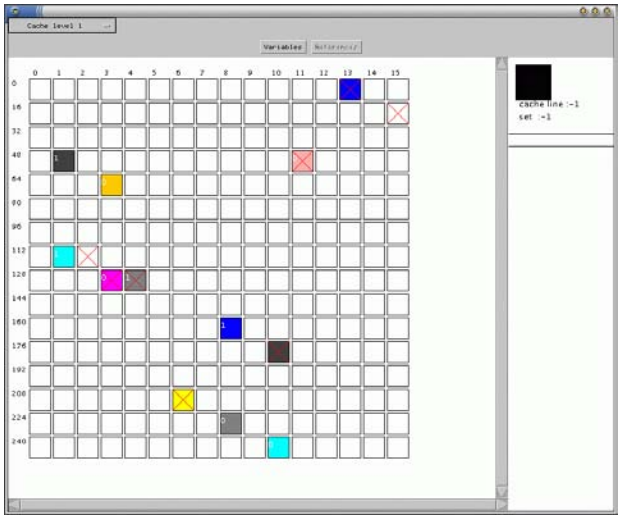


Figure 7 Cache mapping of `calc2` array references after padding.

Intra-Nest Capacity Misses. Because of the large number of arrays, the cache is too small to exploit group-dependences like `CV(i,j), CV(i,j+1)`. As a result the remaining 12.5% misses are almost all capacity misses. Note that these capacity misses are a combination of intra-nest and inter-nest capacity misses. For the moment we focus on intra-nest capacity misses. *To remove intra-nest capacity misses, we block loop I* as shown below. The resulting loop nest has almost no L1 capacity miss but *execution time increases by 14%*, and moreover the loop nest exhibits conflict misses again.

```
DO II=1,M,BI
DO 200 J=1,N
DO 200 I=1,MIN(M,II+BI-1)
      ...
200 CONTINUE
```

Example 3 Blocking the innermost loop.

In fact, capacity misses were *hiding* conflict misses that are exposed once capacity misses are removed. Using the cache debugger, we find several *internal cross-interferences that disrupt group-dependence reuse*.

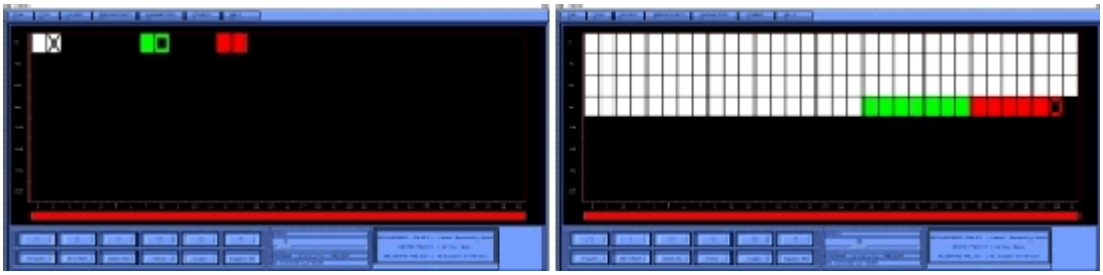


Figure 8 Cross-interference disrupting group-dependence reuse.

For example reference `POLD(i,j)` (in light grey in the above figure) is located in-between references `CV(i,j)` (in white) and `CV(i,j+1)` (in dark grey) so that `POLD` flushes the data referenced by `CV(i,j+1)` before `CV(i,j)` can reuse them (the same phenomenon recurs with other references). By changing the padding again, we artificially shift arrays base addresses and we eliminate these interferences, as shown in the above figure. Execution time decreases but still remains higher than before blocking was applied.

TLB misses. *Blocking* on the innermost loop has reduced capacity misses but it *has increased the number of TLB misses by 36%* as pages are traversed much faster than in the original loop. As the TLB is fully-associative, TLB misses are capacity misses. To reduce such misses, *we simply block the outer loop J*, as shown in the nest below. Finally, we achieve 41% execution time reduction.

```
DO 200 JJ = 1,N, BJ
DO 200 II=1, M, BI
DO 200 J = JJ, MIN(N,JJ+BJ-1)
DO 200 I = II, MIN(M,II+BI-1)
        ...
200 CONTINUE
```

Example 4 Blocking both loops.

Inter-Nest Misses. We now focus on inter-nest locality. In fact, *most remaining misses are inter-nest capacity misses*. To remove these misses we have extensively used *loop merging/fusion*. We apply loop fusion on the three main loops of the three main routines to reduce reuse distances. Though loop fusion raises numerous dependence and loop boundary issues, they are out of the scope of this section. We have applied fusion on the original nests, so we had to pad and block again the merged loop nest because spatial interference misses are even more numerous in the merged loop nest than in the original loop nest. With these different transformations, we achieve an overall execution time reduction of 45%.

```
DO 10 j = 1, n
  DO 10 i = 1, m
    u(i,j) = ...
    v(i,j) = ...
10 CONTINUE
```

```
DO 20 j = 1, n
  DO 20 i = 1, m
    unew(i,j) = u(i,j) + ...
    vnew(i,j) = v(i,j) + ...
20 CONTINUE
```

```
DO 10 j = 1, n
   DO 10 i = 1, m
     u(i,j) = ...
     v(i,j) = ...
     unew(i,j) = u(i,j) + ...
     vnew(i,j) = v(i,j) + ...
10 CONTINUE
```

Example 5 Loop Merging.

Memory-Bound Programs. Either because of programming style or because of transformations imposed by vectorization, *many numerical programs perform very few arithmetic operations per data fetch*. The general programming model is to fetch data, perform one operation, store the result and fetch again the data for another operation. In other terms, functional units are often idle because the processor is busy fetching data from memory *even for programs with low miss ratios*. Therefore, with proper transformations, it is possible to increase the number of arithmetic operations without increasing overall execution time.

Therefore, after merging loop nests, we apply forward substitution to replace memory accesses by computations. For instance `CU(i,j)` is replaced by `0.5*(P(i,j)+P(i-1,j))*U(i,j)`. As a result, the total number of load/stores decreases by 20% but *the total number of arithmetic operations increases by 69%.* Normally, this increase should wipe out any benefit, but as SWIM is largely *memory-bound*, the overall execution time decreases to 97s versus 228s in the original program, i.e., 57% of the original program execution time.

Summary. This case-study illustrates that cache phenomena can abrogate part or all the benefits of classic cache optimizations like loop blocking and loop fusion. A precise understanding of the code cache behavior is often needed to efficiently apply such optimizations. Moreover optimizations that specifically target conflict misses like padding must be used in combination with blocking and fusion lest these optimizations perform poorly. Finally, this case-study illustrates that a manual process for optimizing applications can achieve significant performance gains if it is based on detailed cache performance analysis.

APPLICATION OF GENETIC ALGORITHM TO THE OPTIMIZING PROCESS

Eric Garnier
ONERA

General Considerations

Description of the problem

One of the key problem concerning loop optimization is the so called “black box compiler” concept. The effort of a loop transformation optimization process can be ruined by the compiler on which the user exerts no control. Another problem rapidly which occurs in practical situations is the huge number of possible transformations. Since no one knows really if a given transformation will be efficient or not (thanks to the “black box compiler” effect), all the transformations implemented in the MHAOTEU tool for instance are possible candidates. As these transformations may apply virtually to all loops and all array indexes, the total number of tests grows exponentially with the size of the code.
This suggests the use of an automatic optimization procedure to find the most efficient combination of options. Since genetic algorithm (GA) are proven to be particularly efficient to find a global optimum in a wide search space. Our efforts have then been concentrated on an adaptation of GA to the optimization of code performance through the search of an optimal combination of loop transformation preprocessor options.

Sum-up of some GA characteristics

The basic underlying principle of GA is that of the Darwinian evolutionary principle of natural selection, wherein the fittest members of a species survive and are favored to produce offspring. The mathematical formalism of

evolutionary algorithm is due to Holland [1] and GA have been popularized by the textbook of Goldberg [2].
GAs attempt to find a good solution to some problem (e.g., finding the maximum of a function) by randomly generating a collection a solution of potential solutions to the problem and then manipulating those solutions using genetic operators. In GA terminology, we say we generate a population of solution and refer to each solution as an individual. Each solution is assigned a scalar fitness value which is a numerical assessment of how well it solves the problem. The key idea is to select for reproduction the solutions with higher fitness and apply the genetic operators to these to generate new solutions. Again, in GA terminology these new solutions are called newborn individuals. Through mutation and re-combination (crossover) operations, better newborn solutions are hopefully generated out of the current set of potential solutions. This process continues until some termination condition is met (maximum number of iteration or condition on the solution of the problem to be optimized).

Description of GA components

We now described the main components of a genetic algorithm in the context of numerical optimization.

Representation (genotype)

In order to use GA, it is necessary to map the solutions of the problem to fixed length strings of some alphabet. The resulting strings are called the representation (genotype in GA terminology). The most common representations are binary and floating point. In a binary representation, each component of a solution vector is converted to a binary encoding and then these encodings are concatenated in order to form a binary string which becomes the genotype of the solution (the binary string is also called chromosome). In floating point representation, the component vector is represented with a one-dimensional array (i.e a vector) of floating point numbers.

Initialization strategy

In order to start the GA evolution process, an initial population of individuals must be generated. The most common method to initialize GA is random initialization in which the initial population consists of random binary string or

vectors (depending of the chosen representation) uniformly distributed in the search space hypercube.

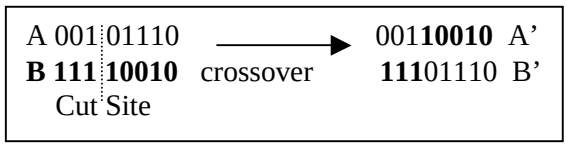
The selection strategy

The selection strategy decides how to select individuals to be parents for newborns. Usually the selection applies some selection pressure by favoring the individuals with better fitness.
The most common section methods are:

- Fitness proportional selection (roulette wheel): Each individual's probability of being selected is proportional to its fitness value.
- Rank-based selection : Each individual's probability of being selected depends on its fitness rank in the population rather than the actual fitness value. The most common rank-based selection methods are:
- Tournament selection: To select an individual reproduction, multiple candidates (usually two) are selected with uniform probability. The fittest of these candidates (the winner of this virtual tournament) is then selected for reproduction.
- Weight series selection: in this method, each individual is assigned a weight that depends on its fitness rank in the population. Proportional selection is then done using the weights rather than the actual fitnesses. The weights are usually taken to be a decreasing arithmetic or geometric series.

Genetic operators

These operators use parent(s) to create newborn(s). These operators are usually either binary (crossover) operator which take two parents and produce one (or two) newborn(s) that resemble both (leading to exchange of genetic material), and unary (mutation) operators, which take one individual and produce a perturbed version of it.
The most known crossover operator is the point crossover which aligns the genotype of the parents. A crossover position is then randomly selected with uniform probability and the part of the first parent's chromosome before the crossover position (also noted cut site) is copied to the corresponding of the newborn. The rest of the newborn comes from its corresponding place in the second parent's chromosome.



The mutation operator is a random alteration of a bit at a string position. It is based on a mutation probability Pm. For a binary representation, a mutation means flipping a bit 0 to 1 and vice-versa. The mutation operator enhances population diversity and enables the optimization to get out of local minima.
The last kind of genetic operator are selection operators which are described in the previous section.

Description of a simple genetic algorithm

To minimize the solution f(x) of a problem P a simple binary coded GA works as follows:
(1) Generate randomly a population of N individuals;
(2) Evaluate the fitness function of each individual genotype;
(3) Select a pair of parents with a probability a selection depending of the value of the fitness function. One individual can be selected several times;
(4) Crossover the selected pair at a randomly selected cutting point with probability Pc to form two new children;
(5) Mutate the two newborns by flipping a bit with probability Pm;
(6) Repeat steps 3,4,5, until a new population (on constant size N) have been generated;
(7) Go to step 2 until convergence.

After several generations, one or several highly fitted individuals in the population represent the solution problem P. The main parameters to adjust for convergence are the size N of the population, the length lc of bit string (length of the chromosome), the probabilities Pc and Pm of crossover and mutation respectively.

Description of the GA code

The main part of the code was found on a web library. It is a C version of Golberg Simple Genetic Algorithm FORTRAN code's rewritten by R. E. Smith (University of Alabama) and latter modified by J. Earickson (Boeing company). Our contribution was the coding of the translator (C routine) which converts binary strings (understandable by the GA code) into preprocessor directives and reciprocally directives into binary strings. This algorithm is designed for maximization problem. The sign of the objective function is supposed to be changed to perform minimization problem. One can also keep in mind that roulette wheel selection only works with positive function (it is then sometimes necessary to add some positive constant to the objective function).

Installation Guide

- Get the sga-1.0.tar file.
- Untar this file (type : *tar xvf sga-1.0.tar*).
- Compile the code (type : *make*), you must have a C compiler installed.
  The executable file is named *sga.* Note that the portability of this code has been successfully assessed on DEC/Alpha and SGI/Origin 2000 servers.

User's Guide

The user is supposed to fill three files. The file named "input" must contain the following information.

- the number of GA run to be performed;
- the size N of the population (even number required);
- the length lc of the chromosome (number of pair of preprocessor options);
- a switch (y/n) for the printing of chromosome strings;
- the number of iteration Ni;
- the probability of crossover Pc;
- the probability of mutation Pm;
- the tournament size for selection (i.e. 2);
- a random seed for the random number generator;
  We give an example of the "input" file.

```
1
4
8
y
10
0.9
0.05
2
0.5
```

The file called "option" must be filled with 2*lc preprocessor options (one per line) considered by pair. Moreover the last line of the file must contain either 1 or –1. If the objective function is always positive (resp. negative), a value a 1 (resp. –1) is chosen for a maximization process whereas –1 (resp. 1) is set for a minimization process. Avoid any change of sign of the objective function during the optimization by adding an appropriate constant. The "option" file can take the following form:

```
-LNO:fission=0
```

```
-LNO:fission=1
-LNO:gather_scatter=0
-LNO:gather_scatter=2
-LNO:non_blocking_loads=OFF
-LNO:non_blocking_loads=ON
-LNO:interchange=OFF
-LNO:interchange=ON
-LNO:prefetch=0
-LNO:prefetch=2
-LNO:prefetch_ahead=1
-LNO:prefetch_ahead=4
-LNO:ou_max=1
-LNO:ou_max=10
-LNO:ou_prod_max=1
-LNO:ou_prod_max=16
-1
```

Finally, fill the file named “routine” with the shell command that the GA code will call. This command must run the evaluation of the fitness (or objective) function. In this study, the “routine“ file is just :

```
sgamake name_of_the_routine_to_be_optimized
```

where `sgamake` is the name of the shell procedure.
The files “outout” and “resput” ensure the communications between the GA code and the fitness function evaluation shell procedure. The “output” file contains the list of preprocessor options given by the GA code and the “resput” file must contain the result of an evaluation of the objective function (one floating point number).
Between these two files, the shell procedure (run by the command line written in the “routine” file) must:

(1) Includes the “output” file in the makefile of the code to be optimized. (For now, optimizations have been undertaken sequentially, routine after routine. So the options of “output” concern only one routine).
(2) Compiles the code
(3) Runs the code with some profiler
(4) Extracts the needed informations given by the profiler

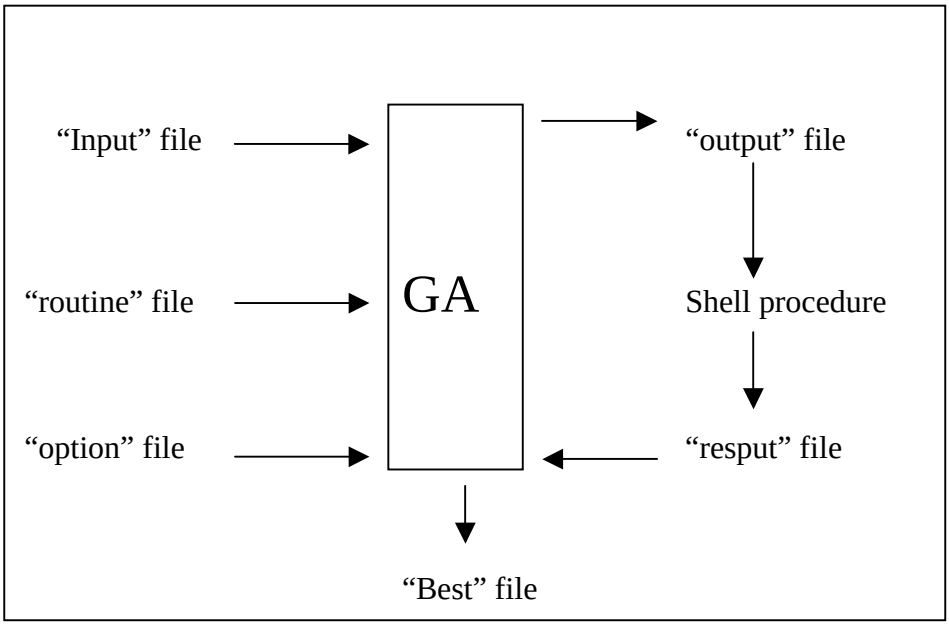
Writes this information (one floating point number) in the “resput” file.

As an example, we give the shell procedure used in this study (“sgamake” file).

```
cd ..
OPTCPL1="  -r4 -i4 -64 -mips4 -extend_source -O3"
cp makefile makefile_s
read A < SGA/output
cat << eof  >>makefile_s
$1.o: $1.f
f90 $OPTCPL1 $A -c $1.f
eof
touch $1.f
```

```
echo "Compilation en cours:" $A
smake -f makefile_s  2>a >s
unlimit stacksize memoryuse
pixie flu3m 2>b >c
echo "Execution en cours"
flu3m.pixie < don100ref >flu3m.output
prof flu3m.Counts > p.Counts
grep $1 p.Counts |awk '{print $2}' > SGA/resput
cat SGA/resput
cd SGA
```

The communications with the GA code are summarized in the following chart.



Note that the best individual found by the GA code is written in the "best" file. The GA code must be run by typing *sga input*. If y is set at the fourth line of the "input" file, some information about the tested chromosomes and the convergence of the GA code are displayed on screen. These information can be redirected in a "out" file by typing *sga input out.* Examples of the "input", "option", "routine" files and a shell procedure (named "sgamake") are provided in the tar file.

In this first attempt to validate the concept of the use of GA for optimization, preprocessor options must be given by pair which are associated to a bit. Typically, the option interchange=ON will be associated to 1 and interchange=OFF will be associated to 0. The effect of an option such as fission=n (with n varying from 0 to 2) can only be investigated by choosing two of the three possible values. Such a limitation could be suppressed in a future version. The number of options (which is the length of the chromosome) is unlimited.

Application to the FLU3M code

Choice of the objective function

Now, the user is supposed to choose the objective function to be minimized. For a code optimization, it can be the overall CPU time given by the command "time" (dependent of the workload of the CPU) or the time spent by the code execution given by a tool like Pixie (independent of the workload). Many other functions can be considered such as the number of hits in the L2 cache (to be maximized).
Moreover, it is evident that an overall measurement will not be efficient and informative enough. And this tool was preferred to be used to minimize the same objective function routine by routine.

Definition of the test case

To demonstrate the interest of using GA to optimize code performance, we have tried to find automatically the best preprocessor options to be used with the six most time-consuming routines of FLU3M. Computations were run on a SGI R12000 server and we use the LNO preprocessor. In this test phase, the effect of only lc=8 pair of options is investigated.
The chosen pair of options are the following :

Bit =0	Bit=1	Comments
-O2	-O3	
fission=0	fission=2	Disable or aggressive fission
gather_scatter=0	gather_scatter=2	Disable or multi-level gather-

		scatter
non_blocking_loads=OFF	non_blocking_loads=ON	
interchange=OFF	interchange=ON	
prefectch=0	prefectch=2	Disable or aggressive prefetching
prefetch_ahead=1	prefetch_ahead=4	1 or 4 lines ahead of the reference
ou_max=1	ou_max=10	1 or 10 possible unrolled copies

For example, a chromosome 10101101 will means that the options –O3, fission=0, gather_scatter=2, non_blocking_loads=OFF, Interchange=ON, prefectch=2, prefetch_ahead=1, ou_max=10 are selected.
The seven last options are arguments of the LNO (which are invoked using the syntax –LNO:fission=0). Note that if –O2 is selected LNO argument are not taken into account. So this test case admits only 2**7+1=129 different combinations.
To avoid a sweep on every possibilities (contrary on GA spirit) the number of individuals N is restricted to 4 and the number of iterations Ni is chosen equal to 10.
Typical probability of crossover Pc are chosen in the interval [0.7 ; 1] and probability of mutation are lower than 0.1. Note that some recent GA codes allow a dynamic (function of the iteration number) variation of these probabilities. In this preliminary test, Pc is chosen equal to 0.9 and Pm equal to 0.05. Selection by deterministic tournament of rank two has been chosen.

The objective function to be minimized is the CPU time given by the tool Pixie. 20 iterations of FLU3M are performed to leave Pixie collect a significant amount of statistic. With 40 (N*Ni) evaluations of the objective function (compilation + execution) multiplied by 6 routines, the CPU time needed to achieve the optimization is about 4 hours on one R12000 SGI processor.

Results

- Routine flroe3n gives 7.73 s whatever the options.
- Routine gradr3 gives 7.63s with –O2 and 7.24 s with –O3 whatever the options of LNO.
- Routine invlus3fm1 gives 4.61 s with –O2 and 5.48 s with –O3 whatever the options of LNO.
- Routine invlui3fm1 gives 4.27 s with –O2 and 4.85 with –O3 whatever the option of LNO.
- Routine flnsc3c gives 3.38 s with –O2 but two results are found with –O3 : 3.369 s or 0.504 s. So, with some combinations of options, –O3 can be better than –O2 but with another combination –O3 is clearly worse than –O2. Let us clarify this

point. A sample of individuals giving both results are reported in the following table. Note that the list individuals giving a Fitness of 3.369 is not exhaustive. Moreover, we recall that the solution of a GA optimization is the ensemble of best fitted individuals.

Fitness : 3.369 s	Fitness 3.504 s
10111010	10111100
10110011	**10011111**
10011011	
11110010	

Comparing the two individuals in bold font which differ only by one bit, we can conclude that for this routine the option prefetch=2 (the sixth option) worsen the CPU time.

- Routine slpmi3 gives 3.69 with –O2 and three different value (3.61s, 3.97s, 4.23s) with –O3 depending of the LNO arguments. We have reported few significant individuals in the following table (the list for the fitness of 3.61s is not exhaustive).

Fitness: 3.61	Fitness : 3.97	Fitness 4.23
11101100	**10110000**	10110111
11101101	10111010	11011111
11100100	10110001	11001101
11100101		**10110100**
11110100		

The comparison of the two individuals in bold font clearly suggests that the option prefetch=2 increases the CPU time spent in the routine. But one can remark that the best individuals (Fitness =3.61s) also use this option. Nevertheless, in their case, the option prefetch=2 is used in combination with fission=1 and gather_scatter=2 whereas the individuals giving a fitness of 4.23 use either fission=1 or gather_scatter=1 but not a combination of both. An additional manual test with the chromosome 11101000 (fission=1 + gather_scatter=2 + prefetch=0) has given a fitness function of 3.61s. This demonstrates that results are insensitive to the prefetching if fission and gather_scatter are combined.
For this routine the gap between best and worst individuals reaches 17 %.
This routine is a clear example of the presence of non cumulative effects in the use of combined options.

If we use one of the best individual for each routine, we can evaluate the gain compared with a user who chose between –O2 or –O3 for all routine. The sum of CPU time for the first four routines (which do not depend on the argument of

LNO) is equal 24.24s if –O2 is always chosen, 25.30s if –O3 is always chosen, and 23.84s is the best option is chosen for each routine. The gain appears to be modest : 1.7 % compared to –O2 alone and 6 % compared to –O3 alone.

Choice of optimization strategy

To optimize a set of chosen routines, two methods are available.
The chosen method will depend mostly and the belief of the user and his knowledge of his code. If the time spent in a routine with a set of preprocessor options is believed to be independent of the set of options used for the other routines, one may prefer a sequential application (routine after routine) of the optimizer. In case of dependence of the objective function of a routine to the set of options of other routines, we must consider a “big chromosome” of length lc1=(number of preprocessor option * number of dependent routines) bits. This is not a problem for the GA code but one must take into account that if a population of N individuals may explore a significant portion of a search space of dimension 2**lc after a reasonable number of iterations Ni, it will be certainly not the case for a search space of dimension 2**lc1. In this demonstrative computation, we have only considered the first strategy.

Conclusion

The black-box effect of the compiler is clearly demonstrated and the application presented illustrates that some gain can be expected of the use of GA for the optimization of code performance. Some work is still necessary to draw clear-cut conclusions. In this study, the choice of the arguments is questionable and it is possible that other arguments have led to larger variations of the objective function. Moreover, a complete study must be undertaken to investigate the dependence of the results to GA parameters. Moreover, the “big chromosome” strategy remains to be evaluated.
Furthermore, GA are naturally parallel algorithm and further development can be considered to take into account of this property running the N evaluation of the fitness function in parallel.

Bibliography

Golberg D. E. (1990) Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Welsey, Reading, Mass.

Holland J. H. (1975) Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor.

END-USER APPLICATION : ONERA'S AERODYNAMIC SOLVER FLU3M

Philippe Guillen, Eric Garnier
ONERA

Sid Ahmed Ali Touati
INRIA

Introduction

This part is devoted to the application of optimization techniques acquired during the MHAOTEU for an end-user application. There are exposed through the optimization of the numerical code FLU3M an aerodynamic flow solver [1], [2], widely used in the french aerospace industry to design launchers, shuttles, missiles, supersonic airplanes, helicopters, in order to show the efficiency of the different techniques implemented a realistic testcase made of the computation of a formula one aerodynamic device is chosen. In a first part, the numerical code is rapidly described as well as the properties of the main CPU consuming routines. In the second part, the different kinds of optimization techniques are exposed. A conclusion is then drawn focussing on the extensibility of the optimizing process to other classical CFD codes.

Presentation of the numerical code

Introduction

The development of industrial numerical codes to solve the equations of fluid mechanics represents an important investment with some uncertain prospective choices to be made. These last years quite an important number of attractive methods have been proposed for the simulation of viscous flows around complex 3D configurations : finite volume or finite element methods, using structured or unstructured grids, having a centered or non-centered numerical scheme, etc. The selection of the most promising method seems to be an hazardous and difficult choice, a matter of compromise between different considerations which are generally contradictory. Moreover some important element of choice such as the available computer technology in the future are difficult to foresee and it is clear that the evaluation of the numerical efficiency could be quite different according to the kind of processing : superscalar, vector or parallel. From an industrial point of view, once the desired level of accuracy is reached, the most important quality of the code is robustness which can lead to select algorithm not very computationally efficient. Another important point to consider is the extension of the

code to more complex set of equations in the future, such as more complex gas or more complex models of turbulence which may affect the design and the architecture of the solver. As an example some well-known commercial CFD codes are integer CPU intensive for generality and modularity reasons.

Architecture of the code : General considerations

The Architecture of the code has been dictated by constraints concerning geometrical considerations and computational aspects

Geometrical Considerations

The treatment of complex geometries has led us to choose a multiblock solver. The computational domain being made of several structured, possibly overlapping or patched domains. The overlapping technique using a so-called Chimera technique. Another interesting possibility has been introduced by enabling different kinds of boundary conditions on a given domain face.

Computational aspects

The constraints concerning computational aspects are clearly linked to the available computer technology. The development of the code has begun in 1988. By that time, the available computer technology did not allow much memory core and the code was designed to limit the memory required. It was built around what we called a plane processor i.e. a piece of software which computes the advancement of a plane k of a structured domain ijk. This technique allows to use only 2D temporary arrays in the numerical scheme. Unfortunately this limitation did not allow to use what we call 3D implicit techniques. Nowadays there is no more strict memory limitations and the code is now based on a “block Processor” which enables to compute the advancement on a full domain ijk, the temporary arrays being 3D i.e. equivalent to a triple index.

Code organisation

It results in a code organization built around three key units : a command interpreter which assumes the user interface, a block monitoring unit which decides of the type of computation and a block processor including the numerical scheme. The block processor being by far the most consuming part, we will focus on it in the next part. We just now detail the two first units.

Command interpreter

The command interpreter is a small language which means that the input file is interpreted dynamically. It owns classical control instructions such as DO loops. Its main functions are the following :

- *Monitoring the core* : it consists in attributing the necessary pointers when a new domain is created for the required metrics.

- *Defining thermodynamic states*;
- *Defining scheme parameter;*
- *Defining boundary conditions*;
- *Initializing domain variables*;

- Extracting Data

- Calling the Block monitor

Block monitoring

As the calculations are based on the computation of separated domains, two kinds of computations can be made according to the way the domains are computed :

- Unsteady flows : the timestep is the same for all domains and they are all updated in a coherent manner;
- Steady Flows : the timestep can be different for each cell and blocks can be computed without any coherence and out of order.

Numerical Scheme

The objective here is just to give an idea of the numerical formulation so that the reader get a clearer view of what follows in the optimization process.

Formulation

The unsteady 3D Navier-Stokes equations are written in conservation form

$$Wt+(F-Fv)x+(G-Gv)y+(H-Hv)z =S$$

Where W is what we call the vector of conservative variables which include density, momentum, energy, etc. F represents the convective fluxes that is to say for instance the mass quantity which goes through a given surface within a unit of time. Fv are the viscous fluxes which depend on the gradients of diverse flow quantities (speed vector). S is a source term necessary to model turbulence growth.

To solve these equations we use an implicit upwind TVD finite volume scheme of van Leer MUSCL type. Basically, the numerical scheme includes the following steps.

1) *Introduction of a linear distribution of variables for each direction in each cell to compute the cell interface (also called slope calculation)*
2) *Computation of gradients quantities for each cell*
3) *Computation of convective fluxes using values obtained at each interfac (step 1)*
4) *Computation of viscous fluxes using gradients*
5) *Computation of source terms*

6) *Computation of explicit part (by summing the different contribution computed before and multiplying by the timestep*
7) *Computation of the implicit part (by solving a linear equation system whose second member is the explicit part computed before).*

Programming Notes

To implement this numerical scheme the following steps are coded in the block processor which advances the solution on a domain

1) *Research of intersecting boundaries : this step determines among the boundary conditions that have been declared the ones concerned with this block.*
2) *Computation of timesteps.*
3) *Computation of gradients*
4) *Treatment of boundary gradients : according to boundary conditions, gradients are modified near boundaries and within fictitious cells.*
5) *Computation of slopes*
6) *Treatment of boundary interfaces : according to boundary conditions, interfaces in the fictitious cells are defined.*
7) *Computation of convective fluxes*
8) *Computation of viscous fluxes*
9) *Treatment of boundary fluxes : according to boundary conditions, fluxes on boundary interfaces are possibly modified.*
10) *Computation of explicit increment*
11) *Computation of implicit coefficients*
12) *Resolution of the implicit system*

The advantage of this coding decomposition is to allow the implementation of a large variety of treatments. Each of these steps corresponds to a library with different subroutines linked to a given model or a given boundary treatment.

The overall code includes 2400 routines and about 1 million lines. It is fully written in FORTRAN77 and has run on a great number of platforms : NEC, CRAY, COMPAQ, SGI, IBM, SUN, LINUX, CONVEX…

Description of main CPU consuming routines

In the table underneath are listed the most important CPU consuming routines with their main purposes and characteristics. The purpose is related to the functionality, the ownership to the explicit or implicit part is also given. The characteristics considered here are genericity and different kinds of locality. Their definitions are detailed in the following paragraphs.

Genericity

Generic means that the routine is just one among several having the same functionality. For instance flroe3n.f is just one routine of fluxes evaluation among a dozen doing the same thing but with some numerical procedure difference.

Strong locality

Strongly local (or **bijective**) means that the evaluation of a local quantity requires values affected to the same locality.

Locality

Local (or **weakly surjective**) means that the evaluation of a local quantity, for instance the density of a cell, or the conservative vector at an interface only requires nearby local quantities (the densities in the adjacent cells for instance).

Out of order

Out of order means that there is no dependency on the order in which the loop iterator describes its space of values. In the explicit phase the main loops are out of order. By extension, a routine is qualified out of order if its main computing loops can be computed out of order (this can be seen if the loop is preceded by the comment CVD$ NODEPCHK). Bijective loops are always out of order.

Globality

Global (or **strongly surjective**) means that the evaluated value requires values affected all over the domain. A Gauss-Seïdel sweep for instance is typical of a strongly surjective loop.

All this notions can be largely extended to many CFD finite volume or finite element codes.

	purpose	Main Characteristics
flroe3n.f	Compute explicit Fluxes	Generic, local, out of order
slpmi3.f	Compute slopes	Generic, out of order
tstb3.f	Compute timestep	Strongly local, out of order
precoe.f	Compute Block Processor Working variables	Strongly local, out of order
spsource.f	Compute the source term of the fluid mechanics equations	Strongly local, out of order
invlui3fm1.f	Forward Substitution Sweep of the Lower-Upper Process	Global

invlus3fm1.f	Backward Substitution Sweep of the Lower-Upper Process	Global
core3as1.f	Explicit Update of Conservative Variables	Strongly local, out of order
implvl3.f	ADI Implicit Backward and Forward Sweep	Global, generic
bgr3.f	Computation of gradients at Boundaries	Generic, local, out of order
bvas3.f	Computation of Slopes at Boundaries	Generic, local, out of order
bfl3.f	Computation of Fluxes at Boundaries	Generic, local, out of order
cprdu3.f	Computation of residuals	Global, out of order
ppfw.f	Computation of Boundary Values	Generic, local

Optimization Process

Some General remarks before starting

When one wants to optimize a numerical code there are two things that must be decided. The first one is the targeted machine. The second one is how to estimate gains (or losses) of performance. The answer to this first question is not at all innocent. Of course optimization depends on the CPU architecture. It is well known for instance that some transformations such as the GIR are very efficient for the 21164 architecture and not so for more recent ones. Howeve, the first question has a clear answer : the targeted machines are the ones used by the clients, and following this consideration the choice is clearly a SGI origin 2000 (R12k proc). The second question is not so clear as it seems. One main problem we have with the Origin architecture is that the CPU time evaluation of a job may depend in a rather strong way to the traffic jam on buses. Real CPU time accuracy is really not better than 15% (sometimes more). In these conditions, the verdict of the efficiency of a given transformation for a routine may be in some way speculative (winning 2% with 15% of inaccuracy becomes a stochastic concept) except if one has all the entire machine for himself, which is not the case.

Two other general remarks have to be done concerning the flow solver. The first one concerns the optimized version. It is a Spallart-Allmaras RANS solver. This solver was not in use at the beginning of the MHAOTEU project and so could not be given as a testcase. However, since it is the most used at ONERA during the third year of the project, we decided to select it for operational reasons. The second one concerns the CPU design status of this solver. ONERA is equipped with one of the most powerful Vector machine : the SX-5 from NEC. So the optimization of this version was done for this kind of architecture. The current performance the code on the NEC is around 4

Gflop/s out of 8 Gflop/s of top performance. In order to get this 50% of efficiency, different vector optimizations that may seem a bit curious were realized. Clearly some of the optimizations made consist in going backwards (see the Suppressing Unnecessary Operations paragraph).

Optimization Process : Suppressing Array References

All the optimizations included in this part require a deep knowledge of the programming architecture of the code. Two optimizations have been considered. The first one breaks an aspect of the code modularity. The second one inhibits a vectorization trick.

Integrated implicit solver

For the sake of modularity, the computation of the implicit phase has been decomposed in two coding steps 11) and 12). This modular decomposition enables to test different implicit terms with different kinds of resolution but has the drawback to imply the storage of the implicit coefficients in memory. This storage represents about 75 words per domain cell which is quite important. If we decide to fuse this two steps, there is a clear possibility to compute the implicit coefficients **while** solving the system. This imply a very good use of cache memory and probably registers.

Suppressing a vectorization indirection

The vectorization of the Lower-Upper solver used in FLU3M uses a judicious remark that states that both the lower and upper sweeps can be done using i+j+k as iterator. The idea is that there is no data dependence between all triplets so that i+j+k equals a given integer constant C. As the number of triplets solutions of this equation is very important provided C is not near to 3 or imax+jmax+kmax, the vectorization of the loop is much more efficient than the one of the alternative which is to perform a triple do-loop on ijk. In order to be efficient, all the triplet solutions for a given C were stored for all domains. This could take a lot of memory (about 25% of all the required memory in some practical cases). All that is no more necessary and has been replaced by the triple loop solution freeing a lot of memory and probably bettering the cache behaviour.

Optimization Process : Suppressing Floating-Point Operations

The FLU3M version chosen to be optimized within the framework of MHAOTEU is a turbulent Navier-Stokes solver using the Spalart-Allmaras one-equation turbulence model recently implemented. It has been fully optimized on a NEC-SX5 vector. On this computer the most efficient reduction of CPU time is obtained using very long loops with no indirection between the iterator and the array reference, provided, of course, that there is no dependency (i.e. they can really be vectorized). Such a loop looks like :

```
do l=1, lmax

a(l)=b(l)* f(l)
```

……..
……..

enddo

In fact many of these operations, i.e. the ones concerning fictitious cells, are not necessary. On our supercomputer the absence of an hardware scatter-gather implies that it is more efficient to compute all cells even with some unnecessary floating point operations than to use an indirection or a triple do-loop to limit them. This optimization is no more effective on a superscalar server such as the SGI O2000.

Optimization Process : Data Partitioning

Another idea to exploit memory cache hierarchy is to try to make what could be an extension of the blocking idea but for bigger datasets sizes corresponding to the L2 size rather than smaller ones. Practically it consists in trying to compute a domain using only some contiguous working arrays which may hold in the L2 cache. In order to do so, two actions have to be achieved. Firstly, we have to organize the block processor so that it uses mainly working arrays and not database arrays. Secondly, we have to minimize the size of these working arrays.

Removing database arrays

In order to do so, a deep knowledge of the code structure is required. The idea is to take the necessary values from the database and to store it in a working array called coe which will be used to make all the computations, the final values being reinserted into the database at the end of the iteration.

Limiting working arrays size

Once classical actions to minimize the size of required memory within the block processor have been achieved, the most efficient way to minimize working arrays size is to reduce the dimensions of the domains, this is also called data partitioning. It consists in dividing big domains into smaller ones, knowing that there will not be any difference in the obtained solution. Classically, we have three solutions to achieve that.

Splitblock Software

The idea of automatically dividing domains before computing has been naturally developed within the context of parallel programming. In this context the objective is to make a repartition of the computational work between several CPUs. In this respect, dividing a small number of big blocks into smaller ones is an obvious practical solution if you are able to distribute each block computation on a different processor. Such a software able to take the input of a multidomain configuration to transform it into an equivalent multidomain configuration with more blocks respecting some criteria (dimensions of the greatest block, total number of blocks, minimization of communications between blocks) has been realized

at ONERA. This Splitblock software consists in an Object-Oriented library with block transformation capabilities.

CAD block topology definition constraint (both for fusion and splitting)

Another possibility is to define the constraint on the dimensions of the blocks during the CAD phase of the grid construction. Generally during this phase, for reasons concerning the complexity of the configuration, the first grid topology generated is made of a great number of small or very small domains. In order to simplify the data input, a great number of these blocks are gathered to make greater ones. This phase could be done with the desired constraints.

CFD software built-in blocking

This last technique consists in modifying the code so that for instance the block processor treats a domain by slices of a few k-planes. It requires a very deep knowledge of the code and a rather important number of modifications in many routines.

Optimization Process : MHAOTEU Transformation Techniques

In this part we take into consideration the different transformations proposed either by the MHAOTEU TOOL or used by one of the partner in the workshop exercise, and we examine how they can be applied for our practical case.

The MHAOTEU server proposes two different kinds of transformations techniques : the first one concerns arrays and the second one loops.

Transformation techniques : General Index Reordering

General Index Reordering is a transformation proposed in the MHAOTEU server which consists in switching the arguments of an array. For instance the array A(i,j,k) will be ordered A(k,j,i) if we decide to switch the first and third arguments. One of the interest of such a change is justified by memory efficiency. For Vector machines, experiments conducted a few years ago have shown that best results were obtained when we try to fetch data with a constant increment but not contiguous. For superscalar machines the opposite is generally true. It results in a practical interest in switching the indexes of the arrays optimized for the Vector machine.
A first difficulty in doing so is that it requires a deeper knowledge of the code that it may seem at a first glance. If reordering the indexes of an array can be done automatically using for instance the ad hoc panel of the MHAOTEU tool, the problem comes from the fact that it has to be done in all routines concerned by that array, knowing that in FORTRAN, arrays do not have necessarily the same name in all routines nor the same ordering.

A second difficulty is that it implies a very great number of changes. To get an idea, you will find in the application paragraph underneath a table made of all routines concerned by this optimization in the code with the required switches. The total number of switch is about 300, which means 300 versions of software through the MHAOTEU tool to perform manually without error. That is why in order to have a practical possibility of achieving the task we have used a small awk program displayed below which enables to make a switch of array indexes. All the necessary commands being kept in an executable and editable shell file.

```
This procedure permutes the indexes of an array in a subroutine. per {nom du tableau}
{numero du premier indice} {numero du second indice} {nom du fichier initial} {nom du
fichier modifie}
#!/bin/csh
alias rm /usr/bin/rm
if ( $#argv != 5 ) then
 echo "permutation varname indice1 indice2 infile outfile"
 goto 10
endif
cat > /tmp/row.awk.$$ <<__END__
BEGIN { s=0
  IGNORECASE = 1
}
/^c/ {
  print \$0
  next
}
/subroutine/ {
  print \$0
  next
}
{
  s=substr(\$0,1)
  for(;;) {
    i=index(s,VAR)
    if ( i == 0 ) break
    s1=substr(s,1,i-1)
    i = i-1+length(VAR)
    printf "%s",substr(s,1,i)
    s=substr(s,i+1)
    c=substr(s1,length(s1),1)
    ok=1
    if ( match(c,"[abcdefghijklmnopqrstuvwxyz_0123456789]") != 0 ) ok=0
    for (i=1;i<=length(s);i++) {
      c=substr(s,i,1)
      if ( c != " " && c != " " ) break
    }
    c=substr(s,i,1)
    if ( match(c,"[abcdefghijklmnopqrstuvwxyz_0123456789]") != 0 ) ok=0
    if ( ok == 1 ) {
      printf "%s",substr(s,1,i)
      s=substr(s,i+1)
      if ( c == "," ) {
        printf "%s with no subscript, line %d\n",VAR,NR >> "/tmp/err.$$"
        printf "%s\n",\$0 >> "/tmp/err.$$"
        continue
      }
      if ( c == "(") {
        i=1
        while ( (j=index(substr(s,i),")")) != 0 ) {
            k=index(substr(s,i),"(")
          if ( k > j || k == 0 ) {
            i=i+j-1
            break
          }
          else {
            i=i+j+1
```

```
          }
        }
        t=substr(s,1,i-1)
        n=0
        ok=0
        k=1
        for (j=1;j<=length(t);j++) {
          c=substr(t,j,1)
          if ( c == "," ) {
            if ( ok > 0 ) continue
            n++
            a[n]=substr(t,k,j-k)
            k=j+1
            continue
          }
          if ( c == "(" ) {
            ok++
            continue
          }
          if ( c == ")" ) {
            ok--
          }
        }
        n++
        a[n]=substr(t,k,j-k)
        if ( IND2 > n || IND1 > n ) {
          printf "erreur, nombre d'indices incoherent,ligne %s\n",NR>> "/tmp/err.$$"
          printf "%s\n",\$0 >> "/tmp/err.$$"
          continue
        }
        else {
          for(j=1;j<IND1;j++) printf "%s,",a[j]
          printf "%s,",a[IND2]
          for(j=IND1+1;j<IND2;j++) printf "%s,",a[j]
          if( IND2 == n) printf "%s",a[IND1]
          else{
           printf "%s,",a[IND1]
            for(j=IND2+1;j<n;j++) printf "%s,",a[j]
            printf "%s",a[n]
          }
        }
      }
    }
    s=substr(s,i)
  }
  printf "%s\n",s
  next
}
__END__
#gawk -f /tmp/row.awk.$$ -v VAR=$1 IND1=$2 IND2=$3 $4 > $5
awk -f /tmp/row.awk.$$ -v VAR=$1 IND1=$2 IND2=$3 $4 > $5
rm -f /tmp/row.awk.$$
rm -f /tmp/err.$$

10:
```

Transformation techniques : Loop optimisation

The second kind of transformation techniques considered within the MHAOTEU project concerns loops. This section is a summary of a pratical case study to check the applicability of code transformation and optimization techniques on the FLU3M code. One of the purposes is to provide the programmer some advices and hints on the way on programming for code optimisation in the conclusion.

In order to fully take advantage from the memory hierarchy, the different following techniques are candidates :

- Blocking and tiling;
- fusion and fission
- unroll and jam
- loop interchange

Furthermore, we inspect other fine grain optimization techniques :

- forward substitution
- unrolling
- software pipelining
- prefetching

The first class of loop transformations requires some pre-conditions on the data dependence information to be applied safely. We have implemented a tool called "visudep" (based on the omega test) which computes the data dependencies between the instructions at the high level. The second class of optimizations do not need safety conditions, but they require some conditions to be efficient.

Blocking and Tiling

Blocking (strip mining) is a special case of tiling since it can be considered as a tiling of the innermost loop. This loop transformation is applied to a loop nest in order to keep the successive accessed data in the cache. To be able to pursue this transformation, some conditions must be satisfied :

1. the array must be accessed in a regular way
2. the array indexes must be loop indexes.

Unfortunately, all of the analysed arrays are not indexed with loop index, but with induction variables. As example, the following code example is extracted from "flns3c" :

```
do  k=1,ka
      lk=lk+nja
      lj=0
      do  j=1,ja
           lj=lj+nia
           li=1
      do  i=1,ia
          li=li+1
          l=li+lj+lk
          l1 = l - inc1
          l2 = l
          u1 = coe(2,l1)
          u2 = coe(2,l2)
          v1 = coe(3,l1)
          v2 = coe(3,l2)
          w1 = coe(4,l1)
          w2 = coe(4,l2)
           .....
```

The array "coe" is accessed with the induction variables l1 and l2. This loop cannot be tiled as it is. Since these induction variables depend linearly on (i,j,k), they must be rewritten as an affine function of (i,j,k). Furthermore, the stride of the array accesses is unknown statically since they depend on unknown variables (inc1, nia, nja). We cannot

quantify statically the cache reuse between the different memory accesses. We can of course assume an optimistic stride factor (we assume a unit stride), but we cannot ensure that the tiling transformation would be efficient.

Loop Fusion

This loop transformation is applied when two or more successive loops exhibit some cache reuse, or when their loop bodies are not sufficiently large. The safety condition to applying it is that the new loop after the fusion does not contain a dependence circuit between statements of the two original loops. Unfortunately, there is no two adjacent loops in the analysed ones. However, unrolling the outer loop produces two or more successive inner loops candidate for fusion. This will be commented in the next section.

Unroll and Jam

Unrolling the outer loops can be done for all the loop nest in order to exhibit adjacent inner loops candidates for fusion. Unfortunately, the array indexes used in the loops make the static data dependence analysis difficult, and hence returns conservative information regarding dependence distances which inhibits loop fusion. As example, the following loop is extracted from "flvle3c.f" :

```
do  k=1,ka,ksaut
      lk=k*nja
      lj=0
      do  j=1,ja,jsaut
            lj=j*nia
            li=1
            do  i=1,ia,isaut
                  li=i+1
                  l=li+lj+lk
                  ...
                  flu(1,l) = tc*fm
```

The instruction in bold characters has a self output dependence with a distance <+,*,*> as reported by our tool. To be conservative, we must consider a distance <1, *, *>. This distance inhibits unrolling the outer loop (k) to jam the two produced inner loops (j). So, to apply the unroll and jam transformation on this loop, all the output dependencies must be eliminated by using intermediate arrays.

Loop Fission

When a loop is very large and the instruction cache is small, or if it accesses multiple large arrays which produce cache conflicts, loop fission (distribution) can be considered. The safety condition before splitting a loop into two smaller loops is to preserve the dependence circuits : each small loop must not broke a strongly connected component of the original dependence graph. All the analysed loops verify this condition and hence are candidates for loop fission.

Loop Interchange

All the array accesses in our loops have a constant in the first dimension and an induction variable in the other dimensions. Since arrays in fortran are column major, loop interchange can be very efficient since it would perform the array access in the innermost loop, and hence they exhibit spatial locality. As example, the following loop is extracted from "implvi3.f" :

```
do  k=1,ka
      lk=lk+nja
      lj=0
      do  j=1,ja
         lj=lj+nia
         li=1
         do  i=1,ia
              li=li+1
              l=li+lj+lk
              tcxn = tijk(1,l)
              tcyn = tijk(2,l)
              tczn = tijk(3,l)
              ....
```

As we see, the arrays are indexed with a constant in the first dimension : loop interchange would not be efficient since the index is not used for the first dimension. To enable loop interchange, arrays must be transformed by a general index reordering, i.e. we inverse the array dimensions. The safety condition to apply loop interchange is that all the dependence distance vectors must stay lexicographic positive after interchanging (i.e. the first non null element in these vectors must be strictly positive). Unfortunately, the array indexes in all the loops are not an affine function of the loop indexes, so the dependence analysis returns distance vectors with unknown elements (*) which inhibits loop interchange.

Loop unrolling

Loop unrolling can be applied for all the loops, especially for the innermost ones. The efficiency (and not the safety) of this optimization technique is conditioned by two factors :

1. the trip count of the loop must be sufficiently large ;
2. the performance of the loop body is mainly constrained by data dependencies instead of resources (processor functional units) availability.

Our dynamic analysis of the flu3m application reveals that the trip counts are very small (~10). These parameters depend on the input data set. The data set that we experimented is representative but is not (in our point of view) as large as the real ones. We think that in the case of larger data set, trip counts become more significant. Even if that was the fact, all the loop bodies are sufficiently large and mainly constrained by resources availability. The inner-iterations dependencies can be sufficiently covered statically by independent operations. However, dynamic events such that cache misses can inverse this fact and hence loop unrolling would produce good performances.

Forward substitution

Forward substitution can be considered for two reasons :

1. avoid loading a data by recomputing it;
2. avoid consuming a register (scalar) during long period of times.

All the analyzed loops are suitable for this optimization technique.

Software Pipelining

It is more suitable than loop unrolling since it can produce better performance with a compact code size. All the innermost loops are suitable for software pipelining. However, lot of the data dependence distances in the innermost loops are not known statically (*) which requires to assume a null conservative distance. This fact limits the efficiency of SWP : array indexes must be rewritten as an affine function of loop indexes in order to be able to extract the intrinsic fine grain parallelism.

Data Prefetching

This optimization techniques can be applied in all the loops in two ways :

1. inserting the “prefetch” instructions into the loops. The condition is that the processor accept this sort of special instructions ;
2. issuing a dependent load well ahead by considering it as statically a cache miss. This solution consumes more registers, so we encourage to use the first solution since the number of available registers is the most critical source of bottleneck in all the loops (too much scalars).

Advice for end-users on the way of Programming for Optimizing an application

A clear conclusion which can be drawn from this analysis is that some very efficient optimization techniques cannot be automatically applied on the code because of the complex programming style of the end users : compilation techniques, in fact, need clear program structure to be able to analyze them and take the full advantage of the machine abilities. So we would like to conclude this section by the most important advice regarding the way of programming. We must note that all these advice are not obligations, and must be done only when if possible an only in the case of time consuming loops.

1. Array declarations : arrays must have a constant declared size. When the size of the data set is not known *a priori* (which is the case in most of situations), it is more convenient to fix an upper bound and declare the size of the array with this limit. This fixed size must be reported in all the subroutines which access the array.
2. Array accesses : array accesses must be done with loop indexes. Furthermore, the stride (the increment) of accessing the successive array elements must be fixed. Finally, we encourage the fact that the innermost loop access the array in the first dimension, as illustrated in the following loop

```
DO i
   DO j
     DO k
       array(function of k, ....)
```

1. Loop Structures : numerical loops are encouraged to be written in a perfectly nested way. Loop splitting (fission) can be employed for this purpose.
2. Eliminating function calls from loops : when a user function is called inside a loop, inlining this function enable to get a loop free of calls.
3. Eliminating branches from loops : some sort of branches can be eliminated from loops :
   - *loop index dependent conditionals* make the branch true for a certain range of the loop indexes. They can be safely eliminated by splitting the loop depending of these ranges. The following example is extracted from "implvi3.f" of the ONERA application

```
     do  i=1,ia
        ....
        if (i .NE. 1)
        ...
        if (i .NE. ia)
        ...
```

     this loop can be rewritten

```
   iteration 1
   do  i=2,ia - 1
     .... (without the branches)
   END
   iteration ia
```

   - *loop invariant conditionals* make the outcome of the branch always the same, i.e. the Boolean value of the condition is not modified by the loop. Consider :

```
do i =1, n
   if (x .NE. 0) then
     A(i) = A(i) *2
   else
    B(i)=A(i)**2
   endif
enddo
```

the outcome of this condition does not depend on the computation inside the loop, and hence we can rewriten as

```
if (x .NE. 0) then
     do i =1, n
        A(i) = A(i) *2
     enddo
else
    do i =1, n
      B(i)=A(i)**2
    enddo
endif
```

Optimization Process : Stochastic Evaluation of Loop Transformations

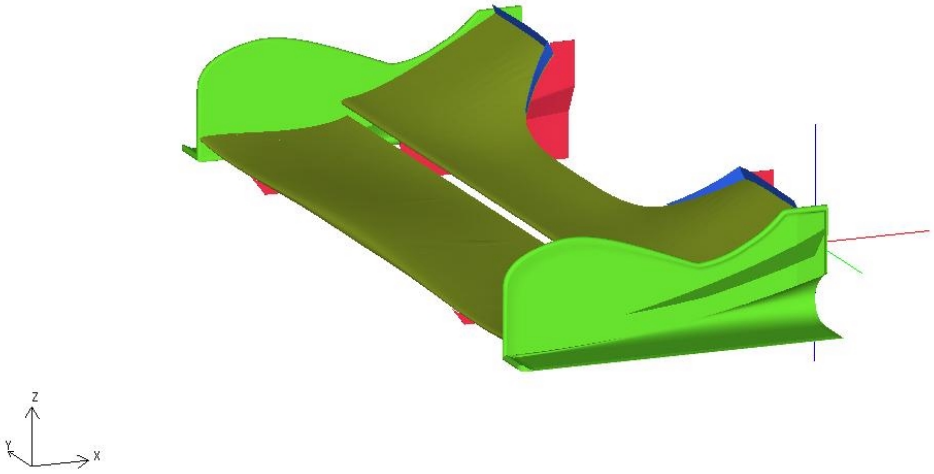
For confidential reasons, the optimization was done on the more recent version of the FLU3M code, it was not possible to use a machine with the MHAOTEU server for this optimization. To fill this gap, the idea is to use the LNO (Loop Nest Optimizer) optimizer of the SGI Origin 2000 which enables to make most of the transformation on loops described just above, and to couple it with the Genetic Algorithm technique demonstrated in the previous part of the M3D3 report.

Application and results

Test case description

This part is devoted to the application of the different optimization processes described above.

To sum up we have divided the full optimizing process into five steps. The test case retained is extracted from a Formulae1 mainplane and flap computation. It consists in a domain of 43 000 cells. The CPU time depends linearly on the number of time iterations. This number is chosen so that CPU time is included in the [60-180] seconds range.



Step 1

The object of this test was to suppress the array references in the Lower-Upper Solver by grouping the construction and the inversion of the implicit phase (which means a little loss of modularity, see the integrated implicit solver part above). The results obtained on an Euler modelization test case are indicated in the following table. They indicate a gain of 71% proving that sometimes some rather minor modifications may lead to very impressive CPU savings.

	CPU Time (s)	L1 Hit Cache Ratio	L2 Hit Cache Ratio	Megaflop/s
Original	246.05	.757	.847	32.12
Optimized	61.80	.883	.944	111.39

Step 2

The objective of this phase is to see what we can grab using the automatic stochastic process. The idea was to see what we could gain with transformation similar to the ones proposed by the MHAOTEU server. For practical reasons it was not possible at the time of the optimization to use the server itself and that is why we decided to use the Loop Nest Optimizer of MIPS which relies on similar ideas. The details or this optimization process can be found in a previous part. The optimization was limited to the most consuming CPU routines and was tested on a Silicon Graphics Workstation equipped of a R12k with 2Mbytes of cache. The LNO options retained by the optimizer are indicated below :

```
flro3n.o: flro3n.f
      f90   -r4 -i4 -mips4 -64 -avoid_gp_overflow -extend_source -O3 -
LNO:fission=0:gather_scatter=2:non_blocking_loads=ON:interchange=ON:pr
efectch=0:prefetch_ahead=4:ou_max=1 -c flro3n.f
gradr3.o: gradr3.f
      f90   -r4 -i4 -mips4 -64 -avoid_gp_overflow -extend_source -O3 -
LNO:fission=0:gather_scatter=2:non_blocking_loads=ON:interchange=ON:pr
efetch=0:prefetch_ahead=4:ou_max=1 -c gradr3.f
invlus3fm1.o: invlus3fm1.f
      f90   -r4 -i4 -mips4 -64 -avoid_gp_overflow -extend_source -O2 -
LNO:fission=1:gather_scatter=2:non_blocking_loads=OFF:interchange=ON:p
refetch=2:prefetch_ahead=1:ou_max=1 -c invlus3fm1.f
invlui3fm1.o: invlui3fm1.f
      f90   -r4 -i4 -mips4 -64 -avoid_gp_overflow -extend_source -O2 -
LNO:fission=1:gather_scatter=2:non_blocking_loads=OFF:interchange=ON:p
refetch=2:prefetch_ahead=1:ou_max=1 -c invlui3fm1.f
flns3c.o: flns3c.f
      f90   -r4 -i4 -mips4 -64 -avoid_gp_overflow -extend_source -O3 -
LNO:fission=0:gather_scatter=2:non_blocking_loads=ON:interchange=ON:pr
efetch=0:prefetch_ahead=4:ou_max=1 -c flns3c.f
slpmi3.o: slpmi3.f
      f90   -r4 -i4 -mips4 -64 -avoid_gp_overflow -extend_source -O3 -
LNO:fission=1:gather_scatter=2:non_blocking_loads=ON:interchange=OFF:p
refetch=2:prefetch_ahead=4:ou_max=10 -c slpmi3.f
```

The table underneath gives the results obtained using the standard O2 optimization of the compiler as well as the standard O3 on the main routines (compiling all routines with O3 does not work). One can see that there is a gain of 8% with the O2 and 2.3 % with the O3.

	CPU Time (s)	L1 Hit Cache Ratio	L2 Hit Cache Ratio	Megaflop/s
Original-O2	159.07	.9032	.6398	68.58
Original-O3	146.32	.8022	.7076	80.21
Optimized	142.88	.8921	.707	81.88

Step 3

In that step, we decided to see the effect of different optimizations discussed above :

- Suppressing unnecessary (vectorized) floating point operations
- Suppressing unnecessary array references
- Suppressing the vectorized indirection in the LU solver

It results in a 35% gain obtained with the modification of about 20 routines.

	CPU Time (s)	L1 Hit Cache Ratio	L2 Hit Cache Ratio	Megaflop/s
Original	123.52	.8905	.9156	121.24
Optimized	79.23	.9349	.9220	137.47

Step 4

The next optimization consisted in applying the Global Index Reordering technique on working arrays. It results in a 4% CPU gain as can be seen in the table

	CPU Time (s)	L1 Hit Cache Ratio	L2 Hit Cache Ratio	Megaflop/s
Original	79.23	.9349	.9220	137.47
Optimized	76.03	.8847	.9245	142.07

In order to have an idea of the number of transformations required, you will find hereafter a shell script containing on each line the name of the routine followed, any time necessary, by the name of the array and the two indexes to switch.

```
init_rotmp.f coe 1 2
precoe.f  coe 1 2
precoefm.f coe 1 2
afvmut.f coe 1 2
tst13.f coe 1 2
tstb3.f coe 1 2
```

```
tstb3c.f coe 1 2
cpcfl3.f coe 1 2
grad3c.f dvardc 1 3 coe 1 2 ti 1 2 tj 1 2 tk 1 2
grad3n.f dvardc 1 3 coe 1 2 dvnu 1 2 dvronu 1 2 ti 1 2 tj 1 2 tk 1 2
gradr3.f dvardc 1 3 coe 1 2
bgrs3.f dvardc 1 3 coe 1 2 tijk 1 2
gradad3.f dvardc 1 3 coe 1 2
gradis3.f dvardc 1 3 coe 1 2
gradis3m.f dvardc 1 3 coe 1 2
centre_m.f coe 1 2
vitesse_m.f coe 1 2
bgre3.f dvardc 1 3 coe 1 2
gradrn.f dvardc 1 3 coe 1 2 dvnu 1 2
bgrsnu3.f coe 1 2 dvnu 1 2 tijk 1 2
gradnu3.f coe 1 2 dvnu 1 2
bgrnu.f coe 1 2 dvnu 1 2
slpnu3.f coe 1 2 qp 1 2 qm 1 2
slpva3.f coe 1 2 qp 1 2 qm 1 2
slpvl3.f coe 1 2 qp 1 2 qm 1 2
slpmi3.f coe 1 2 qp 1 2 qm 1 2
slpsb3.f coe 1 2 qp 1 2 qm 1 2
slpko3.f coe 1 2 qp 1 2 qm 1 2
slpmu3.f coe 1 2 qp 1 2 qm 1 2
slpka3.f coe 1 2 qp 1 2 qm 1 2
slpvm3.f coe 1 2 qp 1 2 qm 1 2
bvas2k.f coe 1 2 q 1 2 qi 1 2
bvas7k.f coe 1 2 q 1 2 qi 1 2
bvas75k.f coe 1 2 q 1 2 qi 1 2
bvas27k.f coe 1 2 q 1 2 qi 1 2
bvasm1.f coe 1 2 q 1 2 qi 1 2
bvasm2.f coe 1 2 q 1 2 qi 1 2
bvas26k.f coe 1 2 q 1 2 qi 1 2
bvas12ck.f coe 1 2 q 1 2 qi 1 2 tijk 1 2
bvasmm2.f coe 1 2 q 1 2 qi 1 2
fsqrtnec.f coe 1 2 qp 1 2 qm 1 2
flvle3.f drodm 1 2 flu 1 2 coe 1 2 qp 1 2 qm 1 2 tijk 1 2
flroe3n.f drodm 1 2 flu 1 2 coe 1 2 qp 1 2 qm 1 2 tijk 1 2
flroe3.f drodm 1 2 flu 1 2 coe 1 2 qp 1 2 qm 1 2 tijk 1 2
flaus3.f drodm 1 2 flu 1 2 coe 1 2 qp 1 2 qm 1 2 tijk 1 2
flvle3r.f drodm 1 2 flu 1 2 coe 1 2 qp 1 2 qm 1 2 tijk 1 2
flroe3r.f drodm 1 2 flu 1 2 coe 1 2 qp 1 2 qm 1 2 tijk 1 2
flns3c.f drodm 1 2 flu 1 2 dvardc 1 3 coe 1 2 tijk 1 2
flns3n.f drodm 1 2 flu 1 2 coe 1 2 dvnu 1 2 tijk 1 2
mjdro3.f drodm 1 2 flu 1 2 coe 1 2
bflco1k.f drodm 1 2 flu 1 2 coe 1 2 q 1 2 tijk 1 2
bflco2k.f drodm 1 2 flu 1 2 coe 1 2 q 1 2 tijk 1 2
bflco0k.f drodm 1 2 flu 1 2 coe 1 2 q 1 2 tijk 1 2
bflco1kr.f drodm 1 2 flu 1 2 coe 1 2 q 1 2 tijk 1 2
bflio1k.f drodm 1 2 flu 1 2 coe 1 2 q 1 2 tijk 1 2
bfliokr.f drodm 1 2 flu 1 2 coe 1 2 q 1 2 tijk 1 2
bflesu1k.f drodm 1 2 flu 1 2 coe 1 2 q 1 2 tijk 1 2
bflssu1k.f drodm 1 2 flu 1 2 coe 1 2 q 1 2 tijk 1 2
bflco1knu.f drodm 1 2 flu 1 2 coe 1 2 q 1 2 tijk 1 2
bflco2knu.f drodm 1 2 flu 1 2 coe 1 2 q 1 2 tijk 1 2
bflco0knu.f drodm 1 2 flu 1 2 coe 1 2 q 1 2 tijk 1 2
spsource.f drodm 1 2 dvardc 1 3 coe 1 2 dvnu 1 2 dvronu 1 2
spsource2.f drodm 1 2 dvardc 1 3 coe 1 2 dvnu 1 2 dvronu 1 2
cprdu3.f drodm 1 2 coe 1 2
core3a.f drodm 1 2 coe 1 2
core3as1.f drodm 1 2 coe 1 2
core3as2.f drodm 1 2 coe 1 2
cprdu3s1.f drodm 1 2 coe 1 2
mjdrof3.f drodm 1 2 coe 1 2
implck3fm.f coe 1 2 ti 1 2 tj 1 2 tk 1 2
impljt3fm.f coe 1 2 ti 1 2 tj 1 2 tk 1 2
invlui3fm1.f drodm 1 2 coe 1 2 ti 1 2 tj 1 2 tk 1 2
invlud3fm.f drodm 1 2 coe 1 2 ti 1 2 tj 1 2 tk 1 2
invlus3fm1.f drodm 1 2 coe 1 2 ti 1 2 tj 1 2 tk 1 2
invlui3fm2.f drodm 1 2 coe 1 2 ti 1 2 tj 1 2 tk 1 2
invlus3fm2.f drodm 1 2 coe 1 2 ti 1 2 tj 1 2 tk 1 2
invlui3nu.f drodm 1 2 coe 1 2 ti 1 2 tj 1 2 tk 1 2
invlud3nu.f drodm 1 2 coe 1 2 ti 1 2 tj 1 2 tk 1 2
invlus3nu.f drodm 1 2 coe 1 2 ti 1 2 tj 1 2 tk 1 2
```

```
impljt3fmr.f coe 1 2 ti 1 2 tj 1 2 tk 1 2
invlui3fmr1.f drodm 1 2 coe 1 2 ti 1 2 tj 1 2 tk 1 2
invlus3fmr1.f drodm 1 2 coe 1 2 ti 1 2 tj 1 2 tk 1 2
invlui3fmr2.f drodm 1 2 coe 1 2 ti 1 2 tj 1 2 tk 1 2
invlus3fmr2.f drodm 1 2 coe 1 2 ti 1 2 tj 1 2 tk 1 2
mjro3s1.f drodm 1 2 coe 1 2
mjro3.f drodm 1 2 coe 1 2
cydm.f coe 1 2
mjdrodm0.f coe 1 2
tabdtop1.f tabp 1 2
```

Step 5

This step concerns the effect of data-partitioning. The previous monodomain computation is replaced by an eleven domain one globally of the same size. A 15 % gain is obtained. One may remark that we have now a very good L2 hit ratio of .995 and 168 Megaflop/s which mean about 28% of the crest speed.

	CPU Time (s)	L1 Hit Cache Ratio	L2 Hit Cache Ratio	Megaflop/s
Original	76.07	.8847	.9245	142.07
Optimized	64.52	.8929	.9951	168.86

Conclusion

This coordinated action between ONERA and INRIA to optimize the FLU3M CFD solver has led us to the main following conclusions.

- Concerning the overall result the action is successful. The final gains obtained are quite interesting. The overall factor 5 with respect to the original version is more than satisfactory. Even, if we do not consider the first optimization step, the integrated implicit solver, the gain is of a very satisfactory factor 2.
- Among the different optimizations that worked, the ones which concern the suppression of "Vector Optimization Programming Techniques" is quite effective (about 35%).
- The other important and efficient technique is data-partitioning (about 15%). One important remark concerning this technique is the strong potential reuse of different actions concerning parallel computing which practically share the same objective.
- The use of the MHAOTEU transformation techniques has permitted some rather limited gains, 3% with the GIR technique and 8% through the LNO optimizer which implements the same loop transformations and with the help of the recently developed GA technique. The use of the MHAOTEU server itself was not possible for practical reasons (the optimized version of FLU3M is classified). However an analysis of the applicability of these techniques has been done and we have included in this part of the M3D3 report some advice for developer's in order to extent the scope of applicability of loop transformation techniques.

Bibliography

Ph. Guillen, M. Dormieux : Design of a 3D multidomain Euler Code, Int Seminar on Supercomputing, Boston (USA), October 3-5, 1989.

L. Cambier, D. Darracq, M. Gazaix, Ph. Guillen, Ch. Jouet, L. Le Toullec, " Améliorations récentes du code de calcul d'écoulements compressibles FLU3M". Progress and Challenges in CFD, Methods and Algorithms, AGARD-CP-578, April 1996.