Applying Interposition Techniques for Performance Analysis of OpenMP Parallel Applications *

Marc González, Albert Serra, Xavier Martorell, José Oliver
Eduard Ayguadé, Jesús Labarta, Nacho Navarro

Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya
C/ Jordi Girona 1-3, Campus Nord, Mòdul C6, 08034 Barcelona, Spain
E-mail: {marc,alberts,joseo,xavim,eduard,jesus,nacho}@ac.upc.es

Abstract

Tuning parallel applications requires the use of effective tools for detecting performance bottlenecks. Along a parallel program execution, many individual situations of performance degradation may arise. We believe that an exhaustive and time–aware tracing at a fine–grain level is essential to capture this kind of situations.

This paper presents a tracing mechanism based on dynamic code interposition, and compares it with the usual compiler–directed code injection. Dynamic code interposition adds monitoring code at run–time to unmodified binaries and shared libraries, making it suitable for environments in which the compiler or the available tools do not offer instrumentation facilities.

Static injection and dynamic interposition techniques are used to collect detailed traces that feed an analysis tool. Both environments meet the accuracy and performance goals required to profile and analyze parallel applications and runtime libraries.

1. Introduction

Shared–memory multiprocessors are becoming more and more affordable and commonplace, encouraging the development of parallel applications that can benefit from this kind of architectures. In order to use these architectures efficiently, programmers and developers of parallel execution environments require accurate information about the behaviour of parallel applications, as well as about the impact on performance due to the parallelizing environment and the underlying hardware platform.

The use of appropriate performance analysis tools can reveal the sources of performance degradation, such as excessive fork/join overhead, synchronization inefficiencies, load unbalancing and poor memory hierarchy behaviour at the application level.

Many performance monitoring approaches are based on static code instrumentation, either at the source level [16] or at the binary level [9]. Our proposal is to add the monitoring code dynamically, at execution time, allowing the use of the same executable and the same libraries both for production executions as well as during performance monitoring sessions.

In this paper, we apply dynamic code interposition to the performance monitoring problem and we compare it with the more usual compiler–directed code injection. Both techniques are used to collect detailed traces that feed our visualization and analysis tool. Our target are OpenMP applications running on SGI Origin2000 systems. As we will show, interposition techniques provide detailed and accurate information, and introduce an overhead comparable to the traditional approach.

The rest of the document is structured as follows: Section 2 describes the methodology used in the development of the parallel performance analysis tools. In Section 3 we evaluate the instrumentation and, finally, Section 4 concludes the paper.

2. Methodology

2.1. Execution traces

We use traces from real executions to analyze the performance of parallel applications. These traces reflect the activity in an OpenMP application through a set of predefined states and events. Instead of providing a summary of the whole application behaviour at different levels (loops, function calls, ...), traces collect the occurence of state changes and events along the application lifetime.

Application states. The analysis of parallel applications is done at thread level. Each thread evolves through a set of states that are representative of the parallel execu-

*This work has been supported by the European Community under the ESPRIT project E21907 (NANOS) and the Ministry of Education of Spain (CICYT) under contract TIC98-0511, and by the Comissionat per a Universitats i Recerca de la Generalitat de Catalunya under the grant FI96-3088

tion. We consider four states: running, idle, scheduling and blocked/synchronizing. Running means that the thread is running code that belongs to the original application; idle means that the thread is searching for work; scheduling means that the thread is executing runtime scheduling code to supply work for other virtual processors that are taking part in the execution; and blocked/synchronizing means that the thread is executing code to synchronize different virtual processors taking part in the execution.

Performance–related events. For each event being traced, the monitoring code obtains the thread identifier, and the time at which the event happened. The time, as well as any relevant performance–related information (cache misses, TLB misses, ...) is acquired using platform–specific mechanisms. In the SGI architecture, we use the SGI memory mapped high resolution clock [4] in order to obtain timestamps that are consistent across virtual processors with low overhead. The performance–related information is obtained by reading the counters included in R10000 processors.

Run–time issues. Each virtual processor registers all the aforementioned information in a buffer. The data structures used by the tracing environment are arranged at initialization time in order to prevent interferences among virtual processors (basically, to prevent false sharing). Therefore, there is no need for locks, synchronization or mutual exclusion in the monitoring code.

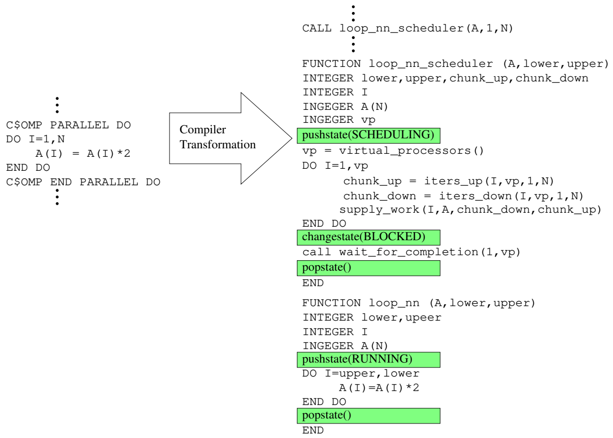

Figure 1. Example of code injection

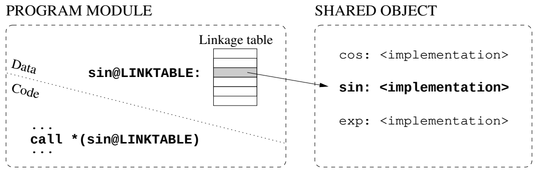

Figure 2. Linkage table

2.2. Code injection approach

When the application source code is available, either a pre–processor or the compiler can be used to statically inject calls to a tracing library. Code injection is the usual mechanism used to instrument applications, and we use it for comparison purposes.

We have developed a parallel tracing library. This library offers routines that can be called from the application to record specific events in the application (entry/exit to/from parallel, work sharing, or synchronization constructs). It also provides routines to record state changes of the calling thread. These calls are inserted by the compiler at specific points in the source code, where the state transitions occur. Figure 1 presents an example of the transformation of a parallel loop into instrumented parallel code.

Code injection has been implemented in the backend of the NANOSCOMPILER [1] (an extension of PARAFRASE–2 [12]). The NANOSCOMPILER uses an internal structure (the Hierarchical Task Graph, HTG) on top of which code transformations take place. This HTG is modified along the compilation process by many stages. The two most important stages from our point of view are the parallelization stage and the instrumentation stage. The parallelization stage transforms the HTG to express parallelism found either by OPENMP directives or by code analysis. This stage keeps enough information associated to each parallelized block to allow the injection of instrumentation code. The instrumentation of the resulting HTG reflects the different virtual processor states that will occur during the application execution. Once these transformations have been done, the compiler generates an instrumented version of the parallel code. The tracing facility includes source code information in the output trace to establish a quick correspondence between the trace and the source program.

2.3. Code interposition approach

When the user is not able to instrument the application using code injection (e.g. because of the compiler does not support it, or because the source code is not available), we propose to dynamically interpose the instrumentation code at run time, using DITOOLS [13] (Dynamic Interposition Tools). DITOOLS offers an environment in which dynamically–linked executables can be extended at runtime with unforeseen functionality (for instance, argument snooping, I/O tracing, alternative service implementations, or performance monitoring).

Dynamic linking is a feature available in many modern operating systems. Program generation tools (compilers and linkers) support dynamic linking via the generation of *linkage tables*. As shown in Figure 2, linkage tables are redirection tables that allow delaying symbol resolution to run time. In this figure, the program modules reaches the implementation of sin through a pointer stored within the linkage table. At program loading time, a system component (the *dynamic linker*) fixes each pointer to the right lo-

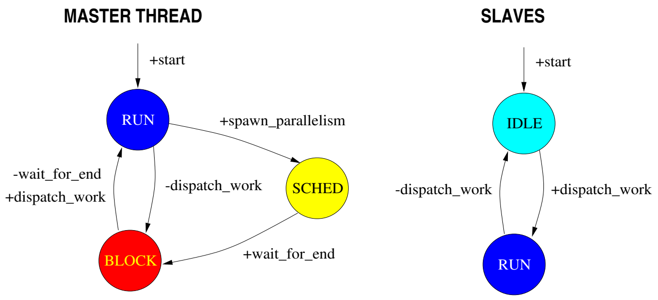

Figure 3. Example of a state transition graph

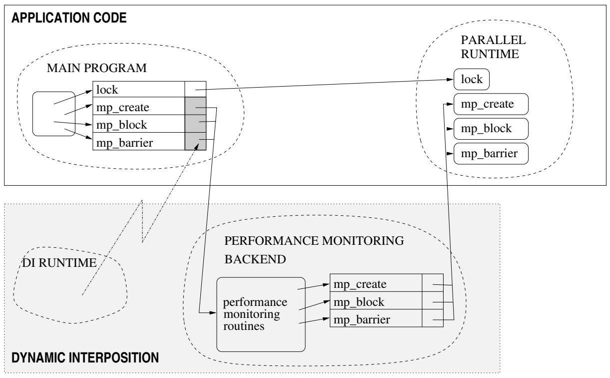

Figure 4. Dynamic Interposition scenario

cation using some predefined resolution policy. Usually, the object file format as well as these data structures are defined by the system Application Binary Interface (ABI). The standardization of the ABI makes possible to take generic approaches to dynamic interposition.

Previous approaches to dynamic interposition make use of system call redirection facilities [6], use binary patching [8], or rely on the symbol resolution policy [5] to insert new code in the execution path. In our approach, interposition is accomplished explicitly, through the declaration of which modules should be added to the image as well as which bindings between references and definitions should be changed. Changes are done directly by changing entries in the linkage tables. Our infrastructure works in multiprocessor/multithreaded platforms.

Dynamic Interposition and application states. The monitoring methodology is based on the fact that, typically, the application will invoke runtime services at key places in order to spawn parallelism, to distribute the work among virtual processors, or to synchronize at the end of a parallel region. We have observed that the presence of procedure boundaries required by the interposition approach is guaranteed by parallelizing compilers themselves; in addition to the insertion of calls to the runtime libraries, parallelizing compilers encapsulate program sections to be executed in parallel within procedures that are created at compile time.

The required knowledge about the execution environment can be expressed using a state transition graph, in which each transition is triggered by a procedure call and/or a procedure return. Figure 3 presents an example of such a state transition graph (very close to the one that represents the SGI-MP runtime library), in which nodes represent states, and arcs correspond to procedure calls (indicated by a + sign) or procedure returns (indicated by a – sign) causing a state transition. This transition graph is then used to derive the interposition routines used to keep track of the state in the performance monitoring backend. These routines are simple wrappers of state–changing functions, like the following one:

```
void dispatch_work_slave_wrapper()
{
  RECORD_TRANSITION(timestamp,IDLE,RUN);
  dispatch_work_slave();
  RECORD_TRANSITION(timestamp,RUN,IDLE);
}
```

Dynamic Interposition scenario. DITOOLS consists of four main pieces: the *DI Runtime* (interposition services required by any configuration), a *DI Backend* (configuration–specific routines to be used to extend applications), a *DI Configuration* (information to assist the DI runtime for a particular use of the tool), and the *DI Shell* (the utility to start a DI session). Whenever a program is launched within a performance monitoring session, the DI Runtime gains control and loads the backend. Then, the DI Runtime interposes monitoring routines, according to the goals of that backend.

Figure 4 shows an scenario in which a program has been extended using the performance monitoring backend. Four objects are shown: the main program, the parallel runtime, the DI Runtime and the performance monitoring backend. The upper box encloses objects that are normally loaded when this application is launched. The lower box encloses objects that have been added because the program has been launched during a dynamic interposition session. Two linkage tables are also shown: one for the main object, and the other for the performance monitoring backend.

The DI Runtime, as the first object receiving control, acts on the linkage table of the main program in order to redirect references to relevant procedures to performance monitoring routines in the backend, which, in turn, may invoke the original definitions. While there are dynamic instrumentation approaches that do runtime code patching for this purpose [7], DITOOLS installs backend routines simply by changing an entry in a linkage table. It is interesting to note that DITOOLS works entirely at user level, without kernel support and without administrator privileges, like the code injection approach.

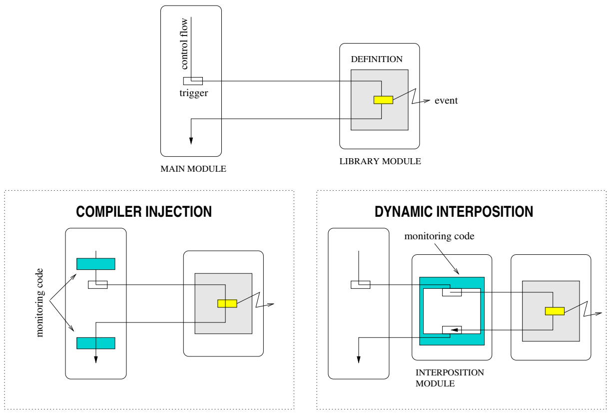

Figure 5. Placement of instrumentation code

2.4. Comparison of instrumentation techniques

The properties of these instrumentation techniques are closely related to the point in which they instrument the application. Static code injection is performed at compile time, acting over the program source code, which is more expressive than executable files. Therefore, injected code can exploit the availability of high–level information to enrich the trace (e.g. to associate source line numbers to profiling events). On the other hand, dynamic interposition works on dynamically–linked binaries, where many high–level symbols and constructs have been processed and potentially transformed by previous stages. Only in special cases, interposition will be able to exploit the presence of high–level information in the executable file itself (e.g. when debug symbols exist).

Figure 5 illustrates the placement of instrumentation code in each approach. The nature of the interposition mechanism requires the presence of a dynamically–linked reference close to the point in which the event is generated, in order to be able to capture this event. On the other hand, the injection mechanism can add monitoring code arbitrarily close to this point, allowing a more fine–grained monitoring. For this reason, injection has more resolution and can monitor a wider set of events than interposition, including events triggered by individual statements.

Finally, interposition avoids the need of compiler support, making unnecessary to rebuild the executable file. Instrumentation code inserted by means of DITOOLS can be efficiently activated, deactivated, and even modified at runtime, because of changing indirections is inexpensive. In this sense, interposition is more flexible than injection.

3. Evaluation

3.1. Hardware and runtime environments

Our experimental environment is based on a 64–processor Origin2000 [14] system, from Silicon Graphics Inc. This system consists of dual–processor nodes as building blocks. Each node contains 2 MIPS R10000 processors, each one with 32 Kbytes of split primary instruction and data cache and up to 4 Mbytes of unified second level cache per processor. The memory in each node can be up to 4 Gbytes of DRAM. The Origin2000 runs IRIX, a highly scalable 64–bit OS with support for multithreading, distributed shared memory and multidisciplinary schedulers for batch, interactive, real–time and parallel processing.

The execution environments that are considered in this study are NANOS [10] and SGI-MP [15]. NANOS is a parallelizing environment composed by an OPENMP–compliant parallelizing compiler (NANOSCOMPILER) and a runtime library (NTHLIB) implementation of the Nano–Threads Programming Model. SGI-MP is the implementation of the OPENMP standard by SGI, composed of two main elements: the MIPSpro F77 compiler and the MP runtime library.

3.2. Analysis tool: PARAVER

PARAVER [11] is a tool to visualize and analyze the performance of parallel programs, which have been instrumented to generate a trace file. PARAVER provides many functionalities to see and analyze quantitatively the trace files. Moreover, the user can add his own analysis functions to extend the functionality of the tool.

3.3. Impact of instrumentation

To evaluate the impact of instrumentation we use three experiments: PRODUCTION, in which we run the unmodified application; STATE, in which the application has been instrumented to monitor thread state changes; and COUNTERS, in which the application has been instrumented to sample performance counters in addition to the state monitoring.

We have evaluated the overhead introduced by our performance monitoring infrastructures to selected individual OPENMP directives. Results are shown in Figure 6. These measures have been obtained using the OPENMP MicroBenchmark Suite [3]. The upper subfigure corresponds to NANOS and the lower subfigure to SGI-MP. Each bar represents the execution time of a synthetic loop, evaluating the OPENMP directive indicated in the associated label.

Next, we have evaluated the effect of instrumentation on an entire application in the two runtime environments. In Figure 7, we show the execution time of the NAS MG benchmark, using from 1 to 32 processors. As can be observed, interposition–based instrumentation performs comparably to the injection approach. The overhead measured in the STATE experiment is always less than 5% of the execution time. The COUNTERS experiment for the interposition approach scales better because the implementation samples counters less frequently. For example, to instrument a PARALLEL directive, injection introduces 12 counter samples (12 system calls) for the master thread, while interposition introduces only two samples.

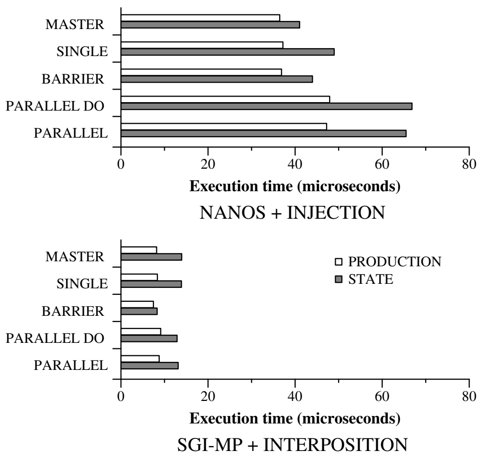

Figure 6. Instrumented OPENMP directives

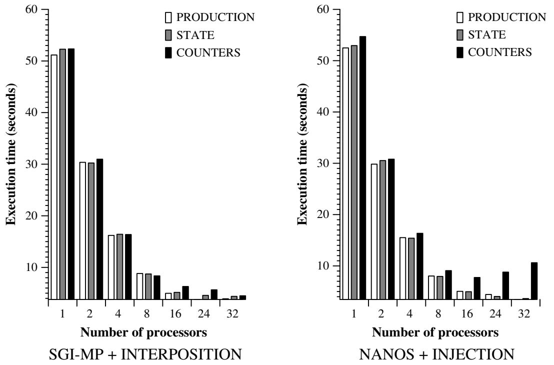

Figure 7. Impact on execution time

Finally, we give the overall overhead for some class–A NAS benchmarks [2] in Figure 8. This graph summarizes execution time for multiple runs of the benchmarks with a number of processors ranging from 8 to 32. For each application we show three bars. The leftmost bar accumulates execution time for the unmodified benchmark from 8 to 32 processors, the central bar accumulates execution time for the STATE experiment, and the righmost bar accumulates execution time for the COUNTERS experiment.

Regardless of differences in execution time due to the runtime environment, STATE instrumentation performs comparably in both environments: the average overhead for the STATE experiment is 3% for interposition and 3% for injection.

3.4. Accuracy and detail of traces

To illustrate the accuracy and detail of traces we will focus on the analysis of thread creation. The example being analyzed (Figure 9) corresponds to a parallel region, extracted from the execution of the NAS MG benchmark. The upper subfigure is for NANOS and the lower subfigure for

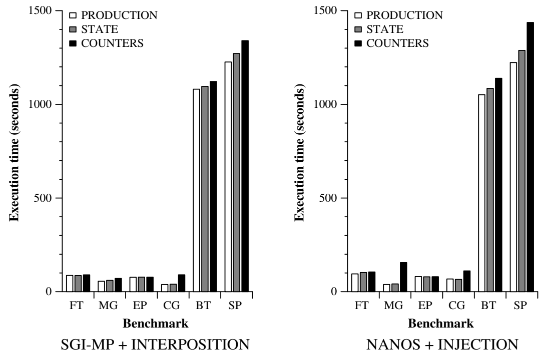

Figure 8. Overall impact of instrumentation

SGI-MP. Each subfigure presents elapsed time in the X–axis (in μseconds) and the processor number in the Y–axis. The colors of the lines displayed for each processor indicate the state of the thread running on that processor. Processor labelled CPU 1 is the master processor of the application.

In the upper subfigure, the master processor begins executing user code while the slave processors are idle, waiting for work. When the master reaches the parallel construct, it schedules the parallelism to be executed. Then, it blocks, waiting for its termination. After blocking, the processor becomes free, and another thread is scheduled in order to get its portion of the parallelism and begin working. Meanwhile, the slave processors detect that there is some work to execute and they also switch to the running state. In the lower subfigure, the master processor takes its portion of the parallel work immediately after the scheduling phase. This is the reason for not appearing the blocking state.

In the NANOS example, the master processor has spent 42 μseconds to schedule the work: 37 μseconds to create the work for the slaves (light portion in the CPU 1 line), and 5 μseconds for blocking and selecting the next thread to run (grey portion after the light portion). Instead, the SGI-MP environment schedules the work in only 8 μseconds. Detecting that kind of situations has been very helpful during the test and tuning of NTHLIB.

4. Summary and conclusions

In this paper, we have described an approach to performance analysis based on dynamic code interposition. This technique enables the extension of programs with new functionality at runtime, keeping the program binaries and libraries unchanged. Dynamic code interposition has been used to analyze the execution of OPENMP parallel applications, and compared against the static code injection approach. We do a trace–based analysis. Traces contain state transitions in a per–thread basis, as well as information from performance counters. The analysis of the traces has been done with PARAVER.

We conclude that dynamic code interposition compares well to the static code injection approach: it offers enough

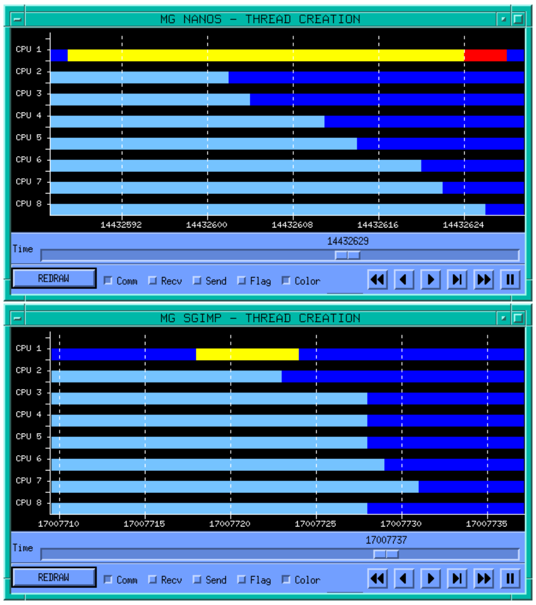

Figure 9. Thread creation (PARALLEL DO)

detail level and similar overhead. Moreover, code interposition has the advantage of not requiring the availability of source code.

Acknowledgments

We thank the insightful comments received from Toni Cortes during the writing. We also thank Jordi Caubet, Oriol Riu and the CEPBA staff for their help during the experimental evaluation. We would also thank the rest of the NANOS project partners for their fruitful discussions along this work.

References

[1] E. Ayguadé, X. Martorell, J. Labarta, M. Gonzàlez and N. Navarro, "Exploiting Parallelism Through Directives on the Nano-Threads Programming Model", Proc. of the 10th. Lang. and Comp. for Parallel Computing, August 1997

[2] D. Bailey, T. Harris, W. Saphir, R. Wijngaart, A. Woo and M. Yarrow, "The NAS Parallel Benchmarks 2.0", TR NAS-95-020, NASA, December 1995

[3] J. M. Bull, "Measuring Synchronisation and Scheduling Overheads in OpenMP", in Proc. of First European Workshop on OpenMP, Sept. 1999, pp. 99-105

[4] D. Cortessi, A. Evans, W. Ferguson and J. Hartman, "Topics in IRIX Programming", Doc. num. 007-2478-006, SGI, http://techpubs.sgi.com, 1998

[5] T. W. Curry, "Profiling and Tracing Dynamic Library Usage Via Interposition", Proc. of the USENIX Summer 1994 Technical Conf., June 1994

[6] M. Jones, "Interposition Agents: Transparently Interposing User Code at the System Interface", Proc. of the 14th ACM Symp. on OS Princ., Dec. 1993

[7] J. Hollingsworth, B. Buck, "DynInstAPI Programmer's Guide", CS Dept., University of Maryland, 1994

[8] G. Hunt, D. Brubacher, "Detours: Binary Interception of Win32 Functions", Microsoft Research, TR -98-33, Feb. 1999

[9] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, T. Newhall, "The Paradyn Parallel Performance Measurement Tools", TR, Dept. of Comp. Science, Univ. of Wisconsin, 1994

[10] NANOS Consortium, "Nano-threads Programming Model Specification", ESPRIT Project No. 21907 (NANOS), Deliverable M1.D1, July 1997. Also available at http://www.ac.upc.es/NANOS

[11] V. Pillet, J. Labarta, T. Cortes, S. Girona. "PARAVER: A Tool to Visualize and Analyze Parallel Code", WoTUG-18, pp 17–31, April 1995

[12] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. P. Leung and D. A. Schouten, "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors", Proc. of the 1989 Int. Conf. on Parallel Processing, 1989

[13] A. Serra, N. Navarro, "Extending the Execution Environment with DITools", TR UPC-DAC-1999-26, 1999

[14] SGI, "Origin200 and Origin2000 Technical Report", 1996

[15] SGI, "Origin2000 and Onyx2 Performance Tuning and Optimization Guide", Doc. num. 007-3430-002, http://techpubs.sgi.com, 1998

[16] A. Voss, "Instrumentation and Measurement of Multithreaded Applications", Thesis, Institut fuer Mathematische Maschinen und Datenverarbeitung, Universitaet Erlangen–Nuernberg. January 1997