

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 ex-

cessive 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 occurrence 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.

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 PARAPHRASE-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

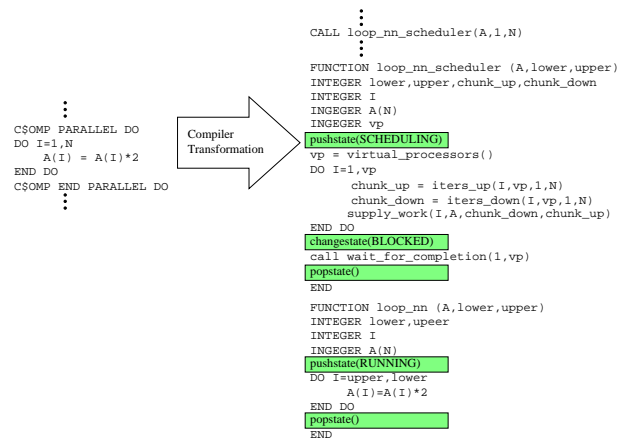


Figure 1. Example of code injection

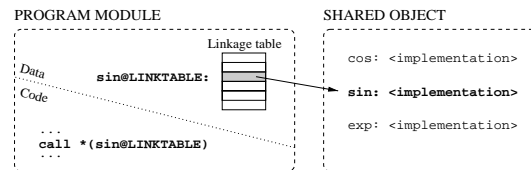


Figure 2. Linkage table

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 DITOOLES [13] (Dynamic Interposition Tools). DITOOLES offers an environment in which dynamically-linked executables can be extended at run-time 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-

