An OpenMP run-time performance analyzer[1]

Van Bui*, Oscar Hernandez*,
Barbara Chapman*,[2], Rick Kufrin†,[3]

* *Department of Computer Science, University of Houston, Houston, Texas*
† *NCSA, University of Illinois at Urbana-Champaign, Urbana, Illinois*

ABSTRACT

Most OpenMP performance analysis tools are usually stand-alone and provide data to the user after an application completes running. Considerable time and effort can be spent learning to use different tools and waiting for results to become available post-execution. It is beneficial to develop a runtime system that exploits the synergistic effects between compilers and performance tools to alleviate this overhead. An added advantage for such a system is that the instrumentation points do not interefere with static compiler optimizations because they reside in the runtime environment. But as with most run-time systems, the problem of runtime overheads must be addressed. In this project, we address these issues and build a prototype implementation of a run-time performance environment that applies a light-weight sampling based technique to extract low level performance metrics and map these metrics to higher levels of abstraction. Experiments are run on our environment using a kernel application and we display preliminary data.

1 Introduction

Performance analysis tools abound which support tuning OpenMP (a parallel programming model) applications on a variety of systems, including software distributed shared memory, large scale SMP, and embedded systems [Oh03]. A central goal of these tools is to significantly reduce the human effort and time involved in the application tuning cycle. Most of these tools provide the user with performance data after an application completes running. Time spent waiting for this performance data can be efficiently utilized by providing the user access to intermediate performance results while an application is running. Performance tools exist that provide the user with performance data during runtime. This functionality is most beneficial to applications that run for long time periods. An added benefit is that since data collection is part of the runtime environment, the process does not interfere with static compiler optimizations. The primary problem encountered by many of these on-line tools is the costly runtime overheads that can result from instrumenting the application while it is running. In order to

[1]This work is supported by the National Science Foundation under grant No. 0444468 and 0444407.
[2]E-mail: {vtbui,oscar,chapman}@cs.uh.edu
[3]E-mail: rkufrin@ncsa.uiuc.edu

mitigate these overheads and interference issues some tools, such as BEA JRockit JVM [Suga05], apply a light weight sampling-based technique to extract performance data during run-time.

We have developed a prototype implementation of a runtime environment supporting performance analysis that is similar to JRockit's dynamic runtime system. Our environment also exploits a low overhead sampling-based technique to extract performance data that are output to the user during run-time. Several differences do exist between our systems nonetheless. JRockit targets Java applications whereas our system targets C/C++ and Fortran applications using OpenMP. JRockit dynamically instruments the application during runtime, while our instrumentation points are part of the OpenMP runtime library. In addition, JRockit profiles methods, locks, garbage collection statistics, optimization decisions and object statistics while we provide a profile of regions of code which use OpenMP constructs. We generally apply different methods towards achieving similar tool functionality and target a different application space.

There are several advantages which our system can provide to the user and compiler. No user instrumentation is required since the instrumentation is internal to the OpenMP run-time library. Overhead for learning to use a new performance tool is eliminated since the tool is integrated into the compiler and they share a common interface. Measurements do not interfere with static compiler optimizations as they are part of the runtime system. There are low runtime overheads since we utilize a light weight sampling-based technique to collect performance data. Tuning time is also reduced for long running applications since the user can observe and begin analyzing performance data once the application starts running.

A disadvantage of this system is that it is very specialized and is only applicable for applications applying OpenMP. However, it is this specialization that separates our approach from previous efforts. In a sense, we are introducing a novel extension to the current functionalities provided by the OpenMP run-time library. We are providing a general framework for retrieving performance data through the runtime system and reframing this data in terms of the respective parallel programming paradigm or language. We contribute a novel environment that integrates a compiler and performance tool to provide OpenMP related performance data to the user at run-time. Such an environment can be very useful for tuning OpenMP applications and can also play a pivotal role within a larger system targeted at tuning a variety of application types.

2 Methods

This system integrates an open source C/C++ and Fortran compiler and hardware counter based performance analysis tool, OpenUH [Liao06] and PerfSuite [Kufr05], respectively. PerfSuite provides library routines that permit a user to tailor performance data collection to their needs. The two different libraries that we exploit from PerfSuite are libperfsuite and libpshwpc. The task is in relating the low level performance data PerfSuite provides to a higher level of abstraction of OpenMP constructs. This mapping is done by understanding the translation process of OpenMP directives within the runtime library. Understanding this translation process informs us on potential locations to insert PerfSuite routines within the runtime library to collect this higher level performance data. The system currently obtains measurements for events occuring within a parallel region, including the following: initialization overheads, interval spent executing assigned tasks by each thread, and parallel region barrier wait time.

To minimize additional overheads in the runtime library, the performance data is extracted in a raw format to an XML file for each thread and processed using a separate parser utility. We implemented a simple C parser utility to process the raw data and convert the data into statistics that are more

intelligible to the user. The wall-clock time in seconds is calculated for each event within a parallel region. The parser utility calculates additional statistics for the event representing the interval of time spent executing the assigned tasks by each thread. Table 1 references some of the statistics that are computed from the raw data.

The OpenMP construct, construct events, and statistics discussed thus far form the foundation for the sort of data we intend to collect. This list will eventually expand as we make further decisions concerning the data that will be most useful to the user during runtime.

3 Experimental Results

To test the functionality of the environment, we performed a sample run using the Additive Schwarz Preconditioned Conjugate Gradient (ASPCG) kernel [Wang99] on four threads. The ASPCG kernel solves a linear system of equations generated by a Laplace equation in Cartesian coordinates. The kernel supports multiple parallel programming models including OpenMP and MPI. The experiments were performed on an SGI Altix system, from NCSA, which consists of two SMP systems running on Linux. Each system has 512 Intel Itanium 2 processors.

The system reports data for 819 parallel regions on ASPCG. This is not reflective of the actual number of parallel regions in the source program. The large number of parallel regions derive from all the different execution contexts of each parallel region. Table 1 provides a sample snapshot of the performance statistics calculated for one execution context of a parallel region from ASPCG. Future implementations will instead provide a snapshot of a summary of performance data for parallel regions as seen from the point of view of the program source file.

Table 1 shows that each thread spends approximately 0.09 seconds executing their respective assigned tasks. Time spent initalizing the parallel region and inside the parallel region barrier seem to be negligible values. Threads 0 and 2 provides some potentially more interesting data. No instructions are issued or completed for an average of 19% of the cycles as Thread 0 is executing its task. Within the same parallel region event, Thread 0 is also stalling on some resource for about 5% of the cycles. Thread 2 exhibits a 0.80 level 3 cache miss ratio, which is a seemingly high ratio value while executing its task. But this is not to indicate that stalls or cache misses are not occurring for the other threads. In some cases, event counts are not available due to multiplexing issues and so the statistic cannot be computed. We are exploring ideas to decrease the number of PerfSuite calls to address this problem. These results should give the reader an idea and visual description of the type of data our system can provide at runtime.

4 Conclusions

A prototype of an OpenMP run-time performance analysis environment was implemented by integrating a performance analysis tool (PerfSuite) and a compiler (OpenUH). PerfSuite provides a useful and flexible set of library routines to obtain performance data using low-cost means, namely via hardware counter sampling. The functionality of OpenUH's OpenMP run-time library is extended to access performance data while the application is executing. Experimental results obtained from running a kernel application are partially displayed to provide the reader with a visual description of the functionality of the system.

Future development efforts will focus on mapping the data back to source so that users can target their tuning efforts toward particularly troubling code regions. In addition, we must explore methods that will ensure that the system properly scales in displaying results when an application is running

Snapshot of Statistics for a Parallel Region					
Parallel region events	Statistic	Thread 0	Thread 1	Thread 2	Thread 3
Init. overhead	Wall clock time (secs)	0	NA	NA	NA
Executing tasks	% cycles w/o instruct. issued	18.7	NV	NV	NV
	% cycles w/o instruct. completed	19.3	NV	NV	NV
	% cycles stalled on any resource	5.61	NV	NV	NV
	L1 cache miss ratio (data)	NV	NV	NV	0.11
	L2 cache miss ratio (data)	NV	NV	NV	0
	L3 cache miss ratio (data)	NV	NV	0.8	NV
	Wall clock time (secs)	0.09	0.09	0.09	0.09
Barrier wait	Wall clock time (secs)	0	NA	NA	NA

Table 1: A snapshot of calculated performance statistics for each thread for a parallel region in ASPCG. Note that only thread 0 contains data for the event associated with initialization overheads and barrier wait time as is appropriate. NV indicates no value can be calculated because insufficient counter data was retrieved, possibly due to multiplexing issues.

and exploiting the resources on large platforms. Plans are also underway to feed this performance data back to the compiler to perform optimizations in subsequent program runs.

References

[Kufr05] R. KUFRIN. PerfSuite: An Accessible, Open Source Performance Analysis Environment for Linux. In *6th International Conference on Linux Clusters: The HPC Revolution 2005*, April 2005.

[Liao06] C. LIAO, O. HERNANDEZ, B. CHAPMAN, W. CHEN, AND W. ZHENG. OpenUH: An Optimizing, Portable OpenMP Compiler. In *12th Workshop on Compilers for Parallel Computers*, January 2006.

[Oh03] J. OH, S. KIM, AND C. KIM. OpenMP and Compilation Issue in Embedded Applications. In *WOMPAT*, pages 109–121, 2003.

[Suga05] T. SUGANUMA, T. YASUE, M. KAWAHITO, H. KOMATSU, AND T. NAKATANI. Design and evaluation of dynamic optimizations for a Java just-in-time compiler. *ACM Trans. Program. Lang. Syst.*, 27(4):732–785, 2005.

[Wang99] G. WANG AND D. TAFTI. Performance Enhancement on Microprocessors with Hierarchical Memory Systems for Solving Large Sparse Linear Systems. *Int. J. High Perform. Comput. Appl.*, 13(1):63–79, 1999.