Understanding Performance of Multi-Core Systems using Trace-based Visualization

Peter F. Sweeney
IBM TJ Watson Research
pfs@us.ibm.com

Matthias Hauswirth
U. of Lugano, Switzerland
Matthias.Hauswirth@unisi.ch

Amer Diwan
U. of Colorado at Boulder
diwan@cs.colorado.edu

Marina Biberstein
IBM Haifa Lab, Isreal
biberstein@il.ibm.com

Yuval Harel
IBM Haifa Lab, Isreal
harely@il.ibm.com

ABSTRACT

Hardware designers are adding additional cores to chips in an attempt to improve throughput performance. These multi-core systems share hardware resources at multiple levels of the system. The question is when multiple applications run on the same chip, how do they interact with the shared resources, and how do these interactions effect performance? Traditional profile based approaches that aggregate statistics across the complete execution of an application are inadequate to illuminate these interactions, because temporal causality is lost in the aggregation of the data.

This paper argues that analyzing multi-core systems requires trace-based statistics and flexible visualization techniques to analyze the trace data.

1. INTRODUCTION

Due to power and thermal constraints, hardware designers have had to look for performance in innovative new ways other than increasing clock speeds. The current trend is to increase parallelism at the task level by placing multiple cores on a chip and by adding hardware support for multiple hardware threads on a core. Although these new hardware features exploit the increase in silicon that is available on a chip, it is not always clear how software can exploit the trend's promise for increased performance.

Specifically, these new hardware features share resources at multiple levels of the system. For example, the Intel Pentium Processor Extreme microprocessor has dual cores on a chip that share a front side bus. In addition to sharing a memory bus, the PowerPC POWER5 microprocessor has dual core's on a chip that share the on chip L2 cache and can access off chip L3 caches of other cores. Both microprocessors provide multiple hardware threads per core where each thread shares the core's caches, pipeline, issue queues, and functional units.[1] The question is when multiple applications run on the same chip, how do they interact with the shared resources, and how do these interactions effect performance?

Summary statistics, which are collected for the complete run of an application, are not sufficient to identify the effect of task parallelism on the shared resources, because temporal causality is lost in the aggregation of data. To capture temporal causality, trace-based statistics need to be collected, where the statistics are periodically collected throughout the execution of an application However once collected, trace-based statistics require visualization techniques to identify the interactions between processors and shared resources. Once the interactions have been characterized, mechanisms are required to tied the characterization back to the source code.

We have successfully gathered trace-based statistics across multiple layers of the execution stack (application, runtime environment, OS and hardware) and used this data to solve performance anomalies in Java applications [4], and to understand the impact of mapping data structures to large pages in scientific applications [1]. In both cases, visualization of the trace-based statistics over time was essential to understand what was going on. We believe that this approach will work well to understand how shared resources are affected by logical processors (multiple cores on a chip and multiple hardware threads on a core) that share these resources.

2. EXAMPLE

Although we currently don't have traces from either a chip with multiple cores or a core that runs multiple hardware threads, we do have traces from a shared memory machine (SMM) where memory is shared between processors and the L3 cache of one processor can be be remotely accessed by another. We use this trace to illustrate processor interactions with shared resources.

2.1 Experimental Methodology

Our traces were collected on a 4-way PowerPC POWER4 1.2 GHz shared memory machine (SMM) running AIX 5.1. In this machine, each chip has a single processor. Each processor has an on-chip L2 cache, its own off-chip L3 cache, and a processor can remotely access the L3 caches of other cores.

Our traces were generated from a modified version of Jikes

[1]Intel's hyper threading (HT) technology and IBM's simultaneous multi-threading (SMT) both support multiple hardware threads per core.

This paper appeared and was presented at the First Workshop on Software Tools for Multi-Core Systems (STMCS06), March 26, 2006, Manhattan, NY.

RVM [5], an open source research virtual machine. Jikes RVM does its own Java thread scheduling by multiplexing Java threads onto underlying operating system threads. When Jikes RVM runs on a SMM, it creates an OS thread for each processor. At each thread switch, the modified version of Jikes RVM writes out to a trace file the statistics for the Java Thread that was just running. Jikes RVM uses a 10 milliseconds scheduling time quantumn.[2] For more details the reader is referred to [6]. In previous work, we found that Java thread-specific, trace-based statistics were important, because different Java threads have significantly different behavior [4].

We traced the execution of a multithreaded Java application, *pseudojbb*, a modified version of the SPECjbb2000 [2] (Java Business Benchmark) that evaluates the performance of server-side Java by modeling a three-tier application server. For benchmarking purposes, *pseudojbb* was modified to run for a fixed number of transactions (120,000 for this experiment) instead of running for a fixed amount of time. We configured *pseudojbb* to create two application threads and then executed this configuration on two processors of the SMM.

2.2 Visualization

In this example, we focus on the L3 cache behavior of the two application threads running on two processors. Figure 1 is a screen shot of the Performance Explorer, a visualization tool, In the upper window, the horizontal-axis is time and the vertical dimension is broken up into three strips that are labeled "Application Thread 1", "Application Thread 2", and "remote L3 accesses". Each strip contains one or more layers, and a layer plots a metric as a signal of line segments over time. The length of the line segment is its duration and its height is the metric's value for that time interval. All threads, other than the two application threads, have been filtered out of the plots. In addition, we have zoomed into a small time period of the psuedojbb's execution.

The first strip of in the upper window of Figure 1, labeled "Application Thread 1", plots two metrics as signals: the gray signal is the number of remote L3 cache accesses divided by cycles, and the solid signal is the number of local L3 cache accesses divided by cycles. We normalized a metric's signal by cycles to ensure that any change in magnitude is due to an intrinsic change in a metric's value and not due to an increase in the length of the time interval.

The second strip, labeled "Application Thread 2", plots the same two signals that were plotted in the first strip, but for the second application thread. The third strip, labeled "remote L3 accesses", plots the remote L3 cache accesses divided by cycles for the two application threads. The gray signal is the first application thread's signal and the solid signal is the second application thread's signal.

In all three strips, the y-axis's lower and upper bounds for a particular metric has been manually adjusted to be the same across the strips, although different metrics may have different upper bounds. Because the x- and y-axis dimensions are consistent across strips, it allows us to visually make some observations. First of all, in the first two strips, the first application thread has more remote L3 cache accesses and fewer local L3 cache accesses than the second application thread in the second strip. The difference can be seen as the distance between the two signals in each strip. This difference implies that there is an imbalance in the way that the application threads accesses the L3 caches. Second, the local and remote L3 cache accesses for a particular thread seem to correlate reasonable well visually; that is, when the line segment in one signal goes up the corresponding line segment in the other signal also goes up, and vice versa. Finally, in the third strip, which plots the remote L3 cache accesses for both application threads, the signals do not seem to correlate visually; that is, when one line segment for one signal goes up the corresponding line segment for the other signal may or may not go up.

[2]A Java thread may be run for less then 10 milliseconds if it yields control, or if it obtains control from another thread that has yielded control, and thus does not get the full 10 milliseconds.

3. CONSTRUCTING VIEWS

From this simple example that visualizes the local and remote L3 cache accesses over time for two application threads running on two processors, we were able to draw some conclusions, because of the signals were juxtaposed either in the same or separate strips. This juxataposing is easily achieved with PE's graphical user interface (GUI) for constructing strips and layers. The lower window, which is labeled "Strip Array Properties" in Figure 1, is the GUI for manipulating the layers in a strip. The layer that is defined in this window is the local L3 cache accesses per cycles signal for the application thread that is illustrated in the first strip. Each layer has a trace attribute that determines which trace the data that is to be plotted comes from; a record type ("record list") that determines which record type in the trace to find the data; and a value attribute that determines which fields in the trace are used to plot the signal. The value attribute determines the y-axis height of the signal at a given time interval. By changing the value attribute, one can quickly visualize a signal for different metric.

A filter attribute allows the data from a subset of the records to be plotted by filtering the value of a field. In this example, the thread id (tid) is selected to be "15", which happens to be one of the application's threads. The start and end time attributes identify the fields in the record that define the start and end of the line segment that is plotted. Finally, the color attribute determines the color of the signal.

The plus (+) and minus (-) buttons in the lower left hand corner of the lower window allow new layers to be added and old layers to be deleted. Final, each layer has a background layer that defines the color of background and an axis layer that defines whether and where vertical lines should be draw, if at all, and their color. In this example, all vertical lines have been eliminated from the strips.

In this example, each strip has two layers, however, the user can add as many layers as they would like. For example, we have constructed strips for MPI applications that have dozens of layers, one layer for each type of MPI function.

4. DISCUSSION

We hope that the reader sees the pontential for using trace-based visualization to understand the interaction between multiple cores and multiple hardware threads on shared resources. The traces that we use in this paper came from a PowerPC POWER4 derivative that does not support hardware threads and has only one processor per chip. In such a machine, resource sharing is limited. Nevertheless, we were

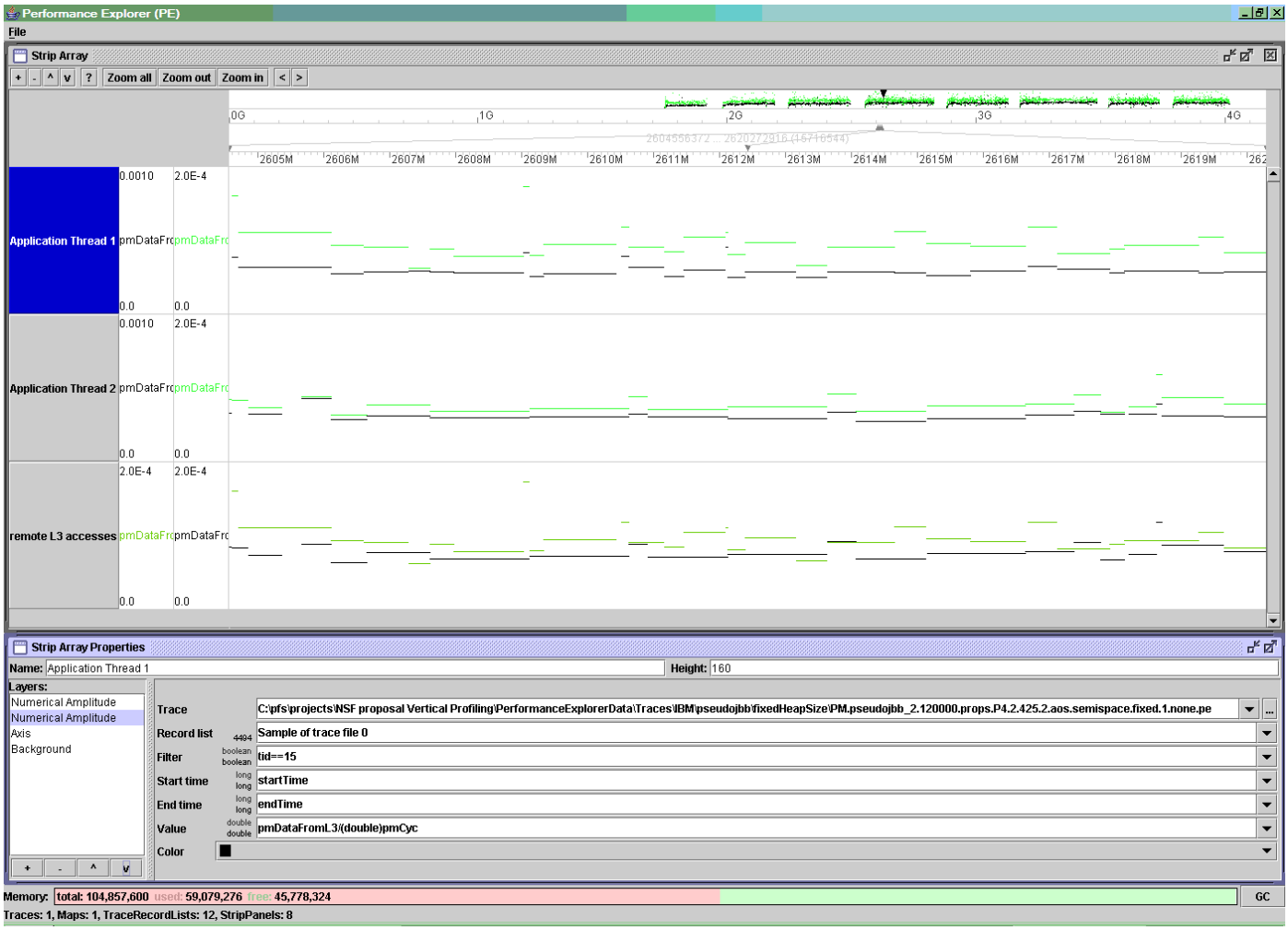

Figure 1: PE screen dump for pseudojbb with two application threads run on two processors of a shared memory machine.

able to illustrate how local and remote L3 cache accesses differ between multiple application threads that are running concurrently. The flexibility to construct views that visually illustrate the impact of sharing resource between logical processors is an important capability that is needed to understand performance anomalies that may arise by hardware support for task parallelism.

5. GETTING BACK TO SOURCE CODE

In many cases, tying the behavior visualized in the trace-based statistics back to the source code can help to understand what optimization is most appropriate to apply. For example, phase markers could be added to the source code to generate a trace record whenever a particluar phase is entered or left. The phase markers could be visualized as a event marker layer. Correlating phase markers with cache behavior provides insight into each phase's impact on data locality. Furthermore, the interaction between phases on logical processors that share resources could be used to better affinity schedule applicaitons that share resources. For example, if the two distinct phases in different applications have poor data locality when executed concurrently, then scheduling these phases to run at different times would be a good scheduling decision.

6. HARDWARE PERFORMANCE MONITORS

Traced-based visualization of logical processors assumes hardware performance monitor (HPM) support that can capture hardware events concurrently that are generated by multiple cores on the same chip and by multiple hardware threads on the same core. Without this HPM support, understanding the interactions between logical processors on shared resources is difficult, if not impossible.

7. FUTURE WORK

Performance Explorer is a stand alone tool that is written from Swing components in Java Foundation Classes and is available as open source from the download page of the Jikes RVM home page [5]. We are in the process of porting PE's functionality to eclipse [3], and open source integrated development environment. Integrating PE's functionality into eclipse provides a framework in which PE can interact with other tools. For example, the eclipse framework will provide the ability to navigate between PE's views and source code or other graphical or statistics views.

We are in the process of porting the Jikes RVM tracing infrastructure of IBM's POWER5 machines and Intel's dual core processors. The step is necessary for our future exploration of multi-core and multi-threaded systems.

7.1 Acknowledgements

This work is supported by NSF ITR grant CCR-0085792, NSF Career CCR-0133457, an IBM faculty partnership award, and Defense Advanced Research Project Agency Contract NBCH30390004. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

8. ADDITIONAL AUTHORS

9. REFERENCES

[1] Călin Caşcaval, Evelyn Duesterwald, Peter F. Sweeney, and Robert W. Wisniewski. Multiple page size modeling and optimization. In *Proceedings of the Fourteenth International Conference on Parallel Architectures and Compiler Technology (PACT05)*. IEEE, Sep. 2005.

[2] Standard Performance Evaluation Corporation. SPECjbb2000 (Java Business Benchmark). http://www.spec.org/jbb2000.

[3] eclipse. http://eclipse.org.

[4] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA04)*. ACM Press, October 2004.

[5] Jikes Research Virtual Machine (RVM). http://www.ibm.com/developerworks/oss/jikesrvm.

[6] Peter F. Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove, and Michael Hind. Using hardware performance monitors to understand the behavior of Java applications. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM'04)*, May 2004.