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  
yuval@il.ibm.com

ABSTRACT

Hardware designers are adding additional cores to chips in an attempt to increase 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? 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 track 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 [2], and to understand the impact of mapping data structures to large pages in scientific applications [11]. 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 multiple cores on a chip (or multiple hardware threads on a core).

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 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 tie 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 [2], and to understand the impact of mapping data structures to large pages in scientific applications [11]. 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 multiple cores on a chip (or multiple hardware threads on a core).

2. EXAMPLE

Although we currently don’t have traces from either a chip with multiple cores or a core with multiple hardware threads running, we do have traces from a shared memory machine (SMM). Even though the resource sharing for a SMM is different, we can use its traces 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 core. Each core has an on-chip L2 cache, its own off-chip L3 cache, and a core can remotely access the L3 caches of other cores. Our traces were generated from a modified version of Jikes RVM [8], an open source research virtual machine. Jikes RVM does its own Java thread scheduling by multiplexing them onto underlying operating system threads. When Jikes RVM runs on a SMM, it creates an OS thread for each pro-

---

1Intel’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.
A Java thread may be run for less than 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.

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.

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 juxtaposition 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 ??, 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 drawn, 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.
Figure 1: PE screen dump for pseudojbb with two application threads.
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 systems.

4.1 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.