PAPI-NUMA: Middleware to Support Hardware Sampling

By: Ivonne Lopez  Supervised by: Dr. Shirley Moore

Abstract

PAPI is a widely used portable library for accessing hardware counters on modern microprocessors. PAPI offers both counting and sampling interfaces, but the sampling interface is extremely limited, consisting of a simple interrupt-driven interface that can periodically report processor state. In the past few years, the hardware and operating systems of modern processors have added support for new more advanced sampling features. These features enable information about non-uniform memory access (NUMA) behavior to be obtained. Currently, performance tool developers who want to provide sampling data to their users must make use of a complex low-level kernel interface, sometimes developing their own kernel patch to access the features they need. This paper reports on initial efforts to develop a middleware layer that will serve as a stable interface and enable tool developers to access sampling data through standard PAPI calls and to obtain data important for NUMA analysis.

Introduction

The Performance Application Interface (PAPI) is a widely used portable library for accessing hardware counters on modern microprocessors [1]. PAPI offers both counting and sampling interfaces, but the sampling interface is extremely limited, consisting of a simple interrupt-driven interface that can periodically report processor state. In the past few years, the hardware and operating systems of modern processors have added support for new more advanced sampling features. For memory access events, information such as access latency, operand address, and data source for loads can be returned. Such information is extremely useful for performance tuning on modern multicore systems. In a non-uniform memory access (NUMA) system, last level cache and main memory are distributed across the different sockets on a node. Non-optimal memory layout and access patterns can seriously degrade performance on a NUMA system. Sampling memory accesses that exceed a specified latency threshold and mapping these accesses to the data addresses responsible can help code developers analyze and improve data layout and performance.

Cache Hierarchy

Computer systems are only able to fetch data and instructions from cache memory. This memory has to be loaded from the main memory in the system, the Random Access Memory or RAM for short. Memory is divided into two categories: volatile and nonvolatile memories, with the former requiring the system to be power ON in order to maintain the data storage.

• Cache (MB): Fastest accessible memory of a computer system. It is volatile and expensive.
• Main memory (GB): Main memory claims to be the most used memory with the advantage that it is reasonably fast.
• Secondary storage (TB): Nonvolatile data storage units which are external to the computer system, examples of these are hard drives.
• Tertiary storage (PB): This type of storage is designed with the sole purpose to serve for data backup. Examples of these are tape drives.

Non-Uniform Memory Access

Non-uniform memory access (NUMA) is used in multiprocessing as a computer memory design to determine memory access time depending on the memory location in the processor. The main advantage of processors under NUMA is the capacity to access its own local memory faster than remote memory. NUMA allows multiprocessing systems to share memory locally and improve performance.

Testing

We have tested our PAPI sampling interface on an 8-core Intel Westmere machine, named Dracula, at the University of Oregon that is running Linux 4.0.7 with a patch to enable per-thread counts. Dracula is a NUMA machine with two sockets, each with four cores. To test per-thread counts, we instrumented the OpenMP version of the STREAM memory benchmark code [8] which was instrumented with PAPI by (PAPI初始化函数) that allowed us to set up eight counters for the eight logical CPUs and ran with eight OpenMP threads. We sampled the native event MEM INST RETIRED: LATENCY ABOVE THRESHOLD. We set the sample type to PERF_SAMPLESUMER_IF | PERF_SAMPLE_WEIGHT | PERF_SAMPLE_DATA_SRC. We instrumented and ran versions of STREAM with first touch (memory initialized inside a parallel loop) and without first touch (all memory initialized by the master thread).

Table 1: STREAM Benchmark Results – with First Touch

<table>
<thead>
<tr>
<th>Function</th>
<th>Bandwidth (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy</td>
<td>21995.0</td>
</tr>
<tr>
<td>Stencil</td>
<td>96.0</td>
</tr>
<tr>
<td>Add</td>
<td>24164.0</td>
</tr>
<tr>
<td>Tiled</td>
<td>24280.0</td>
</tr>
</tbody>
</table>

Table 2: STREAM Benchmark Results – with First Touch

<table>
<thead>
<tr>
<th>Function</th>
<th>Bandwidth (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy</td>
<td>11841.3</td>
</tr>
<tr>
<td>Stencil</td>
<td>96.0</td>
</tr>
<tr>
<td>Add</td>
<td>12787.7</td>
</tr>
<tr>
<td>Tiled</td>
<td>13907.3</td>
</tr>
</tbody>
</table>

Results

Our initial implementation of a PAPI sampling interface is a low-level interface intended for performance tool developers. We plan to make our implementation, which we call PAPI-NUMA, available to tool developers in a development branch of PAPI so that they can try it out and we can get feedback on its usefulness and on features they would like to have added or modified. We also plan to design a higher-level interface that will not require the user to provide the signal handler nor parse the mmap buffer.

There is an unavoidable problem that the capabilities needed rely on a recent Linux kernel and on the required patch being part of the kernel. Production machines typically run older kernel versions, and there is unfortunately no way that we can make PAPI-NUMA compatible with older kernel versions.

References