Mnemosyne: A Tool for Temporal Memory Access Analysis in HPC Applications

Shahrukh R. Tarapore and Matthew Burkholder
Lockheed Martin Advanced Technology Laboratories
{shahrukh.r.tarapore, matthew.burkholder}@lmco.com

Introduction

An increasingly common cause of performance degradation in HPC applications is the ineffective access of data as memory technology performance lags behind its processor counterparts [1]. Even today, access to off-chip memories is significantly slower than access to on-chip registers and caches (e.g., register reference 0.25ns, cache 1.0ns, and main memory 100ns). These observed slowdowns are predominantly due to the drastically varying frequencies at which DRAM chips run compared to on-chip storage. In addition, address translation, multiplexing, memory controller overheads, bus arbitration and speed, and DRAM refresh rates have increasing influence on access times.

Phenomena such as data striding, misaligned cache access and false sharing use to be small anomalies with a negligible effect on application performance. However, as the gap has grown and continues to grow between CPU and DRAM performance, these phenomena pose a measurable and significant source of performance degradation.

Mnemosyne Overview

Mnemosyne is a dynamic program analysis tool that is able to collect information about memory access over time, which can be used to directly assist HPC code optimization efforts.

Mnemosyne uses binary instrumentation to collect detailed (bit-level) information about memory accesses during the execution of the program. The tool leverages a novel combination of dynamic binary rewriting and detailed symbolic information to facilitate measurement of any x86-based (e.g., Intel Pentium/Xeon, AMD Athlon/Opteron) compiled binaries. The tool performs real-time "distillation" of the data to avoid unmanageable data bloat. Once analysis is completed, the results are linked to respective lines of source code to aid in maintenance and optimization during the software development cycle (Figure 1).

For a given input executable, the output of the supervised execution tool is a set of temporal control and memory events that can be used to identify performance degrading issues in HPC codes.

To usefully interpret the collected data, Mnemosyne requires a model of the underlying hardware cache architecture. This model defines attributes of the cache, such as cache size, associativity level, and cache line size, and can be extended as new analyzers are built requiring more knowledge of the hardware architecture.

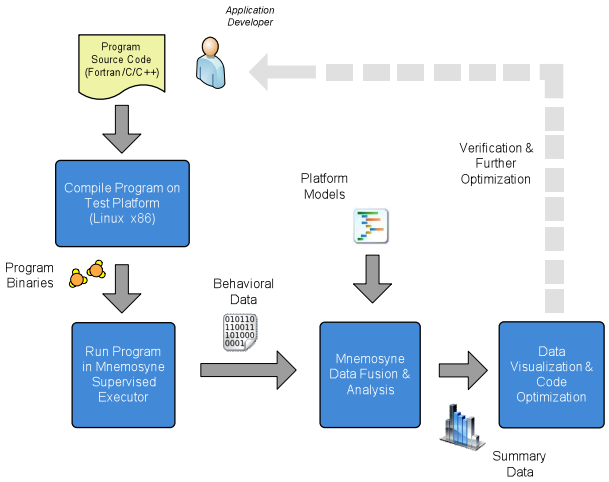

Figure 1. Mnemosyne Tool Components

Mnemosyne Analyzers

Memory Stride Analysis

Striding is the amount of data or interval between successive reads and/or writes. The memory stride analysis monitors the patterns in which an application traverses its memory space. We are interested in detecting two forms of striding. First, we find erratic striding which negates the spatial locality and increases cache misses due to cache line conflicts. Second, we detect regular and predictable striding patterns that force unexpected capacity and conflict misses [2].

Mnemosyne's memory stride analysis detects striding patterns in an application as it traverses the memory space. As patterns appear and are recorded, their predictability is measured. The prototype analysis currently rates the predictability of an individual pattern against all other detected patterns. The respective code segments that exhibit the striding patterns can then later be evaluated for erratic access patterns and unnecessary cache misses.

Cache Misaligned Access Analysis

The cache misaligned access analysis searches for addresses of memory requests that cause two cache lines to be accessed, thus stalling the pipeline for longer than expected by instruction schedulers. Our analysis mimics modern cache architectures by comparing address tags. If the address boundaries of the request have two different tags, the request is misaligned and reported to the developer.

Cache False Sharing Analysis

The cache false sharing analysis determines whether two or more threads of a multi-threaded application requesting two distinct data elements on the same cache line are thrashing the cache as they move data from one processor to another. This behavior, if left unchecked, can lead to performance degradation as the coherency protocol and data transfer between caches dominates execution time.

To mimic the concurrent behavior of threads, we model the expected behavior of threads with a sliding window of memory accesses. If two memory requests fall on the same cache line (but not the same address) within the predefined window, the requesting thread is considered to be competing with another thread for that cache line. If this behavior is seen in multiple requests, the behavior is reported along with the subset of threads involved in the false sharing. There are limitations to our scheme. The sliding window, if specified too large, could incur many false positives, and if the window is configured too small, could incur false negatives.

Invariant Analysis

The invariant analysis finds invariant branches and function calls that are unnecessary to preserve program semantics. Control flow constructs such as branches pose a difficult challenge to compilers trying to schedule instructions to execute. Additionally, execution time can be dominated by loops, so it is vital to keep loops as small and efficient as possible.

Our analyzer keeps a running history of frequently executed invariant variables in the code. The user can specify a requisite invariance tolerance level (typically near 100%). As the code is executed, a running history is updated depending on how the potential invariant is resolved in that instance. Variables below the user defined threshold are thrown out. These variables are too erratic and exhibit no invariance in the program semantics. Variables that stay in the history and have invariance levels above the threshold are reported as potential invariants or as sufficiently consistent variables. The invariant variables can be safely eliminated from the application code and those that are sufficiently consistent can be reexamined and reorganized to make them invariant if possible.

Tool Evaluation and Results

We used open source, SPEC [3], and customer codes to evaluate Mnemosyne. Our results are shown in Table 1. The codes were hand-optimized based on the Mnemosyne analysis, and the applications were rerun to verify functional correctness and to compare total execution time.

Code	Speedup	Behaviors
LibQuantum	2x	Memory striding Function Invariants
GNU Go	1.05x	Memory striding Branch invariants
OpenLB Poiseuille2D	1.02x	Memory striding Branch invariants
ICEPIC	1x	Memory striding Branch invariants Misaligned Accesses

Table 1. Evaluation Results

While we do expect Mnemosyne to find many common bottlenecks in HPC codes, it would be unrealistic to compare an automated tool to human developers seeking out and fixing those same bottlenecks. The fact that Mnemosyne was able to find deficiencies even after decades of human optimization efforts speaks to the extent of its usefulness. In our future work, Mnemosyne will be evaluating new applications as they are being developed. Our goal is to assist developers in quickly and easily detecting inefficient anomalies in their code and provide them exact source code lines to inspect for further optimization, reducing long optimization efforts.

References

[1] John Hennessy and David Patterson. Computer Architecture - A Quantitative Approach. Morgan Kaufmann, 2003.

[2] M.D. Hill and A.J. Smith, “Evaluating Associativity in CPU Caches”, IEEE Transactions on Computers, vol. 38, issue 12, Dec. 1989, pp. 1623-1630.

[3] http://www.spec.org/cpu2006/