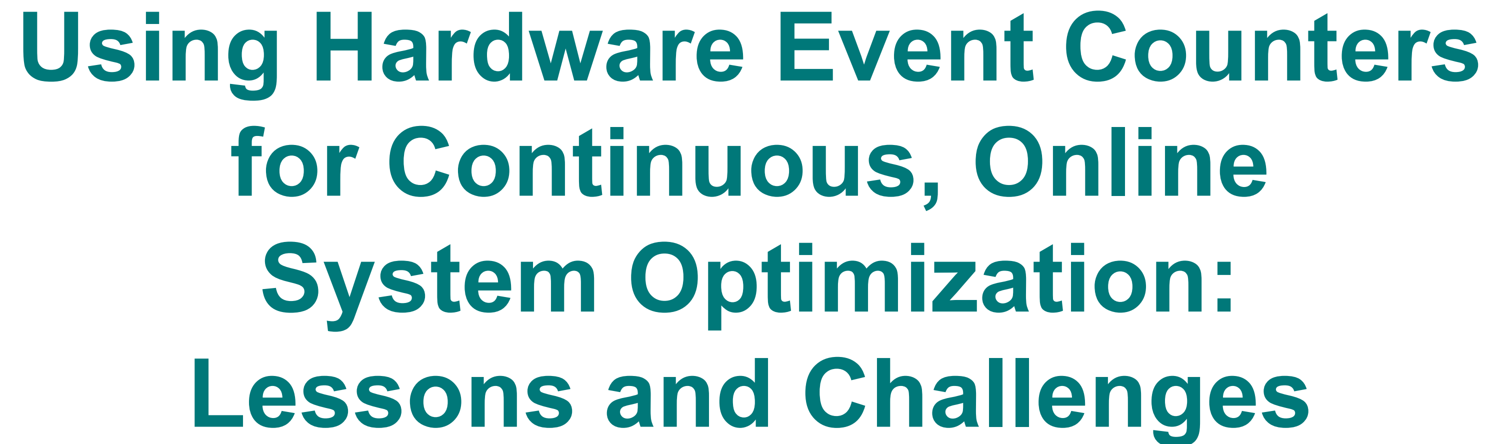


Using Hardware Event Counters for Continuous, Online System Optimization: Lessons and Challenges

Christos D. Antonopoulos & Dimitrios S. Nikolopoulos

Computer Science Department
The College of William & Mary,
Williamsburg, VA, U.S.A.
{cda, dsn}@cs.wm.edu

Workshop on Hardware Performance Monitor Design and Functionality

11th International Symposium on High-Performance Computer Architecture

1. Introduction

2. Our Experience

3. Suggestions

4. Conclusions

- Most modern general-purpose CPUs offer ***H**ard**w**are support for **P**erformance **M**onitoring (PMHw)*
- How is PMHw exploited?
  - Application programmers:
    - Post-mortem analysis
    - Identification of performance bottlenecks in applications
    - Resolution of bottlenecks
  - Hardware manufacturers:
    - Collect information on the performance of their products
    - Use this information during the design of future products

1. Introduction

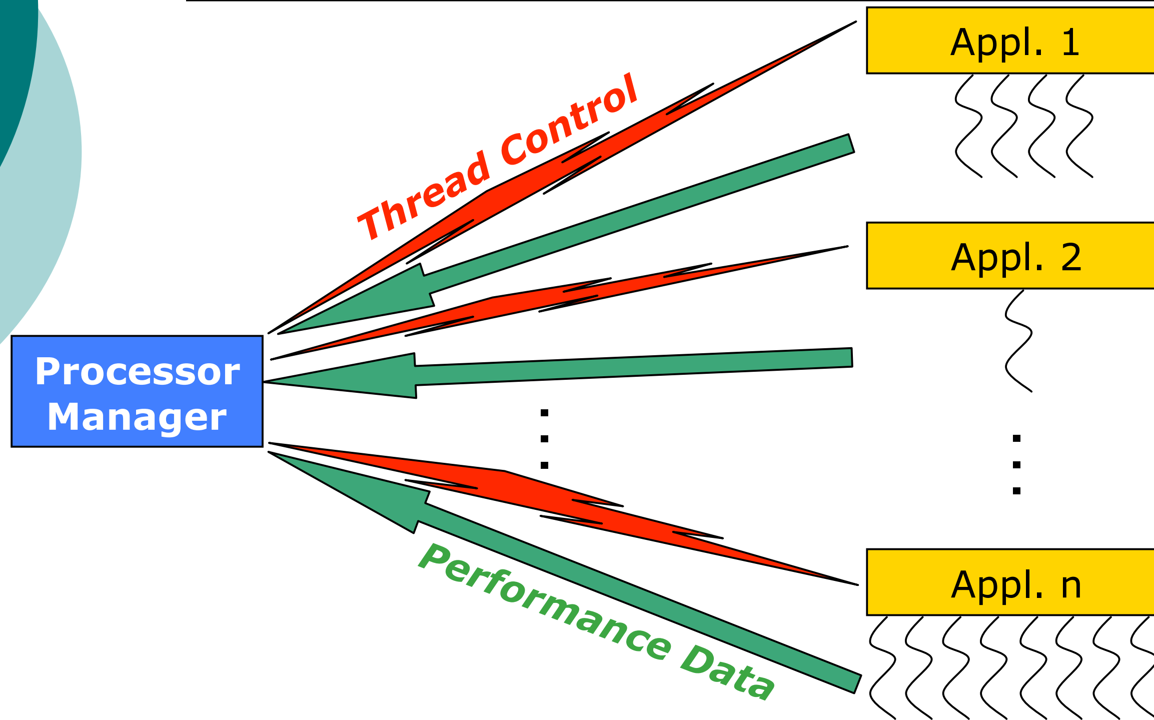
2. Our Experience

3. Suggestions

4. Conclusions

- **Online** monitoring of h/w events
- Performance **feedback** to OS schedulers
  - **Awareness** of the dynamically changing characteristics of the execution environment
  - **Adaptivity** to these characteristics
  - More educated scheduling decisions
- Policies implemented in a processor manager

MOHCA: *MO*nitoring of *H*ardware for *C*ontinuous *A*daptation (2/2)



- Implemented on Linux.
- Applied on Intel HT-based SMPs
- Negligible monitoring / thread control overhead
- Overlooked by OS schedulers
- Significant performance gains
- Gains projected to be higher in future multicore architectures

MOHCA: Main Problems & Workarounds

- Sharing of performance monitoring hardware between both execution context of an HT processor
  - Potential Conflicts
  - System software disallows the execution of 2 threads using performance monitoring on the same CPU
- Workaround
  - Configure 2 sets of counters for each measured event
  - Use each set to measure the events of triggered by a specific execution context
  - Attribute the events to the thread executing on the specific execution context
    - Possible because the processor manager has complete control on & knowledge about the execution and placement of threads to execution contexts

1. Introduction

2. Our Experience

3. Suggestions

4. Conclusions

Cost Effective, Thread-Local Event Monitoring

- Ideally: Replication of performance monitoring hardware once per execution context
- Workaround: Deal with the problem at the system software-level
  - Provide the virtual notion of per thread performance monitoring hardware
    - At the expense of limitations in the number of concurrently measurable events

Overhead and Intrusiveness

- Low overhead allows finer-grained sampling of events
  - Better association with the context of the program
- Offer low overhead, user-level access to performance counters
  - Prerequisite: Resolution of inter-thread conflicts for shared hardware
- Offer hardware support for non-intrusive performance monitoring
  - Typical example: Liquid architectures (Jones *et al.* 2004)
  - Specialized hardware allows offloading monitoring overhead from the processor
    - Facilitates extremely fine-grained monitoring and adaptation

- Lack of standardization
  - Set of measurable events
    - Basic metrics such as read/write misses to all levels of the cache hierarchy are not always supported
    - Readings are sometimes very inaccurate
  - Portability problems
  - API to the programmer
    - PAPI: Good effort in the direction of API standarization
    - Hindered by the heterogeneity of events and event semantics offered by different processors

Conveying Information on Contention

- SMTs & CMPs introduce high degrees of resource sharing inside the processor
- Performance monitoring should focus on attaining information about contention
  - Average length of queues: Indication of latency for access to the resources
    - Available for some queues (access to FSB or to cache)
    - Required for _op queues etc.
- Tagging for contention characterization
  - Tag _ops & cache lines according to the execution context that created / touched them
  - Estimate interference with other threads
  - Estimate footprints of threads in cache / conflict misses

Characterizing Misses

- Miss characterization absent from current performance monitoring infrastructures
  - Complexity
  - However some information can be attained:
    - Cross thread eviction: conflicts between co-executing threads
    - Per-thread population of cache lines: affinity policies
    - Per-thread population per cache bank / page-size region: page coloring / remapping
  - Dynamic memory optimizations expected to be critical in multithreaded / multicore processors

Monitoring Memory Traffic on Specific Address Ranges

- Monitoring virtual address ranges:
  - Useful for cache indexing
    - Identification of hot and cold areas in the virtual address space
  - Useful for associating objects with addresses at run-time
    - Feedback to run-time optimizers and profile-guided compilation
- Monitoring physical address ranges:
  - Useful for NUMA systems
    - Optimize data placement for locality or…
    - … energy control

1. Introduction

2. Our Experience

3. Suggestions

4. Conclusions

- Exploited hardware performance counters for continuous online optimization in kernel schedulers
- Identified drawbacks
- Coordinated effort required from H/W manufacturers and system S/W developers
  - Hardware: Indicators of performance loss and of its sources / contention for shared resources
  - Software: Integrate low-overhead, non-intrusive monitoring into transparent system modules
- System- and application-level adaptation via hardware monitoring is promising