Performance Optimization for Stampede

Jim Browne, Leo Fialho and Ashay Rane

XSEDE 2013

THE UNIVERSITY OF TEXAS AT AUSTIN

Agenda

- Why another performance tool?
- What is MACPO?
- What does MACPO tell you?
- How to use MACPO?
- Details of MACPO metrics

State of the art

- Modern processors can record performance events
- Performance events provide accurate view of CPU execution
- Various tools exist to correlate perormance events to user code
- Ex: PerfExpert, TAU, HPCToolkit, VTune, Scalasca, etc.

But memory is a bigger problem than CPU

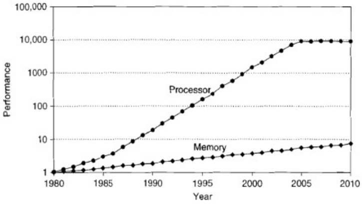


Figure 2.2 Starting with 1980 performance as a baseline, the gap in performance, measured as the difference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access, is plotted over time.

Source: Computer Architecture: A Quantitative Approach, 4th Edition by Hennessy and Patterson

But memory is a bigger problem than CPU

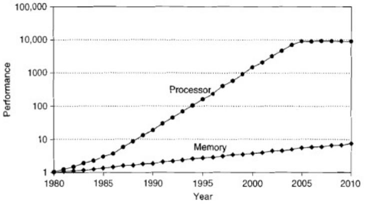


Figure 2.2 Starting with 1980 performance as a baseline, the gap in performance, measured as the difference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access, is plotted over time.

Source: Computer Architecture: A Quantitative Approach, 4th Edition by Hennessy and Patterson

- For data-intensive applications, application performance usually dependent on memory and not the processor.
- Performance events provide a CPU-centric, not a memory-centric view.

Limitations for performance events: #1: Fine-grain measurements

- Performance events are measured on execution of each instruction
- `mov ah, 16` causes cache miss $\Rightarrow$ increment cache miss counter
- But memory is optimized for stream traffic and regular accesses (locality, bandwidth, reuse)
- Hence gap between measurements and optimization techniques

Limitations for performance events: #2: Ambiguous interpretation

- Same symptoms but different root causes
- For instance, L3 cache misses could mean any of the following:
  - Capacity misses (cold cache)
  - Poor locality (less reuse of data structures)
  - False sharing (two processors writing to same cache line)

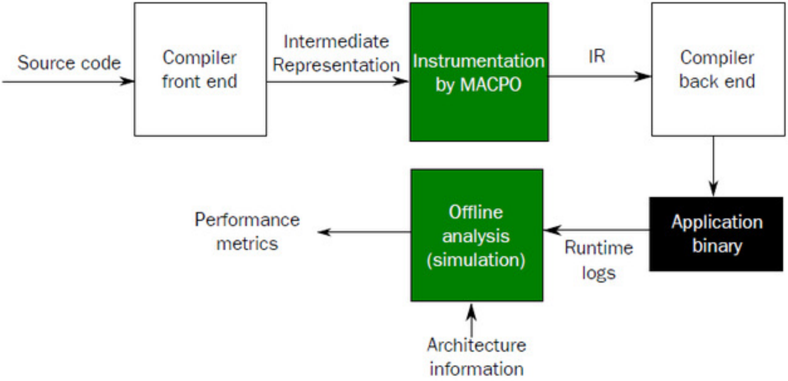
Limitations for performance events: #3: Little or no scoping

- Performance events are triggered for all instructions
- Hence all memory accesses are profiled
- But information about only specific data structures is desirable
- Can greatly speed up problem resolution

Hence, MACPO
Memory Access Centric Performance Optimization

- Analyzes access patterns to find sources of inefficiency
- Used as part of the compilation process
- Tracks accesses to arrays and structures within a function
- Tags each such access with source code location
- Analyzes accesses to see if access patterns can be improved
- Works with C, C++ and Fortran code [+ Pthreads, OpenMP]

MACPO workflow



Combines information from compiler, architecture and simulation

What does MACPO tell you?

For each important variable, MACPO shows:

- Access strides and the frequency of occurence
- Presence or absence of cache thrashing and the frequency
- NUMA misses
- Reuse factors for data caches

Sample output

```
Var "counts", seen 1668 times, estimated to cost 147.12 cycles on every access
Stride of 0 cache lines was observed 1585 times (100.00%).

Level 1 data cache conflicts = 78.22% [################################        ]
Level 2 data cache conflicts = 63.37% [##########################              ]
NUMA data conflicts = 43.56%          [##################                      ]

Level 1 data cache reuse factor = 97.0% [####################################### ]
Level 2 data cache reuse factor = 3.0% [##                                      ]
Level 3 data cache reuse factor = 0.0% [                                        ]
```

How to use MACPO?

- Compile application using `macpo.sh`
- Run application as usual
- Analyze macpo logs using macpo-analyze

How to use MACPO?

```
# Compile the application using macpo.sh
# Specify code section using --macpo:function or --macpo:loop
macpo.sh --macpo:function=thread_func -c mcpi.cc
macpo.sh --macpo:function=thread_func -o mcpi mcpi.o

# Run the application as usual
./mcpi

# Post-process logs to get analysis output
macpo-analyze macpo.out
```

Understanding MACPO metrics

- Access strides
- Cache conflicts
- NUMA misses
- Reuse factor for data caches

Metric #1: Cycles per access

Example:

```
Var "counts", seen 1073 times,
estimated to cost 8.98 cycles on every access
```

- Provides esimate of performance impact of accesses to variable
- Can be used to rule-out variables from further consideration

Metric #2: Access strides

Example:

```
Stride of 0 cache lines was observed 983 times (97.62%).
Stride of 2 cache lines was observed 24 times (2.38%).
```

- Programs that have unit strides or small regular stride values generally execute fast
- If stide value is high, look for inverted loops affecting the row-major or column-major ordering

Metric #3: Cache conflicts

Example:

```
Level 1 data cache conflicts = 78.22%
Level 2 data cache conflicts = 63.37%
```

- Indicates multiple cores writing to the same cache line
- Add dummy bytes to the array so that each processor writes to a different cache line

Metric #4: NUMA misses

Example:

```
NUMA data conflicts = 43.56%
```

- NUMA misses generally arise from one processor initializing all of the shared memory
- To eliminate NUMA misses, have each processor initialize it's portion of shared memory

Metric #5: Reuse factors

Example:

```
Level 1 data cache reuse factor = 94.1%
Level 2 data cache reuse factor = 5.9%
Level 3 data cache reuse factor = 0.0%
```

- Reuse factor indicates the number of times a cache was reused before it was evicted
- Improve reuse factors by using techniques to improve locality

Summary

- MACPO is a tool to analyze memory access patterns
- NOT a replacement for PerfExpert. Instead, complements PerfExpert's diagnosis.
- Allows collection of memory traces for arrays and structures
- Analyzes traces offline to calculate performance metrics
- This is an early release, so help us squash the bugs! :)

Sample application

- Monte-Carlo computation of Pi
- Source code online at: `http://goo.gl/uEVrh`
- Uses basic C++, parallelized using Pthreads
- Tasks performed by each thread:
  - Generates a buffer of random numbers
  - For each pair of random numbers, calculates `z`
  - Checks a condition on `z`, based on the result increments a counter

Thread function

```
int t;
float x, y, z;
thread_info_t* thread_info = (thread_info_t*) arg;
for (int repeat=0; repeat<REPEAT_COUNT; repeat++)
{
  for (int i=0; i<ITERATIONS; i++)
  {
    t = i+thread_info->tid;
    x = random_numbers[t%RANDOM_BUFFER_SIZE];
    y = random_numbers[(1+t)%RANDOM_BUFFER_SIZE];

    z = x*x + y*y;
    if (z < 1) counts[thread_info->tid]++;
  }
}
```

MACPO commands for sample code

```
# Compile the application using macpo.sh
# Specify code section using --macpo:function or --macpo:loop
macpo.sh --macpo:function=thread_func source.c compute.c -o mm

# Run the application as usual
./mm

# Post-process logs to get analysis output
macpo-analyze macpo.out
```

MACPO analysis output (truncated)

```
macpo-analyze macpo.out

Var "counts", seen 1668 times,
estimated to cost 147.12 cycles on every access

Stride of 0 cache lines was observed 1585 times (100.00%).

Level 1 data cache conflicts = 78.22%
Level 2 data cache conflicts = 63.37%
NUMA data conflicts = 43.56%

Level 1 data cache reuse factor = 97.0%
Level 2 data cache reuse factor = 3.0%
Level 3 data cache reuse factor = 0.0%
```

Problem resolution

- MACPO shows cache thrashing for `counts` variable.
- Solution: Add dummy bytes, thus all processors write to different cache lines
- Optimized code in `monte-carlo-v2.cc`

MACPO analysis output for optimized code (truncated)

```
macpo-analyze macpo.out

Var "counts", seen 1073 times,
estimated to cost 8.98 cycles on every access

Stride of 0 cache lines was observed 983 times (97.62%).
Stride of 2 cache lines was observed 24 times (2.38%).

Level 1 data cache conflicts = 0.00%
Level 2 data cache conflicts = 0.00%
NUMA data conflicts = 0.00%

Level 1 data cache reuse factor = 94.1%
Level 2 data cache reuse factor = 5.9%
Level 3 data cache reuse factor = 0.0%
```

Review

- Compiled application using `macpo.sh`
- Discovered cache thrashing for the `counts` array
- Padding the array reduced cache conflicts from 70% to 0%
- Execution time: 9.14s to 3.17s (65% improvement)

Summary of MACPO instructions

```
# Compile the application using macpo.sh
# Specify code section using --macpo:function or --macpo:loop
macpo.sh --macpo:function=thread_func source.c compute.c -o mm

# Run the application as usual
./mm

# Post-process logs to get analysis output
macpo-analyze macpo.out
```