Performance Optimization for Stampede

Jim Browne, Ashay Rane and Leo Fialho

XSEDE 2013

THE UNIVERSITY OF
TEXAS
AT AUSTIN

Agenda

1 Introduction

2 The User Perspective

3 Hands-on

4 PerfExpert Architecture

5 Conclusions

Agenda

In the morning:

- Introduction and motivation
- What PerfExpert can provide to you?
- Enhancing PerfExpert with MACPO analysis
- Hands on tutorial
- Bring your own code!
- How PerfExpert does that? (opening Pandora's box)
- Closure
- Do not worry, there will be a break (coffee?)

Overview: why PerfExpert?

Problem: HPC systems operate far below peak

- Chip/node architectural complexity is growing rapidly
- Performance optimization for these chips requires deep knowledge of architectures, code patterns, compilers, etc.

Performance optimization tools

- Powerful in the hands of experts
- Require detailed performance and system expertise
- HPC application developers are domain experts, not computer gurus

Result: Many HPC programmers do not use these tools

(seriously)

Goal for PerfExpert: democratize optimization!

Subgoals:

- Make use of the tool as simple as possible
- Start with only chip/node level optimization
- Make it adaptable across multiple architectures
- Design for extension to communication and I/O performance

How to accomplish?

- Formulate the performance optimization task as a workflow of subtasks
- Leverage the state-of-the-art: build on the best available tools for the subtasks to minimize the effort and cost of development
- Automate the entire workflow

Introduction

The four stages of automatic performance optimization:

- Measurement and attribution (1)
- Analysis, diagnosis and identification of bottlenecks (2)
- Selection of effective optimizations (3)
- Implementation of optimizations (4)

Use of State-of-the-Art:

- HPCToolkit/Intel VTune, **MACPO** based on ROSE (1)
- **PerfExpert Team** (2 and 3)
- **PerfExpert Team** based on ROSE, PIPS, Bison and Flex (4)

Introduction

Uniqueness of PerfExpert:

- Nearly complete optimization first three stages of optimization for chip/node level
- Framework for implementing optimizations is complete and several optimizations are completed
- Integrates code segment focused and data structure based measurements (**MACPO**)
- Workflow will apply to communication and I/O optimization as well

Introduction

Small break: what is MACPO?

- 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 fairly accurate view of CPU execution
- Various tools exist to correlate performance events to user code
- Examples: PerfExpert, TAU, HPCToolkit, VTune, Scalasca, etc.

Introduction

But memory is a bigger problem

- 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.

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

Introduction

Performance counter info is limited

- Performance events are too-fine grained for memory profiling
- Some performance events are ambiguous to interpret
- Performance event measurements cannot be scoped to just the important variables

Fine-grain measurements

- Performance events are measured on execution of each instruction
- `mov ah, [1234h]` causes cache miss ⇒ increment cache miss counter
- But memory is optimized for stream traffic and regular accesses (locality, bandwidth, reuse)
- Hence gap between measurements and optimization techniques

Introduction

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)

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

Introduction

Hence, MACPO

- Memory Access Centric Performance Optimization
- Performance tool that analyzes memory 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 and other data
- Analyzes accesses to see if access patterns can be improved
- Works with C, C++ and Fortran code [+ Pthreads, OpenMP]

Introduction

Unique properties of MACPO (integrated into PerfExpert):

- Multicore resolved traces
- Code segment local measurement
- Data structure specific traces
- Order of magnitude lower overhead of measurement
- More accurate (associative) cache models
- Strides by data structure and code segment
- Architecture “independent” metrics

What PerfExpert can provide to you?

Performance report:

- Identification of bottlenecks by relevance
- Performance analysis based on performance metrics
- Recommendations for optimization

There are three possible outputs:

- Performance report only
- List of recommendations
- Fully automated code transformation

Performance Report

```
Loop in function compute() at mm.c:8 (99.8% of the total runtime)
===============================================================================
ratio to total instrns       % 0.........25..........50.........75........100
   - floating point      :   100 ***********************************************
   - data accesses       :    25 ************
* GFLOPS (% max)         :    12 ******
   - packed              :     0 *
   - scalar              :    12 ******
-------------------------------------------------------------------------------
performance assessment     LCPI good......okay......fair......poor......bad....
* overall                :   3.0 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
upper bound estimates
* data accesses          :   9.6 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
   - L1d hits            :   0.9 >>>>>>>>>>>>>>>>>
   - L2d hits            :   1.8 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
   - L2d misses          :   6.9 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
* instruction accesses   :   0.1 >
   - L1i hits            :   0.0 >
   - L2i hits            :   0.0 >
   - L2i misses          :   0.1 >
* data TLB               :   4.6 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
* instruction TLB        :   0.0 >
* branch instructions    :   0.1 >>
   - correctly predicted :   0.1 >>
   - mispredicted        :   0.0 >
* floating-point instr   :   5.1 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
   - fast FP instr       :   5.1 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
   - slow FP instr       :   0.0 >
```

Performance Report (MACPO)

```
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%).

Level1 data cache conflicts = 78.22% [##########################       ]
Level2 data cache conflicts = 63.37% [####################             ]
NUMA data conflicts = 43.56%         [###############                  ]

Level1 data cache reuse factor = 97.0% [############################## ]
Level2 data cache reuse factor = 3.0%  [##                             ]
Level3 data cache reuse factor = 0.0%  [                               ]
```

List of Recommendations

```
#--------------------------------------------------
# Recommendations for mm.c:8
#--------------------------------------------------
# This is a possible recommendation for this code segment
#
Description: change the order of loops
Reason: this optimization may improve the memory access
pattern and make it more cache and TLB friendly
Pattern Recognizers: c_loop2 f_loop2
Code example:
loop i {
  loop j {...}
}
=====> loop j {
  loop i {...}
}
```

Fully Automated Code Transformation

Before:

```
void compute() {
 register int i, j, k;
 for (i = 0; i < 3000; i++)
  for (j = 0; j < 3000; j++)
   for (k = 0; k < 3000; k++)
    c[i][j] += (a[i][k] * b[k][j]);
}
```

After:

```
void compute() {
 register int i, j, k;
 for (i = 0; i <= 2999; i++)
  for (k = 0; k <= 2999; jp += 1)
   for (kp = 0; kp <= 2999; kp += 1)
   c[i][j] += a[i][k] * b[k][j];
}
```

Agenda

1. Introduction
2. The User Perspective
3. Hands-on
4. PerfExpert Architecture
5. Conclusions

Basic Usage of PerfExpert

Making PerfExpert Available

```
$ module load papi perfexpert
```

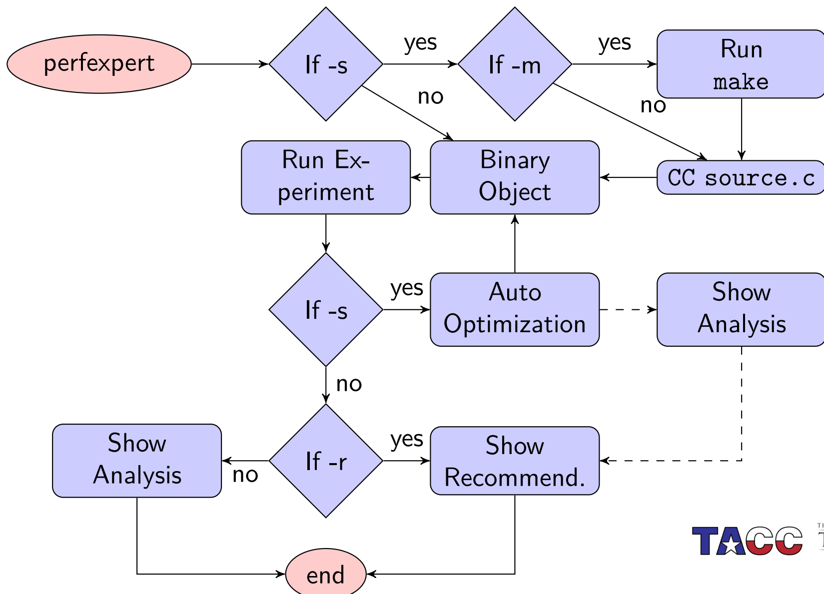
Execution Options

```
Usage:  perfexpert [-ghmvq] [-w DIR] [-s FILE] [-r COUNT]
                   program_executable [program_arguments]

-s    Use FILE as the source code
-m    Use 'make' to compile source code
-q    Disable verbose mode
-w    Use DIR as temporary directory
-g    Do not remove the temporary directory
-r    Use COUNT as the number of recommendation to show

Use CC, CFLAGS and LDFLAGS to select compiler and
compilation/linkage flags
```

Basic Usage of PerfExpert



TACC THE UNIVERSITY OF TEXAS AT AUSTIN

Basic Usage of PerfExpert

In Other Words...

- No source code, no automatic optimization
- No source code, choose between analysis or recommendation
- Source code, enable automatic optimization
- Source code, choose the compilation method (-m) and options (CC, CFLAGS and LDFLAGS)
- Source code, show analysis and recommendation after all the possible automatic optimizations have been applied

Examples:

```
$ perfexpert my_program param1 param2
$ perfexpert -r 5 my_program param1 param2
$ perfexpert -s my_program.c my_program param1 param2
$ perfexpert -m -s my_program.c my_program param1 param2
```

Understanding PerfExpert Analysis

On the The Analysis Report...

- The more "expensive" comes first
- Tells user where the slow code sections are as well as why they perform poorly
- Every function or loop which takes more than 1% of the execution time is analyzed (default value)
- Yes, we rely on performance metrics (but not only and not the raw ones)
- No, we do not rely on hardware specs
- If you are not using properly the node PerfExpert may conclude everything is fine (use a representative workload)

Metrics used by PerfExpert

Source Code

- Language (C, C++, Fortran)
- File name and line number
- Type (loop or function)
- Function name and “deepness”
- Representativeness (percentage of execution time)

Execution Performance

- Raw data (PAPI)
- LCPI: local cycles per instruction (PerfExpert Analyzer)

Metrics used by PerfExpert

Data Access Performance (from MACPO)

- Access strides and the frequency of occurrence (*)
- Presence or absence of cache thrashing and the frequency (*)
- Estimated cost (cycles) per access (*)
- NUMA misses (*)
- Reuse factors for data caches (*)
- Stream count

(*) *per variable*

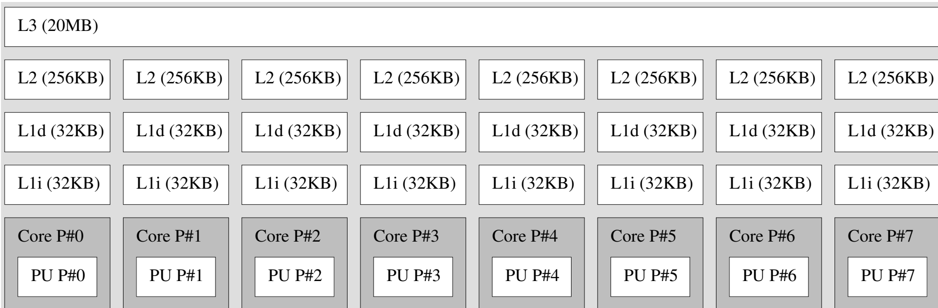
Metrics used by PerfExpert

Architecture Characteristics

- Memory access latency: L1, L2, L3 and main memory (based on micro-benchmarks)
- Memory hierarchy, topology and size (based on hwlock)
- Branch latency and missed branch latency (based on micro-benchmarks)
- Float-point operation latency (based on micro-benchmarks)
- Micro-architecture (in progress)

TACC THE UNIVERSITY OF TEXAS AT AUSTIN

How Stampede CPUs Looks Like?



Just to mention some numbers...

- Memory access latency:
  — L1: ~3.5 cycles
  — L2: ~11.5 cycles
  — L3: ~16.5 cycles
  — Main memory: ~270 cycles

How Stampede nodes Looks Like?

NUMANode P#0 (16GB)

Socket P#0

L3 (20MB)

L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB)

L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB)

L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB)

Core P#0 PU P#0 | Core P#1 PU P#1 | Core P#2 PU P#2 | Core P#3 PU P#3 | Core P#4 PU P#4 | Core P#5 PU P#5 | Core P#6 PU P#6 | Core P#7 PU P#7

PCI 8086:1521 eth0

PCI 8086:1521 eth1

PCI 1a03:2000

PCI 8086:1d02 sda

NUMANode P#1 (16GB)

Socket P#1

L3 (20MB)

L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB) L2 (256KB)

L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB) L1d (32KB)

L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB) L1i (32KB)

Core P#0 PU P#8 | Core P#1 PU P#9 | Core P#2 PU P#10 | Core P#3 PU P#11 | Core P#4 PU P#12 | Core P#5 PU P#13 | Core P#6 PU P#14 | Core P#7 PU P#15

PCI 15b3:1003 ib0 mlx4_0

PCI 8086:225c

Performance Report

```
Loop in function compute() at mm.c:8 (99.8% of the total runtime)
===============================================================================
ratio to total instrns       % 0.........25...........50.........75........100
   - floating point       :   100 ***********************************************
   - data accesses        :    25 ************
* GFLOPS (% max)          :    12 ******
   - packed               :     0 *
   - scalar               :    12 ******
-------------------------------------------------------------------------------
performance assessment      LCPI good......okay......fair......poor......bad....
* overall                 :   3.0 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
upper bound estimates
* data accesses           :   9.6 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
   - L1d hits             :   0.9 >>>>>>>>>>>>>>>>>
   - L2d hits             :   1.8 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
   - L2d misses           :   6.9 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
* instruction accesses    :   0.1 >
   - L1i hits             :   0.0 >
   - L2i hits             :   0.0 >
   - L2i misses           :   0.1 >
* data TLB                :   4.6 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
* instruction TLB         :   0.0 >
* branch instructions     :   0.1 >>
   - correctly predicted  :   0.1 >>
   - mispredicted         :   0.0 >
* floating-point instr    :   5.1 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
   - fast FP instr        :   5.1 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
   - slow FP instr        :   0.0 >
```

Performance Report

```
Loop in function compute() at mm.c:8 (99.8% of the total runtime)
===============================================================================
ratio to total instrns        % 0.........25..........50.........75........100
   - floating point       :   100 ***********************************************
   - data accesses        :    25 ************
```

Interpretation

- What percentage of the total instructions were computational (floating-point instructions)
- What percentage were instructions that accessed data
- So, whether optimizing the program for either data accesses or floating-point instructions would have a significant impact on the total running time of the program?

Performance Report

```
Loop in function compute() at mm.c:8 (99.8% of the total runtime)
...
* GFLOPS (% max)          :     12 ******
   - packed               :      0 *
   - scalar               :     12 ******
...
```

Interpretation

- GFLOPs rating, which is the number of floating-point operations executed per second
- This metric is displayed as a percentage of the maximum possible GFLOP value for that particular machine
- It is rare for real-world programs to match even 50% of the maximum value

TACC THE UNIVERSITY OF TEXAS AT AUSTIN

Performance Report

```
Loop in function compute() at mm.c:8 (99.8% of the total runtime)
...
performance assessment     LCPI good......okay......fair......poor......bad....
* overall                 :   3.0 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
...
```

Interpretation

- LCPI values: is the ratio of cycles spent in the code segment for a specific category, divided by the total number of instructions in the code segment
- The overall value is the ratio of the total cycles taken by the the code segment to the total instructions executed in the code segment
- Generally, a value of 0.5 or lower for the CPI is considered to be good

Performance Report

```
Loop in function compute() at mm.c:8 (99.8% of the total runtime)
...
* data accesses         :   9.6 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
   - L1d hits           :   0.9 >>>>>>>>>>>>>>>>>
   - L2d hits           :   1.8 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
   - L2d misses         :   6.9 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
* instruction accesses  :   0.1 >
   - L1i hits           :   0.0 >
   - L2i hits           :   0.0 >
   - L2i misses         :   0.1 >
...
```

Interpretation

- LCPI arising from accesses to memory for program variables
- LCPI arising from memory accesses for code (functions and loops)
- Shows different levels of memory (L1, L2, etc.)

TACC THE UNIVERSITY OF TEXAS AT AUSTIN

Performance Report

```
Loop in function compute() at mm.c:8 (99.8% of the total runtime)
...
* data TLB              :   4.6 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
* instruction TLB       :   0.0 >
* branch instructions   :   0.1 >>
   - correctly predicted :   0.1 >>
   - mispredicted        :   0.0 >
* floating-point instr  :   5.1 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
   - fast FP instr       :   5.1 >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>+
   - slow FP instr       :   0.0 >
```

Interpretation

- Data TLB: provides an approximate measure of penalty arising from strides in accesses or regularity of accesses
- Instruction TLB: reflects penalty from fetching instructions due to irregular accesses
- Branch instructions: counts penalty from jumps (i.e. if statements, loop conditions, etc.)
- Floating-point instructions: counts LCPI from executing computational (floating-point) instructions

PerfExpert: A Simple Example

Automatic Optimization of a C Code

```
$ perfexpert -s code.c code
```

Optimization Steps

- One full optimization cycle
- Runs out of automatic optimizations during the second cycle
- Shows the analysis report as well as recommendations
- Execution time: from 88.856 seconds to 6.967 seconds
- There is (still) room for improvement

Comparing Codes

Before:

```
void compute() {
 register int i, j, k;




 for (i = 0; i < 1000; i++)


  for (j = 0; j < 1000; j++)

   for (k = 0; k < 1000; k++)
    c[i][j] += (a[i][k] * b[k][j]);
}
```

After:

```
void compute() {
 register int i, j, k;
 //PIPS generated variable
 register int jp, kp;
 /* PERFEXPERT: start work here */
 /* PERFEXPERT: grandparent loop */
 loop_6:
 for (i = 0; i <= 999; i++)
  /* PERFEXPERT: parent loop */
  loop_7:
  for(jp = 0; jp <= 999; jp += 1)
   /* PERFEXPERT: bottleneck */
   for(kp = 0; kp <= 999; kp += 1)
   c[i][kp] += a[i][jp]*b[jp][kp];
}
```

Comparing Reports

Before: 88.856 sec

```
ratio to total instrns              %
    - floating point        :     100
    - data accesses         :      25
* GFLOPS (% max)            :      13
    - packed                :       0
    - scalar                :      13
---------------------------------
performance assessment           LCPI
* overall                   :     3.7
* data accesses             :    40.6
    - L1d hits              :     2.3
    - L2d hits              :     4.9
    - L2d misses            :    33.4
* instruction accesses      :     0.1
...
* data TLB                  :     4.5
* instruction TLB           :     0.0
* branch instructions       :     0.1
...
* floating-point instr      :     5.7
    - fast FP instr         :     5.7
    - slow FP instr         :     0.0
```

After: 6.967 sec (12x faster)

```
ratio to total instrns              %
    - floating point        :     100
    - data accesses         :      29
* GFLOPS (% max)            :      29
    - packed                :      17
    - scalar                :      12
---------------------------------
performance assessment           LCPI
* overall                   :     0.7
* data accesses             :    10.5
    - L1d hits              :     2.6
    - L2d hits              :     0.9
    - L2d misses            :     7.0
* instruction accesses      :     0.0
...
* data TLB                  :     0.0
* instruction TLB           :     0.0
* branch instructions       :     0.1
...
* floating-point instr      :     1.7
    - fast FP instr         :     1.7
    - slow FP instr         :     0.0
```

Exploring the Temporary Directory

Automatic Optimization of a C Code

- Each optimization cycle has it's own subdirectory containing:
  - Source code directory
  - Debug file and intermediary file for every optimization step (5)
  - Analysis report
  - Directory containing the code fragments identified as bottleneck
  - Directory containing the optimized source code
- Workflow log file

MACPO as a Standalone Tool

How to use MACPO?

- Compile the application using `macpo.sh` and either `--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 application as usual

```
$ ./mcpi
```

- Analyze macpo logs using macpo-analyze

```
$ macpo-analyze macpo.out
```

Understanding MACPO Metrics

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

```
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%).

Level1 data cache conflicts = 78.22% [#########################       ]
Level2 data cache conflicts = 63.37% [###################             ]
NUMA data conflicts = 43.56%         [##############                  ]

Level1 data cache reuse factor = 97.0% [############################# ]
Level2 data cache reuse factor = 3.0%  [##                            ]
Level3 data cache reuse factor = 0.0%  [                              ]
```

Understanding MACPO Metrics

Report

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

...
```

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

TACC THE UNIVERSITY OF TEXAS AT AUSTIN

Understanding MACPO Metrics

Report

```
...
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

TACC THE UNIVERSITY OF TEXAS AT AUSTIN

Understanding MACPO Metrics

```
...
Level1 data cache conflicts = 78.22% [##########################        ]
Level2 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

Understanding MACPO Metrics

```
...
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

Understanding MACPO Metrics

```
...
Level1 data cache reuse factor = 97.0% [################################ ]
Level2 data cache reuse factor = 3.0%  [##                               ]
Level3 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

MACPO as a Standalone Tool: Summary

Quick summary of MACPO

- 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, please help us squash the bugs! :)

Agenda

1. Introduction
2. The User Perspective
3. Hands-on
4. PerfExpert Architecture
5. Conclusions

Hands-on Tutorial

Guided examples

- PerfExpert automatic optimization using on a matrix multiply code
- PerfExpert report analysis using LULESH
- PerfExpert report analysis using Rodinia Back Propagation (using Makefile to compile)
- MACPO report analysis using a Monte-Carlo computation of Pi

Open laboratory

- Suggestion: try the guided examples with different number of threads
- Bring your own code

Hands-on Tutorial

Setting up your environment

- After entering you login and password, run the following command:
  — `$ tar -xzvf ~fialho/xsede.tar.gz`

WARNING!

- Do not run any example on the login nodes!
- If you can see something like `login2$` in the terminal, you are in a login node (the number varies, but it also starts with `login`
- Request a node using `~/reserve` to run the examples

TACC THE UNIVERSITY OF TEXAS AT AUSTIN

PerfExpert Automatic Optimization

Parallel Matrix Multiply using OpenMP

- Source code available at the 3 directory
- Pure C code, parallelized using OpenMP
- The `compute` function:
  - — Reads from matrix `a` and `b`
  - — Write on matrix `c`

Original compute function

```
void compute(void) {
#pragma omp parallel shared(a, b, c, chunk) private(i, j, k)
#pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)
    for (j=0; j<NCB; j++)
      for (k=0; k<NCA; k++)
        c[i][j] += a[i][k] * b[k][j];

}
```

IVERSITY OF XAS AUSTIN

PerfExpert Automatic Optimization

Running the matrix multiply with PerfExpert

```
$ cd
$ ./reserve
$ cd 3
$ OMP_NUM_THREADS=16 perfexpert -s mm_omp.c mm_omp
```

Checking PerfExpert and MACPO analyses

```
$ cat perfexpert-temp-XXXXXX/*/analyzer2.output.txt
$ cat perfexpert-temp-XXXXXX/*/macpo.output.txt
```

Checking the modified source code

```
$ cat new_mm_omp.c
```

PerfExpert Automatic Optimization

Before: 22.375 sec

```
ratio to total instrns              %
   - floating point        :      100
   - data accesses         :       25
* GFLOPS (% max)           :       13
   - packed                :        0
   - scalar                :       13
---------------------------------
performance assessment          LCPI
* overall                  :      4.4
* data accesses            :     44.7
   - L1d hits              :      0.9
   - L2d hits              :      2.6
   - L2d misses            :     41.2
* instruction accesses     :      0.1
...
* data TLB                 :      4.6
* instruction TLB          :      0.0
* branch instructions      :      0.1
...
* floating-point instr     :      7.7
   - fast FP instr         :      7.7
   - slow FP instr         :      0.0
```

After: 2.08 sec (11x faster)

```
ratio to total instrns              %
   - floating point        :      100
   - data accesses         :       29
* GFLOPS (% max)           :       29
   - packed                :       13
   - scalar                :       13
---------------------------------
performance assessment          LCPI
* overall                  :      0.9
* data accesses            :      7.9
   - L1d hits              :      1.0
   - L2d hits              :      0.8
   - L2d misses            :      6.1
* instruction accesses     :      0.0
...
* data TLB                 :      0.0
* instruction TLB          :      0.0
* branch instructions      :      0.1
...
* floating-point instr     :      1.7
   - fast FP instr         :      1.7
   - slow FP instr         :      0.0
```

PerfExpert Automatic Optimization

Before: 22.375 sec

```
Var "a", seen 2498 times, cost 22.93
Stride of 0 cache lines 87.42%
Stride of 1 cache lines 12.58%
...
Level1 cache reuse factor = 85.7%
Level2 cache reuse factor = 14.3%

Var "b", seen 2295 times, cost 62.93
Stride of 375 cache lines 100.00%

...
Level1 cache reuse factor = 50.0%
Level2 cache reuse factor = 50.0%

Var "c", seen 2546 times, cost 12.38
Stride of 0 cache lines 100.00%

...
Level1 cache reuse factor = 50.0%
Level2 cache reuse factor = 50.0%
```

After: 2.08 sec (11x faster)

```
Var "a", seen 2247 times, cost 11.61
Stride of 0 cache lines 100.00%

...
Level1 cache reuse factor = 94.1%
Level2 cache reuse factor = 5.9%

Var "b", seen 1889 times, cost 12.05
Stride of 0 cache lines 86.92%
Stride of 1 cache lines 13.08%
...
Level1 cache reuse factor = 78.9%
Level2 cache reuse factor = 21.1%

Var "c", seen 2546 times, cost 23.11
Stride of 0 cache lines 87.84%
Stride of 1 cache lines 12.16%
...
Level1 cache reuse factor = 97.0%
Level2 cache reuse factor = 3.0%
```

PerfExpert Automatic Optimization

Parallel Matrix Multiply using OpenMP

- The indexes of the two inner loops do not make a good use of the cache
  - the strides are too long
  - increases the TLB misses
  - prevents a cache reuse for matrix b and c

Modified compute function

```
void compute(void) {
#pragma omp parallel shared(a, b, c, chunk) private(i, j, k)
#pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)
    for (k=0; k<NCB; k++)
      for (j=0; j<NCA; j++)
        c[i][j] += a[i][k] * b[k][j];
}
```

LULESH Example

Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics

- Source code available in the directory 4 of your environment, or online at: `https://codesign.llnl.gov/lulesh.php`
- Uses basic C++, parallelized using OpenMP
- Using 50 as problem size, it runs in 31 seconds
- Function `CalcFBHourglassForceForElems()` takes 10% of the execution time
  — plus an OpenMP runtime system function (`omp_get_num_procs()`) which takes other 53% of the execution time

TACC THE UNIVERSITY OF TEXAS AT AUSTIN

LULESH Example

Running LULESH with PerfExpert and MACPO

- Please, refer to the hands-out for this example, the following is just a reference:
  - `$ g++ -g -O3 -fopenmp -o lulesh lulesh.cc`
  - `$ OMP_NUM_THREADS=16 perfexpert lulesh 50`
  - `$ macpo.sh --macpo:function=CalcFBHourglassForceForElems -g -O3 -fopenmp -o lulesh lulesh.cc`
- Analyze PerfExpert and MACPO reports
- Run PerfExpert again, asking for recommendations:
  - `$ OMP_NUM_THREADS=16 perfexpert -r 5 lulesh 50`

LULESH Example

The optimized version of this code...

- ...is also available in the directory 4 of your environment, or online at: `https://codesign.llnl.gov/lulesh.php`
- Optimized by performance experts from LLNL and UTEP
- They followed the following recommendations:
  — **move loop invariant memory accesses out of loop**: memory is unnecessarily allocated and deallocated at each time step – LLNL moved this outside of the main loop, which reduced execution time
  — **unroll outer loop**: loop unrolling was applied as part of the aggressive technique to increase vectorization
  — **enable the use of vector instructions to transfer more data per access**: LLNL vectorized this loop but it was not done in this way – nonetheless, the MACPO output shows that it was rather successful
  — **componentize important loops by factoring them into their own subroutines**: LLNL did this a bit excessively at first – then they fused some functions to regain performance

LULESH Example

The optimized version of this code...

- Let's run the optimized version (please, refer to the hands-out for this example, the following is just a reference):
  — `$ g++ -g -O3 -fopenmp -o lulesh_opt lulesh_opt.cc`
  — `$ OMP_NUM_THREADS=16 perfexpert lulesh_opt 50`
  — `$ macpo.sh --macpo:function=CalcFBHourglassForceForElems -g -O3 -fopenmp -o lulesh_opt lulesh_opt.cc`
- Analyze PerfExpert and MACPO reports
- Now the code runs in 18.5 seconds!
- Function `CalcFBHourglassForceForElems()` takes 23% of the execution time and the bottleneck on the OpenMP runtime system was gone!

MACPO as a Standalone Tool

Monte-Carlo computation of Pi

- Source code online at: `http://goo.gl/uEVrh` or at: `/home1/0003/train300/sample-programs.tar.gz`
- 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

MACPO as a Standalone Tool

Thread function

```
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++)
  {
     x = random_numbers[(i+thread_info->tid)%RANDOM_BUFFER_SIZE];
     y = random_numbers[(1+i+thread_info->tid)%RANDOM_BUFFER_SIZE];

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

MACPO as a Standalone Tool

Compiling with MACPO

```
# Compile the application using macpo.sh
$ macpo.sh --macpo:function=thread_func monte-carlo.cc -o mm -lpthread
-lrt

# Run the application as usual
$ ./mm

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

MACPO as a Standalone Tool

Report

```
$ 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%  [                                        ]
```

Analysis

- 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`

TACC | THE UNIVERSITY OF TEXAS AT AUSTIN

MACPO as a Standalone Tool

Report

```
$ 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%  [                                        ]
```

Analysis

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

Agenda

1 Introduction

2 The User Perspective

3 Hands-on

4 PerfExpert Architecture

5 Conclusions

How PerfExpert does that: The Big Picture

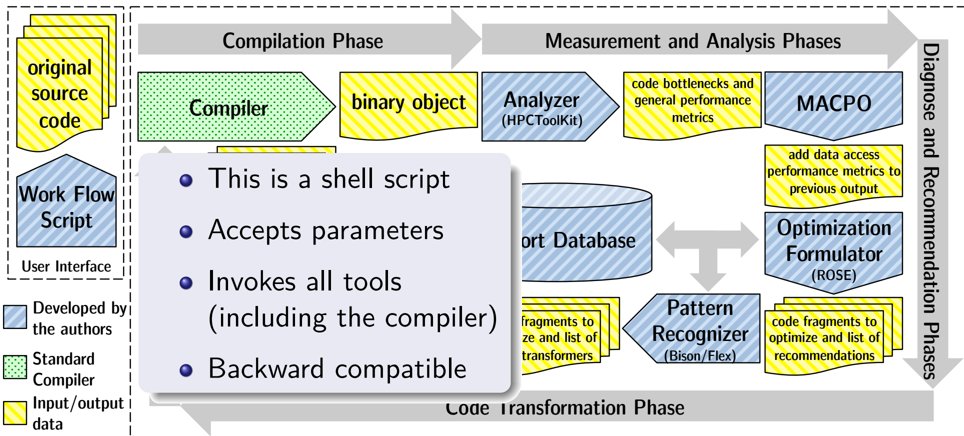
Compilation Phase
Measurement and Analysis Phases
Diagnose and Recommendation Phases
Code Transformation Phase
Code Integration Phase

original source code
Work Flow Script
User Interface

Compiler
binary object
Analyzer (HPCToolKit)
code bottlenecks and general performance metrics
MACPO
add data access performance metrics to previous output
Optimization Formulator (ROSE)
code fragments to optimize and list of recommendations
Pattern Recognizer (Bison/Flex)
code fragments to optimize and list of code transformers
Transformer (PIPS/ROSE)
optimized code fragments
Integrator (ROSE)
optimized source code
Support Database

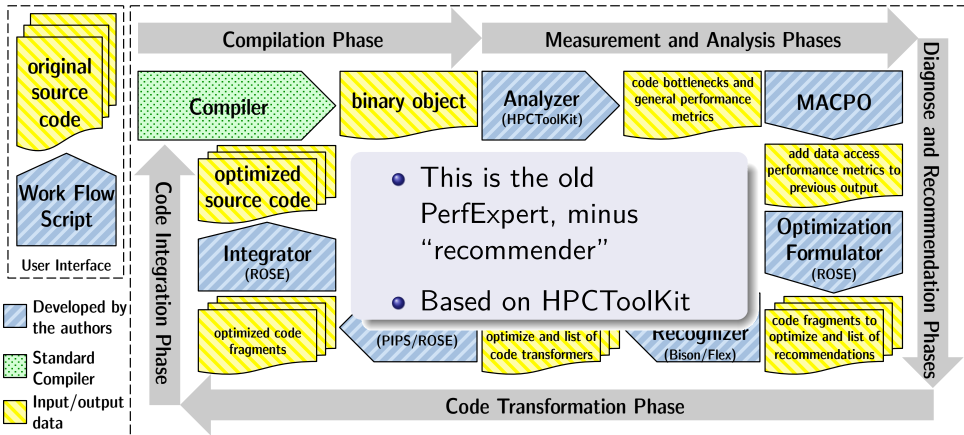
Developed by the authors
Standard Compiler
Input/output data

How PerfExpert does that: Work Flow Script



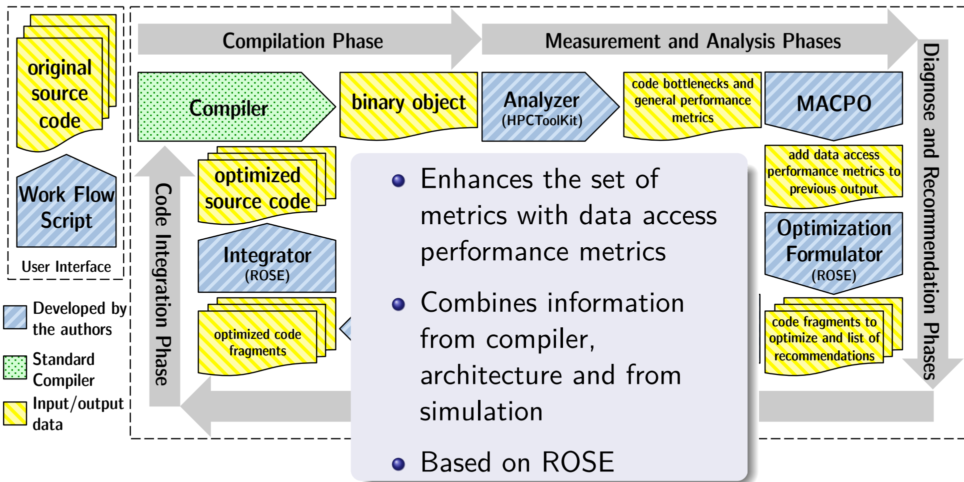
- This is a shell script
- Accepts parameters
- Invokes all tools (including the compiler)
- Backward compatible

How PerfExpert does that: Analyzer



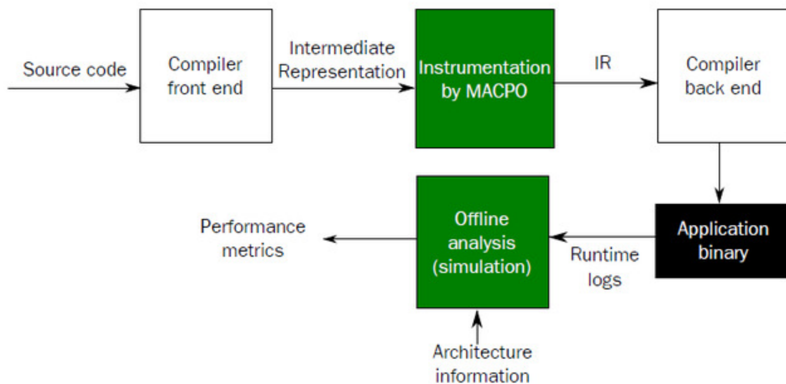
- This is the old PerfExpert, minus "recommender"
- Based on HPCToolKit

How PerfExpert does that: MACPO



- Enhances the set of metrics with data access performance metrics
- Combines information from compiler, architecture and from simulation
- Based on ROSE

How PerfExpert does that: MACPO

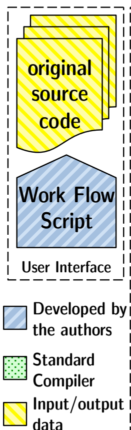


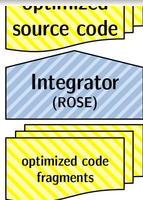
How PerfExpert does that: Optimization Formulator

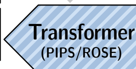
- Loads performance metrics on the Support Database
- Runs all *"recommendation selection functions"*
- Concatenates and ranks the list of recommendations
- Extracts code fragments identified as bottlenecks
- Based on ROSE
- **Extendable:** accepts user-defined performance metrics
- **Extendable:** it is possible to write new *"recommendation selection functions"* (SQL query)

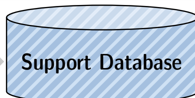
How PerfExpert does that: Support Database

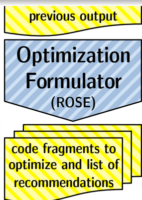
- This is a SQLite database
- Stores the list of *"recommendation selection functions"*, *"pattern recognizers"* and *"code transformers"*
- Engine to run the *"recommendation selection functions"*











Code Integration Phase

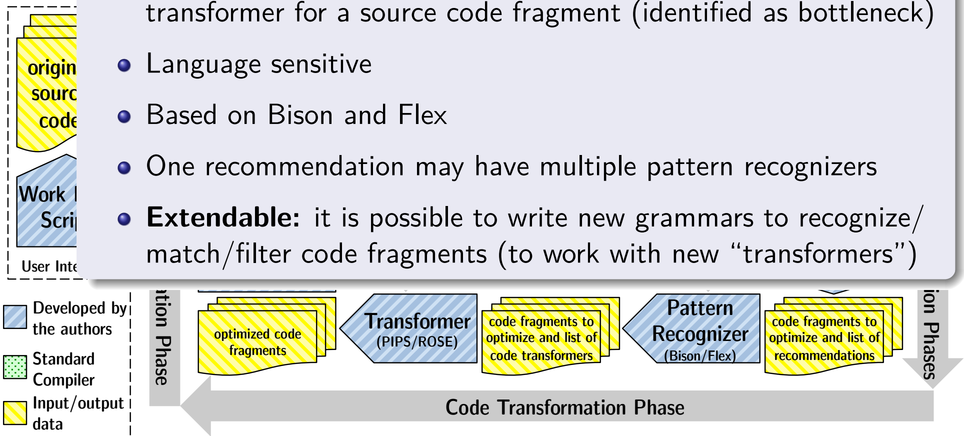
code fragments to optimize and list of code transformers

Code Transformation Phase

ecommendation Phases

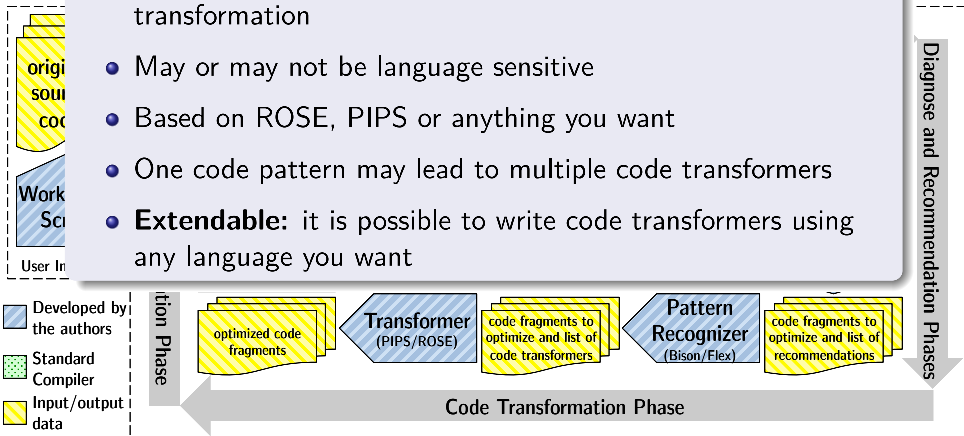
How PerfExpert does that: Pattern Recognizer

- Acts as a "filter" trying to find (match) the right code transformer for a source code fragment (identified as bottleneck)
- Language sensitive
- Based on Bison and Flex
- One recommendation may have multiple pattern recognizers
- **Extendable:** it is possible to write new grammars to recognize/match/filter code fragments (to work with new "transformers")

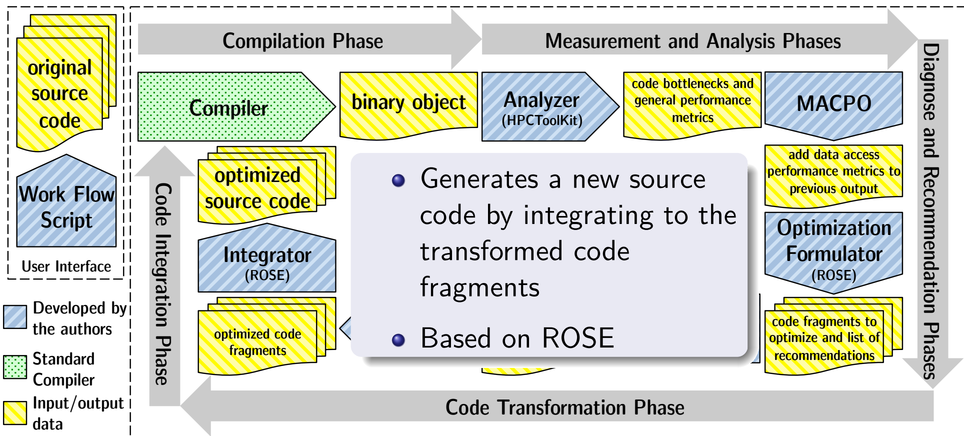


How PerfExpert does that: Transformer

- Implements the recommendation by applying source code transformation
- May or may not be language sensitive
- Based on ROSE, PIPS or anything you want
- One code pattern may lead to multiple code transformers
- **Extendable:** it is possible to write code transformers using any language you want



How PerfExpert does that: Integrator



- Generates a new source code by integrating to the transformed code fragments
- Based on ROSE

How PerfExpert does that: Key Points

Why is this performance optimization "architecture" strong?

- Each piece of the tool chain can be updated/upgraded individually
- It is flexible: you can add new metrics as well as plug new tools to measure application performance
- **It is extendable**: new recommendations, transformations and strategies to select recommendations
- Multi-language, **multi-architecture**, open-source and built on top of well-established tools (HPCToolKit, ROSE, PIPS, etc.)
- Easy to use and lightweight!

Agenda

1. Introduction
2. The User Perspective
3. Hands-on
4. PerfExpert Architecture
5. Conclusions

Conclusions

- This is the first end-to-end open-source performance optimization tool (as far as we know)
- It will become more and more powerful as new recommendations, transformations and features are added
- Different from (most of) the available performance optimization tools, there is no "big code" (to increase in complexity until it become unusable or too hard to maintain)

TACC THE UNIVERSITY OF TEXAS AT AUSTIN

Next Steps

Major Goals

- Improve analysis based on the data access (in progress)
- Increase the number of recommendations and possible code transformations (continuously)
- New algorithms for recommendations selection (in progress)
- Add support to MIC architecture (in progress)
- Add support to MPI-related recommendations (medium term)
- Add support to MPI-related code transformations (long term)

Next Steps

Minor Goals

- Support "Makefile"-based source code/compilation tree (done!)
- Make the required packages installation process easier (done!)
- Add a test suite based on established benchmark codes (in progress)
- Easy-to-use interface to manipulate the support database (medium term)

TACC THE UNIVERSITY OF TEXAS AT AUSTIN

We Are Ready To Help You

- The group of people developing PerfExpert is ready to help you!
- There are several other folks at TACC who also use PerfExpert and will be glad to help users get started
- Do not be shy, send us an email, we are here for that!
  - http://www.tacc.utexas.edu/perfexpert
  - fialho@utexas.edu
- We will also be happy to help you install PerfExpert on your system

Thank You

THE UNIVERSITY OF
TEXAS
AT AUSTIN