Building Your Own Performance Tools

Sameer Shende

Department of Computer and Information Science,
University of Oregon
sameer@cs.uoregon.edu
http://www.cs.uoregon.edu/research/paracomp/tau

Motivation

To discuss issues in instrumentation, measurement, analysis that highlight the choices available for building new tools for evaluating the performance of applications.

Outline

- Introduction to performance evaluation
- Instrumentation techniques
- Measurement techniques
  - Profiling
  - Tracing, synchronization issues
- Analysis techniques
  - Visualization of performance data
- Problems
- Conclusions

Introduction

- Understanding the behavior of parallel programs
  - Performance profiling: What is the relative contribution of routines?
  - Tracing: When do events take place?
  - Bottleneck detection: Where do bottlenecks lie?
  - Debugging: How can I correct the problem?

Understanding Application Performance

- **instrumentation** or modification of the program to generate performance data
- **measurement** of interesting aspects of execution
- **analysis** of the performance data.

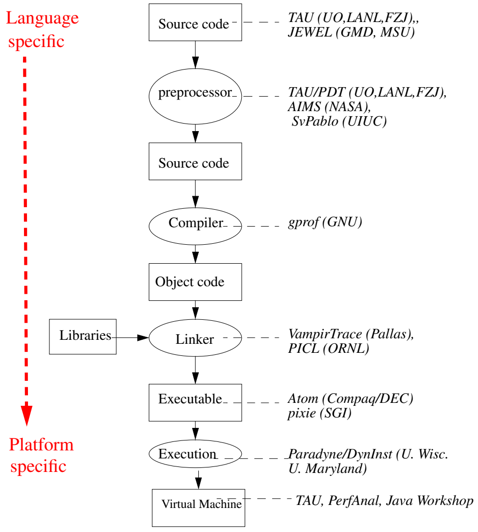
Instrumentation

- **What** is an event?
- **When** does an event get triggered?
- **How** do we add instrumentation to the program?

When does an event get triggered?

- When some point is reached during an execution
  - breakpoint/watchpoint
  - synchronization operation
  - routine entry/exit
- When some internal condition is satisfied
  - Interrupt (time/counts) for sampling
- When some external condition is satisfied
  - signal by debugger/user

When is instrumentation added?



Instrumentation Approaches

- Manual source code instrumentation
  - Recompile the application
  - Instrumentation API
- Preprocessor
  - Source -to- source transformation
  - Requires a parser for each language
- Compiler
  - Access to code mappings
- Library level instrumentation
  - Interposition libraries: wrappers & callbacks
- Binary Instrumentation
  - Binary rewriting
  - Runtime instrumentation
- Virtual Machine instrumentation

Instrumentation

- Which is the best instrumentation approach?
- Is a combination better in some cases?
- Simplicity vs. Flexibility?

Multi-Level Instrumentation : Example

- Multi-language applications (Java, C++, C, Fortran)
- Hybrid execution models (Java threads, MPI)
- JNI/native Java implementations of MPI Java Interface
  - Java Virtual Machine Profiler Interface (JVMPI)
  - Java Native Interface (JNI)
  - MPI Profiling Interlace

Java Virtual Machine Profiler Interface (JVMPI)

- Profiling Hooks into the Virtual Machine
- In-process profiling agent instruments Java application
- No changes to the Java source code, bytecode, or the executable code of the JVM
- Two-way call interface
- Profiler agent is a shared object (libTAU.so) loaded at runtime
- Agent registers events to the JVMPI
- JVMPI notifies events to the agent at runtime
- Agent uses JNI to invoke JVMPI control routines

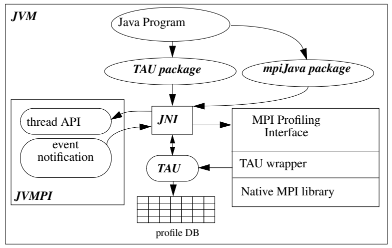
JVMPI Events

- Method transition events triggered at method entry and exits
- Memory events triggered when an object is allocated, moved, or deleted
- Heap arena events triggered when an arena is created or destroyed
- Garbage collection start and finish events
- Loading and unloading in memory events for classes and compiled methods
- JNI global and weak global reference allocation and deallocation events
- Monitor events for contended Java and raw monitors triggered when a thread attempts to enter, actually enters, or exits a monitor that is accessed by more than one thread
- Monitor wait events triggered when a thread is about to wait or finishes waiting on an object
- Thread start and end events when a thread starts or stops executing in the virtual machine
- Events that request a dump or resetting of the profiling data gathered by the in-process profiling agent
- Virtual machine initialization and shutdown events

Agent JVMPI interaction

- create a daemon thread in the virtual machine
- enable or disable the notification of an event
- enable, disable or force a garbage collection in the virtual machine
- obtain information regarding the current method call stack trace for a given thread
- obtain the accumulated CPU time consumed by the current thread
- obtain information about the object where a method took place
- get or set a pointer-sized thread-local storage data structure that can be used to record per-thread profiling data
- create or destroy a raw monitor. Raw monitors are not associated with Java objects and can be used by the profiler agent to maintain consistency of multi-threaded profiling data
- enter, exit or wait on a raw monitor for mutual exclusion. It can also notify all threads that are waiting on a raw monitor or specify a time-out period while waiting
- resume or suspend a thread
- exit the virtual machine

Integration of Multi-Level Instrumentation APIs



- Common TAU database for multiple sources

Outline

- Introduction to performance evaluation
- Instrumentation techniques
- Measurement techniques
  - Profiling
  - Tracing, synchronization issues
- Analysis techniques
  - Visualization of performance data
- Problems
- Conclusions

Measurement : Profiling

- shows summary statistics of performance **metrics**
  - CPU time spent in a routine
  - no. of secondary data cache misses for a statement
  - number of profiled routines invoked by a routine...
- presented as sorted lists showing **contribution** of routines
- implemented by **sampling** or **measured** process timing

Profiling Techniques

- sampling (PC/Callstack)
  - time based (hardware interval timer)

    `prof, gprof (GNU)`
  - hardware performance counters based (after *n* instructions, data cache misses...)

    `SpeedShop (SGI), PCL(FZJ), PAPI (UTK)`

➠ Estimates profile, low overhead

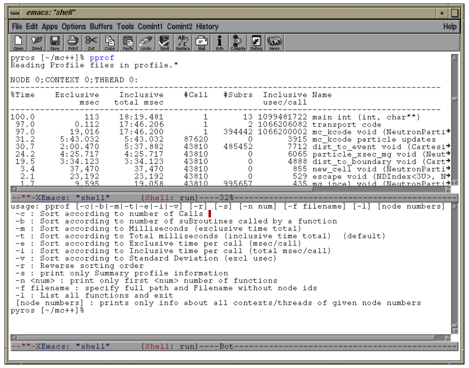
- measured process timing
  - routine entry/exit

    `TAU (U. Oregon, LANL, FZJ)`

➠ Accurate, overhead depends on frequency of invocation

Example of Profiling using TAU

- pprof sorts lists of performance metrics



Example: TAU Profiling Package

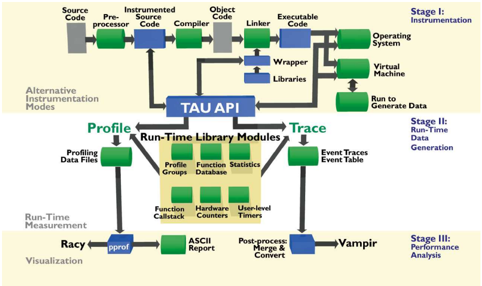
- RACY

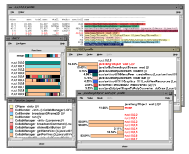
Tracing

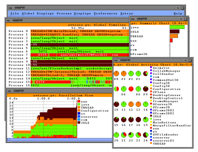
- tracing highlights the temporal aspect of performance variations, showing **when** and **where** in the code performance is achieved
- logging events (routine transitions/messages/user-def.)
  - event identifier
  - timestamp when the event occurred
  - where it occurred (node, context, thread ids)
  - optional field of event specific information
  - plus, event headers (event characteristics)

Architecture of TAU

Tuning and Analysis Utilities



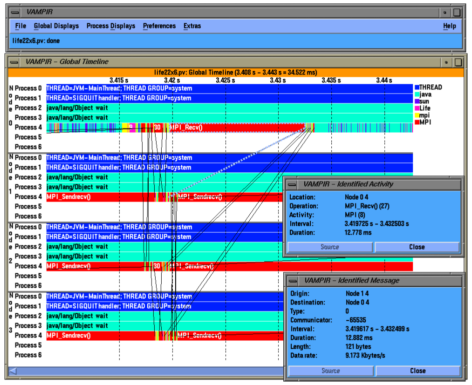




Example

- Tracing: Visualization in Vampir [http://www.pallas.de]

```
% prunjava 4 Life
% tau_merge tautrace*.trc Life.trc
% tau_convert -vampir Life.trc tau.edf Life.pv
% vampir Life.pv
```



Analysis of Performance Data

- Pablo (U. Illinois, Urbana)
  - User-directed analysis using performance data transformation modules that are interconnected
  - http://www-pablo.cs.uiuc.edu/
- Vampir (FZJ, Pallas GmbH)
  - Commercial trace visualization tool
  - http://www.pallas.de
- ParaGraph (NCSA, UIUC)
  - Rich set of visualizations, extensible
  - http://www.ncsa.uiuc.edu/

Problems...

- how do we profile/trace in the presence of
  - optimizations (PETE/C++, ZPL)
  - code transformations (Opus/HPF, Fortran-D)?
- how can we compensate for the perturbation caused by the instrumentation?
- how can we map performance data between layers?
- how can we produce meaningful visualizations that can scale to thousands of processors?
- how can we show performance data at a level of abstraction that the user understands?

Conclusions

- Effective choices
  - instrumentation
  - measurement
  - analysis
- Bridging the “semantic-gap”
- Problems and constraints

Unless tools can present performance data in ways that are meaningful to the user, and are consistent with the user’s mental model of abstractions, their success will be limited.