Motivation

- What communities can benefit from automated collection of performance data? A (partial) listing:
  - Application Developers
  - Performance Analysts/Engineers
  - Computational Resource Providers
  - Educational Audience
  - …etc
Motivation, cont’d

- Some factors impacting cluster application performance:
  - CPU capabilities
  - Memory subsystem
  - Network hardware/software stack
  - Compilers
  - Libraries
  - I/O subsystem

The Basic Tuning Process

- Select “best” compiler flags
- Select/interface with “best” libraries
- Measure
- Validate
- Hand-tune (routine/loop-level tuning)
- ... iterate
Performance Analysis/Tuning in Practice

- Observation: many application developers don’t use performance tools at all (or rarely)
- Why?
  - Learning curve can be steep
  - Results can be difficult to understand
  - Investment (time) can be substantial
  - Maturity/availability of various tools
  - Not everyone is a computer scientist

Automated Performance Tools for Linux Clusters

- Not a lot has been available in standard distributions
  - This is changing! Several high-quality products emerging from numerous groups
- The remainder of this talk describes three separate but related examples of ongoing work at UIUC/UTK
What Does “Automatic” Mean???

- Can be thought of in many ways, for example:
  - Collection of performance-related data without need for user intervention
    - At the application level
    - At the system level
  - Presentation, post-processing
  - Analysis, diagnostics
  - Prediction

PerfSuite

- Design Goals
  - Remove the barriers to the initial steps of performance analysis (don’t make it hard)
  - Separate data collection from presentation
  - Machine-independent representation
  - Holistic viewpoint: compiler, hardware counters, message-passing, etc.
  - Focus on the “Big Picture”
- A primary goal is to provide an “entry point” to help decide how to proceed
- WWW:
  - [http://perfsuite.ncsa.uiuc.edu/](http://perfsuite.ncsa.uiuc.edu/)
PerfSuite Hardware Counter Utilities

- Four performance counter-related utilities:
  - psconfig - configure / select performance events
  - psinv - query events and machine information
  - psrun - generate raw counter or statistical profiling data from an unmodified binary
  - psprocess - pre- and post-process data
- There are also user-accessible libraries (API) for finer control

PerfSuite “psrun”

- Hardware performance counting and profiling with unmodified dynamically-linked executables
- Available for x86, x86-64, and ia64
- POSIX threads support
- Automatic multiplexing
- Can be used with MPI
- Optionally collects resource usage
- Supports all PAPI standard events
- Input/Output = XML documents (can request plain text)
PerfSuite “psprocess”

- Output content and style customizable
- Generated using XML Transformations

$ psrun a.out
$ mpirun -np 128 psrun a.out
$ psprocess psrun.xml

Case Study: Automatic Performance Collection on NCSA Linux Clusters

- NCSA transition (c. 2000) from shared-memory “traditional” supercomputers to cluster technology is a major shift:
  - Does it translate in practice to high-performance cycles delivered?
  - What is the percentage of users making efficient use of the resource?
  - How can knowledge improve services (i.e., feedback loop)?
Project Definition (Jan 2003)

- Measure the aggregate performance of all user applications on Linux clusters, (new) IBM p690, and (retiring) Origin 2000 systems
- Unmodified binaries – no impact on or effort required of users
- Operational within existing job management system – no “special queues” or contacts. Avoid self-selecting users.
- In-place and operational by March ’03 in order to gather sufficient data for NSF reporting by late summer.

Project Implementation

- Focus narrowed to Linux clusters and PerfSuite used to gather the performance data 24x7
- Implemented as a wrapper to the standard MPI launch commands
  - Could be extended to serial applications relatively easily
- Integrated with system support efforts (file management) and a relational database back-end
- By Supercomputing ’03, nearly 5 million records of performance data gathered
Job Scale (time, processors)

Pentium III FY03

Floating Point Performance

- 10% of peak or greater: 12% on Pentium III, 7% on Itanium
- Note: vector/SIMD instructions not counted as FP_INS / FP_OPS by PAPI
Are Performance Counters Enough?

- Performance counters provide valuable information required for an analysis like this, but:
  - They only provide a CPU-centric view
  - They are not directly comparable across architectures
  - There is no single metric suitable for determining whether an arbitrary application is making “good use” of a machine
  - Extensions are being planned to address “off-chip” performance factors

What is DynaProf?

- A portable tool to dynamically and selectively instrument serial and parallel programs for the purpose of performance analysis and application understanding.
Why the “Dyna” in DynaProf?

- Dynamic Instrumentation means the application's object code is modified at run-time.
- The instrumentation is contained in simple shared libraries DynaProf calls “probes”.
- Object code to those functions is generated and then inserted into the program’s address space.
- DPCL and DynInst do all the dirty work.

DPCL vs. DynInst

- **DPCL:**
  - Based on an early version of DynInst.
  - Supports Asynch./Sync. Operation.
  - Provides functions for getting data back to tool.
  - Integrated with IBM's Parallel Operating Environment.
  - It's Stale! And it requires a working rsh/ssh. (AIX only)

- **DynInst:**
  - Shared libraries, Loops, Basic blocks, Arbitrary locations
  - Provides breakpoints, CFG
  - Single process model.
  - Actively supported on many platforms.
History of Dynamic Instrumentation

- Popularized by James Larus with EEL: An Executable Editor Library at U. Wisc.
- Technology matured by Dr. Bart Miller and Dr. Jeff Hollingsworth at U. Wisc.
  - DynInst Project at U. Maryland & U. Wisconsin
  - IBM's DPCL: A DynInst Mutation

DynaProf Goals

- Make collection of run-time performance data easy!
- Avoiding the instrumentation/recompilation cycle.
- Avoiding interference with compiler optimization.
- Using the same tool with different probes.
- Providing useful and meaningful probe data.
- Providing different kinds of probes.
- Allow easy development of custom probes.
- Providing complete language independence.
Dynaprof Probes

- papiprobe
  - Measure any combination of PAPI presets and native events
- wallclockprobe
  - Highly accurate elapsed wallclock time in microseconds.
- perfometerprobe
  - Visualize hardware counter traces in pseudo real-time.
- tauprobe
  - Support all TAU measurement methodologies including timing, memory tracking, hardware counters and call stack tracing.
- vmonprobe
  - Statistical profiling of hardware counter events ala gprof.

Dynaprof Probe Design

- Probes export a few functions with loosely standardized interfaces.
- Easy to roll your own.
  - If you can code a timer, you can write a probe.
- Dynaprof detects thread model and will load a special version of the probe.
- The probes dictate how the data is recorded and visualized.
Papiprobe & Wallclockprobe

- These are well tested probes.
- papiprobe
  - Counts hardware counters using PAPI, either PAPI presets or Native events.
  - Supports counter multiplexing:
    - Not good for fine grained instrumentation.
- wallclockprobe
  - Counts microseconds using real time cycle counter available on each platform.

Reporting Probe Data

- The wallclock and PAPI probes produce very similar data.
- Both use a parsing script written in Perl.
  - wallclockrpt <file>
  - papiproberpt <file>
- Produce 3 profiles
  - Inclusive: $T_{function} = T_{self} + T_{children}$
  - Exclusive: $T_{function} = T_{self}$
  - 1-Level Call Tree: $T_{child} = Inclusive T_{function}$
Instrumenting SWIM for IPC

$./dynaprof
(dynaprof) load tests/swim
(dynaprof) list DEFAULT_MODULE
swim.F
libm.so.6
libc.so.6
(dynaprof) list swim.F
MAIN
  initial
  calc1
  calc2
  calc3
(dynaprof) use probes/papiprobe PAPI_TOT_CYC, PAPI_TOT_INS
Module papiprobe.so was loaded.
Module libpapi.so was loaded.
Module libperfctr.so was loaded.
(dynaprof) instr function swim.F calc*
swim.F, inserted 4 instrumentation points
(dynaprof) instr
  calc1
  calc2
  calc3
  calc3z
(dynaprof) run
papiprobe: output goes to /home/mucci/dynaprof/tests/swim.1671

Swim Benchmark: Instructions & Cycles

Exclusive Profile of Metric PAPI_TOT_INS.

<table>
<thead>
<tr>
<th>Name</th>
<th>Percent</th>
<th>Total</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL</td>
<td>100</td>
<td>1.723e+09</td>
<td>1</td>
</tr>
<tr>
<td>calc2</td>
<td>38.28</td>
<td>6.598e+08</td>
<td>120</td>
</tr>
<tr>
<td>calc1</td>
<td>32.31</td>
<td>5.567e+08</td>
<td>120</td>
</tr>
<tr>
<td>calc3</td>
<td>22.33</td>
<td>3.847e+08</td>
<td>118</td>
</tr>
<tr>
<td>unknown</td>
<td>7.084</td>
<td>1.221e+08</td>
<td>1</td>
</tr>
</tbody>
</table>

Inclusive Profile of Metric PAPI_TOT_INS.

<table>
<thead>
<tr>
<th>Name</th>
<th>Percent</th>
<th>Total</th>
<th>SubCall</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL</td>
<td>100</td>
<td>1.723e+09</td>
<td>0</td>
</tr>
<tr>
<td>calc2</td>
<td>39.42</td>
<td>6.793e+08</td>
<td>168</td>
</tr>
<tr>
<td>calc1</td>
<td>35.28</td>
<td>6.08e+08</td>
<td>180</td>
</tr>
<tr>
<td>calc3</td>
<td>22.87</td>
<td>3.942e+08</td>
<td>165</td>
</tr>
</tbody>
</table>

1-Level Inclusive Call Tree of Metric PAPI_TOT_INS.

<table>
<thead>
<tr>
<th>Parent/Child</th>
<th>Percent</th>
<th>Total</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL</td>
<td>100</td>
<td>1.723e+09</td>
<td>1</td>
</tr>
<tr>
<td>calc1</td>
<td>100</td>
<td>6.08e+08</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>frsav</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.02065</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.03132</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.05911</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.06434</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.9013</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.03132</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.03132</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.05356</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.05079</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6.813</td>
<td>120</td>
</tr>
<tr>
<td>calc2</td>
<td>100</td>
<td>1.144e+09</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>frsav</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.01848</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.02804</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.02804</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.07762</td>
<td>120</td>
</tr>
</tbody>
</table>

Exclusive Profile of Metric PAPI_TOT_CYC.

<table>
<thead>
<tr>
<th>Name</th>
<th>Percent</th>
<th>Total</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL</td>
<td>100</td>
<td>3.181e+09</td>
<td>1</td>
</tr>
<tr>
<td>calc2</td>
<td>34.85</td>
<td>1.108e+09</td>
<td>120</td>
</tr>
<tr>
<td>calc1</td>
<td>33.48</td>
<td>1.065e+09</td>
<td>120</td>
</tr>
<tr>
<td>calc3</td>
<td>26.1</td>
<td>8.301e+08</td>
<td>118</td>
</tr>
<tr>
<td>unknown</td>
<td>5.568</td>
<td>1.771e+08</td>
<td>1</td>
</tr>
</tbody>
</table>

Inclusive Profile of Metric PAPI_TOT_CYC.

<table>
<thead>
<tr>
<th>Name</th>
<th>Percent</th>
<th>Total</th>
<th>SubCall</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL</td>
<td>100</td>
<td>3.181e+09</td>
<td>0</td>
</tr>
<tr>
<td>calc2</td>
<td>35.98</td>
<td>1.144e+09</td>
<td>1680</td>
</tr>
<tr>
<td>calc1</td>
<td>35.61</td>
<td>1.133e+09</td>
<td>1800</td>
</tr>
<tr>
<td>calc3</td>
<td>26.88</td>
<td>8.55e+08</td>
<td>1652</td>
</tr>
</tbody>
</table>

1-Level Inclusive Call Tree of Metric PAPI_TOT_CYC.

<table>
<thead>
<tr>
<th>Parent/Child</th>
<th>Percent</th>
<th>Total</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL</td>
<td>100</td>
<td>3.181e+09</td>
<td>1</td>
</tr>
<tr>
<td>calc1</td>
<td>100</td>
<td>1.133e+09</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>frsav</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.03432</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.07356</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.06630</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.07390</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.71890</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.16460</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.03407</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.18670</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.06067</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.22</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.03979</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.03008</td>
<td>120</td>
</tr>
<tr>
<td>calc2</td>
<td>100</td>
<td>1.144e+09</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>frsav</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.03382</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.03222</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.03554</td>
<td>120</td>
</tr>
</tbody>
</table>

1-Level Inclusive Call Tree of Metric PAPI_TOT_CYC.

<table>
<thead>
<tr>
<th>Parent/Child</th>
<th>Percent</th>
<th>Total</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL</td>
<td>100</td>
<td>3.181e+09</td>
<td>1</td>
</tr>
<tr>
<td>calc1</td>
<td>100</td>
<td>1.133e+09</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>frsav</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.03432</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.07356</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.06630</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.07390</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.71890</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.16460</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.03407</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.18670</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.06067</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.22</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.03979</td>
<td>120</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.03008</td>
<td>120</td>
</tr>
</tbody>
</table>
### Swim Benchmark: Instructions per Cycle

<table>
<thead>
<tr>
<th>Name</th>
<th>Percent</th>
<th>Total</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL</td>
<td></td>
<td>1.72e+05</td>
<td></td>
</tr>
<tr>
<td>calc</td>
<td>38.34%</td>
<td>6.54e+05</td>
<td>120</td>
</tr>
<tr>
<td>unknown</td>
<td>22.33%</td>
<td>3.88e+05</td>
<td>120</td>
</tr>
<tr>
<td>fsav</td>
<td>0.05%</td>
<td>1.10e+05</td>
<td>120</td>
</tr>
</tbody>
</table>

**Calc2**: 0.59IPC  
**Calc1**: 0.53IPC  
**Calc3**: 0.46IPC

### DynaProf Links

- [http://icl.cs.utk.edu/papi](http://icl.cs.utk.edu/papi)
- [http://www.cs.utk.edu/~mucci/dynaprof](http://www.cs.utk.edu/~mucci/dynaprof)
- [http://www.dyninst.org](http://www.dyninst.org)
- [http://www.paradyn.org](http://www.paradyn.org)
- [http://www.cs.wisc.edu/~larus/eel.html](http://www.cs.wisc.edu/~larus/eel.html)
- [http://www.cs.uoregon.edu/research/paracomp/tau/tautools](http://www.cs.uoregon.edu/research/paracomp/tau/tautools)
KOJAK / CUBE

- Collaborative research project between
  - Forschungszentrum Jülich
  - University of Tennessee
- Automated support of
  - Application instrumentation
  - Off-line analysis of collected data
  - Comparison of multiple experiments
- WWW
  - http://www.fz-juelich.de/zam/kojak/
  - http://icl.cs.utk.edu/kojak/

Automatic Performance Analysis

- Transformation of low-level performance data
- Take event traces of MPI/OpenMP applications
- Search for execution patterns
- Calculate mapping
  - Problem, call path, system resource $\Rightarrow$ time
- Display in performance browser
Example: Late Sender

KOJAK / CUBE Architecture
KOJAK Architecture

- Automatic instrumentation
  - Profiling interface PGI compiler
- Abstract representation of event trace
  - Simplified specification of performance problem
  - Simplified extension of predefined problems
- Analysis
  - Automatic classification and quantification of performance behavior
  - Automatic comparison of multiple experiments
- Presentation
  - Navigating / browsing through performance space
  - Can be combined with time-line display

CUBE Uniform Behavioral Encoding

- Abstract data model of performance behavior
- Portable data format (XML)
- Documented C++ API to write CUBE files
- Generic presentation component
- Performance-data algebra
CUBE data model

- Most performance data are mappings of aggregated metric values onto program and system resources
  - Performance metrics
    - Execution time, floating-point operations, cache misses
  - Program resources (static and dynamic)
    - Functions, call paths
  - System resources
    - Cluster nodes, processes, threads
- Hierarchical organization of each dimension
  - Inclusion of metrics, e.g., cache misses $\subseteq$ memory accesses
  - Source code hierarchy, call tree
  - Nodes hosting processes, processes spawning threads

CUBE GUI

- Design emphasizes simplicity by combining a small number of orthogonal features
- Three coupled tree browsers
- Each node labeled with metric value
- Limited set of actions
- Selecting a metric / call path
  - Break down of aggregated values
- Expanding / collapsing nodes
  - Collapsed node represents entire subtree
  - Expanded node represents only itself without children
- Scalable because level of detail can be adjusted
Which type of situation caused the problem?
Where in the source code?
Which call path?
Which process/thread?
How severe is the problem?

Performance Algebra

- Cross-experiment analysis
  - Different execution configuration
  - Different measurement tools
  - Different random errors
- Arithmetic operations on CUBE instances
  - Difference, mean, merge
  - Obtain CUBE instance as result
  - Display it like ordinary CUBE instance
Comparing domain decompositions of CX3D

- Difference between 4 x 4 and 16 x 1 decompositions
- Sunken relief indicates where 4 x 4 is more efficient
- 4 x 4 has better overall performance

In spite of better overall performance, the late sender problem became worse.

Call tree shows where