Dynamic Tuning of Parallel Programs with DynInst

Anna Morajko, Tomàs Margalef, Emilio Luque
Universitat Autònoma de Barcelona
Content

- Our goals - dynamic performance tuning
- Steering loop
  - Dynamic tracing
  - Performance analysis
  - Dynamic tuning
- DynInst
- Application development framework
- Conclusions and future work
The objective is to:

- create a system that improves performance of parallel applications during their execution without recompiling and rerunning them

Our approach to performance improvement:

- Phases:
  - Tracing
  - Performance analysis
  - Tuning
- All done during run-time
Dynamic performance tuning

- Application development
- Source
- Application
- Execution
- Instrumentation
- Events
- Tracing
- Performance analysis
- Suggestions
- Tuning
- Modifications
- Problem / Solution

User

Tool
Dynamic performance tuning
Dynamic performance tuning

Requirements:

- all optimization phases done during run time
- improve performance changing execution of an application
- without source code access, re-compile, re-link and re-run

**Dynamic instrumentation (DynInst)**

instrumentation and tracing

tuning
Dynamic performance tuning

• **Analysis**
  • application dependent performance model:
    - what to measure
    - how to detect bottlenecks
    - how to resolve it
  • simple analysis – decisions must be taken in a short time

• **Execution modifications**
  • what can be changed - complexity when performing application changes without knowing its internal code
  • and how it can be changed

• **Intrusion**
  • minimization of intrusion caused by system modules, instrumentation, event generation and communication
Content

• Our goals - dynamic performance tuning

• **Steering loop**
  - Dynamic tracing
  - Performance analysis
  - Dynamic tuning

• DynInst

• Application development framework

• Conclusions and future work
Dynamic tracer

Goals of the dynamic tracing tool:

- control of the distributed application
- application instrumentation management
- event generation
- intrusion minimization
- portability and extensibility

Version:

- implementation in C++ for PVM based applications
- Sun Solaris 2.x / SPARC
Dynamic tracer

Distributed application control:

- Process control with PVM Tasker service
- VM control with Hoster service
- Master tracer / Slave tracers
- Clock synchronization (event timestamp with global order)
Dynamic tracer

Application instrumentation management:

• Instrumentation
  • inserted into a set of selected functions (configurable)
  • this set can be changed during execution (insert, remove)

• Different snippets:
  • Event generation (function entry, exit)
  • Line number information (function call points)

• Snippets call tracing library dynamically loaded into a running task
Dynamic tracer

Event generation:

- When - global timestamp
- Where - task
- What - type of event
- Additional information (i.e. function call params, source code line number)
Dynamic tracer

Event Collection:
- Centralized event collector receives events
- Global ordering
- Flow of ordered events is passed to the analyzer
- To consider, e.g. event buffering, periodical flush
Performance analysis

Clue questions:
• What should/should not happen during execution (what kind of problems, performance bottlenecks)?

• What events are needed (what should be collected to help discover what really happened)?

• How to deduce what really happened (finding bottlenecks and causes)?

• How to improve performance (finding solutions)?
Performance analysis

- Performance model
  - some fixed model for an application - need to determine desired performance characteristics
  - model determines what to measure

- Analyzing events that come from event collector

- Evaluating the model

- Giving solutions - the best possible changes that will improve the performance

- Sending solutions to tuner
Dynamic tuning

What can be changed?

• parameters: changing parameter value
  • application contains well-known variables
  • application is prepared for changes

• set of pre-defined snippets
  • choosing strategy – function replacement
  • changing order of functions

• dynamic snippet creation
  • it depends on the solutions generated by analyzer
  • very difficult
Dynamic tuning

Where can be changed?
- All tuners receives changes (i.e. SPMD)
- One concrete tuner that tunes one concrete process (i.e. MW)
Steering loop

Machine 1
- Task₁
  - lib
  - Tracer
  - pvmd
  - Change instrumentation
  - Events
  - Apply solutions to Tuner

Machine 2
- Task₁
  - lib
  - Tuner
  - Performance Tuning Protocol
  - Change instrumentation
  - Events

Machine 3
- Analyzer

Connections:
- Machine 1 to Machine 2: Apply solutions
- Task₁ to Tuner: Performance Tuning Protocol
- Tuner to Task₁: Apply solutions

Content

• Our goals - dynamic performance tuning
• Steering loop
  ➢ Dynamic tracing
  ➢ Performance analysis
  ➢ Dynamic tuning
• DynInst
• Application development framework
• Conclusions and future work
DynInst

Two principal reasons for using DynInst:

• **instrumentation and tracing**
  • collect information about application behavior

• **tuning**
  • insert/remove/change code in order to improve application performance
DynInst

Instrumentation and tracing:

Tracer

... thread.loadLibrary("libLog.so");

BPatch_function f = image.findFunction(logEvent);

BPatch_funcCallExpr log (f);

BPatch_point point (image, pvm_send, BPatch_entry);

thread.insertSnippet (log, point);

PVM application

foo_1 (...
{
  pvm_send ();
}

pvm_send (params)
{
  ...
}

Snippet

log ()
{
  logEvent(format, params)
}

logEvent(format,...)
{
  // transfer event
}

Dynamic library (libLog.so)
DynInst

Event line number:

logLineNumber

PVM application

pvm_send(...)
{
}
...

numr = DIM/(nunw);
extra = DIM % (nunw);
pvm_pkint (x,1,2);
pvm_send (...);

numr = DIM/(nunw);
extra = DIM % (nunw);
pvm_pkint (x,1,2);
pvm_send (...);

numr = DIM/(nunw);
extra = DIM % (nunw);
pvm_pkint (x,1,2);
pvm_send (...);

numr = DIM/(nunw);
extra = DIM % (nunw);
pvm_pkint (x,1,2);
pvm_send (...);

numr = DIM/(nunw);
extra = DIM % (nunw);
pvm_pkint (x,1,2);
pvm_send (...);

numr = DIM/(nunw);
extra = DIM % (nunw);
pvm_pkint (x,1,2);
pvm_send (...);

numr = DIM/(nunw);
extra = DIM % (nunw);
pvm_pkint (x,1,2);
pvm_send (...);
Event line number:

```cpp
BPatch_function f=image.findFunction (logLine);
BPatch_funcCallExpr logLineNumber (f);

CallPointList cp ("pvm_send")
for each cp
    addr = cp.getAddress ();
    thread.getLineAndFile (addr);

    thread.insertSnippetBefore
        (logLineNumber, cp);
```

PVM application

```cpp
foo_1 (...)
{
    pvm_send ();
    pvm_recv ();
}
...
```

```cpp
logLine ()
{
    logLineNumber (params)
}
```
DynInst

changeParam (param, value)
{
    send_req (tuner, param, value);
}

recv_req (param, value)
{
    BPatch_variable variable =
        image.FindVariable (param);
    variable.writeValue (value);
}

main ()
{
    int tuneParam; int param;
    if (param != tuneParam)
        ...
}
DynInst

Problems when using DynInst:

- Line number
  - We need to scan all functions to be instrumented and additionally instrument them (before execution)
  - Insert snippet that logs line number (before function call)
  - Could be nice to access callee address (call stack??)

- Limited number of parameters that can be passed to library function call (max. 5 for Sun Solaris)

- Problems with scanning function call points (e.g. C++ mutatee with operator<<, function sleep)
  - BPatch_function::findPoint (subroutine)
  - BPatch_point::getCalledFunction ()
Content

• Our goals - dynamic performance tuning
• Steering loop
  ➢ Dynamic tracing
  ➢ Performance analysis
  ➢ Dynamic tuning
• DynInst
• Application development framework
• Conclusions and future work
**Framework** for parallel application development

- Based on patterns, e.g. master worker, pipeline, divide and conquer
- Support for dynamic tuning system:
  - Measure points, performance model, tuning points

<table>
<thead>
<tr>
<th>Parallel pattern</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Application Framework</td>
</tr>
<tr>
<td></td>
<td>Perf. model</td>
</tr>
<tr>
<td>Measure points</td>
<td>Tuning points</td>
</tr>
</tbody>
</table>

- Support for parallel application developer:
  - hides implementation details
  - developer provides only specific functionality
Dynamic performance tuning

User

Application Development

Application

Execution

Tracing

Performance analysis

Suggestions

Problem / Solution

Modifications

API

Application Framework

Instrumentation

Measure points

Events

Perf. model

Tuning points

Tool

UAB
Content

• Our goals - dynamic performance tuning
• Steering loop
  ➢ Dynamic tracing
  ➢ Performance analysis
  ➢ Dynamic tuning
• DynInst
• Application development framework
• Conclusions and future work
Conclusions

• Dynamic tuning
  • Design of dynamic tuning tool
  • Characteristics

• Steering loop
  • Design details
  • Implementation of dynamic tracer
  • Bases of performance analysis
  • Tuning methods

• How we use DynInst

• Parallel application development framework
Future work

• Tracer
  • event buffering, changing instrumentation on demand, performance measurements

• Analyzer
  • performance analysis

• Tuner
  • other solutions that can be applied when changing???

• Find and analyze more examples of what to measure, how analyze and what to change
Thank you very much 😊