The New Call Graph Based Performance Consultant

Trey Cain
cain@cs.wisc.edu

Computer Sciences Department
University of Wisconsin
1210 W. Dayton St.
Madison, WI 53706-1685
USA
The Performance Consultant (PC)

- Uses two main Paradyn technologies
  - Dynamic instrumentation
  - Automated bottleneck search
- Original version had difficulty searching large applications
- Our solution: direct PC search using application call graph
Outline

- Paradyn Basics
- Original Search Strategy
- Call Graph Based Search Strategy
- Dynamic Call Site Instrumentation
- Performance Comparison
- Conclusion
Paradyn Basics:
Resource Hierarchies

testutil.C
  debugA
dbgB

code
  main.C
    main
  vect.C
    vect::insert
dev::delete
dev::size

printstatus

dev::insert
dev::delete
dev::size

main.C

Machine
  Host1
    Process1
      Thread1
    Process2
      Thread1

Host2
  Process1
    Thread1
    Thread2

SyncObject
  Barrier
    b1
  Message
    g1
g2
  Semaphore
    sem1
  Spinlock
    spin1
Paradyn Basics: Resource Hierarchies

Example Focus: {/Code/testutil.C/printstatus, /Machine/host1/process1, /SyncObject }
Paradyn Basics: Performance Metrics

• Metrics are measurable performance characteristics such as CPU time, function calls, I/O bytes transferred

• Performance data collected for metric/focus pair

• Example metric/focus pairs:
  • cpu:{/Code/mod1/func1 }
  • msgs:{/Code/mod1/func1, /Machine/host1/proc4/thread2, /SyncObject/Message/1/0}
## Performance Consultant Basics

- **Why is the application running slowly?**
  - Test bottleneck hypotheses
    - CPU Bound?
    - I/O Wait Bound?
    - Synchronization Wait Bound?
    - Memory Bound?
  - Performance metric associated with each hypothesis
- **Which part of the application is slow?**
  - Isolates bottleneck to part of resource hierarchy

---

Cain@cs.wisc.edu

Call Graph Based Performance Consultant
Original PC Example

Top Level Hypothesis
- CPUBound
- SyncWaitBound
- I/OWaitBound
Original PC Example

Top Level Hypothesis

SyncWaitBound
I/OWaitBound

CPUBound
Original PC Example

Top Level Hypothesis

SyncWaitBound
I/OWaitBound

CPUBound

Module1.o
Module2.o
Module3.o
Module4.so
Original PC Example

Top Level Hypothesis

SyncWaitBound
I/OWaitBound

CPUBound

Module1.o
Module3.o
Module4.so

Module2.o
Original PC Example

Top Level Hypothesis

SyncWaitBound
I/OWaitBound

CPUBound

Module1.o
Module3.o
Module4.so

Module2.o

Function1
Function2
Function3
Function4
Original PC Example

Top Level Hypothesis

SyncWaitBound
I/OWaitBound

CPUBound

Module1.o
Module3.o
Module4.so

Function2
Function3
Function4

Module2.o

function1

Host1
Host2
Host3
Problem: Traversing the code hierarchy does not scale

- Search space too large: too many modules, too many functions
- Module instrumentation is not cheaper than instrumenting all of module’s functions
- Exclusive metrics are costly
- We would like to avoid excessive instrumentation
Too many modules and functions

Module 1

Module 2

Module 3

Code

...
Too many modules and functions

Module 1
Module 2
Module 3
Code

Function 1
Function 2
Function 3
...
Function 100

Module 1000
PC Timing Metrics

• Performance Consultant based on the idea that coarse grained instrumentation is cheaper than fine grained...

• But instrumenting a module has the same cost as instrumenting each function in the module individually.
Exclusive vs. Inclusive Metrics

```c
foo() {
    bar();
    car();
}
```

Exclusive Timer
Exclusive vs. Inclusive Metrics

```
foo() {
    bar();
    car();
}
```

Exclusive Timer
Exclusive vs. Inclusive Metrics

**Exclusive Timer**

foo() {
  startTimer(t)
  bar();
  startTimer(t)
  car();
  stopTimer(t)
}

**Inclusive Timer**

foo() {
  startTimer(t)
  bar();
  stopTimer(t)
  car();
  stopTimer(t)
}
Call Graph Based Performance Consultant

• Based on application’s call graph
• Code hierarchy search starts at function main, search continues to main’s children

• Advantages: Lots!
  • It’s Scalable: Natural hierarchical refinement from course grained search to fine grained search
  • Uses less costly inclusive metrics
  • Functions which are not part of call graph will never be instrumented
Call Graph Based PC Example

Top Level Hypothesis

SyncWaitBound

CPUBound

I/OWaitBound
Call Graph Based PC Example

Top Level Hypothesis

- SyncWaitBound
- I/OWaitBound
- CPUBound
- main
Call Graph Based PC Example

Top Level Hypothesis

- SyncWaitBound
  - I/OWaitBound

- CPUBound
  - main
    - a1
    - a2
    - a3
    - a4
Call Graph Based PC Example

Top Level Hypothesis

SyncWaitBound
I/OWaitBound

CPUBound

main

a1
a2
a3
a4
Call Graph Based PC Example

Top Level Hypothesis

SyncWaitBound

I/OWaitBound

CPUBound

main

a1

a2

a3

b1

b2

b3
Call Graph Based Performance Consultant

cain@cs.wisc.edu

Call Graph Based PC Example

Top Level Hypothesis

SyncWaitBound

I/OWaitBound

CPUBound

main

a1

a2

a4

a3

b1

b3

b2
Call Graph Construction

- Problem: targets of calls using function pointers and virtual functions are not statically determinable.

- Unknown callees in static call graph may cause blind spots in new PC search

- We resolve dynamic callee addresses at run time

- Strategy:
  - Build static call graph at program start
  - Fill in dynamic call graph on demand.
Dynamic Call Sites

• Characterized by keeping the address of a callee in a register or memory location

• New type of instrumentation necessary to determine callee

• Examples:

<table>
<thead>
<tr>
<th>Instruction Set</th>
<th>Call Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPS</td>
<td>jalr $t9</td>
</tr>
<tr>
<td>X86</td>
<td>call [%edi]</td>
</tr>
</tbody>
</table>
Call Site Instrumentation: Chain of Events

Paradyn Front-end → Code Generator → Notifier → Application

```
main()
{
  fp=bar;
}

foo()
{
  (*fp)();
}

bar()
{
}
```
## Performance Results

<table>
<thead>
<tr>
<th>Application</th>
<th>Bottlenecks found in complete search</th>
<th>Instrumentation Mini-tramps Used</th>
<th>Required Search Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Draco</td>
<td>Original: 3, Call Graph: 5</td>
<td>Original: 14,317, Call Graph: 228</td>
<td>Original: 1,006, Call Graph: 322</td>
</tr>
<tr>
<td>go</td>
<td>Original: 2, Call Graph: 4</td>
<td>Original: 12,570, Call Graph: 284</td>
<td>Original: 755, Call Graph: 278</td>
</tr>
<tr>
<td>Fpppp</td>
<td>Original: 3, Call Graph: 3</td>
<td>Original: 474, Call Graph: 96</td>
<td>Original: 141, Call Graph: 186</td>
</tr>
<tr>
<td>ssTwod (MPI)</td>
<td>Original: 9, Call Graph: 9</td>
<td>Original: 43,230, Call Graph: 11,496</td>
<td>Original: 461, Call Graph: 316</td>
</tr>
<tr>
<td>OM3 (MPI)</td>
<td>Original: 13, Call Graph: 16</td>
<td>Original: 184,382, Call Graph: 60,670</td>
<td>Original: 2,515, Call Graph: 957</td>
</tr>
</tbody>
</table>
Conclusion

• Call graph based search strategy perturbs application less than old search
• New search also faster than old search
• New version of PC available in Paradyn 3.0
• Room for future work…
  • Exclusive bottleneck verification
  • Finding a way to avoid potential blind spots.
Potential Blind Spot for New PC

A rare scenario: we haven’t seen it happen yet.
Retroactive Instrumentation

- Problem: Find CPU Time for a function if we are executing in one of its children.
- When do we start the timer for the entry to function?
- Need mechanism to trigger instrumentation code.
- Retroactive instrumentation walks stack, triggering outstanding timers