Call Paths for Pin Tools

Milind Chabbi, Xu Liu, and John Mellor-Crummey
Department of Computer Science
Rice University
CGO'14, Orlando, FL
February 17, 2014
What is a Call Path?

Chain of function calls that led to the current point in the program. (a.k.a Calling Context / Call Stack / Backtrace / Activation Record)
What is a Call Path?

A chain of function calls that led to the current point in the program. (a.k.a Calling Context / Call Stack / Backtrace / Activation Record)
Need: Ubiquitous Call Paths

Fine-grained monitoring tools

• Correctness

• Performance
Need: Ubiquitous Call Paths

Fine-grained monitoring tools

- Correctness
  - Data race detection

- Performance
Need: Ubiquitous Call Paths

Fine-grained monitoring tools

• Correctness
  ✦ Data race detection

• Performance

Attribute each conflicting access to its full call path
Need: Ubiquitous Call Paths

Fine-grained monitoring tools

• Correctness
  ✦ Data race detection
  ✦ Taint analysis
  ✦ Array out of bound detection

• Performance
Need: Ubiquitous Call Paths

Fine-grained monitoring tools

• Correctness
  ✦ Data race detection
  ✦ Taint analysis
  ✦ Array out of bound detection

• Performance
  ✦ Reuse-distance analysis
    [Liu et al. ISPASS’13]

Attribute distance between “use” and “reuse” to references in full context
Need: Ubiquitous Call Paths

Fine-grained monitoring tools

• Correctness
  ✦ Data race detection
  ✦ Taint analysis
  ✦ Array out of bound detection

• Performance
  ✦ Reuse-distance analysis
    [Liu et al. ISPASS’13]

Attribute distance between “use” and “reuse” to references in full context
Need: Ubiquitous Call Paths

Fine-grained monitoring tools

• Correctness
  ✦ Data race detection
  ✦ Taint analysis
  ✦ Array out of bound detection

• Performance
  ✦ Reuse-distance analysis
  ✦ Cache simulation
  ✦ False sharing detection
  ✦ Redundancy detection (e.g. dead writes)
Need: Ubiquitous Call Paths

Fine-grained monitoring tools

• Correctness
  ✦ Data race detection
  ✦ Taint analysis
  ✦ Array out of bound detection

• Performance
  ✦ Reuse-distance analysis
  ✦ Cache simulation
  ✦ False sharing detection
  ✦ Redundancy detection (e.g. dead writes)

• Other tools
  ✦ Debugging, testing, resiliency, replay, etc.
State-of-the-art in Collecting Ubiquitous Call Paths

"It will slow down execution by a factor of several thousand compared to native execution -- I'd guess -- so you'll wind up with something that is unusably slow on anything except the smallest problems."

"If you tried to invoke Thread::getCallStack on every memory access there would be very serious performance problems … your program would probably never reach main."

No support for collecting calling contexts.
State-of-the-art in Collecting Ubiquitous Call Paths

“It will slow down execution by a factor of several thousand compared to native execution -- I'd guess -- so you'll wind up with something that is unusably slow on anything except the smallest problems.”

“If you tried to invoke Thread::getCallStack on every memory access there would be very serious performance problems … your program would probably never reach main.”

No support for collecting calling contexts

We built one ourselves—CCTLib
Roadmap

CCTLib

- Ubiquitous call path collection
- Attributing costs to data objects
- Evaluation
- Conclusions
Roadmap

CCTLib

- Ubiquitous call path collection
- Attributing costs to data objects
- Evaluation
- Conclusions
Top Three Challenges

1. Overhead (Space)
2. Overhead (Time)
3. Overhead (Parallel scaling)
## Top Three Challenges

1. Overhead
2. Overhead (Time)
3. Overhead (Parallel scaling)
Top Three Challenges

1. Overhead (Space)
2. Overhead (Time)
3. Overhead (Parallel scaling)
Top Three Challenges

1. Overhead
2. Overhead
3. Overhead
Top Three Challenges

1. Overhead (Space)
2. Overhead (Time)
3. Overhead (Parallel scaling)
Top Three Challenges

1. Overhead (Space)
2. Overhead (Time)
3. Overhead
Top Three Challenges

1. Overhead (Space)
2. Overhead (Time)
3. Overhead (Parallel scaling)
Problem:
Deluge of call paths
Store History of Contexts Compactly

Problem:
Deluge of call paths

Instruction stream
Store History of Contexts Compactly

Problem:
Deluge of call paths

Instruction stream
Store History of Contexts Compactly

Problem:
Deluge of call paths

Instruction stream
Store History of Contexts Compactly

Problem:
Deluge of call paths

Solution
• Call paths share common prefix
• Store call paths as a calling context tree (CCT)
• One CCT per thread
Shadow Stack to Avoid Unwinding Overhead

Problem:
Unwinding overhead

```
Main()

P()

Foo() {
    *ptr = 100;
    x = 42;
}
```
Shadow Stack to Avoid Unwinding Overhead

Problem:
Unwinding overhead

```c
Main()
P()
Foo() {
    *ptr = 100;
x = 42; }
```
Shadow Stack to Avoid Unwinding Overhead

Problem: Unwinding overhead

Solution: Reverse the process. Eagerly build a replica/shadow stack on-the-fly.

```c
Main()
P()
Foo() {
  *ptr = 100;
  x = 42;
}
```
Problem: Unwinding overhead

Solution: Reverse the process. Eagerly build a replica/shadow stack on-the-fly.

```c
Main()
P()
Foo() {
    *ptr = 100;
    x = 42;
}
```
Shadow Stack to Avoid Unwinding Overhead

Problem: Unwinding overhead

Solution: Reverse the process. Eagerly build a replica/shadow stack on-the-fly.

```c
Main()

P()

Foo() {
    *ptr = 100;
    x = 42;
}

*ptr = 100;

x = 42; }
```
Shadow Stack to Avoid Unwinding Overhead

**Problem:**
Unwinding overhead

**Solution:**
Reverse the process. Eagerly build a replica/shadow stack on-the-fly.

```c
Foo() {
    *ptr = 100;
    x = 42;
}
```

```
Main()
P()
```

```
CTXT
Main()
P()
```
Shadow Stack to Avoid Unwinding Overhead

Problem: Unwinding overhead

Solution: Reverse the process. Eagerly build a replica/shadow stack on-the-fly.

Tools can obtain pointer to the current context via \texttt{CTXT} in constant time.
Shadow Stack to Avoid Unwinding Overhead

Problem: Unwinding overhead

Solution: Reverse the process. Eagerly build a replica/shadow stack on-the-fly.

Tools can obtain pointer to the current context via "CTX" in constant time.
Shadow Stack to Avoid Unwinding Overhead

Problem: Unwinding overhead

Solution: Reverse the process. Eagerly build a replica/shadow stack on-the-fly.

Tools can obtain pointer to the current context via \texttt{CTXT} in constant time.
Maintaining

`Main()`

`P()`

`W()` `Z()`
Maintaining CTXT

Return to caller:
Constant time update

CTXT

Main()

P()

W() ...

Z()
Maintaining
Finding a callee from its caller involves a lookup.
Maintaining CTXT

Finding a callee from its caller involves a lookup
Maintaining CTXT

Finding a callee from its caller involves a **lookup**
Accelerate Lookup with Splay Trees

Splay tree [“Self-adjusting binary search trees” by Sleator et al. 1985] ensures frequently called functions are near the root of the tree.
Splay tree ["Self-adjusting binary search trees" by Sleator et al. 1985] ensures frequently called functions are near the root of the tree.
Accelerate Lookup with Splay Trees

Splay tree [“Self-adjusting binary search trees” by Sleator et al. 1985] ensures frequently called functions are near the root of the tree.
Splay tree [“Self-adjusting binary search trees” by Sleator et al. 1985] ensures frequently called functions are near the root of the tree.
Splay tree [“Self-adjusting binary search trees” by Sleator et al. 1985] ensures frequently called functions are near the root of the tree.
Splay tree [“Self-adjusting binary search trees” by Sleator et al. 1985] ensures frequently called functions are near the root of the tree.
Context Should Incorporate Instruction Pointer

```c
main()

Foo()
    P()
        *ptr = 100;
        x = 42;
    }
```
Context Should Incorporate Instruction Pointer

main() -> P() -> Foo()

Foo() {
  *ptr = 100;
  x = 42; }

CTXT = Foo: INS 1
Context Should Incorporate Instruction Pointer

```c
main()

Foo()

P()

Foo()
{
    *ptr = 100;
    x = 42;
}
```

- CTXT = Foo: INS 1
- CTXT = Foo: INS 2
Attributing to Instructions

A CCT node represents a *Pin trace*

- CCTLib maintains node ➞ *Pin trace* mapping
- Each slot in a node represents an instruction in a *Pin trace*
Attributing to Instructions

• Problem: Mapping IP to Slot at runtime
  ✦ Variable size x86 instructions
  ✦ Non-sequential control flow

• Solution:
  ✦ Pin’s *trace-instrumentation* to hardwire Slot# as argument to context query routine for an IP

• Result:
  ✦ Constant time to query
Attributing to Instructions

- Problem: Mapping IP to Slot at runtime
  - Variable size x86 instructions
  - Non-sequential control flow

- Solution:
  - Pin’s trace-instrumentation to hardwire Slot# as argument to context query routine for an IP

- Result:
  - Constant time to query
Attributing to Instructions

- Problem: Mapping IP to Slot at runtime
  - Variable size x86 instructions
  - Non-sequential control flow

- Solution:
  - Pin’s trace-instrumentation to hardwire Slot# as argument to context query routine for an IP

- Result:
  - Constant time to query
Attributing to Instructions

• Problem: Mapping \textit{IP} to \textit{Slot} at runtime
  ✦ Variable size x86 instructions
  ✦ Non-sequential control flow

• Solution:
  ✦ Pin’s \textit{trace-instrumentation} to hardwire \textit{Slot#} as argument to context query routine for an \textit{IP}

• Result:
  ✦ Constant time to query
Attributing to Instructions

• Problem: Mapping IP to Slot at runtime
  ✦ Variable size x86 instructions
  ✦ Non-sequential control flow

• Solution:
  ✦ Pin’s trace-instrumentation to hardwire Slot# as argument to context query routine for an IP

• Result:
  ✦ Constant time to query CTXT
Attributing to Instructions

- **Problem**: Mapping IP to Slot at runtime
  - Variable size x86 instructions
  - Non-sequential control flow

- **Solution**:
  - Pin’s *trace-instrumentation* to hardwire Slot# as argument to context query routine for an IP

- **Result**:
  - Constant time to query CTXT
Attributing to Instructions

• Problem: Mapping IP to Slot at runtime
  ✦ Variable size x86 instructions
  ✦ Non-sequential control flow

• Solution:
  ✦ Pin’s trace-instrumentation to hardwire Slot# as argument to context query routine for an IP

• Result:
  ✦ Constant time to query
Roadmap

CCTLib

- Ubiquitous call path collection
- Attributing costs to data objects
- Evaluation
- Conclusions
Data-Centric Attribution in CCTLib

```c
int MyArray[SZ];

int * Create() {
    return malloc(...);
}

void Update(int * ptr) {
    for (...)
        ptr[i]++;
}

int main() {
    int * p;
    if (...) {
        p = Create();
    } else {
        p = MyArray;
    }
    Update(p);
}
```

- Associate each data access to its data object
- Data object
  - Dynamic allocation: Call path of allocation site
  - Static objects: Variable name
Data-Centric Attribution in CCTLib

```c
int MyArray[SZ];

int * Create(){
    return malloc(…);
}

void Update(int * ptr) {
    for( … )
        ptr[i]++;
}

int main(){
    int * p;
    if (…)
        p = Create();
    else
        p = MyArray;
    Update(p);
}
```

- Associate each *data access* to its *data object*
- **Data object**
  - Dynamic allocation: Call path of allocation site
  - Static objects: Variable name
Data-Centric Attribution in CCTLib

```c
int MyArray[SZ];

int * Create(){
    return malloc(...);
}

void Update(int * ptr) {
    for( ... )
        ptr[i]++;
}

int main(){
    int * p;
    if (...)
        p = Create();
    else
        p = MyArray;
    Update(p);
}
```

- Associate each *data access* to its *data object*
- **Data object**
  - Dynamic allocation: Call path of allocation site
  - Static objects: Variable name
Data-Centric Attribution in CCTLib

```c
int MyArray[SZ];

int * Create(){
    return malloc(...);
}

void Update(int * ptr) {
    for( ... )
        ptr[i]++;
}

int main(){
    int * p;
    if (...)
        p = Create();
    else
        p = MyArray;
    Update(p);
}
```

- Associate each *data access* to its *data object*
- **Data object**
  - Dynamic allocation: Call path of allocation site
  - Static objects: Variable name
Data-Centric Attribution in CCTLib

```c
int MyArray[SZ];

int * Create(){
    return malloc(...);
}

void Update(int * ptr) {
    for( ... )
        ptr[i]++;
}

int main(){
    int * p;
    if (...)
        p = Create();
    else
        p = MyArray;
    Update(p);
}
```

- Associate each data access to its data object
- Data object
  - Dynamic allocation: Call path of allocation site
  - Static objects: Variable name
Data-Centric Attribution in CCTLib

```c
int MyArray[SZ];

int * Create(){
    return malloc(...);
}

void Update(int * ptr) {
    for( ... )
        ptr[i]++;
}

int main(){
    int * p;
    if (...)
        p = Create();
    else
        p = MyArray;
    Update(p);
}
```

- Associate each *data access* to its *data object*
- Data object
  - Dynamic allocation: Call path of allocation site
  - Static objects: Variable name
Data-Centric Attribution in CCTLib

```c
int MyArray[SZ];

int * Create(){
    return malloc(...);
}

void Update(int * ptr) {
    for( ... )
        ptr[i]++;
}

int main(){
    int * p;
    if (...) 
        p = Create();
    else
        p = MyArray;
    Update(p);
}
```

- Associate each data access to its data object
- Data object
  - Dynamic allocation: Call path of allocation site
  - Static objects: Variable name
Data-Centric Attribution in CCTLib

```c
int MyArray[SZ];

int * Create(){
    return malloc(...);
}

void Update(int * ptr) {
    for( ... )
        ptr[i]++;
}

int main(){
    int * p;
    if (...)
        p = Create();
    else
        p = MyArray;
    Update(p);
}
```

- Associate each data access to its data object
- Data object
  - Dynamic allocation: Call path of allocation site
  - Static objects: Variable name
Data-Centric Attribution

• How?
  ✦ Record all `<AddressRange, VariableName>` tuples in a map
  ✦ Instrument all allocation/free routines and maintain `<AddressRange, CallPath>` tuples in the map
  ✦ At each memory access: search the map for the address

• Problems
  ✦ Searching the map on each access is expensive
  ✦ Map needs to be concurrent for threaded programs
Data-Centric Attribution using a Balanced Tree

• Observation:
  ✦ Updates to the map are infrequent
  ✦ Lookups in the maps are frequent

• Solution #1: sorted map
  ✦ Keep <AddressRange, Object> in a balanced binary tree
  ✦ Low memory cost—O(N)
  ✦ Moderate lookup cost—O(log N)
  ✦ Concurrent access is handled by a novel replicated tree data structure
Data-Centric Attribution using Shadow Memory

- Solution #2: shadow memory

  Application

  CCTLib
Data-Centric Attribution using Shadow Memory

- Solution #2: shadow memory
Data-Centric Attribution using Shadow Memory

- Solution #2: shadow memory

![Diagram showing Application and CCTLib with ObjA objects]
Data-Centric Attribution using Shadow Memory

- Solution #2: shadow memory
Data-Centric Attribution using Shadow Memory

- Solution #2: shadow memory
Data-Centric Attribution using Shadow Memory

• Solution #2: shadow memory

  For each memory cell, a shadow cell holds a handle for the memory cell’s data object
  ✦ Low lookup cost—O(1), high memory cost—
  ✦ Shadow memory supports concurrent access

• CCTLib supports both solutions, clients can choose
Roadmap

CCTLib

• Ubiquitous call path collection
• Attributing costs to data objects

• Evaluation
• Conclusions
## Evaluation

### Experimental setup:
- 2.2GHz Intel Sandy Bridge
- 128GB DDR3
- GNU 4.4.6 tool chain

<table>
<thead>
<tr>
<th>Program</th>
<th>Running time in sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>astr</td>
<td>361</td>
</tr>
<tr>
<td>bzip2</td>
<td>161</td>
</tr>
<tr>
<td>gcc</td>
<td>70</td>
</tr>
<tr>
<td>h264ref</td>
<td>618</td>
</tr>
<tr>
<td>hmmmer</td>
<td>446</td>
</tr>
<tr>
<td>libquantum</td>
<td>462</td>
</tr>
<tr>
<td>mcf</td>
<td>320</td>
</tr>
<tr>
<td>omnetpp</td>
<td>352</td>
</tr>
<tr>
<td>Xalan</td>
<td>295</td>
</tr>
<tr>
<td>ROSE</td>
<td>24</td>
</tr>
<tr>
<td>LAMMPS</td>
<td>99</td>
</tr>
<tr>
<td>LULESH</td>
<td>67</td>
</tr>
</tbody>
</table>
## Evaluation

**Spec Int 2006 reference benchmark**

<table>
<thead>
<tr>
<th>Program</th>
<th>Running time in sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>astr</td>
<td>361</td>
</tr>
<tr>
<td>bzip2</td>
<td>161</td>
</tr>
<tr>
<td>gcc</td>
<td>70</td>
</tr>
<tr>
<td>h264ref</td>
<td>618</td>
</tr>
<tr>
<td>hmmmer</td>
<td>446</td>
</tr>
<tr>
<td>libquantum</td>
<td>462</td>
</tr>
<tr>
<td>mcf</td>
<td>320</td>
</tr>
<tr>
<td>omnetpp</td>
<td>352</td>
</tr>
<tr>
<td>Xalan</td>
<td>295</td>
</tr>
<tr>
<td>ROSE</td>
<td>24</td>
</tr>
<tr>
<td>LAMMPS</td>
<td>99</td>
</tr>
<tr>
<td>LULESHE</td>
<td>67</td>
</tr>
</tbody>
</table>

**Experimental setup:**
- 2.2GHz Intel Sandy Bridge
- 128GB DDR3
- GNU 4.4.6 tool chain
### Evaluation

**Spec Int 2006 reference benchmark**

<table>
<thead>
<tr>
<th>Program</th>
<th>Running time in sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>astr</td>
<td>361</td>
</tr>
<tr>
<td>bzip2</td>
<td>161</td>
</tr>
<tr>
<td>gcc</td>
<td>70</td>
</tr>
<tr>
<td>h264ref</td>
<td>618</td>
</tr>
<tr>
<td>hmmer</td>
<td>446</td>
</tr>
<tr>
<td>libquantum</td>
<td>462</td>
</tr>
<tr>
<td>mcf</td>
<td>320</td>
</tr>
<tr>
<td>omnetpp</td>
<td>352</td>
</tr>
<tr>
<td>Xalan</td>
<td>295</td>
</tr>
<tr>
<td>ROSE</td>
<td>24</td>
</tr>
<tr>
<td>LAMMPS</td>
<td>99</td>
</tr>
<tr>
<td>LULESH</td>
<td>67</td>
</tr>
</tbody>
</table>

**Experimental setup:**
- 2.2GHz Intel Sandy Bridge
- 128GB DDR3
- GNU 4.4.6 tool chain

**Source-to-source compiler from LLNL**
- 3M LOC compiling 70K LOC
- Deep call chains
Evaluation

<table>
<thead>
<tr>
<th>Program</th>
<th>Running time in sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>astr</td>
<td>361</td>
</tr>
<tr>
<td>bzip2</td>
<td>161</td>
</tr>
<tr>
<td>gcc</td>
<td>70</td>
</tr>
<tr>
<td>h264ref</td>
<td>618</td>
</tr>
<tr>
<td>hmmer</td>
<td>446</td>
</tr>
<tr>
<td>libquantum</td>
<td>462</td>
</tr>
<tr>
<td>mcf</td>
<td>320</td>
</tr>
<tr>
<td>omnetpp</td>
<td>352</td>
</tr>
<tr>
<td>Xalan</td>
<td>295</td>
</tr>
<tr>
<td>ROSE</td>
<td>24</td>
</tr>
<tr>
<td>LAMMPS</td>
<td>99</td>
</tr>
<tr>
<td>LULESH</td>
<td>67</td>
</tr>
</tbody>
</table>

Experimental setup:
- 2.2GHz Intel Sandy Bridge
- 128GB DDR3
- GNU 4.4.6 tool chain

Source-to-source compiler from LLNL
3M LOC compiling 70K LOC
Deep call chains

Molecular dynamics code
500K LOC
Deep call chains
Multithreaded
# Evaluation

## Spec Int 2006 reference benchmark

<table>
<thead>
<tr>
<th>Program</th>
<th>Running time in sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>astr</td>
<td>361</td>
</tr>
<tr>
<td>bzip2</td>
<td>161</td>
</tr>
<tr>
<td>gcc</td>
<td>70</td>
</tr>
<tr>
<td>h264ref</td>
<td>618</td>
</tr>
<tr>
<td>hmmer</td>
<td>446</td>
</tr>
<tr>
<td>libquantum</td>
<td>462</td>
</tr>
<tr>
<td>mcf</td>
<td>320</td>
</tr>
<tr>
<td>omnetpp</td>
<td>352</td>
</tr>
<tr>
<td>Xalan</td>
<td>295</td>
</tr>
<tr>
<td>ROSE</td>
<td>24</td>
</tr>
<tr>
<td>LAMMPS</td>
<td>99</td>
</tr>
<tr>
<td>LULESH</td>
<td>67</td>
</tr>
</tbody>
</table>

## Experimental setup:
- 2.2GHz Intel Sandy Bridge
- 128GB DDR3
- GNU 4.4.6 tool chain

### Source-to-source compiler from LLNL
- 3M LOC compiling 70K LOC
- Deep call chains

### Molecular dynamics code
- 500K LOC
- Deep call chains
- Multithreaded

### Hydrodynamics mini-app from LLNL
- Frequent data allocation and de-allocations
- Memory bound
- Multithreaded, Poor scaling
# Overhead Analysis

<table>
<thead>
<tr>
<th>Call path collection</th>
<th>Time overhead relative to original program (Null Pin tool)</th>
<th>Time overhead relative to simple instruction counting Pin tool</th>
<th>Memory overhead relative to original program</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>30x</td>
<td>1.7x</td>
<td>1.8x</td>
</tr>
</tbody>
</table>

**Data-centric attribution**
- Balanced Tree
- Shadow Memory
# Overhead Analysis

<table>
<thead>
<tr>
<th></th>
<th>Call path collection</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Call path</strong></td>
<td></td>
</tr>
<tr>
<td><strong>collection</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Time overhead</strong></td>
<td></td>
</tr>
<tr>
<td>relative to original</td>
<td></td>
</tr>
<tr>
<td>program (Null Pin</td>
<td>30x</td>
</tr>
<tr>
<td>tool)</td>
<td></td>
</tr>
<tr>
<td><strong>Time overhead</strong></td>
<td></td>
</tr>
<tr>
<td>relative to simple</td>
<td>1.7x</td>
</tr>
<tr>
<td>instruction counting</td>
<td></td>
</tr>
<tr>
<td>Pin tool</td>
<td></td>
</tr>
<tr>
<td><strong>Memory overhead</strong></td>
<td></td>
</tr>
<tr>
<td>relative to original</td>
<td>1.8x</td>
</tr>
<tr>
<td>program</td>
<td></td>
</tr>
</tbody>
</table>

Data-centric attribution

- Balanced Tree
- Shadow Memory
## Overhead Analysis

<table>
<thead>
<tr>
<th></th>
<th>Call path collection</th>
<th>Data-centric attribution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Balanced Tree</td>
<td>Shadow Memory</td>
</tr>
<tr>
<td><strong>Time overhead</strong></td>
<td>1.7x</td>
<td>4.5x</td>
</tr>
<tr>
<td>relative to simple</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction counting</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pin tool</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Memory overhead</strong></td>
<td>1.8x</td>
<td>2.0x</td>
</tr>
<tr>
<td>relative to original</td>
<td></td>
<td></td>
</tr>
<tr>
<td>program</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Overhead Analysis

<table>
<thead>
<tr>
<th>Call path collection</th>
<th>Data-centric attribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time overhead relative to simple instruction counting Pin tool</td>
<td>Balanced Tree</td>
</tr>
<tr>
<td>Memory overhead relative to original program</td>
<td>1.7x</td>
</tr>
<tr>
<td></td>
<td>1.8x</td>
</tr>
</tbody>
</table>
CCTLib Scales to Multiple Threads

CCTLib overhead of N threads: 

CCTLib scalability of N threads:

Higher scalability is better, 1.0 is ideal

CCTLib scalability on LAMMPS

Call path collection
Data-centric attribution via Sorted maps
Data-centric attribution via Shadow memoory
Conclusions

- Many tools can benefit from attributing metrics to full calling contexts and/or data objects
- Ubiquitous calling context collection was previously considered prohibitively expensive
- Fine-grain attribution of metrics to calling contexts and data objects is practical
- Full-precision call path collection and data-centric attribution require only modest space and time overhead

✦ Choice of algorithms and data structures was a key to success
Conclusions

- Many tools can benefit from attributing metrics to full calling contexts and/or data objects.
- Ubiquitous calling context collection was previously considered prohibitively expensive.
- Fine-grain attribution of metrics to calling contexts and data objects is practical.
- Full-precision call path collection and data-centric attribution require only modest space and time overhead.
  ✦ Choice of algorithms and data structures was a key to success.
Conclusions

- Many tools can benefit from attributing metrics to full calling contexts and/or data objects
- Ubiquitous calling context collection was previously considered prohibitively expensive
- Fine-grain attribution of metrics to calling contexts and data objects is practical
- Full-precision call path collection and data-centric attribution require only modest space and time overhead
- Choice of algorithms and data structures was a key to success

http://code.google.com/p/cctlib/
Other Complications in Real Programs

- Complex control flow
  - Signal handling
  - Setjmp-Longjmp
  - C++ exceptions (try-catch)

- Thread creation and destruction
  - Maintaining parent-child relationships between threads
  - Scalability to large number of threads