CCTLib: Pinpointing Software Inefficiencies with Fine-grained Program Monitoring

Milind Chabbi
Shasha Wen
Xu Liu
Who Are We?

Xu Liu
Assistant Professor
Department of Computer Science
College of William and Mary
xl10@cs.wm.edu

Shasha Wen
Ph.D. student
Department of Computer Science
College of William and Mary
swen@email.wm.edu

Milind Chabbi
Research Scientist
Hewlett Packard Labs
Palo Alto, CA
milind.chabbi@hpe.com

Introduction
Importance of Code Efficiency
Importance of Code Efficiency
Importance of Code Efficiency
Importance of Code Efficiency
Importance of Code Efficiency

Programs need to be efficient at all scales
Sources of Performance Bottlenecks

• Code design
  ✦ Algorithms
  ✦ Data structures

• Programming practice
  ✦ Aware of functionality but not performance

• Compiler optimization
  ✦ Sometimes optimization may cause more harm than good
  ✦ Code must be tailored to enable some optimization
Sources of Performance Bottlenecks

- Code design
  - Algorithms
  - Data structures

- Programming practice
  - Aware of functionality but not performance

- Compiler optimization
  - Sometimes optimization may cause more harm than good
  - Code must be tailored to enable some optimization

A tool set is necessary to pinpoint inefficiencies
Classical Performance Analysis

• Identify hot spots — high resource utilization
  ✦ Time / CPU cycles
  ✦ Cache misses on different levels
  ✦ Floating point operations, SIMD
  ✦ Derived metrics such as instruction per cycle (IPC)

• Improve code in hot spots

• Hot spot analysis is indispensable, but
  ✦ Cannot tell if resources were “well spent”
  ✦ Hot spots may be symptoms of performance problems
  ✦ Need significant manual efforts to investigate root causes
From Resource Usage to Wastage

• Wasted data movement
  ✦ Redundant memory accesses
    * Redundant stores: write same values to a memory location
  ✦ Useless memory accesses
    * Dead stores: stored value got overwritten without use

• Wasted arithmetic computation
  ✦ Symbolic equivalent computation
    * a=b+c; d=b+c
  ✦ Result equivalent computation
    * a=b*b-c*c; d=(b+c)*(b-c)

• Unnecessary synchronization (locks and barriers)
From Resource Usage to Wastage

• Wasted data movement
  ✦ Redundant memory accesses
    * Redundant stores: write same values to a memory location
  ✦ Useless memory accesses
    * Dead stores: stored value got overwritten without use

• Wasted arithmetic computation
  ✦ Symbolic equivalent computation
    * a=b+c; d=b+c
  ✦ Result equivalent computation
    * a=b*b-c*c; d=(b+c)*(b-c)

• Unnecessary synchronization (locks and barriers)

Need new profiling techniques
fine-grained profiling
HMMER: A Example for Resource Wastage

<table>
<thead>
<tr>
<th>Unoptimized</th>
<th>-O3 optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i = 1; i &lt;= L; i++) {</td>
<td>for (i = 1; i &lt;= L; i++) {</td>
</tr>
<tr>
<td>for (k = 1; k &lt;= M; k++) {</td>
<td>for (k = 1; k &lt;= M; k++) {</td>
</tr>
<tr>
<td>mc[k] = mpp[k-1] + tpmm[k-1];</td>
<td>R1 = mpp[k-1] + tpmm[k-1];</td>
</tr>
<tr>
<td>if ((sc = ip[k-1] + tpim[k-1]) &gt; mc[k])</td>
<td>mc[k] = R1;</td>
</tr>
<tr>
<td>mc[k] = sc;</td>
<td>if ((sc = ip[k-1] + tpim[k-1]) &gt; R1)</td>
</tr>
<tr>
<td></td>
<td>mc[k] = sc;</td>
</tr>
</tbody>
</table>
HMMER: A Example for Resource Wastage

Unoptimized

for (i = 1; i <= L; i++) {
    for (k = 1; k <= M; k++) {
        mc[k] = mpp[k-1] + tpmm[k-1];
        if ((sc = ip[k-1] + tpim[k-1]) > mc[k])
            mc[k] = sc;
    }
}

-O3 optimized

for (i = 1; i <= L; i++) {
    for (k = 1; k <= M; k++) {
        R1 = mpp[k-1] + tpmm[k-1];
        mc[k] = R1;
        if ((sc = ip[k-1] + tpim[k-1]) > R1)
            mc[k] = sc;
        else
            mc[k] = R1;
    }
}
HMMER: A Example for Resource Wastage

Unoptimized

```c
for (i = 1; i <= L; i++) {
    for (k = 1; k <= M; k++) {
        mc[k] = mpp[k-1] + tpmm[k-1];
        if ((sc = ip[k-1] + tpim[k-1]) > mc[k])
            mc[k] = sc;
    }
}
```

-O3 optimized

```c
for (i = 1; i <= L; i++) {
    for (k = 1; k <= M; k++) {
        R1 = mpp[k-1] + tpmm[k-1];
        mc[k] = R1;
        if ((sc = ip[k-1] + tpim[k-1]) > R1)
            mc[k] = sc;
        else
            mc[k] = R1;
    }
}
```

Never Alias.
Declare as “restrict” pointers.
Can vectorize.
**HMMER: A Example for Resource Wastage**

<table>
<thead>
<tr>
<th>Unoptimized</th>
<th>-O3 optimized</th>
</tr>
</thead>
</table>
| for (i = 1; i <= L; i++) {
  for (k = 1; k <= M; k++) {
    mc[k] = mpp[k-1] + tpmm[k-1];
    if ((sc = ip[k-1] + tpim[k-1]) > mc[k])
      mc[k] = sc;
  }
}
| for (i = 1; i <= L; i++) {
  for (k = 1; k <= M; k++) {
    R1 = mpp[k-1] + tpmm[k-1];
    mc[k] = R1;
    if ((sc = ip[k-1] + tpim[k-1]) > R1)
      mc[k] = sc;
    else
      mc[k] = R1;
  }
}

- **Never Alias.**
- Declare as “restrict” pointers.
- Can vectorize.

> > 16% running time improvement
> > 40% with vectorization
Compilers Do NOT Eliminate All Inefficiencies

• Compilers have limitations with their static analysis
  ✦ Aliasing and pointers
  ✦ Limited optimization scopes: compilation units
  ✦ Input-sensitive inefficiencies
  ✦ Flow-sensitive inefficiencies
Coarse-grained Profilers Lack

• State-of-the-art coarse-grained profilers
  ✦ Intel VTune
  ✦ Rice HPCToolkit
  ✦ Oracle Solaris Studio
  ✦ ARM allinea

• Coarse-grained analysis
  ✦ Sample instructions or events via hardware performance monitoring units (PMU)
    ✴ One sample per 1M instructions
  ✦ Do not track consecutive sequence of instructions or memory references —> cannot detect wasteful operations
  ✦ Never capture semantic meaning of execution
Fine-grained Profiling

- Track each instruction
  - Operator
  - Operands

- Track each register
  - General registers
  - SIMD registers

- Track each memory location
  - Effective addresses

- Track each value in storage location
  - Value in registers
  - Value in memory

- One step closer to reconstructing the semantic meaning (or lack there of) in execution
for (i = 1; i <= L; i++) {
    for (k = 1; k <= M; k++) {
        R1 = mpp[k-1] + tpmm[k-1];
        mc[k] = R1;
        if ((sc = ip[k-1] + tpim[k-1]) > R1)
            mc[k] = sc;
HMMER Example

```c
for (i = 1; i <= L; i++) {
    for (k = 1; k <= M; k++) {
        R1 = mpp[k-1] + tpmm[k-1];
        mc[k] = R1;
        if ((sc = ip[k-1] + tpim[k-1]) > R1) mc[k] = sc;
    }
}
```

```assembly
1   mov %r10,%rax,4),%ecx
2   add 0x0(%r13,%rax,4),%ecx #mpp[k-1]+tpmm[k-1]
3   mov %ecx, 0x4(%rdx)           #assign mc[k]
4   mov 0x18(%rsp),%rbx
5   mov (%r9,%rax,4),%r15d
6   add (%rbx,%rax,4),%r15d      #dpp[k-1]+tpdm[k-1]
7   mov 0x20(%rsp),%rbx
8   cmp   %ecx,%r15d               #%ecx is mc[k]
9   cmovge %r15d, %ecx
10  mov %ecx, 0x4(%rdx)         #assign mc[k]
```
HMMER Example

for (i = 1; i <= L; i++) {
    for (k = 1; k <= M; k++) {
        R1 = mpp[k-1] + tpmm[k-1];
        mc[k] = R1;
        if ((sc = ip[k-1] + tpim[k-1]) > R1)
            mc[k] = sc;
    }
}

1. mov %r10,%rax,4,%ecx
2. add 0x0(%r13,%rax,4),%ecx #mpp[k-1]+tpmm[k-1]
3. mov %ecx, 0x4(%rdx) #assign mc[k]
4. mov 0x18(%rsp),%rbx
5. mov (%r9,%rax,4),%r15d
6. add (%rbx,%rax,4),%r15d #dpp[k-1]+tpdm[k-1]
7. mov 0x20(%rsp),%rbx
8. cmp %ecx,%r15d #%ecx is mc[k]
9. cmovge %r15d, %ecx
10. mov %ecx, 0x4(%rdx) #assign mc[k]
```c
for (i = 1; i <= L; i++) {
    for (k = 1; k <= M; k++) {
        R1 = mpp[k-1] + tpmm[k-1];
        mc[k] = R1;
        if ((sc = ip[k-1] + tpim[k-1]) > R1)
            mc[k] = sc;
    }
}
```

---

1. `mov %r10,%rax,4,%ecx`
2. `add 0x0(%r13,%rax,4),%ecx #mpp[k-1]+tpmm[k-1]`
3. `mov %ecx, 0x4(%rdx) #assign mc[k]`
4. `mov 0x18(%rsp),%rbx`
5. `mov (%r9,%rax,4),%r15d`
6. `add (%rbx,%rax,4),%r15d #dpp[k-1]+tpdm[k-1]`
7. `mov 0x20(%rsp),%rbx`
8. `cmp %ecx,%r15d #%ecx is mc[k]`
9. `cmovge %r15d, %ecx`
10. `mov %ecx, 0x4(%rdx) #assign mc[k]`
HMMER Example

for (i = 1; i <= L; i++) {
    for (k = 1; k <= M; k++) {
        R1 = mpp[k-1] + tpmm[k-1];
        mc[k] = R1;
        if ((sc = ip[k-1] + tpim[k-1]) > R1)
            mc[k] = sc;
    }
}

1   mov %r10,%rax,4),%ecx
2   add 0x0(%r13,%rax,4),%ecx #mpp[k-1]+tpmm[k-1]
3   mov %ecx, 0x4(%rdx)           #assign mc[k]
4   mov 0x18(%rsp),%rbx
5   mov (%r9,%rax,4),%r15d
6   add (%rbx,%rax,4),%r15d    #dpp[k-1]+tpdm[k-1]
7   mov   0x20(%rsp),%rbx
8   cmp   %ecx,%r15d               #%ecx is mc[k]
9   cmovge %r15d, %ecx
10  mov %ecx, 0x4(%rdx)         #assign mc[k]

the value in memory location 0x4(%rdx) is unused
HMMER Example

```
for (i = 1; i <= L; i++) {
    for (k = 1; k <= M; k++) {
        R1 = mpp[k-1] + tpmm[k-1];
        mc[k] = R1;
        if ((sc = ip[k-1] + tpim[k-1]) > R1)
            mc[k] = sc;
    }
}
```

1. `mov %r10,%rax,4),%ecx`
2. `add 0x0(%r13,%rax,4),%ecx #mpp[k-1]+tpmm[k-1]`
3. `mov %ecx, 0x4(%rdx) #assign mc[k]`
4. `mov %ecx, 0x4(%rdx) #assign mc[k]`
5. `mov 0x18(%rsp),%rbx`
6. `mov (%r9,%rax,4),%r15d #dpp[k-1]+tpdm[k-1]`
7. `add (%rbx,%rax,4),%r15d`
8. `cmp %ecx,%r15d #%ecx is mc[k]`
9. `cmovge %r15d, %ecx`
10. `mov %ecx, 0x4(%rdx) #assign mc[k]`

The value in memory location 0x4(%rdx) is unused.
Call Path Profiling for Fine-grained Analysis

• Associate problematic instructions with their call paths
  ✦ Expose more semantic information about the instructions
  ✦ Understand context-sensitive performance issues

• If no call path collected for fine-grained analysis
  ✦ Do not provide root causes of the problem
  ✦ Do not guide source code optimization
An Example: SPEC bwaves

A pair of redundant computation

movsdq 0x8(%rdi,%r10,8), %xmm0:__mul:<no src>

---------------------
REDUNDANT WITH
---------------------
movsdq 0x8(%rdi,%r10,8), %xmm0:__mul:<no src>
An Example: SPEC bwaves

A pair of redundant computation

```
movsdq 0x8(%rdi,%r10,8), %xmm0:__mul:<no src>
__dvd: <no src>
__mpexp:<no src>
__mplog:<no src>
__slowpow:<no src>
__ieee754_pow_sse2:<no src>
pow:<no src>
jacobian_:jacobian_lam.f:47
shell_:shell_lam.f:193
MAIN__:flow_lam.f:63
main:flow_lam.f:67
```

REDUNDANT WITH

```
movsdq 0x8(%rdi,%r10,8), %xmm0:__mul:<no src>
__dvd: <no src>
__mpexp:<no src>
__mplog:<no src>
__slowpow:<no src>
__ieee754_pow_sse2:<no src>
pow:<no src>
jacobian_:jacobian_lam.f:47
shell_:shell_lam.f:193
MAIN__:flow_lam.f:63
main:flow_lam.f:67
```
An Example: SPEC bwaves

A pair of redundant computation

movsdq 0x8(%rdi,%r10,8), %xmm0:__mul:<no src>
__dvd: <no src>
__mpexp:<no src>
__mplog:<no src>
__slowpow:<no src>
__ieee754_pow_sse2:<no src>
pow:<no src>
  jacobian_:jacobian_lam.f:47
  shell_:shell_lam.f:193
  MAIN__:flow_lam.f:63
  main:flow_lam.f:67

REDUNDANT WITH *************

movsdq 0x8(%rdi,%r10,8), %xmm0:__mul:<no src>
__dvd:<no src>
__mpexp:<no src>
__mplog:<no src>
__slowpow:<no src>
__ieee754_pow_sse2:<no src>
pow:<no src>
  jacobian_:jacobian_lam.f:47
  shell_:shell_lam.f:193
  MAIN__:flow_lam.f:63
  main:flow_lam.f:67

REDUNDANT WITH *************
An Example: SPEC bwaves

41. \( \text{ros} = q(1, \text{ip1}, \text{jp1}, \text{kp1}) \)
42. \( \text{us} = q(2, \text{ip1}, \text{jp1}, \text{kp1})/\text{ros} \)

\[ \ldots \]

47. \( \mu = \left( \mu + \frac{(g_m-1.0d0) \times \left( q(5, \text{ip1}, \text{jp1}, \text{kp1})/\text{ros} - 0.5d0 \times (u^2 + v^2 + w^2) \right)}{2.0d0} \right)^{0.75d0} \)

A pair of redundant computation

\[
\text{movsdq 0x8(%rdi,%r10,8), %xmm0:__mul:<no src>}
\]

\[
\text{__dvd: <no src>}
\]

\[
\text{__mpexp:<no src>}
\]

\[
\text{__mplog:<no src>}
\]

\[
\text{__slowpow:<no src>}
\]

\[
\text{__ieee754_pow_sse2:<no src>}
\]

\[
\text{pow:<no src>}
\]

\[
\text{jacobian_:jacobian_lam.f:47}
\]

\[
\text{shell_:shell_lam.f:193}
\]

\[
\text{MAIN__:flow_lam.f:63}
\]

\[
\text{main:flow_lam.f:67}
\]

************** REDUNDANT WITH *************
An Example: SPEC bwaves

41. \( \text{ros} = q(1, \text{ip1}, \text{jp1}, \text{kp1}) \)
42. \( \text{us} = q(2, \text{ip1}, \text{jp1}, \text{kp1})/\text{ros} \)

…

47. \( \text{mu} = (\text{mu} + (\text{gm}-1.0d0)*(q(5,\text{ip1},\text{jp1},\text{kp1})/\text{ros}-0.5d0*(\text{us}^2+\text{vs}^2+\text{ws}^2))^0.75d0)/2.0d0 \)

A pair of redundant computation

```
movsdq  0x8(%rdi,%r10,8), %xmm0:__mul:<no src>
__dvd: <no src>
__mpexp:<no src>
__mplog:<no src>
__slowpow:<no src>
__ieee754_pow_sse2:<no src>
pow:<no src>
```

```
jacobian_:jacobian_lam.f:47
shell_:shell_lam.f:193
MAIN__:flow_lam.f:63
main:flow_lam.f:67
```

************** REDUNDANT WITH **************

```
movsdq 0x8(%rdi,%r10,8), %xmm0:__mul:<no src>
__dvd:<no src>
__mpexp:<no src>
__mplog:<no src>
__slowpow:<no src>
__ieee754_pow_sse2:<no src>
pow:<no src>
```

```
jacobian_:jacobian_lam.f:47
shell_:shell_lam.f:193
MAIN__:flow_lam.f:63
main:flow_lam.f:67
```
An Example: SPEC bwaves

41. \( \text{ros} = q(1, \text{ip1, jp1, kp1}) \)
42. \( \text{us} = q(2, \text{ip1, jp1, kp1})/\text{ros} \)
43. \[ \text{...} \]
47. \( \text{mu} = (\text{mu} + (\text{gm}-1.0d0)*(q(5,\text{ip1,jp1,kp1})/\text{ros}-0.5d0*(\text{us*us+vs*vs+ws*ws}))^{0.75d0})/2.0d0 \)

No insights without call path profiling

A pair of redundant computation

\[
\text{movsdq 0x8(\%rdi,\%r10,8), \%xmm:__mul:<no src>  
__dvd:<no src>  
__mpexp:<no src>  
__mplog:<no src>  
__slowpow:<no src>  
__ieee754_pow_sse2:<no src>  
\text{pow:<no src>  
}  
\text{jacobian_:jacobian_lam.f:47}  
\text{shell_:shell_lam.f:193}  
\text{MAIN__:flow_lam.f:63}  
\text{main:flow_lam.f:67}  
\]
An Example: SPEC bwaves

41. \( \text{ros} = q(1, \text{ip}1, \text{jp}1, \text{kp}1) \)
42. \( \text{us} = q(2, \text{ip}1, \text{jp}1, \text{kp}1)/\text{ros} \)

\[ \ldots \]
47. \( \text{mu} = (\text{mu} + (\text{gm} - 1.0d0) * (q(5, \text{ip}1, \text{jp}1, \text{kp}1)/\text{ros} - 0.5d0 * (\text{us} \times \text{us} + \text{vs} \times \text{vs} + \text{ws} \times \text{ws}))^{0.75d0})/2.0d0 \)

No insights without call path profiling

CCTLib: a framework that collects calling context for fine-grained profilers

A pair of redundant computation

\[
\text{movsdq 0x8(%rdi,%r10,8), %xmm0:<no src>}
\text{__mul:<no src>}
\text{__dvd:<no src>}
\text{__mpexp:<no src>}
\text{__mplog:<no src>}
\text{__slowpow:<no src>}
\text{__ieee754_pow_sse2:<no src>}
\text{pow:<no src>}
\text{jacobian_:jacobian_lam.f:47}
\text{shell_:shell_lam.f:193}
\text{MAIN__:flow_lam.f:63}
\text{main:flow_lam.f:67}
\]

**************REDUNDANT WITH **************

\[
\text{movsdq 0x8(%rdi,%r10,8), %xmm0:<no src>}
\text{__mul:<no src>}
\text{__dvd:<no src>}
\text{__mpexp:<no src>}
\text{__mplog:<no src>}
\text{__slowpow:<no src>}
\text{__ieee754_pow_sse2:<no src>}
\]

main:flow_lam.f:67
CCTLib Overview

• Functionality
  ✦ Can capture call path for each dynamic instruction
  ✦ Can capture the data object read/written by each memory access
    ✦ Heap data objects: call paths to the allocations
    ✦ Static data objects: names from symbol table

• Programmability
  ✦ APIs provide request-based service for clients

• Overhead
  ✦ Moderate overhead in both runtime and space
CCTLib Software

- git clone https://github.com/CCTLib/cctlib.git
  - Supported on x86_64 linux, gcc > 4.8.2
  - Pin 3.0 not yet supported.
- cd cctlib
- sh build.sh

```
PIN_ROOT is NOT set!
+ echo  (1) Download Pin from the WWW and automatically set PIN_ROOT?
    (2) Enter PIN_ROOT in the commandline?
    (any key) Exit?
(1) Download Pin from the WWW and automatically set PIN_ROOT?
    (2) Enter PIN_ROOT in the commandline?
    (any key) Exit?
```

- Choose (1)
- Successful installation will end with this message

```
**************************************************
*************** ALL TESTS PASSED ********************
****************************************************
```
CCTLib Software

• Distributed under MIT license

The MIT License (MIT)

Copyright 2014-2017 CCTLib team and contributors

Permission is hereby granted, free of charge, to any person obtaining a copy of this
software and associated documentation files (the "Software"), to deal in the Software
without restriction, including without limitation the rights to use, copy, modify, merge,
publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons
to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or
substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE
FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.

• Contact

cctlib-forum@lists.wm.edu
## Scale of Call Paths

<table>
<thead>
<tr>
<th>Description</th>
<th>Original program running for 10 minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Debuggers</strong></td>
<td>&lt; $10^3$</td>
</tr>
<tr>
<td>On each break point</td>
<td></td>
</tr>
<tr>
<td><strong>Performance analysis tools</strong></td>
<td>$6 \times 10^5$</td>
</tr>
<tr>
<td>On each sample (1 sample/ms)</td>
<td></td>
</tr>
<tr>
<td><strong>Fine-grained instrumentation tools</strong></td>
<td>$1.2 \times 10^{12}$</td>
</tr>
<tr>
<td>On each instruction (2GHz CPU)</td>
<td></td>
</tr>
</tbody>
</table>
Challenges in Ubiquitous Call Path Collection

1. Overhead (space)
2. Overhead (time)
3. Overhead (parallel scaling)
Store History of Contexts Compactly

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

Solution
• Call paths share common prefix
• Store call paths as a calling context tree (CCT)
• One CCT per thread

Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling: Ammons et al. PLDI’97
Shadow Stack to Avoid Unwinding Overhead

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

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.

```
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.

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.

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

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

```
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.

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

call

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

Tools can obtain pointer to the current context via 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.

Main()

P()

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

*ptr = 100;

x = 42;

return

Main()

P()

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

*ptr = 100;

x = 42;

return

Tools can obtain pointer to the current context via 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.

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

Tools can obtain pointer to the current context via in constant time
Two Ways to Use CCTLib By Pin Tools

- Option 1: Store context handles (ContextHandle_t) within the Pin tool and access the context (traverse full call chain) as needed
Two Ways to Use CCTLib By Pin Tools

- Option 2: Associate a user-defined “metric” with each ContextHandle_t and store it in the calling context tree. Perform a tree traversal as needed.
Associating Address to Data Objects

- Static objects
  - Record all `<AddressRange, VariableName>` tuples in a map

- Dynamic allocations
  - Instrument all allocation/free routines
  - Maintain `<AddressRange, ContextId>` 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
Context To DOT

Execution-wide calling context tree for NWChem—a six-million line computational chemistry code

/* Description: 
   Dumps all CCTs into DOT files for visualization.
*/

void DottifyAllCCTs();
CCTLib Client Tools

• DeadSpy: Pinpointing dead stores in a program
  ✦ Detects dead writes in an execution

• RedSpy: Pinpointing silent stores in a program
  ✦ Detects redundant data movement in an execution

• RVN: Runtime Value Numbering
  ✦ Detects useless computations in an execution

• Metric client
  ✦ Captures hot paths

• Footprint client
  ✦ Computes context-sensitive memory footprint of a data object
Pinpointing Useless Memory Accesses

• Accessing memory is expensive on modern architectures
  ✦ Multiple levels of hierarchy
  ✦ Cores share cache
  ✦ Limited bandwidth per core

• Unnecessary writes
  ✦ Cause unnecessary capacity miss and coherence traffic —→ affects resource shared system
  ✦ Wear out NVM-based or disk-based memory

**Dead write:** Two writes happen to the same memory location without an intervening read

```
int x = 0;
x = 20;
```

```
int x = 0;
Print(x);
x = 20;
```

```
int x = 0;
x = 20;
```

**Killing write**
Dead Writes: Example

- Chombo: AMR framework for solving PDEs

- Compilers can’t eliminate all dead writes because of:
  - Aliasing / ambiguity
  - Aggregate variables
  - Function boundaries
  - Late binding
  - Partial deadness
Dead Writes: Example

Code lacked “design for performance”

Better code: Use else-if nesting

```fortran
do k
  do j
    do i
      Wgdnv(i, j, k, 0) = ...
      Wgdnv(i, j, k, inorm) = ...
      Wgdnv(i, j, k, 4) = ...
    endif
    if (spout.le.0.0d0) then
      Wgdnv(i, j, k, 0) = ...
      Wgdnv(i, j, k, inorm) = ...
      Wgdnv(i, j, k, 4) = ...
    endif
    if (spin.gt.0.0d0) then
      Wgdnv(i, j, k, 0) = ...
      Wgdnv(i, j, k, inorm) = ...
      Wgdnv(i, j, k, 4) = ...
    endif
  enddo
enddo
```

```fortran
if (spout.le.0.0d0) then
  Wgdnv(i, j, k, 0) = ...
  Wgdnv(i, j, k, inorm) = ...
  Wgdnv(i, j, k, 4) = ...
elsif (spout.le.0.0d0) then
  Wgdnv(i, j, k, 0) = ...
  Wgdnv(i, j, k, inorm) = ...
  Wgdnv(i, j, k, 4) = ...
else
  Wgdnv(i, j, k, 0) = ...
  Wgdnv(i, j, k, inorm) = ...
  Wgdnv(i, j, k, 4) = ...
endif
```
Detection Scheme

- Monitor every load and store in a program
- Maintain state information for each memory byte referenced by the program
- Detect every dead write in an execution with an automaton

[CGO’12] “DeadSpy: A Tool to Pinpoint Program Inefficiencies”
Detection Scheme

- Monitor every load and store in a program
- Maintain state information for each memory byte referenced by the program
- Detect every dead write in an execution with an automaton

[CGO’12] “DeadSpy: A Tool to Pinpoint Program Inefficiencies”
Detection Scheme

- Monitor every load and store in a program
- Maintain state information for each memory byte referenced by the program
- Detect every dead write in an execution with an automaton

[CGO’12] “DeadSpy: A Tool to Pinpoint Program Inefficiencies”
Detection Scheme

• Monitor every load and store in a program
• Maintain state information for each memory byte referenced by the program
• Detect every dead write in an execution with an automaton

[CGO’12] “DeadSpy: A Tool to Pinpoint Program Inefficiencies”
Detection Scheme

- Monitor every load and store in a program
- Maintain state information for each memory byte referenced by the program
- Detect every dead write in an execution with an automaton

[CGO’12] “DeadSpy: A Tool to Pinpoint Program Inefficiencies”
Detection Scheme

- Monitor every load and store in a program
- Maintain state information for each memory byte referenced by the program
- Detect every dead write in an execution with an automaton

[CGO’12] “DeadSpy: A Tool to Pinpoint Program Inefficiencies”
Detection Scheme

- Monitor every load and store in a program
- Maintain state information for each memory byte referenced by the program
- Detect every dead write in an execution with an automaton

[CGO’12] “DeadSpy: A Tool to Pinpoint Program Inefficiencies”
Detection Scheme

- Monitor every load and store in a program
- Maintain state information for each memory byte referenced by the program
- Detect every dead write in an execution with an automaton

[CGO’12] “DeadSpy: A Tool to Pinpoint Program Inefficiencies”
DeadSpy: Measurement and Attribution

• Precise measurement
  ♦ No false positives and no false negatives

• Precise attribution
  ♦ Source-level feedback with calling context of dead and killing writes
  ♦ On each dead write record <old-ctxt-handle, cur-ctxt-handle>

```c
int x = 0;
Dead write
```

```c
x = 20;
Killing write
```
DeadSpy + CCTLib in Action

Main()

Memory
DeadSpy + CCTLib in Action

Main()

A()

Memory
DeadSpy + CCTLib in Action

Memory
DeadSpy + CCTLib in Action

Main()

A()

B()

Memory

W
DeadSpy + CCTLib in Action

Main()

A()

B()

W

Memory

M
DeadSpy + CCTLib in Action

Main()

A()

B()

W

C₁ = GetContextHandle()

Memory

M
DeadSpy + CCTLib in Action

Main()

A()

B()

C₁ = GetContextHandle()

Memory

Shadow Memory

W

C₁

W

M
DeadSpy + CCTLib in Action

Main()

A() -> C()

B()

C₁ = GetContextHandle()

Memory

Shadow Memory

W  C₁
DeadSpy + CCTLib in Action

C1 = GetContextHandle()

M

Memory

Shadow Memory

W C1
DeadSpy + CCTLib in Action

```
Main()
  A()
  B()
W

C()
W

C₁ = GetContextHandle()
M

Memory
```

```
<table>
<thead>
<tr>
<th>W</th>
<th>C₁</th>
</tr>
</thead>
</table>
```
DeadSpy + CCTLib in Action

C1 = GetContextHandle()

C2 = GetContextHandle()

M

Memory

Shadow Memory

W C1
DeadSpy + CCTLib in Action

Main()

A()

B()

W

C()

W

C1 = GetContextHandle()

C2 = GetContextHandle()

Memory

Shadow Memory

Dead writes

<table>
<thead>
<tr>
<th>C1:C2</th>
<th>+1</th>
</tr>
</thead>
</table>

W C1
DeadSpy + CCTLib in Action

Dead writes

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C1:C2</td>
<td>1234</td>
</tr>
<tr>
<td>C3:C23</td>
<td>959</td>
</tr>
<tr>
<td>C2:C14</td>
<td>546</td>
</tr>
</tbody>
</table>

Main()

M

W

B()

A()

C()

C1 = GetContextHandle()

C2 = GetContextHandle()

W

Memory

Shadow Memory

W   C1
DeadSpy + CCTLib in Action

Main()

A()

B()

C()

\[ C_1 = \text{GetContextHandle()} \]

Memory

\[ C_2 = \text{GetContextHandle()} \]

Shadow Memory

\[ \begin{array}{c|c}
C_1 & C_2 \\
\hline
1234 & 959 \\
\hline
546 & \\
\end{array} \]

Dead writes

\[ C_1: C_2 \]

\[ C_3: C_{23} \]

\[ C_2: C_{14} \]
DeadSpy + CCTLib in Action

Main()

A()

B()

C()

W

C1 = GetContextHandle()

M

W

C2 = GetContextHandle()

Dead writes

<table>
<thead>
<tr>
<th>Context</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1:C2</td>
<td>1234</td>
</tr>
<tr>
<td>C3:C23</td>
<td>959</td>
</tr>
<tr>
<td>C2:C14</td>
<td>546</td>
</tr>
</tbody>
</table>

Memory

Shadow Memory
DeadSpy + CCTLib in Action

Main()

A()

B()

C()

Dead writes

<table>
<thead>
<tr>
<th>Context</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1:C2</td>
<td>1234</td>
</tr>
<tr>
<td>C3:C23</td>
<td>959</td>
</tr>
<tr>
<td>C2:C14</td>
<td>546</td>
</tr>
</tbody>
</table>

C1 = GetContextHandle()
C2 = GetContextHandle()

Memory

Shadow Memory

M
Dead Writes in SPEC CPU 2006

Lower is better

The number of dead writes is surprisingly high!

Across compilers and optimization levels
HMMER: Lack of Design for Performance

Unoptimized

```c
for (i = 1; i <= L; i++) {
    for (k = 1; k <= M; k++) {
        ...
        ic[k] = mpp[k] + tpmi[k];
        if ((sc = ip[k] + tpii[k]) > ic[k])
            ic[k] = sc;
    }
}
```

-O3 optimized

```c
for (i = 1; i <= L; i++) {
    for (k = 1; k <= M; k++) {
        ...
        R1 = mpp[k] + tpmi[k];
        ic[k] = R1;
        if ((sc = ip[k] + tpii[k]) > R1)
            ic[k] = sc;
    }
}
```
HMMER: Lack of Design for Performance

Unoptimized

for (i = 1; i <= L; i++) {
    for (k = 1; k <= M; k++) {
        ...
        ic[k] = mpp[k] + tpmi[k];
        if ((sc = ip[k] + tpii[k]) > ic[k])
            ic[k] = sc;
    }
}

-O3 optimized

for (i = 1; i <= L; i++) {
    for (k = 1; k <= M; k++) {
        ...
        R1 = mpp[k] + tpmi[k];
        ic[k] = R1;
        if ((sc = ip[k] + tpii[k]) > R1)
            ic[k] = sc;
    }
    else
        ic[k] = R1;
HMMER: Lack of Design for Performance

Unoptimized

```c
for (i = 1; i <= L; i++) {
    for (k = 1; k <= M; k++) {
        ...
        ic[k] = mpp[k] + tpmi[k];
        if ((sc = ip[k] + tpii[k]) > ic[k])
            ic[k] = sc;
    }
}
```

-O3 optimized

```c
for (i = 1; i <= L; i++) {
    for (k = 1; k <= M; k++) {
        ...
        R1 = mpp[k] + tpmi[k];
        ic[k] = R1;
        if ((sc = ip[k] + tpii[k]) > R1)
            ic[k] = sc;
        else
            ic[k] = R1;
    }
}
```

Never Alias.
Declare as "restrict" pointers.
Can vectorize.
HMMER: Lack of Design for Performance

Unoptimized

for (i = 1; i <= L; i++) {
    for (k = 1; k <= M; k++) {
        ... 
        ic[k] = mpp[k] + tpmi[k];
        if ((sc = ip[k] + tpII[k]) > ic[k])
            ic[k] = sc;
    }
}

-O3 optimized

for (i = 1; i <= L; i++) {
    for (k = 1; k <= M; k++) {
        ... 
        R1 = mpp[k] + tpmi[k];
        ic[k] = R1;
        if ((sc = ip[k] + tpII[k]) > R1)
            ic[k] = sc;
        else
            ic[k] = R1;
    }
}

Never Alias.
Declare as “restrict” pointers.
Can vectorize.

> 16% running time improvement
> 40% with vectorization
A ***silent store*** is the one that does not change the system state.
Pinpointing Silent Store

A **silent store** is the one that does not change the system state.

```c
/**  Func has no side-effect **/
for (int i = 0 ; i < N; i++) {
    A[i] = 2 * Func(i);
    ...  = A[i];
    A[i] = Func(i)+Func(i);
    ...  = A[i];
}
```
A **silent store** is the one that does not change the system state.

```java
/** Func has no side-effect **/
for (int i = 0 ; i < N; i++) {
    A[i] = 2 * Func(i);
    ... = A[i];
    A[i] = Func(i)+Func(i);
    ... = A[i];
}
```
A **silent store** is the one that does not change the system state.

```cpp
/** Func has no side-effect **/
for (int i = 0 ; i < N; i++) {
    A[i] = 2 * Func(i);
    ... = A[i];
    A[i] = Func(i)+Func(i);
    ... = A[i];
}
```
A *silent store* is the one that does not change the system state.

```java
/** Func has no side-effect **/
for (int i = 0 ; i < N; i++) {
    A[i] = 2 * Func(i);
    ... = A[i];
    A[i] = Func(i)+Func(i);
    ... = A[i];
}
```
A **silent store** is the one that does not change the system state.

```c
/**  Func has no side-effect **/
for (int i = 0 ; i < N; i++) {
    A[i] = 2 * Func(i);
    ... = A[i];
    A[i] = Func(i)+Func(i);
    ... = A[i];
}
```

DeadSpy and traditional value profiling cannot detect this redundancy.
Value Agnostic vs. Value Aware

• DeadSpy: Value Agnostic
  ✦ Does not inspect the value at a location; merely inspects the operation (read/write) on a location

• RedSpy: Value Aware
  ✦ Inspects value produced by each operation
Value Agnostic vs. Value Aware

- DeadSpy: Value Agnostic
  - Does not inspect the value at a location; merely inspects the operation (read/write) on a location

- RedSpy: Value Aware
  - Inspects value produced by each operation
Value Agnostic vs. Value Aware

• DeadSpy: Value Agnostic
  ✦ Does not inspect the value at a location; merely inspects the operation (read/write) on a location

• RedSpy: Value Aware
  ✦ Inspects value produced by each operation

silent store

silent write to same location (memory/register..)
Value Agnostic vs. Value Aware

• DeadSpy: Value Agnostic
  ✦ Does not inspect the value at a location; merely inspects the operation (read/write) on a location

• RedSpy: Value Aware
  ✦ Inspects value produced by each operation
Value Agnostic vs. Value Aware

• DeadSpy: Value Agnostic
  ✦ Does not inspect the value at a location; merely inspects the operation (read/write) on a location

• RedSpy: Value Aware
  ✦ Inspects value produced by each operation

silent store ↔ silent write to same location (memory/register..) ↔ silent write to adjacent locations ↔ Almost silent (approximate computing)
RedSpy: Value Locality

*Value Locality* implies producing the same value that is already present

*Value Locality* is often a symptom of some kinds of redundancy
RedSpy: Value Locality

Value Locality implies producing the same value that is already present

Value Locality is often a symptom of some kinds of redundancy

- Temporal value locality
  - The same value overwrites the same storage location
  - In memory or in register

- Spatial value locality
  - Nearby storage locations share a common value
  - Mainly in memory, big arrays
RedSpy: Value Locality

*Value Locality* implies producing the same value that is already present

*Value Locality* is often a symptom of some kinds of redundancy

- Temporal value locality
  - The same value overwrites the same storage location
  - In memory or in register

- Spatial value locality
  - Nearby storage locations share a common value
  - Mainly in memory, big arrays

v1 = a + a;
... = v1;
v1 = a * 2;
**Value Locality implies producing the same value that is already present**

**Value Locality** is often a symptom of some kinds of redundancy

- **Temporal value locality**
  - The same value overwrites the same storage location
  - In memory or in register

- **Spatial value locality**
  - Nearby storage locations share a common value
  - Mainly in memory, big arrays

```c
v1 = a + a;  // Temporal value locality
... = v1;
v1 = a * 2;  // Temporal value locality
```

```c
for(i=0; i<N; ++i){
a[i]=i/2+1; // i is int
b[i] = Func(a[i]);
}
```

```c
for(i=0; i<N; ++i){
a[i]=i/2+1; // i is int
b[i] = Func(a[i]);
}
```
Value Locality implies producing the same value that is already present

Value Locality is often a symptom of some kinds of redundancy

- Temporal value locality
  - The same value overwrites the same storage location
  - In memory or in register

- Spatial value locality
  - Nearby storage locations share a common value
  - Mainly in memory, big arrays

```c
v1 = a + a;
... = v1;
v1 = a * 2;
for(i=0; i<N; ++i){
a[i]=i/2+1; // i is int
b[i] = Func(a[i]);
}
```
Value Locality implies producing the same value that is already present.

**Value Locality** is often a symptom of some kinds of redundancy.

- **Temporal value locality**
  - The same value overwrites the same storage location
  - In memory or in register

- **Spatial value locality**
  - Nearby storage locations share a common value
  - Mainly in memory, big arrays

```cpp
v1 = a + a;  // approximately the same
... = v1;
v1 = a * 2;
```

```cpp
for(i=0; i<N; ++i){
    a[i]=i/2+1;  // i is int
    b[i] = Func(a[i]);
}
```
RedSpy: Value Locality

Value Locality implies producing the same value that is already present

**Value Locality** is often a symptom of some kinds of redundancy

- Temporal value locality
  - The same value overwrites the same storage location
  - In memory or in register

- Spatial value locality
  - Nearby storage locations share a common value
  - Mainly in memory, big arrays

\[
\begin{align*}
v1 &= a + a; \\
\ldots &= v1; \\
v1 &= a \times 2; \\
\end{align*}
\]

\[
\text{for}(i=0; i<N; ++i)\
\begin{align*}
a[i] &= i/2 + 1; // i is int \\
b[i] &= \text{Func}(a[i]); \\
\end{align*}
\]

Exact & approximate
RedSpy: Value Locality Detection

- Temporal value locality (temporal redundancy)
  - Monitor memory write
  - Monitor register write
RedSpy: Value Locality Detection

- Temporal value locality (temporal redundancy)
  - Monitor memory write
  - Monitor register write

  target location $t$ is written
RedSpy: Value Locality Detection

- Temporal value locality (temporal redundancy)
  - Monitor memory write
  - Monitor register write

$\text{target location } t \text{ is written}$
RedSpy: Value Locality Detection

- Temporal value locality (temporal redundancy)
  - Monitor memory write
  - Monitor register write

    target location $t$ is written

    new value
RedSpy: Value Locality Detection

- Temporal value locality (temporal redundancy)
  - Monitor memory write
  - Monitor register write

```
target location \( t \) is written

old value \quad \downarrow \quad \text{new value}
```
RedSpy: Value Locality Detection

- Temporal value locality (temporal redundancy)
  - Monitor memory write
  - Monitor register write

Target location $t$ is written

old value $= (\approx) \rightarrow$ new value
RedSpy: Value Locality Detection

• Temporal value locality (temporal redundancy)
  ✦ Monitor memory write
  ✦ Monitor register write

\[
\text{target location } t \text{ is written} \\
\downarrow \\
\text{old value} = (\approx) \text{ new value}
\]

old operation
RedSpy: Value Locality Detection

- Temporal value locality (temporal redundancy)
  - Monitor memory write
  - Monitor register write

\[
\text{target location } t \text{ is written} \\
\downarrow \\
\text{old value} = (\approx) \text{ new value}
\]

old operation
current operation
RedSpy: Value Locality Detection

- Temporal value locality (temporal redundancy)
  - Monitor memory write
  - Monitor register write

\[
\text{target location } t \text{ is written} \\
\text{old value } = (\approx) \text{ new value}
\]

old operation <redundant with> current operation
RedSpy: Value Locality Detection

- Temporal value locality (temporal redundancy)
  - Monitor memory write
  - Monitor register write

\[
\text{target location } t \text{ is written} \\
\downarrow \\
\text{old value} \quad = \quad (\approx) \quad \text{new value}
\]

old operation <redundant with> current operation

CCTLib
RedSpy: Value Locality Detection

- Temporal value locality (temporal redundancy)
  - Monitor memory write
  - Monitor register write

\[
\text{target location } t \text{ is written} \\
\text{old value} \quad = \quad (\approx) \quad \text{new value}
\]
RedSpy: Spatial Value Locality

- User provides instrumentation points
  - Where? call predefined function
  - How? array, size, checking stride, approximation

- CCTLib scans while data structure and identifies the ratio of unique values to total elements

- CCTLib Data-Centric APIs
  - Type: static, dynamic, stack…
  - Name/allocation point
  - Begin address
  - End address
RedSpy: Experiments

- Temporal redundancy

  ✦ GCC 4.8.5 -O3 PGO

GeoMean precise reg-reg = 4.46%
GeoMean approximate reg-reg = 1.71%
GeoMean precise load = 4.45%
GeoMean approximate load = 3.37%
GeoMean precise store = 3.13%
GeoMean approximate store = 2.33%
Case Study: h264ref SPEC CPU2006

- Redundant writes to same location (temporal redundancy)
  - 13% loads and 13% stores are redundant

```c
for (pos = 0; pos < max_pos; pos++) {
    ...
    if(...) PelYline_11 = FastLine16Y_11;
    else PelYline_11 = UMVLine16Y_11;

    for (blkx = 0; blkx < 4; blkx++) {
        for (y = 0; y < 4; y++) {
            refptr = PelYline_11(ref_pic, abs_y++, abs_x, img_height, img_width);
            ... }
        }
    }
}
```
Case Study: h264ref SPEC CPU2006

- Redundant writes to same location (temporal redundancy)
  - 13% loads and 13% stores are redundant

```c
for (pos = 0; pos < max_pos; pos++) {
    ...
    if(...) PelYline_11 = FastLine16Y_11;
    else PelYline_11 = UMVLine16Y_11;

for (blky = 0; blky < 4; blky++) {
    for (y = 0; y < 4; y++) {
        refptr = PelYline_11(ref_pic, abs_y++, abs_x, img_height, img_width);
        ...
    }
}
```

parameters seldom change
Case Study: h264ref SPEC CPU2006

- Redundant writes to same location (temporal redundancy)
  - 13% loads and 13% stores are redundant

```c
for (pos = 0; pos < max-pos; pos++) {
    ...
    if(...) PelYline_11 = FastLine16Y_11;
    else PelYline_11 = UMVLine16Y_11;

for (blky = 0; blky < 4; blky++) {
    for (y = 0; y < 4; y++) {
       reffptr = PelYline_11(ref_pic, abs_y++, abs_x, img_height, img_width);
        ...
    }
}
```

- Parameters seldom change
- Push same value
Case Study: h264ref SPEC CPU2006

- Redundant writes to same location (temporal redundancy)
  - 13% loads and 13% stores are redundant

```c
for (pos = 0; pos < max_pos; pos++) {
    ...
    if(...) PelYline_11 = FastLine16Y_11;
    else PelYline_11 = UMVLine16Y_11;

for (blky = 0; blky < 4; blky++) {
    for (y = 0; y < 4; y++) {
        refptr = PelYline_11(ref_pic, abs_y++, abs_x, img_height, img_width);
        ...
    }
}
```

- Prevent from "inline"
- Push same value
- Parameters seldom change
Case Study: h264ref SPEC CPU2006

• Redundant writes to same location (temporal redundancy)
  ✦ 13% loads and 13% stores are redundant

```c
for (pos = 0; pos < max_pos; pos++) {
    ...
    if(...) PelYline_11 = FastLine16Y_11;
    else PelYline_11 = UMVLine16Y_11;

    for (blky = 0; blky < 4; blky++) {
        for (y = 0; y < 4; y++) {
            refptr = PelYline_11(ref_pic, abs_y++, abs_x, img_height, img_width);
            ...
        }
    }
```

• Optimization
  ✦ Inline the two functions
  ✦ 1.34x speedup; 23% energy saving
Case Study: Rodinia hotspot

- Approximately same values
  - Elements in array `result` are similar (<1%)

```c
for (r = 0; r < row; r++) {
    for (c = 0; c < col; c++) {
        ...{
            delta = (step / Cap)*(power[r*col+c] + (temp[(r+1)*col+c]+temp[(r-1)*col+c]
                    -2.0*temp[r*col+c]) / Ry + (temp[r*col+c+1]+ temp[r*col+c-1] -
                    2.0*temp[r*col+c]) / Rx + (amb_temp - temp[r*col+c]) / Rz);
        }
        result[r*col+c] =temp[r*col+c] + delta;
    }
}
```
Case Study: Rodinia hotspot

- Approximately same values
  - Elements in array $\text{result}$ are similar ($<1\%$)

```c
for (r = 0; r < row; r++) {
    for (c = 0; c < col; c++) {
        delta = (step / Cap) * (power[r*col+c] + (temp[(r+1)*col+c]+temp[(r-1)*col +c]
        -2.0*temp[r*col+c]) / Ry + (temp[r*col+c+1]+ temp[r*col+c-1] -
        2.0*temp[r*col+c]) / Rx + (amb_temp - temp[r*col+c]) / Rz);
        result[r*col+c] = temp[r*col+c] + delta;
    }
}
```
Case Study: Rodinia hotspot

- Approximately same values
  - Elements in array `result` are similar (<1%)
Case Study: Rodinia hotspot

- Approximately same values
  - Elements in array `result` are similar (<1%)

```java
for (r = 0; r < row; r++) {
    for (c = 0; c < col; c++) {
        delta = (step / Cap)*(power[r*col+c] + (temp[(r+1)*col+c]+temp[(r-1)*col+c] - 2.0*temp[r*col+c]) / Ry + (temp[r*col+c+1]+temp[r*col+c-1] - 2.0*temp[r*col+c]) / Rx + (amb_temp - temp[r*col+c]) / Rz);
        result[r*col+c] = temp[r*col+c] + delta;
    }
}
```

- Optimization
  - calculate the first and middle column
  - 2.21x speedup; 70% power saving
  - mean relative error: <0.6%
Case Study: Rodinia hotspot

- Approximately same values
  - Elements in array `result` are similar (<1%)

```java
for (r = 0; r < row; r++) {
    for (c = 0; c < col; c++) {
        delta = (step / Cap)*(power[r*col+c] + (temp[(r+1)*col+c]+temp[(r-1)*col +c] - 2.0*temp[r*col+c]) / Ry + (temp[r*col+c+1]+ temp[r*col+c-1] - 2.0*temp[r*col+c]) / Rx + (amb_temp - temp[r*col+c]) / Rz);
        result[r*col+c] =temp[r*col+c] + delta;
    }
}
```

- Optimization
  - calculate the first and middle column
  - 2.21x speedup; 70% power saving
  - mean relative error: <0.6%
Case Study: Rodinia hotspot

• Approximately same values
  ✦ Elements in array result are similar (<1%)

```plaintext
for (r = 0; r < row; r++) {
    for (c = 0; c < col; c++) {
        ...{
            delta = (step / Cap)*(power[r*col+c] + (temp[(r+1)*col+c]+temp[(r-1)*col +c]
            -2.0*temp[r*col+c]) / Ry + (temp[r*col+c+1]+ temp[r*col+c-1] -
            2.0*temp[r*col+c]) / Rx + (amb_temp - temp[r*col+c]) / Rz);
        }
        result[r*col+c] =temp[r*col+c]+ delta;
    }
```

• Optimization
  ✦ calculate the first and middle column
  ✦ 2.21x speedup; 70% power saving
  ✦ mean relative error: <0.6%
Case Study: Rodinia hotspot

- Approximately same values
  - Elements in array `result` are similar (<1%)

```java
for (r = 0; r < row; r++) {
    for (c = 0; c < col; c++) {
        delta = (step / Cap)*(power[r*col+c] + (temp[(r+1)*col+c]+temp[(r-1)*col +c] -2.0*temp[r*col+c]) / Ry + (temp[r*col+c+1] + temp[r*col+c-1] - 2.0*temp[r*col+c]) / Rx + (amb_temp - temp[r*col+c]) / Rz);
        result[r*col+c] = temp[r*col+c] + delta;
    }
}
```

- Optimization
  - calculate the first and middle column
  - 2.21x speedup; 70% power saving
  - mean relative error: <0.6%
Case Study: Rodinia hotspot

- Approximately same values
  - Elements in array `result` are similar (<1%)

```cpp
for (r = 0; r < row; r++) {
    for (c = 0; c < col; c++) {
        ...
        delta = (step / Cap) * (power[r*col+c] + (temp[(r+1)*col+c] + temp[(r-1)*col+c] - 2.0*temp[r*col+c]) / Ry + (temp[r*col+c+1] + temp[r*col+c-1] - 2.0*temp[r*col+c]) / Rx + (amb_temp - temp[r*col+c]) / Rz);
        result[r*col+c] = temp[r*col+c] + delta;
    }
}
```

- Optimization
  - calculate the first and middle column
  - 2.21x speedup; 70% power saving
  - mean relative error: <0.6%
Summary

- Production programs suffer from myriad inefficiencies in software
  - Compilers and traditional tools are insufficient
- Fine-grained monitoring tools are necessary for identifying several kinds of program inefficiencies
  - Fine-grained tools can provide semantic information for developer productivity
- CCTLib provides efficient calling context collection for production workloads at moderate overhead
- CCTLib is open source: [https://github.com/CCTLib/cctlib](https://github.com/CCTLib/cctlib)
- Pin tools built with CCTLib pinpoint software inefficiencies and offer new venues to tuning