DeadSpy
A Tool to Pinpoint
Program Inefficiencies

Milind Chabbi and John Mellor-Crummey
Department of Computer Science
Rice University

Funded by the DARPA through the Air Force Research Lab (AFRL) and the TI Fellowship
Impact of Code Efficiency

Programs need to be efficient at all scales.
Causes of Inefficiencies

- Program design
  - Algorithms
  - Data structures
- Programming practice
  - Lack of design for performance
- Compiler optimizations
  - Sometimes optimizations can do more harm than good
  - Code must be tailored to enable some optimizations

One tool to pinpoint inefficiencies: DeadSpy
Classical Performance Analysis

- Focus on code regions with "high resource utilization"
- Look for "hot spots" in program
  - Time
  - Cache misses
  - Energy
- Improve code in these "hot spots"

Hot spot analysis is indispensable, but ...

Can’t identify if resources were "well spent"

Hot spots may be symptoms of problems

The onus is on the programmer to investigate causes
DeadSpy Philosophy

- Shift focus from “resource usage” to “resource wastage”
- Identify causes of wasteful resource consumption
  - E.g. Wasted memory accesses

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

<table>
<thead>
<tr>
<th>Dead write</th>
<th>Dead write</th>
<th>Not a dead write</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int x = 0;</code>&lt;br&gt;<code>x = 20;</code></td>
<td><code>int x = 0;</code>&lt;br&gt;<code>Print(x);</code>&lt;br&gt;<code>x = 20;</code></td>
<td></td>
</tr>
</tbody>
</table>

DeadSpy: Motivating Example

- **Chombo**: Framework for Adaptive Mesh Refinement (AMR) for solving partial differential equations

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

- Code lacks design for performance

C KERNEL: RIEMANNF 8 0

\[
\begin{align*}
&\text{do } k = \text{iboxlo2,iboxhi2} \\
&\text{do } j = \text{iboxlo1,iboxhi1} \\
&\text{do } i = \text{iboxlo0,iboxhi0}
\end{align*}
\]

\[
\begin{align*}
\text{Wgdnv}(i,j,k,0) &= \text{ro} + \text{frac}*(\text{rstar} - \text{ro}) \\
\text{Wgdnv}(i,j,k,inorm) &= \text{uno} + \text{frac}*(\text{ustar} - \text{uno}) \\
\text{Wgdnv}(i,j,k,4) &= \text{po} + \text{frac}*(\text{pstar} - \text{po})
\end{align*}
\]

if \((\text{spin.g}\text{t}(0.0d0))\) then
\[
\begin{align*}
\text{Wgdnv}(i,j,k,0) &= \text{rstar} \\
\text{Wgdnv}(i,j,k,inorm) &= \text{ustar} \\
\text{Wgdnv}(i,j,k,4) &= \text{pstar}
\end{align*}
\]
endif

Else if \((\text{spout.le.}(0.0d0))\) then
\[
\begin{align*}
\text{Wgdnv}(i,j,k,0) &= \text{ro} \\
\text{Wgdnv}(i,j,k,inorm) &= \text{uno} \\
\text{Wgdnv}(i,j,k,4) &= \text{po}
\end{align*}
\]
endif

Else
\[
\begin{align*}
\text{Wgdnv}(i,j,k,0) &= \text{ro} + \text{frac}*(\text{rstar} - \text{ro}) \\
\text{Wgdnv}(i,j,k,inorm) &= \text{uno} + \text{frac}*(\text{ustar} - \text{uno}) \\
\text{Wgdnv}(i,j,k,4) &= \text{po} + \text{frac}*(\text{pstar} - \text{po})
\end{align*}
\]
Endif

Spends 20% time

3-level nested loop

Correct code:

- Else-if nesting
DeadSpy: A Tool to Pinpoint Dead Writes

- Monitor every *load* and *store* operation in a program
- Maintain state information for each memory byte
- Detect every dead write in an execution with an automaton
DeadSpy: Implementation

- Intel Pin instruments code to monitor each \textit{load} and \textit{store}
  - Pin - dynamic binary rewriting tool

- Shadow memory maintains state
  - 1-1 mapping from a program address to the corresponding state information

- Instrumentation code identifies dead writes
  - Check and update the state before performing any memory operation
  - Record necessary information to report \( W \rightarrow W \) state transitions
DeadSpy: Measurement and Attribution

- Instrumentation ensures precise measurement
  - No false positives and no false negatives
- DeadSpy provides useful (source-level) feedback with calling context for dead and killing writes
  - Build **Calling Context Tree (CCT)** on-the-fly by instrumenting function entry and exit

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

Dead write

Killing write
DeadSpy in Action

CCT

Main()
DeadSpy in Action

CCT

Main()

A()
DeadSpy in Action

CCT

Main()

A()

B()

C()

Memory

Shadow Memory
DeadSpy in Action

CCT

Main()

A()

B()

C()

Memory

Shadow Memory

Dead table

<table>
<thead>
<tr>
<th>Dead</th>
<th>Killing</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>+1</td>
</tr>
</tbody>
</table>
DeadSpy in Action

Main()

A()

B()

C()

CCT

Dead Context

Killing Context

Memory

Shadow Memory

Dead table

<table>
<thead>
<tr>
<th>Dead</th>
<th>Killing</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2e10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>250</td>
</tr>
<tr>
<td></td>
<td></td>
<td>80</td>
</tr>
</tbody>
</table>

- Implementation challenges due to limitations of binary analysis
- Cost of instrumentation
Formalizing Dead Writes in an Execution

- **Deadness**
  \[
  \text{Deadness} = \frac{\text{NumBytesDead}}{\text{NumBytesWritten}} = \frac{\sum_i \sum_j F(C_{ij})}{\text{NumBytesWritten}}
  \]

- **Contextual Deadness**
  \[
  \text{ContextualDeadness} (C_{ij}) = \frac{F(C_{ij})}{\sum_i \sum_j F(C_{ij})}
  \]

Higher Deadness → Poor performance
Deadness in SPEC CPU2006 Integer Reference Benchmarks

The number of dead writes is surprisingly high!

Across compilers and optimization levels

Compiled with gcc 4.1.2 with –O2
Case Study: GCC

Use of Inappropriate Data Structure

```c
static void loop_regs_scan (const struct loop * loop, int extra_size) {

  ...
  last_set = (rtx *) xrealloc (regs->num, sizeof (rtx));

  /* Scan the loop, recording register usage. */
  for (Instruction insn in loop) {
    if (INSN_P (insn)) {
      ...
      if (PATTERN (insn) sets a register)
        count_one_set (regs, insn, PATTERN (insn), last_set);
      ...
    }
    if (Label(insn)) // new BBL
      memset (last_set, 0, regs->num * sizeof (rtx));
  }

  Calloc 16937 elements (132KB) array

  Performance implications on a JIT compiler
  • Basic blocks are short
  • Median use 2 unique elements
  • Dense array is a poor data structure choice

  Reinitialize 16937 elements (132KB) each time

  > 28% running time improvement
```
Case Study: Bzip2
Overly Aggressive Compiler Optimization

Original code

```
Bool mainGtU ( UInt32 i1, UInt32 i2, UChar* block, ...) {
    Int32 k; UChar c1, c2; UInt16 s1, s2;
    /* 1 */
    c1 = block[i1]; c2 = block[i2];
    if (c1 != c2) return (c1 > c2);
    i1++; i2++;
    /* 2 */
    c1 = block[i1]; c2 = block[i2];
    if (c1 != c2) return (c1 > c2);
    i1++; i2++;
    /* 3 */
    c1 = block[i1]; c2 = block[i2];
    if (c1 != c2) return (c1 > c2);
    i1++; i2++;
    ...
    /* 12 */
    c1 = block[i1]; c2 = block[i2];
    if (c1 != c2) return (c1 > c2);
    i1++; i2++;

    REST OF THE FUNCTION
```

Optimized Code (gcc 4.1.2)

```
Bool mainGtU ( UInt32 i1, UInt32 i2, UChar* block, ...) {
    Int32 k; UChar c1, c2; UInt16 s1, s2;
    tmp1 = i1 + 1;
    tmp2 = i1 + 2;
    tmp12 = i1 + 12;
    /* 1 */
    c1 = block[i1]; c2 = block[i2];
    if (c1 != c2) return (c1 > c2);
    i1++;
    i2++;
    /* 2 */
    c1 = block[tmp1]; c2 = block[i2];
    if (c1 != c2) return (c1 > c2);
    i1++;
    i2++;
    /* 3 */
    c1 = block[tmp2]; c2 = block[i2];
    if (c1 != c2) return (c1 > c2);
    i1++;
    i2++;
    /* 12 */
    c1 = block[tmp11]; c2 = block[i2];
    if (c1 != c2) return (c1 > c2);
    i1++;
    i2++;

    REST OF THE FUNCTION
```

>14% running time improvement
Case Study: HMMER
Lack of Design for Performance

- HMMER: code for sequence alignment
- Dead writes

```
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;
    }
}
```

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

Deadness in unoptimized: 0.3%
Deadness in optimized : 30%
54 times more absolute no. of dead writes in O2

>16% running time improvement.
> 40% with vectorization.

Never Alias.
Declare as “restrict” pointers.
Can vectorize.
Speedup Based on DeadSpy’s Advice

- Poor data structure choice
- Lack of design for performance
- Overly aggressive compiler optimization
- Lack of design for performance

% Speedup

Benchmarks

- gcc: 14.3
- hmmer: 15.7
- bzip2: 7.2
- chombo: 6.6

> 40% on Intel compiler

AMD Opteron 6168@1.9GHz, 128GB of 1333MHz DDR3, CentOS 5.
Conclusions

- Dead writes are a common symptom of inefficiency
  - DeadSpy pinpoints inefficiencies
  - Improvements in tuned codes

- Lessons for compiler writers
  - Pay attention to code generation algorithms
    - Understand when optimizations do more harm than good (bzip2)
    - Eliminate dead writes when possible
  - Choose compiler data structures prudently (gcc)

- Focus on wasteful resource consumption
  - New avenue for performance tuning

DeadSpy: [http://www.cs.rice.edu/~mc29/deadspy.tgz](http://www.cs.rice.edu/~mc29/deadspy.tgz) (Linux x86_64 only)
HPCToolkit: [http://hpctoolkit.org](http://hpctoolkit.org)
Backup: DeadSpy Overhead

![Backup: DeadSpy Overhead Graph](image-url)

- **Benchmarks**: astar, bzip2, gcc, h264ref, hmer, libquantum, mcf, omnetpp, perlbench, sjeng, xalan, Average
- **Slowdown (times)**: 0, 10, 20, 30, 40, 50, 60, 70, 80, 90
- **Markers**:
  - Yellow: CCT only
  - Blue: CCT + IP

Average Slowdown:
- CCT only: 22.3
- CCT + IP: 41.2
Deadness in Multi-Threaded Codes

OpenMP NAS Parallel Benchmarks

- Intrer-Thread
- Intra-Thread
- Total
# Deadness Across Compilers and Optimization Levels

<table>
<thead>
<tr>
<th>Program</th>
<th>Intel 11.1</th>
<th></th>
<th>PGI 10.5</th>
<th></th>
<th>GNU 4.1.2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>None</td>
<td>Default</td>
<td>Max</td>
<td>None</td>
<td>Default</td>
<td>Max</td>
</tr>
<tr>
<td>aistar</td>
<td>2.3</td>
<td>8.4</td>
<td>5.0</td>
<td>5.2</td>
<td>1.1</td>
<td>5.7</td>
</tr>
<tr>
<td>bzip2</td>
<td>4.7</td>
<td>8.6</td>
<td>8.9</td>
<td>4.9</td>
<td>9.9</td>
<td>12.8</td>
</tr>
<tr>
<td>gcc</td>
<td>39.3</td>
<td>67.8</td>
<td>67.2</td>
<td>40.8</td>
<td>53.3</td>
<td>51.9</td>
</tr>
<tr>
<td>gobmk</td>
<td>15.7</td>
<td>21.3</td>
<td>22.7</td>
<td>17.4</td>
<td>19.1</td>
<td>20.7</td>
</tr>
<tr>
<td>h264ref</td>
<td>14.4</td>
<td>28.4</td>
<td>38.3</td>
<td>15.1</td>
<td>24.5</td>
<td>26.2</td>
</tr>
<tr>
<td>hmmer</td>
<td>31.3</td>
<td>68.7</td>
<td>68.8</td>
<td>31.5</td>
<td>67.6</td>
<td>67.9</td>
</tr>
<tr>
<td>perlbench</td>
<td>15.3</td>
<td>18.0</td>
<td>20.0</td>
<td>16.7</td>
<td>16.5</td>
<td>16.8</td>
</tr>
<tr>
<td>libquantum</td>
<td>1.7</td>
<td>6.0</td>
<td>0.3</td>
<td>2.8</td>
<td>7.1</td>
<td>7.5</td>
</tr>
<tr>
<td>mcf</td>
<td>16.6</td>
<td>49.4</td>
<td>49.5</td>
<td>17.2</td>
<td>27.6</td>
<td>47.2</td>
</tr>
<tr>
<td>omnetpp</td>
<td>4.8</td>
<td>19.4</td>
<td>22.1</td>
<td>11.8</td>
<td>11.2</td>
<td>27.7</td>
</tr>
<tr>
<td>sjeng</td>
<td>9.6</td>
<td>20.4</td>
<td>17.8</td>
<td>10.3</td>
<td>11.4</td>
<td>13.9</td>
</tr>
<tr>
<td>xalan</td>
<td>1.5</td>
<td>6.6</td>
<td>6.7</td>
<td>5.1</td>
<td>4.7</td>
<td>9.4</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>13.1</strong></td>
<td><strong>26.9</strong></td>
<td><strong>27.3</strong></td>
<td><strong>14.9</strong></td>
<td><strong>21.2</strong></td>
<td><strong>25.6</strong></td>
</tr>
</tbody>
</table>

Deadness in SPEC CPU2006-INT with different compilers and optimization levels.