The test-debug cycle

• The test/debug cycle
  – Develop;
  – While(testing shows up errors)
    {Debug;Fix};

• How do we know when to stop testing (later)

• How do we use a failed test to debug?
  – Go to point where error occurred
  – Consider path up to that point
  – Only some of the program is relevant

• CFG and dataflow is useful

• Extend these techniques – **Program slicing**
  (A slice is a subset of the program statements relevant to some aspect of the program)

Slicing and Debugging

Some of these slides are re-used with kind permission of Professor Paolo Tonella of Centro per la Ricerca Scientifica e Tecnologica Trento, Italy.

Prof. Tonella is an internationally leading expert on source code analysis and manipulation.
Program slicing is a decomposition technique aimed at determining the subset of statements relevant to a computation of interest. It provides the program statements which directly or indirectly contribute to the value of a given variable \( x \) at a given statement \( n \).

Example

Example program

1. \texttt{Read(n)}
2. \texttt{i:=1;}
3. \texttt{sum := 0;}
4. \texttt{prod := 1;}
5. \texttt{while( i <= n ) do}
6. \texttt{sum := sum + i;}
7. \texttt{prod := prod * i;}
8. \texttt{i := i+1;}
9. \texttt{write(sum);}
10. \texttt{write(prod)}

Slice w.r.t. \((10, \texttt{prod})\)

1. \texttt{Read(n)}
2. \texttt{i:=1;}
3. \texttt{prod := 1;}
4. \texttt{while( i <= n ) do}
5. \texttt{prod := prod * i;}
6. \texttt{i := i+1;}
7. \texttt{write(prod)}
Definition of Program Slicing

• Program slicing
  – The process of computing a *program slice*

• Program slice
  – The part of a program that affect the values computed at some *slicing criterion*

• Slicing Criterion
  – is usually a pair (line, variable)

• Static Program Slicing
  – By analysing the text of the program (only)

• Dynamic Slicing
  – By making assumptions about the input
    – eg by executing the program for some input

---

**program slicing**

the essential idea ...  

which other lines affect the selected line?  

---  

don’t waste time on the grey part  

reuse only the red part

we only care about this line
Application of slicing to Debugging

– Mark Weiser invented slicing in his 1979 PhD thesis
– Weiser claimed this is what programmers do anyway when debugging
– Say Program computes x erroneously
  • Look at statements that can affect x
  • Working backwards from occurrence of error
  • Why consider LoC which cannot contribute to error?

Approaches to static slicing

• CFG/Dataflow based
• Program Dependency Graph (PDG) based
• PDG
  – Nodes for nodes in CFG
  – Edges for data and control dependencies
  – Slicing criterion is node in PDG
  – Slice corresponds to reachable part of PDG
• **Backward and Forward slicing**
  - **Backward:**
    - All statements that affect the slicing criterion
  - **Forward:**
    - All statements that are affected by the slicing criterion

• **Affect/Affected by:**
  - through dataflow or control flow
  - Dataflow affect - the value of the variable
  - Control flow affect –whether the statement is executed
Example: data and control dependence

1. Read(n)
2. i:=1;
3. sum := 0;
4. prod := 1;
5. while( i <= n ) do
6. sum := sum + i;
7. prod := prod * i;
8. i := i+1;
9. write(sum);
10. write(prod)

• 7 is data dependent on 4
  – 4,5,6,7 is d-clear du for prod

• 7 is control dependent on 5
  – Node 5 decides whether
    Node 7 is definitely executed
    … or
    Possibly avoided

Static Slicing – data dependence

• Uses the DU annotated CFG

• Node j data dependant on node i if
  – i is a def of some variable x
  – j is a use of x
  – there is a path from i to j (which is d-clear for x)

• Def of x at node i is a reaching definition for node j
Static slicing - control dependence

• Node $i$ is \textit{post dominated} by node $j$ if:
  – All paths from $i$ to STOP pass through $j$
• Node $j$ is \textit{control dependent} on node $i$ if:
  – $i$ is not post dominated by $j$, and
  – there is a path $P$ from $i$ to $j$,
    s.t. for any node $k$, $k \neq i$, in $P$,
    $k$ is post-dominated by $j$

Control Dependence – easy version

Node $j$ is control dependent on node $i$ iff $i$ “decides” (or controls, if you prefer) whether $j$ is definitely executed or whether it might possibly be avoided. So $i$ must be a predicate node (having two edges). One edge always reaches $j$ The other edge might manage to reach the stop node without going through $j$. 
Example: data and control dependence

1. Read(n)
2. i:=1;
3. sum := 0;
4. prod := 1;
5. while( i <= n ) do
6. sum := sum + i;
7. prod := prod * i;
8. i := i+1;
9. write(sum);
10. write(prod)

• 7 is \textit{data dependent} on 4
  
  \begin{itemize}
  \item 4,5,6,7 is d-clear du for prod
  \end{itemize}

• 7 is \textit{control dependent} on 5
  
  \begin{itemize}
  \item 6 is post dominated by 7
  \item 5 is not post dominated by 7
  \end{itemize}

Applications of program slicing

• Debugging (as we saw).
• Testing (regression).
• Parallelization.
• Integration of different program versions.
• Understanding (divide and conquer).
• Maintenance.
• Measurement (cohesion).
Emerging New Applications of program slicing

- Testing (reducing search space size)
- Obfuscation
- Security
- Virus removal
- Modal checking
- Clone detection
- Data mining
- Understanding business processes?

Control flow graph

```
1 x = 1;
2 y = 2;
3 if (c)
4   x++;  
5 while (x > 0) {
6   x--;  
7    y += x;  
8 printf("%d %d\n", x, y);
```
Data dependences

**Data dependence**: A data dependence holds between nodes \( n \) and \( m \) on variable \( x \) if \( n \) defines \( x \), \( m \) uses \( x \) and a path exists in the control flow graph from \( n \) to \( m \) along which \( x \) is not redefined.

```
x = 1;
x++; while (x>0) x--; y += x;
x = 1;
y = 2;
if (c) x++; 
while (x > 0) {
x--;  
y += x;
}
printf("%d %d\n", x, y);
```
Example of data dependences

main()
{
    scanf("%d", &a); {a:2,3,4,5,8}
    if (a == 3) {}
    x = a; {x:8}
    else if (a == 4) {}
        a++; {a:8}
    else while (x) {}  
        x--; {x:6,7,8}
    printf("%d %d", a, x); {}  
}

Control dependences

Control dependence:
node \( n \) holds a control dependence on node \( m \) if a path exists in the CFG from \( n \) to \( m \), whose intermediate nodes are postdominated by \( m \), while \( n \) is not postdominated by \( m \).
Example of control dependences

```c
main()
{
    if (a == c) {
        d--;      {}
        while (a < d)        {3, 4}  
            a++; }            {}
    else while (a > d) {    {5, 6, 7}
        a--;                 {} 
        c++; }              {} 
    printf("%d", a);        {} 
}
```

Slicing algorithm

A static slice $S(x, n)$ can be computed as the transitive closure of the data dependences ending at $n$ on $x$, and of the control dependences ending at $n$.

```latex
S(x, n) := data and control dep. ending at n 
while $S(x, n)$ increases 
    $S(x, n) := S(x, n) \cup$ data and control dep. ending at a node in $S(x, n)$ 
end while
```
main() {
  inword = NO;
  nl = 0;
  nw = 0;
  nc = 0;
  c = getchar();
  while (c != EOF) {
    nc = nc + 1;
    if (c == '
' || c == ' ' || c == '	')
      inword = NO;
    else if (inword == NO) {
      inword = YES;
      nw = nw + 1;
    }
    c = getchar();
  }
  printf("%d \n", nl);
  printf("%d \n", nw);
  printf("%d \n", nc);
}

Slicing example: S(nl, 16)

main() {
  inword = NO;
  nl = 0;
  nw = 0;
  nc = 0;
  c = getchar();
  while (c != EOF) {
    nc = nc + 1;
    if (c == ' ' || c == '
' || c == '	')
      inword = NO;
    else if (inword == NO) {
      inword = YES;
      nw = nw + 1;
    }
    c = getchar();
  }
  printf("%d \n", nl);
  printf("%d \n", nw);
  printf("%d \n", nc);
}

Slicing example: S(nw, 17)
Slicing example:
S(nc, 18)

```c
main() {
    inword = NO;
    nl = 0;
    nw = 0;
    nc = 0;
    c = getchar();
    while (c != EOF) {
        nc = nc + 1;
        if (c == '
')
            nl = nl + 1;
        if (c == ' ' || c == '
' || c == '	')
            inword = NO;
        else if (inword == NO) {
            inword = YES;
            nw = nw + 1;
        }
        c = getchar();
    }
    printf("%d \n", nl);
    printf("%d \n", nw);
    printf("%d \n", nc);
}
```

Program dependence graph

Slice = backward reachable nodes

S(y, 8)

```c
1 x = 1;
2 y = 2;
3 if (c)
4   x++;
5 while(d) {
6   x--;
7   y += 1;
   }
8  printf("%d", y);
```
Problems with slicing real code

- Unstructured code (goto’s)
- Interprocedural slicing
- Pointers
- Parameter aliasing

Empirical results on Slicing
does size matter?

you bet it does

in all applications

so … just how big is a typical slice?

does calling context matter?

some authors say yes

some authors say no

The results I report come from a large scale study I did with Dave Binkley
A lot of C code was analysed over the whole study.

### Program Statistics

<table>
<thead>
<tr>
<th>Program</th>
<th>Size (LOC)</th>
<th>Vertices</th>
<th>Edges</th>
<th>Slices</th>
<th>Time (sec)</th>
<th>Slicing Pace (KLOC/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>replace</td>
<td>563</td>
<td>1,097</td>
<td>3,500</td>
<td>867</td>
<td>0.02</td>
<td>9.417</td>
</tr>
<tr>
<td>which</td>
<td>6,033</td>
<td>4,583</td>
<td>14,692</td>
<td>1,049</td>
<td>0.03</td>
<td>14.922</td>
</tr>
<tr>
<td>compres</td>
<td>2,134</td>
<td>5,211</td>
<td>14,749</td>
<td>1,085</td>
<td>0.06</td>
<td>2.805</td>
</tr>
<tr>
<td>wdiff.0.5</td>
<td>8,988</td>
<td>7,211</td>
<td>21,793</td>
<td>2,421</td>
<td>0.12</td>
<td>3.296</td>
</tr>
<tr>
<td>termtools</td>
<td>6,697</td>
<td>8,869</td>
<td>27,442</td>
<td></td>
<td></td>
<td>32.424</td>
</tr>
<tr>
<td>bison</td>
<td>5,622</td>
<td>10,916</td>
<td>35,796</td>
<td></td>
<td></td>
<td>3.215</td>
</tr>
<tr>
<td>gcc</td>
<td>14,609</td>
<td>14,903</td>
<td>49,533</td>
<td>5,705</td>
<td>0.75</td>
<td>6.169</td>
</tr>
<tr>
<td>gcc-cpp</td>
<td>9,536</td>
<td>18,331</td>
<td>80,584</td>
<td>7,322</td>
<td>0.97</td>
<td>6.783</td>
</tr>
<tr>
<td>spacec</td>
<td>4,079</td>
<td>9,093</td>
<td>29,513</td>
<td>8,093</td>
<td>0.97</td>
<td>2.572</td>
</tr>
<tr>
<td>propel</td>
<td>9,126</td>
<td>20,018</td>
<td>64,537</td>
<td>10,579</td>
<td>1.35</td>
<td>7.240</td>
</tr>
<tr>
<td>orgmode2</td>
<td>14,328</td>
<td>20,578</td>
<td>65,253</td>
<td>10,579</td>
<td>1.35</td>
<td>5.473</td>
</tr>
<tr>
<td>EPNIC-x1</td>
<td>14,326</td>
<td>20,564</td>
<td>65,734</td>
<td>10,846</td>
<td>1.38</td>
<td>4.329</td>
</tr>
<tr>
<td>groops</td>
<td>2,218</td>
<td>22,217</td>
<td>79,026</td>
<td>12,236</td>
<td>1.04</td>
<td>5.267</td>
</tr>
<tr>
<td>indent</td>
<td>7,229</td>
<td>25,138</td>
<td>80,922</td>
<td>12,236</td>
<td>1.04</td>
<td>4.139</td>
</tr>
<tr>
<td>finndie6</td>
<td>6,100</td>
<td>25,737</td>
<td>582,728</td>
<td>6,602</td>
<td>0.44</td>
<td>1.511</td>
</tr>
<tr>
<td>finndie5</td>
<td>18,691</td>
<td>30,328</td>
<td>170,697</td>
<td>14,370</td>
<td>2.31</td>
<td>3.215</td>
</tr>
<tr>
<td>byacc</td>
<td>6,337</td>
<td>34,010</td>
<td>104,341</td>
<td>978</td>
<td>0.16</td>
<td>1.098</td>
</tr>
<tr>
<td>diffutils</td>
<td>18,374</td>
<td>36,075</td>
<td>166,603</td>
<td>16,665</td>
<td>1.04</td>
<td>2.902</td>
</tr>
<tr>
<td>flex2-4-7</td>
<td>15,143</td>
<td>39,037</td>
<td>186,523</td>
<td>11,104</td>
<td>1.09</td>
<td>1.756</td>
</tr>
<tr>
<td>caldp</td>
<td>11,068</td>
<td>42,912</td>
<td>193,210</td>
<td>16,665</td>
<td>1.04</td>
<td>1.451</td>
</tr>
<tr>
<td>copas</td>
<td>1,170</td>
<td>8,848</td>
<td>35,347</td>
<td>3,672</td>
<td>0.33</td>
<td>4.174</td>
</tr>
<tr>
<td>el</td>
<td>12,493</td>
<td>18,672</td>
<td>64,655</td>
<td>16,665</td>
<td>1.04</td>
<td>1.662</td>
</tr>
<tr>
<td>gnuchock</td>
<td>16,659</td>
<td>14,657</td>
<td>163,465</td>
<td>16,665</td>
<td>1.04</td>
<td>1.701</td>
</tr>
<tr>
<td>wpst</td>
<td>17,321</td>
<td>34,638</td>
<td>209,465</td>
<td>20,667</td>
<td>1.09</td>
<td>1,316</td>
</tr>
<tr>
<td>flex2-5-4</td>
<td>20,252</td>
<td>55,019</td>
<td>356,413</td>
<td>41,114</td>
<td>1.61</td>
<td>1,108</td>
</tr>
<tr>
<td>rlpd</td>
<td>15,914</td>
<td>56,981</td>
<td>366,093</td>
<td>24,820</td>
<td>1.25</td>
<td>1,225</td>
</tr>
<tr>
<td>tbl-fort.h</td>
<td>3,717</td>
<td>59,247</td>
<td>215,255</td>
<td></td>
<td></td>
<td>142.44</td>
</tr>
<tr>
<td>libpq</td>
<td>24,822</td>
<td>62,698</td>
<td>265,639</td>
<td></td>
<td></td>
<td>193.84</td>
</tr>
<tr>
<td>user=0.95</td>
<td>7,150</td>
<td>70,796</td>
<td>272,726</td>
<td>12,511</td>
<td>1.04</td>
<td>270</td>
</tr>
<tr>
<td>expresso</td>
<td>22,059</td>
<td>93,326</td>
<td>467,165</td>
<td>19,044</td>
<td>1.04</td>
<td>523</td>
</tr>
<tr>
<td>go</td>
<td>23,547</td>
<td>111,246</td>
<td>416,460</td>
<td></td>
<td></td>
<td>2.845</td>
</tr>
<tr>
<td>rings</td>
<td>16,946</td>
<td>120,144</td>
<td>422,051</td>
<td></td>
<td></td>
<td>3.364</td>
</tr>
<tr>
<td>mntd</td>
<td>45,647</td>
<td>204,420</td>
<td>804,652</td>
<td>39,573</td>
<td>1.04</td>
<td>600</td>
</tr>
<tr>
<td>groups</td>
<td>15,217</td>
<td>278,760</td>
<td>1,044,524</td>
<td>56,373</td>
<td>1.04</td>
<td>147</td>
</tr>
<tr>
<td>csiufglpk</td>
<td>36,593</td>
<td>259,220</td>
<td>973,760</td>
<td></td>
<td></td>
<td>1,363</td>
</tr>
<tr>
<td>saps</td>
<td>53,900</td>
<td>417,513</td>
<td>6,303,609</td>
<td></td>
<td></td>
<td>3,835</td>
</tr>
<tr>
<td>empire</td>
<td>53,893</td>
<td>446,833</td>
<td>4,902,235</td>
<td></td>
<td></td>
<td>7,664</td>
</tr>
<tr>
<td>li</td>
<td>6,916</td>
<td>889,260</td>
<td>3,762,858</td>
<td>13,251</td>
<td>1.04</td>
<td>518.00</td>
</tr>
<tr>
<td>sendmail</td>
<td>25,195</td>
<td>1,111,054</td>
<td>37,774,239</td>
<td></td>
<td></td>
<td>35,017</td>
</tr>
<tr>
<td>ssh</td>
<td>25,195</td>
<td>1,111,054</td>
<td>37,774,239</td>
<td></td>
<td></td>
<td>35,017</td>
</tr>
</tbody>
</table>

**Figure 1.** The subject programs studied (sorted by number of vertices)

The study I report to you today was

**43 C programs**

over **one million lines of code**

**2,353,598 slices**

the largest study to date
how do you choose slicing criterion?

if you want to produce a benchmark what criteria should you slice on?

right!
simply used every possible criteria

two details …
what about pointers?

flow sensitive algorithms are impractical
context sensitive algorithms are impractical
Steensgaard is too imprecise
currently Andersen is the right trade off

2.5 increase in slice size for ijpeg

what about structure fields?

expanded or collapsed?

a speed/precision trade off

particular strong effect for function pointers

we give data for both
<table>
<thead>
<tr>
<th>Program</th>
<th>Structure Fields Collapsed</th>
<th>Structure Fields Expanded</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LOC</td>
<td>Slice Size</td>
</tr>
<tr>
<td>replace</td>
<td>356</td>
<td>122</td>
</tr>
<tr>
<td>copia</td>
<td>1,170</td>
<td>226</td>
</tr>
<tr>
<td>compress</td>
<td>1,234</td>
<td>307</td>
</tr>
<tr>
<td>file-dirth-2</td>
<td>3,717</td>
<td>1,059</td>
</tr>
<tr>
<td>gec-cpp</td>
<td>4,079</td>
<td>1,065</td>
</tr>
<tr>
<td>which</td>
<td>4,880</td>
<td>1,484</td>
</tr>
<tr>
<td>barcode</td>
<td>5,562</td>
<td>1,763</td>
</tr>
<tr>
<td>widf-0.5</td>
<td>5,958</td>
<td>509</td>
</tr>
<tr>
<td>time-1.0</td>
<td>6,033</td>
<td>319</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>LOC</th>
<th>Slice Size</th>
<th>Percentage</th>
<th>LOC</th>
<th>Slice Size</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>replace-1.10</td>
<td>39</td>
<td>250</td>
<td>65.2%</td>
<td>39</td>
<td>250</td>
<td>65.2%</td>
</tr>
<tr>
<td>byacc</td>
<td>6,337</td>
<td>1,147</td>
<td>18.1%</td>
<td>1,147</td>
<td>18.1%</td>
<td>1,147</td>
</tr>
<tr>
<td>tmultm</td>
<td>6,907</td>
<td>1,094</td>
<td>25.5%</td>
<td>1,094</td>
<td>25.5%</td>
<td>1,094</td>
</tr>
<tr>
<td>li</td>
<td>9,196</td>
<td>2,060</td>
<td>38.9%</td>
<td>2,060</td>
<td>38.9%</td>
<td>2,060</td>
</tr>
<tr>
<td>user-0.95-0.0</td>
<td>7,150</td>
<td>1,480</td>
<td>20.7%</td>
<td>1,480</td>
<td>20.7%</td>
<td>1,480</td>
</tr>
<tr>
<td>pmultm</td>
<td>7,229</td>
<td>2,082</td>
<td>28.8%</td>
<td>2,082</td>
<td>28.8%</td>
<td>2,082</td>
</tr>
<tr>
<td>epwps-1</td>
<td>8,631</td>
<td>1,010</td>
<td>11.7%</td>
<td>1,010</td>
<td>11.7%</td>
<td>1,010</td>
</tr>
<tr>
<td>spare</td>
<td>9,170</td>
<td>1,933</td>
<td>21.2%</td>
<td>1,933</td>
<td>21.2%</td>
<td>1,933</td>
</tr>
<tr>
<td>dec-5.3</td>
<td>9,536</td>
<td>715</td>
<td>7.2%</td>
<td>715</td>
<td>7.2%</td>
<td>715</td>
</tr>
<tr>
<td>cdsp</td>
<td>11,698</td>
<td>1,478</td>
<td>9.3%</td>
<td>1,478</td>
<td>9.3%</td>
<td>1,478</td>
</tr>
<tr>
<td>ed</td>
<td>12,493</td>
<td>6,759</td>
<td>54.1%</td>
<td>6,759</td>
<td>54.1%</td>
<td>6,759</td>
</tr>
<tr>
<td>oracolo2</td>
<td>14,326</td>
<td>2,822</td>
<td>19.2%</td>
<td>2,822</td>
<td>19.2%</td>
<td>2,822</td>
</tr>
<tr>
<td>ipreg</td>
<td>14,328</td>
<td>2,780</td>
<td>19.4%</td>
<td>2,780</td>
<td>19.4%</td>
<td>2,780</td>
</tr>
<tr>
<td>bc</td>
<td>14,609</td>
<td>7,173</td>
<td>49.1%</td>
<td>7,173</td>
<td>49.1%</td>
<td>7,173</td>
</tr>
<tr>
<td>fls-2.4-7</td>
<td>15,143</td>
<td>3,755</td>
<td>24.8%</td>
<td>3,755</td>
<td>24.8%</td>
<td>3,755</td>
</tr>
</tbody>
</table>

| fls-2.4-7 | 15,143 | 3,755 | 24.8% | 3,755 | 24.8% | 3,755 | 24.8% |

Figure 3. Slices sizes for backward and forward interprocedural slices using both collapsed and expanded structure fields.

Figure 3. The effect of calling context on slice size and computation time.
summary of our findings

Slicing is now fast: 3MLoC / Sec
‘Typical’ slice: 1/3rd of a program
Calling context matters: Size↑50%  Time↑66%

A Survey of Program Slicing
This and much more in:
• Frank Tip
  A Survey of Program Slicing Techniques
  • Technical Report,
  • Computer Science / Dept of Software Technology
  • Centrum voor Wiskunde en Informatica, Netherlands
  • Number CS-R9438, 1994
  • This is quite a technical paper. On the next slide I suggest two easier papers by yours truly and colleagues…
Other surveys

David Binkley and Mark Harman.
_A Survey of Empirical Results on Program Slicing._
Advances in Computers

Mark Harman and Rob Hierons.
_An Overview of Program Slicing_
(A postscript version of also available.)

These are both available from my publications page at

_http://www.dcs.kcl.ac.uk/staff/mark/papers.html_