Decreasing Process Memory Requirements by Overlapping Program Portions

by

Richard Bowman
Emily Ratliff
David Whalley

Computer Science Department
Florida State University
Motivation for Decreasing Process Memory Requirements

• May allow embedded systems to meet their strict limitations on program size.

• May improve memory hierarchy performance.
  — reduce cache misses
  — reduce page faults

• May help offset increases in code size due to code increasing compiler transformations.

• Automatic overlapping supports the software engineering principle of using descriptive variable names.
Areas for Overlapping Program Portions

1. overlap run-time stack data
2. overlap uninitialized static data
3. overlap uninitialized static data and initialized static data
4. overlap instructions
5. overlap uninitialized static data and instructions
Overlapping Data

- Used a graph coloring approach to detect conflicting live ranges.

- Issues
  - Detecting indirectly referenced live ranges.
  - Detecting live ranges of static data used in more than one function.
  - Assigning memory locations to live ranges.
Indirectly Referenced Live Ranges

- Indirectly referenced variables are treated as having a single live range.
- Interference graph nodes not directly connected can be overlapped in memory.

control flow graph

interference graph

live range = possible predecessors \( \cap \) possible successors

live range of \( a[] \) = [1,2,3,4,5] \( \cap \) [2,3,4,5,6,7,8,9] = [2,3,4,5]
live range of \( b[] \) = [1,2,3,4] \( \cap \) [2,3,4,5,6,7,8,9] = [2,3,4]
live range of \( c[] \) = [1,2,3,4,5,6,7,8] \( \cap \) [6,7,8,9] = [6,7,8]
live range of \( d[] \) = [1,2,3,4,5,6,7,8] \( \cap \) [5,6,7,8,9] = [5,6,7,8]
Determining Where Indirectly Taken Addresses are Dereferenced

Source Code:    
\[
\text{main()}
\{
    \text{int } a[100][100];
    \text{int } i, j;
    \text{for (} i=0; i<100; i++ \text{)}
    \\text{for (} j=0; j<100; j++ \text{)}
        a[i][j]=0;
\}
\]

Machine Instructions:

1. \( \text{r}[12]=\text{r}[14]+a; \)
2. \( \text{r}[5]=0; \)
3. \( \text{r}[6]=\text{HI}[40000]; \)
4. \( \text{r}[6]=\text{r}[6]+\text{LO}[40000]; \)
5. \( \text{r}[1]=\text{r}[5]+\text{r}[12]; \)
6. \( \text{r}[2]=\text{r}[1]+400; \)
7. \( \text{M[}\text{r}[1]]\text{=}0; \)
8. \( \text{r}[1]=\text{r}[1]+4; \)
9. \( \text{IC=}\text{r}[1]?\text{r}[2]; \)
10. \( \text{PC=}\text{IC}<0->\text{L19}; \)
11. \( \text{r}[5]=\text{r}[5]+400; \)
12. \( \text{IC=}\text{r}[5]?\text{r}[6]; \)
13. \( \text{PC=}\text{IC}<0->\text{L16}; \)
14. \( \text{PC=}\text{RT}; \)
Detecting Live Ranges across Functions

- Calculate live ranges without propagating information into called functions.
  
  initial live range of $x = [1,2,4,5,6] \cap [3,5,6,7] = [5,6]$
  
- Include blocks within the functions that are called within the live range.
  
  updated live range of $x = [5,6] \cup [11,12,13] = [5,6,11,12,13]$
Assigning Variables to Memory Locations

```c
int x[10];
int y[] = { 0, 1 };
int g = -1;
short s;
...
printf("Data: ");
```

(a) C Code Segment

```assembly
.seg "data"
.global _x
_x:
.L19:
.ascii "Data: \0"
.skip 1
.global _s
_s:
.skip 2
.global _y
_y:
.word 0
.word 1
.global _g
_g:
.word -1
.global _main
_main:
.save %sp,-96,%sp ! first inst within _main
...
.seg "text" ! switch to the code segment
... ! all insts not overlapped with data
```

(b) Offset Assignment

(c) SPARC Assembly Directives and Code
Overlapping Instructions by Cross Jumping

- Performed on jumps and calls.
- The compiler examines sets and uses to allow cross jumping of noncontiguous sequences of instructions.

<table>
<thead>
<tr>
<th>Before Cross Jumping</th>
<th>After Cross Jumping</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Call 1</strong></td>
<td><strong>Call 1</strong></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$r[10] = 1$</td>
<td>$r[10] = 1$</td>
</tr>
<tr>
<td>CALL _pfnote();</td>
<td>CALL _newlabel();</td>
</tr>
</tbody>
</table>

| **Call 2**           | **Call 2**          |
| ...                  | ...                |
| CALL _pfnote();      | CALL _pfnote();    |

| **Call 3**           | **Call 3**          |
| ...                  | ...                |
| $r[10] = 1$          | $r[10] = 1$        |
| CALL _pfnote();      | CALL _newlabel();  |
| ...                  | ...                |

**function entry**

_pfnote:


...
Abstracting Relocatable Code Portions

- Can overlap a relocatable code portion with a subset of another.

<table>
<thead>
<tr>
<th>Before Overlapping</th>
<th>After Overlapping</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image.png" alt="Diagram" /></td>
<td><img src="image.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>
Overlapping Static Data and Instructions

- Nonconflicting relocatable code portions and uninitialized static data can be overlapped in the initialized data segment.

```c
... char string[432];

main(argc, argv) char *argv[];
{
    int y, i, j;
    int m;
    
    if(argc < 2) {
        printf(...);
        exit(0);
    }
    
    m = number(argv[1]);
    
    cal(m,y,string,24);
    
}  

number(str)
char *str;
{
    ... 
}
...

(a) Portion of cal Program

(b) Mapping string with Static Data and Relocatable Code Segments

<table>
<thead>
<tr>
<th>name</th>
<th>address range</th>
<th>num bytes</th>
<th>bytes saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>000-431</td>
<td>432</td>
<td>0</td>
</tr>
<tr>
<td>L31</td>
<td>000-024</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>L74</td>
<td>025-038</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>L43</td>
<td>039-048</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>L55</td>
<td>049-056</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>L54</td>
<td>057-060</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>L44</td>
<td>061-064</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>L56</td>
<td>065-066</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>block range</th>
<th>address range</th>
<th>num bytes</th>
<th>bytes saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-3</td>
<td>068-103</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>42-44</td>
<td>104-123</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>45-45</td>
<td>124-135</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>46-50</td>
<td>136-199</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>51-51</td>
<td>200-207</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>4-18</td>
<td>268-483</td>
<td>216</td>
<td>164</td>
</tr>
</tbody>
</table>
```
Results after Inlining and Cloning

- Code increasing transformations provide additional overlapping opportunities.

<table>
<thead>
<tr>
<th>Program</th>
<th>Overlapping Run-Time Stack Data with Inlining</th>
<th>Overlapping Instructions with Cloning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bytes Orig</td>
<td>Pct Less</td>
</tr>
<tr>
<td>cal</td>
<td>232</td>
<td>3.45%</td>
</tr>
<tr>
<td>cmp</td>
<td>192</td>
<td>0.00%</td>
</tr>
<tr>
<td>csplit</td>
<td>728</td>
<td>0.00%</td>
</tr>
<tr>
<td>ctags</td>
<td>24544</td>
<td>0.36%</td>
</tr>
<tr>
<td>dhrystone</td>
<td>200</td>
<td>4.00%</td>
</tr>
<tr>
<td>grep</td>
<td>304</td>
<td>0.00%</td>
</tr>
<tr>
<td>join</td>
<td>96</td>
<td>0.00%</td>
</tr>
<tr>
<td>lex</td>
<td>7208</td>
<td>0.11%</td>
</tr>
<tr>
<td>linpack</td>
<td>3312</td>
<td>3.38%</td>
</tr>
<tr>
<td>mincost</td>
<td>192</td>
<td>4.17%</td>
</tr>
<tr>
<td>sdiff</td>
<td>5784</td>
<td>0.28%</td>
</tr>
<tr>
<td>tr</td>
<td>96</td>
<td>0.00%</td>
</tr>
<tr>
<td>tsp</td>
<td>2216</td>
<td>2.53%</td>
</tr>
<tr>
<td>whetstone</td>
<td>488</td>
<td>60.66%</td>
</tr>
<tr>
<td>yacc</td>
<td>1360</td>
<td>0.59%</td>
</tr>
<tr>
<td>average</td>
<td>3130</td>
<td>5.30%</td>
</tr>
</tbody>
</table>
Future Work

• Obtain more accurate live ranges of arrays.

• Overlap fields within a structure.

• Measure effect on unified secondary caches and paging.
Conclusions

• Overlapping uninitialized static data with static data and instructions was shown to be quite beneficial.

• Over 10% of the memory requirements of a program was eliminated.

• Code increasing transformations provide additional overlapping opportunities for instructions and run-time stack data.

• More accurate live range analysis of arrays should result in improved results.