MATCHUP: Memory Abstractions for Heap Manipulating Programs

Felix Winterstein, Kermin Fleming, Hsin-Jung Yang, Samuel Bayliss, George Constantinides

23 February 2015
Background

Good HLS tools (Vivado HLS, LegUp, etc.) ...

```c
void Sobel (...) {
    ...
    for (y = 1; y < N; y++) {
        for (x = 1; x < M; x++) {
            pixel_value = 0;
            for (j = -1; j <= 1; j++) {
                for (i = -1; i <= 1; i++) {
                    pixel_value +=
                        weight[j + 1][i + 1] * 
                        image[y + j][x + i];
                }
            }
        }
    }
    ...
```
void Sobel (...) {

... 
for (y = 1; y < N; y++) {
    for (x = 1; x < M; x++) {
        pixel_value = 0;
        for (j = -1; j <= 1; j++) {
            for (i = -1; i <= 1; i++) {
                pixel_value += weight[j + 1][i + 1] * image[y + j][x + i];
            }
        }
        ...
    }
}

Good HLS results
Good HLS tools (Vivado HLS, LegUp, etc.) ...

```c
void Sobel (...) {
    ...
    for (y = 1; y < N; y++) {
        for (x = 1; x < M; x++) {
            pixel_value = 0;
            for (j = -1; j <= 1; j++) {
                for (i = -1; i <= 1; i++) {
                    pixel_value += weight[j + 1][i + 1] * image[y + j][x + i];
                }
            }
        }
    }
    ...
}
```

```c
s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
    t = s;
    u = t->u;
    s = t->n;
    delete t;
    ... do something
    if (u->left!= 0) && (u->right!=0) then
        s = PUSH(u->right, s);
        s = PUSH(u->left, s);
    end if
    delete u;
end while
```
void Sobel (...) {
  ...
  for (y = 1; y < N; y++) {
    for (x = 1; x < M; x++) {
      pixel_value = 0;
      for (j = -1; j <= 1; j++) {
        for (i = -1; i <= 1; i++) {
          pixel_value += weight[j + 1][i + 1] * image[y + j][x + i];
        }
      }
    }
  }
  ...
}

s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
  t = s;
  u = t->u;
  s = t->n;
  delete t;
  ... do something
  if (u->left!= 0) && (u->right!=0) then
    s = PUSH(u->right, s);
    s = PUSH(u->left, s);
  end if
  delete u;
end while
Good HLS tools (Vivado HLS, LegUp, etc.) ...

```c
void Sobel (...) {
    ...
    for (y = 1; y < N; y++) {
        for (x = 1; x < M; x++) {
            pixel_value = 0;
            for (j = -1; j <= 1; j++) {
                for (i = -1; i <= 1; i++) {
                    pixel_value += weight[j + 1][i + 1] * image[y + j][x + i];
                }
            }
        }
    }
    ...
}
```

Good HLS results

```c
s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
    t = s;
    u = t->u;
    s = t->n;
    delete t;
    ... do something
    if (u->left!= 0) && (u->right!=0) then
        s = PUSH(u->right, s);
        s = PUSH(u->left, s);
    end if
    delete u;
end while
```
Good HLS tools (Vivado HLS, LegUp, etc.) ... 

```c
void Sobel (...) {
    ... 
    for (y = 1; y < N; y++) {
        for (x = 1; x < M; x++) {
            pixel_value = 0;
            for (j = -1; j <= 1; j++) {
                for (i = -1; i <= 1; i++) {
                    pixel_value += weight[j + 1][i + 1] * image[y + j][x + i];
                }
            }
        }
    }
    ... 
}
```

Good HLS results

```c
s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
    t = s;
    u = t->u;
    s = t->n;
end while
```

Doesn’t synthesize

```c
s = PUSH(u->left, s);
end if
delete u;
end while
```
Background

Challenges

• Memory grows at run-time
• Parallelization: Determine data dependencies (pointer aliasing)

```
s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
    t = s;
    u = t->u;
    s = t->n;
```
Off-chip memory (low bandwidth)

Interface controller

FPGA

memory

heap

s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
    t = s;
    u = t->u;
    s = t->n;
Off-chip memory (low bandwidth)

Interface controller

FPGA

HLS

memory

heap

s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
    t = s;
    u = t->u;
    s = t->n;
Off-chip memory (low bandwidth)

Interface controller

FPGA

memory

private

private

heap

s = new stackRecord;

s->u = root;

s->n = 0;

while s!=0 do

    t = s;

    u = t->u;

    s = t->n;


Tailor made memory system

Off-chip memory (low bandwidth)

Interface controller

HLS

Coherency network

memory

private

private

shared

heap

s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
    t = s;
    u = t->u;
    s = t->n;
Static program analysis

• for pointer-based programs
• Identify private memory regions:
  – Synthesize “private” caches
  – Independent, cheap, fast
• Identify shared memory regions:
  – Synthesize “coherent” caches
  – Complex, expensive, slow(er)

Automated synthesis tool

• Application specific caching scheme
• Parallelization

memory

private

s = new stackRecord;
s->u = root;
s->n = 0;
while s != 0 do
  t = s;
  u = t->u;
  s = t->n;
Executive summary

- 8x speed-up from parallel caches (average)
- 49% area-time savings from application specificity (average)
1. Find private heap regions
2. Find shared heap regions
3. Legal parallelization in the presence of shared heap
Find private heap regions

- Private regions are independent
- Statements access different memory locations
- What is the problem with pointers?

```cpp
s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
t = s;
u = t->u;
s = t->n;
delete t;
... do something
if (u->left!= 0) && (u->right!=0) then
    s = PUSH(u->right, s);
    s = PUSH(u->left, s);
end if
delete u;
end while
```
Find private heap regions

- Private regions are independent
- Statements access different memory locations
- What is the problem with pointers?

```c
s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
  t = s;
  u = t->u;
  s = t->n;
  delete t;
  ... do something
  if (u->left!= 0) && (u->right!=0)
    s = PUSH(u->right, s);
    s = PUSH(u->left, s);
  end if
  delete u;
end while
```
• Private regions are independent
• Statements access different memory locations
• What is the problem with pointers?

```
s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
    t = s;
    u = t->u;
    s = t->n;
    delete t;
    ... do something
    if (u->left!= 0) & (u->right!=0)
        s = PUSH(u->right, s);
        s = PUSH(u->left, s);
    end if
    delete u;
end while
```

Independent? 1st loop iteration - NO
Find private heap regions

- Private regions are independent
- Statements access different memory locations
- What is the problem with pointers?

```c
s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
    t = s;
    u = t->u;
    s = t->n;
    delete t;
    ... do something
    if (u->left!= 0) && (u->right!=0)
        s = PUSH(u->right, s);
        s = PUSH(u->left, s);
    end if
    delete u;
end while
```

Independent?
1\(^{st}\) loop iteration
- NO
2\(^{nd}\) loop iteration
- YES
All other iterations
- YES
Find private heap regions

• Private regions are independent
• Statements access different memory locations
• What is the problem with pointers?
• Pointers change at runtime
• Syntax analysis doesn’t work
• Our analysis “symbolically executes” the program

```
s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
  t = s;
  u = t->u;
  s = t->n;
  delete t;
  ... do something
  if (u->left!= 0) && (u->right!=0)
    s = PUSH(u->right, s);
  s = PUSH(u->left, s);
end if
delete u;
end while
```

Independent?
1\textsuperscript{st} loop iteration
- NO
2\textsuperscript{nd} loop iteration
- YES
All other iterations
- YES
Real execution (run time)

Heap layout

```
stackRecord 7
stackRecord 6
stackRecord 5
stackRecord 4
stackRecord 3
stackRecord 2
```

```
treeNode 7
treeNode 6
treeNode 5
treeNode 4
treeNode 3
treeNode 2
```

```
s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
  t = s;
  u = t->u;
  s = t->n;
  delete t;
  ... do something
  if (u->left!= 0) && (u->right!=0) then
    s = PUSH(u->right, s);
    s = PUSH(u->left, s);
  end if
  delete u;
end while
```
### Symbolic execution

**Real execution** (run time)

- Heap layout
  - stackRecord 7
  - stackRecord 6
  - stackRecord 5
  - **stackRecord 4**
  - stackRecord 3
  - stackRecord 2
  - treeNode 7
  - treeNode 6
  - treeNode 5
  - treeNode 4
  - treeNode 3
  - treeNode 2

**Symbolic execution** (compile time)

- Formal layout

```java
s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
    t = s;
    u = t->u;
    s = t->n;
    delete t;
    ... do something
    if (u->left!= 0) && (u->right!=0) then
        s = PUSH(u->right, s);
        s = PUSH(u->left, s);
    end if
    delete u;
end while
```

*s points to a record with fields u and n*
Symbolic execution

Real execution (run time)

Heap layout

<table>
<thead>
<tr>
<th>stackRecord 7</th>
<th>stackRecord 6</th>
<th>stackRecord 5</th>
<th>stackRecord 4</th>
<th>stackRecord 3</th>
<th>stackRecord 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>stackRecord 4</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>stackRecord 3</td>
<td></td>
<td><strong>stackRecord 2</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>treeNode 7</td>
<td></td>
<td>treeNode 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>treeNode 6</td>
<td></td>
<td>treeNode 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>treeNode 5</td>
<td></td>
<td>treeNode 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>treeNode 4</td>
<td></td>
<td>treeNode 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>treeNode 3</td>
<td></td>
<td>treeNode 2</td>
</tr>
</tbody>
</table>

Symbolic execution (compile time)

Formal layout

\[
\begin{align*}
\text{s} & \rightarrow [u: u_2', n: s_3'] \\
\text{u} & \rightarrow [l: u_4', r: u_5']
\end{align*}
\]

\[
s = \text{new stackRecord}; \\
\text{s}->u = \text{root}; \\
\text{s}->n = 0; \\
\text{while} \ s! = 0 \ \text{do} \\
\text{t} = \text{s}; \\
\text{u} = \text{t}->u; \\
\text{s} = \text{t}->n; \\
\text{delete t}; \\
\text{... do something} \\
\text{if} \ (\text{u}->\text{left}\neq 0) \ \&\& \ (\text{u}->\text{right}\neq 0) \ \text{then} \\
\text{s} = \text{PUSH}(\text{u}->\text{right}, \text{s}); \\
\text{s} = \text{PUSH}(\text{u}->\text{left}, \text{s}); \\
\text{end if} \\
\text{delete u}; \\
\text{end while}
\]
Symbolic execution

Real execution (run time)

Heap layout

StackRecord 7
StackRecord 6
StackRecord 5
**StackRecord 4**
StackRecord 3
**StackRecord 2**
TreeNode 7
TreeNode 6
TreeNode 5
TreeNode 4
TreeNode 3
TreeNode 2

Symbolic execution (compile time)

Formal layout

\[
\begin{align*}
s_7' & \rightarrow [u: u_7', n: 0] \\
s_6' & \rightarrow [u: u_6', n: s_7'] \\
s_5' & \rightarrow [u: u_5', n: 0] \\
s_4' & \rightarrow [u: u_4', n: s_5'] \\
s_3' & \rightarrow [u: u_3', n: 0] \\
s & \rightarrow [u: u_2', n: s_3'] \\
u_7' & \rightarrow [l: 0, r: 0] \\
u_6' & \rightarrow [l: 0, r: 0] \\
u_5' & \rightarrow [l: 0, r: 0] \\
u_4' & \rightarrow [l: 0, r: 0] \\
u_3' & \rightarrow [l: u_6', r: u_7'] \\
u & \rightarrow [l: u_4', r: u_5']
\end{align*}
\]

Real execution

(s = new stackRecord;
  s->u = root;
  s->n = 0;
  while s!=0 do
    t = s;
    u = t->u;
    s = t->n;
    delete t;
    ... do something
    if (u->left!= 0) && (u->right!=0) then
      s = PUSH(u->right, s);
      s = PUSH(u->left, s);
    end if
    delete u;
  end while)
Symbolic execution

Real execution (run time)

Heap layout

- StackRecord 7
- StackRecord 6
- StackRecord 5
- **StackRecord 4**
- StackRecord 3
- **StackRecord 2**
- TreeNode 7
- TreeNode 6
- TreeNode 5
- TreeNode 4
- TreeNode 3
- TreeNode 2

Symbolic execution (compile time)

Formal layout

- $s_7' \rightarrow [u: u_7', n: 0]$ *
- $s_6' \rightarrow [u: u_6', n: s_7']$ *
- $s_5' \rightarrow [u: u_5', n: 0]$ *
- $s_4' \rightarrow [u: u_4', n: s_5']$ *
- $s_3' \rightarrow [u: u_3', n: 0]$ *
- $s \rightarrow [u: u_2', n: s_3']$ *
- $u_7' \rightarrow [l: 0, r: 0]$ *
- $u_6' \rightarrow [l: 0, r: 0]$ *
- $u_5' \rightarrow [l: 0, r: 0]$ *
- $u_4' \rightarrow [l: 0, r: 0]$ *
- $u_3' \rightarrow [l: u_6', r: u_7']$ *
- $u \rightarrow [l: u_4', r: u_5']$

Separation logic, see paper

Describes heap state and aliasing information

```
s = t->n;
delete t;
... do something
if (u->left!= 0) && (u->right!=0) then
    s = PUSH(u->right, s);
    s = PUSH(u->left, s);
end if
delete u;
end while
```
Symbolic execution

Real execution (run time)

Heap layout

- stackRecord 7
- stackRecord 6
- stackRecord 5
- stackRecord 4
- stackRecord 3
- stackRecord 2
- treeNode 7
- treeNode 6
- treeNode 5
- treeNode 4
- treeNode 3
- treeNode 2

Symbolic execution (compile time)

Formal layout

- \( s_7' \rightarrow [u: u_7', n: 0] \)
- \( s_6' \rightarrow [u: u_6', n: s_7'] \)
- \( s_5' \rightarrow [u: u_5', n: 0] \)
- \( s_4' \rightarrow [u: u_4', n: s_5'] \)
- \( s_3' \rightarrow [u: u_3', n: 0] \)
- \( s \rightarrow [u: u_2', n: s_3'] \)
- \( u_7' \rightarrow [l: 0, r: 0] \)
- \( u_6' \rightarrow [l: 0, r: 0] \)
- \( u_5' \rightarrow [l: 0, r: 0] \)
- \( u_4' \rightarrow [l: 0, r: 0] \)
- \( u_3' \rightarrow [l: u_6', r: u_7'] \)
- \( u \rightarrow [l: u_4', r: u_5'] \)

Separation logic, see paper

Describes heap state and aliasing information

- \( s \rightarrow [u: x', n: y'] \)
- delete t;
- \( \ldots \) do something
- if (u->left!= 0) & & (u->right!=0) then
  - s = PUSH(u->right, s);
  - s = PUSH(u->left, s);
- end if
- delete u;
- end while
Symbolic execution

Real execution (run time)

Symbolic execution (compile time)

Heap layout

<table>
<thead>
<tr>
<th>StackRecord 7</th>
<th>StackRecord 6</th>
<th>StackRecord 5</th>
<th>StackRecord 4</th>
<th>StackRecord 3</th>
<th>StackRecord 2</th>
<th>TreeNode 7</th>
<th>TreeNode 6</th>
<th>TreeNode 5</th>
<th>TreeNode 4</th>
<th>TreeNode 3</th>
<th>TreeNode 2</th>
</tr>
</thead>
</table>

Separation logic, see paper
Describes heap state and aliasing information

\[ s \rightarrow [u: x', n: y'] \]

delete t;
... do something
if (u->left!= 0) && (u->right!=0) then
    s = PUSH(u->right, s);
    s = PUSH(u->left, s);
end if
delete u;
end while

Formal layout
\[
\begin{align*}
    s_7' & \rightarrow [u: u_7', n: 0] * \\
    s_6' & \rightarrow [u: u_6', n: s_7'] * \\
    s_5' & \rightarrow [u: u_5', n: 0] * \\
    s_4' & \rightarrow [u: u_4', n: s_5'] * \\
    s_3' & \rightarrow [u: u_3', n: 0] * \\
    s & \rightarrow [u: u_2', n: s_3'] * \\
    u_7' & \rightarrow [l: 0, r: 0] * \\
    u_6' & \rightarrow [l: 0, r: 0] * \\
    u_5' & \rightarrow [l: 0, r: 0] * \\
    u_4' & \rightarrow [l: 0, r: 0] * \\
    u_3' & \rightarrow [l: u_6', r: u_7'] * \\
    u & \rightarrow [l: u_4', r: u_5']
\end{align*}
\]
Symbolic execution

Real execution (run time)

- StackRecord 7
- StackRecord 6
- StackRecord 5
- **StackRecord 4**
- StackRecord 3
- **StackRecord 2**
- TreeNode 7
- TreeNode 6
- TreeNode 5
- TreeNode 4
- TreeNode 3
- TreeNode 2

Symbolic execution (compile time)

Formal layout

- \( s_7' \rightarrow [u: u_7', n: 0] \)
- \( s_6' \rightarrow [u: u_6', n: s_7'] \)
- \( s_5' \rightarrow [u: u_5', n: 0] \)
- \( s_4' \rightarrow [u: u_4', n: s_5'] \)
- \( s_3' \rightarrow [u: u_3', n: 0] \)
- \( s \rightarrow [u: u_2', n: s_3'] \)

Real execution:

```c
s = t->n;
delete t;
... do something
if (u->left!= 0) && (u->right!=0) then
    s = PUSH(u->right, s);
    s = PUSH(u->left, s);
end if
delete u;
end while
```

Separation logic, see paper

Describes heap state and aliasing information
Real execution (run time)

Heap layout
- stackRecord 7
- stackRecord 6
- stackRecord 5
- stackRecord 3
- stackRecord 2
- treeNode 7
- treeNode 6
- treeNode 5
- treeNode 4
- treeNode 3
- treeNode 2

Symbolic execution (compile time)

Formal layout

$s_7' \rightarrow [u: u_7', n: 0]$
$s_6' \rightarrow [u: u_6', n: s_7']$
$s_5' \rightarrow [u: u_5', n: 0]$
$s_4' \rightarrow [u: u_4', n: s_5']$
$s_3' \rightarrow [u: u_3', n: 0]$
$s \rightarrow [u: u_2', n: s_3']$
$u_7' \rightarrow [l: 0, r: 0]$
$u_6' \rightarrow [l: 0, r: 0]$
$u_5' \rightarrow [l: 0, r: 0]$
$u_4' \rightarrow [l: 0, r: 0]$
$u_3' \rightarrow [l: u_6', r: u_7']$
$u \rightarrow [l: u_4', r: u_5']$

Symbolic execution

$s = \text{new} \ stackRecord$
$s->u = root$
$s->n = 0$

while $s != 0$

do

t = s

u = t->u

s = t->n

delete t
end do

... do something

if $(u->left != 0) \&\& (u->right != 0)$

do

s = PUSH(u->right, s)

s = PUSH(u->left, s)
end if

delete u
end while
Symbolic execution

Real execution (run time)

Heap layout

- stackRecord 7
- stackRecord 6
- stackRecord 3
- treeNode 7
- treeNode 6
- treeNode 5
- treeNode 4
- treeNode 3

Symbolic execution (compile time)

Formal layout

\[
\begin{align*}
    s_7' & \rightarrow [u: u_7', n: 0] \quad * \\
    s_6' & \rightarrow [u: u_6', n: s_7'] \quad * \\
    s_5' & \rightarrow [u: u_5', n: 0] \quad * \\
    s_4' & \rightarrow [u: u_4', n: s_5'] \quad * \\
    s_3' & \rightarrow [u: u_3', n: 0] \quad * \\
    s & \rightarrow [u: u_2', n: s_3'] \quad * \\
    u_7' & \rightarrow [l: 0, r: 0] \quad * \\
    u_6' & \rightarrow [l: 0, r: 0] \quad * \\
    u_5' & \rightarrow [l: 0, r: 0] \quad * \\
    u_4' & \rightarrow [l: 0, r: 0] \quad * \\
    u_3' & \rightarrow [l: u_6', r: u_7'] \quad * \\
    u & \rightarrow [l: u_4', r: u_5']
\end{align*}
\]

\[s = \text{new} \ \text{stackRecord};\]
\[s->u = \text{root};\]
\[s->n = 0;\]
\[\textbf{while} \ s!=0 \ \textbf{do}\]
\[\quad t = s;\]
\[\quad u = t->u;\]
\[\quad s = t->n;\]
\[\quad \text{delete} \ t;\]
\[\quad \ldots \ \text{do} \ \text{something}\]
\[\quad \textbf{if} \ (u->\text{left}!= 0) \ \&\& \ (u->\text{right}!=0) \ \textbf{then}\]
\[\quad \quad s = \text{PUSH}(u->\text{right}, s);\]
\[\quad \quad s = \text{PUSH}(u->\text{left}, s);\]
\[\quad \textbf{end if}\]
\[\quad \text{delete} \ u;\]
\[\textbf{end while}\]
Real execution (run time)

Heap layout

| IT 5, IT 7 | IT 5, IT 6 | IT 2, IT 4 | IT 2, IT 3 | IT 5 | IT 2 | IT 7 | IT 6 | IT 4 | IT 3 | IT 5 | IT 2 |

Symbolic execution (compile time)

Formal layout

\[ s_7' \rightarrow [u: u_7', n: 0] \quad * \]
\[ s_6' \rightarrow [u: u_6', n: s_7'] \quad * \]
\[ s_5' \rightarrow [u: u_5', n: 0] \quad * \]
\[ s_4' \rightarrow [u: u_4', n: s_5'] \quad * \]
\[ s_3' \rightarrow [u: u_3', n: 0] \quad * \]
\[ s \rightarrow [u: u_2', n: s_3'] \quad * \]
\[ u_7' \rightarrow [l: 0, r: 0] \quad * \]
\[ u_6' \rightarrow [l: 0, r: 0] \quad * \]
\[ u_5' \rightarrow [l: 0, r: 0] \quad * \]
\[ u_4' \rightarrow [l: 0, r: 0] \quad * \]
\[ u_3' \rightarrow [l: u_6', r: u_7'] \quad * \]
\[ u \rightarrow [l: u_4', r: u_5'] \]

Real execution (run time)

```
s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
  t = s;
  u = t->u;
  s = t->n;
  delete t;
  ... do something
  if (u->left!= 0) && (u->right!=0) then
    s = PUSH(u->right, s);
    s = PUSH(u->left, s);
  end if
  delete u;
end while
```
Heap footprint analysis

Real execution

Heap layout

- IT 5, IT 7
- IT 5, IT 6
- IT 2, IT 4
- IT 2, IT 3
- IT 5
- IT 2
- IT 7
- IT 6
- IT 4
- IT 3
- IT 5
- IT 2

Dependency between iteration 2 and 4
## Heap footprint analysis

### Real execution

<table>
<thead>
<tr>
<th>Heap layout</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>IT 5, IT 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IT 5, IT 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IT 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IT 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IT 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IT 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IT 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IT 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dependency between iteration 2 and 4

Dependency groups:
- **Group A**: IT 2, 3, 4
Heap footprint analysis

Real execution

Heap layout

Dependency between iteration 2 and 4
Dependency groups:
  - Group A: IT 2, 3, 4
  - Group B: IT 5, 6, 7
Heap footprint analysis

Real execution
Heap layout

Source transformation
• Annotate new/delete commands
Heap footprint analysis

Source transformation
• Annotate new/delete commands
Heap footprint analysis

Source transformation

- Annotate new/delete commands
- Parallelization: Split loop

Real execution

Heap layout

... 
while \( s_B \neq 0 \) do
  ... loop body (access memory partition B)
end while

while \( s_A \neq 0 \) do
  ... loop body (access memory partition A)
end while
Heap footprint analysis

Source transformation
• Annotate new/delete commands
• Parallelization: Split loop

Cache synthesis
• Private cache for each loop kernel

```
... 
while s_B != 0 do
  ... loop body (access memory partition B)
end while
while s_A != 0 do
  ... loop body (access memory partition A)
end while
```
Remainder of this talk

1. Find private heap regions
2. Find shared heap regions
3. Legal parallelization in the presence of shared heap
Detecting shared memory

s->u = root;
s->n = 0;
while s!=0 do
    t = s;
    u = t->u;
    s = t->n;
delete t;
... do something
if (u->left!= 0) && (u->right!=0) then
    s = PUSH(u->right, s);
    s = PUSH(u->left, s);
else
    w_prev = z->w;
    z->w = w_prev + x;
end if
delete u;
end while
Detecting shared memory

Heap layout

s->u = root;
s->n = 0;

while s!=0 do
  t = s;
  u = t->u;
  s = t->n;
  delete t;

  ... do something

  if (u->left!= 0) && (u->right!=0) then
    s = PUSH(u->right, s);
    s = PUSH(u->left, s);
  else
    w_prev = z->w;
    z->w = w_prev + x;
  end if

  delete u;
end while
Detecting shared memory

- Run heap footprint analysis until depth K

```
... do something
if (u->left!= 0) && (u->right!=0) then
    s = PUSH(u->right, s);
    s = PUSH(u->left, s);
else
    w_prev = z->w;
    z->w = w_prev + x;
end if
delete u;
end while
```
Detecting shared memory

- Run heap footprint analysis until depth K
- Mark offending heap portions as shared

```
s->u = root;
s->n = 0;
while s!=0 do
  t = s;
  u = t->u;
  s = t->n;
  delete t;
  ... do something
  if (u->left!= 0) && (u->right!=0) then
    s = PUSH(u->right, s);
    s = PUSH(u->left, s);
  else
    w_prev = z->w;
    z->w = w_prev + x;
  end if
  delete u;
end while
```
Detecting shared memory

Heap layout

- Run heap footprint analysis until depth K
- Mark offending heap portions as shared
- Continue partitioning analysis without them

```c
s->u = root;
s->n = 0;
while s!=0 do
    t = s;
    u = t->u;
    s = t->n;
    delete t;
    … do something
    if (u->left!=0) && (u->right!=0) then
        s = PUSH(u->right, s);
        s = PUSH(u->left, s);
    else
        w_prev = z->w;
        z->w = w_prev + x;
    end if
    delete u;
end while
```
Remainder of this talk

1. Find private heap regions
2. Find shared heap regions
3. Legal parallelization in the presence of shared heap
Accessing shared memory

Assume:
Statement executes in IT 4 and IT 7

- Two cases:
  1. Original program: IT 4 before IT 7
  2. Parallelized program: IT 4 possibly after IT 7

\[ z->w = w_{prev} + x; \]
Accessing shared memory

Assume:
Statement executes in IT 4 and IT 7

• Two cases:
  1. Original program: IT 4 before IT 7
  2. Parallelized program: IT 4 possibly after IT 7
• Does it matter?

z->w = w_prev + x;
Accessing shared memory

Assume:
Statement executes in IT 4 and IT 7

- Two cases:
  1. Original program: IT 4 before IT 7
  2. Parallelized program: IT 4 possibly after IT 7
- Does it matter? **NO!**

1. \[ w_1 = w_{prev} + x^{(IT\ 4)} + y^{(IT\ 7)} \]
2. \[ w_2 = w_{prev} + y^{(IT\ 7)} + x^{(IT\ 4)} \]

\[ w_1 = w_2 \]

\[ z->w \] has the same final value in both cases
Accessing shared memory

Assume:
Statement executes in IT 4 and IT 7

- Two cases:
  1. Original program: IT 4 before IT 7
  2. Parallelized program: IT 4 possibly after IT 7

- Does it matter? **NO!**

1. \[ w_1 = w_{prev} + x^{(IT\ 4)} + y^{(IT\ 7)} \]

2. \[ w_2 = w_{prev} + y^{(IT\ 7)} + x^{(IT\ 4)} \]

- How can a tool decide this?

\[ z\rightarrow w = w_{prev} + x; \]

\[ w_1 = w_2 \]

**z\rightarrow w** has the same final value in both cases
Commutativity analysis

- Idea: Offload verification to SMT solver

\[ w_1 = w_{\text{prev}} + x^{(IT\ 4)} + y^{(IT\ 7)} \]

\[ w_2 = w_{\text{prev}} + y^{(IT\ 7)} + x^{(IT\ 4)} \]

\[ w_1 = w_2 \]

- Not satisfiable: Prove legality of parallelization
Implementation

Parallelized HLS implementation

Loop kernel a
Memory interfaces:

Private SP 0
Private SP 1
Private SP 2

Priv. SP
(owner bit)

Priv. SP
(data)

Coherent SP controller

Coherent SP 0
Coherent SP 1

Platform controller

on-chip

off-chip

LEAP platform

DDR3 DRAM

Host system memory

Coherency network
## Results

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>BRAM</th>
<th>Clock</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Merger</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baseline (no par., no caches)</td>
<td>1</td>
<td>42</td>
<td>10.0 ns</td>
<td>1258 ms</td>
</tr>
<tr>
<td>Parallelization (no caches)</td>
<td>4</td>
<td>62</td>
<td>10.0 ns</td>
<td>539 ms</td>
</tr>
<tr>
<td>Parallelization (with caches)</td>
<td>4</td>
<td>72</td>
<td>10.0 ns</td>
<td>115 ms</td>
</tr>
<tr>
<td><strong>Tree deletion</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baseline (no par., no caches)</td>
<td>1</td>
<td>52</td>
<td>10.0 ns</td>
<td>6575 us</td>
</tr>
<tr>
<td>Parallelization (no caches)</td>
<td>4</td>
<td>91</td>
<td>10.0 ns</td>
<td>2208 us</td>
</tr>
<tr>
<td>Parallelization (with caches)</td>
<td>4</td>
<td>202</td>
<td>10.5 ns</td>
<td>711 us</td>
</tr>
<tr>
<td><strong>K-means clustering</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baseline (no par., no caches)</td>
<td>1</td>
<td>69</td>
<td>10.0 ns</td>
<td>136 ms</td>
</tr>
<tr>
<td>Parallelization (no caches)</td>
<td>4</td>
<td>125</td>
<td>10.0 ns</td>
<td>62 ms</td>
</tr>
<tr>
<td>Parallelization (with caches)</td>
<td>4</td>
<td>272</td>
<td>11.1 ns</td>
<td>42 ms</td>
</tr>
</tbody>
</table>
Tree deletion

Latency [us]

Slices

Hybrid caches (this work)

all-coherent caches (default)

P=1

P=2

P=4
Results

Tree deletion

Latency [us]

Slices

P=1

P=2

P=4

all-coherent caches (default)

Hybrid caches (this work)

Benefit of application specificity: 56% improvement of area-time product
Conclusion

```
stack_record_type *r = new stack_record_type;
r->u = root;
r->d = true;
r->c = centre_list_idx;
r->k = k;
r->next = stackPointer;
stackPointer = r;

while (stackPointer != NULL) {
    // fetch head of stack
    tree_node_type *u;
    centre_set_type *c;
    bool d;
    uint tmp_k;
    stack_record_type *n;
    //stackPointer = pop_node(&u, &d, &c, &tmp_k,
    stackPointer);
    d = stackPointer->d;
    c = stackPointer->c;
    u = stackPointer->u;
    tmp_k = stackPointer->k;
    n = stackPointer->next;
    delete stackPointer;
    stackPointer = n;

    uint c_set[K];
    for (uint i=0; i<tmp_k; i++) {
        uint tmp_idx;
        tmp_idx = c->idx[i];
        c_set[i] = tmp_idx;
    }

    tree_node_type tmp_u;
    delete u;
    data_type_ext comp_point;
    if ((tree_set) ft != NULL) &
```
Conclusion

- Not synthesizable
- Not parallelizable
Conclusion

stack_record_type *r = new stack_record_type;
r->u = root;
r->d = true;
r->c = centre_list_idx;
r->k = k;
r->next = stackPointer;
stackPointer = r;

while (stackPointer != NULL) {
  // fetch head of stack
  tree_node_type *u;
  centre_set_type *c;
  bool d;
  uint tmp_k;
  stack_record_type *n;
  //stackPointer = pop_node(&u, &d, &c, &tmp_k, stackPointer);
  d = stackPointer->d;
  c = stackPointer->c;
  u = stackPointer->u;
  tmp_k = stackPointer->k;
  n = stackPointer->next;
  delete stackPointer;
  stackPointer = n;

  uint c_set[K];
  for (uint i=0; i<tmp_k; i++) {
    uint tmp_idx;
    tmp_idx = c->idx[i];
    c_set[i] = tmp_idx;
  }

  tree_node_type tmp_u;
  delete u;
  data_type_ext comp_point;
  if ((tree_node_type *t = (tree_node_type *) &tmp_u) &&
      (tree_node_type *t = (tree_node_type *) &comp_point)) {
    orig_pointerType_1 r_ptr;
    r_ptr = make_pointer<orig_pointerType_1>(heap_1_0, r);
    r_ptr->stack_record_t::u = root;
    r_ptr = make_pointer<orig_pointerType_1>(heap_1_0, r);
    r_ptr->stack_record_t::d = true;
    r_ptr = make_pointer<orig_pointerType_1>(heap_1_0, r);
    r = r_ptr;
  }
}

new_pointerType_1 r = malloc<new_pointerType_1>({freelist_1_0,&nextFreeLocation_1_0});
orig_pointerType_1 r_ptr;
  r_ptr = make_pointer<orig_pointerType_1>({heap_1_0}, r);
  r_ptr->stack_record_t::u = root;
  r_ptr = make_pointer<orig_pointerType_1>({heap_1_0}, r);
  r_ptr->stack_record_t::d = true;
  r_ptr = make_pointer<orig_pointerType_1>({heap_1_0}, r);
  r = r_ptr;

MATCHUP

• Not synthesizable
• Not parallelizable
• Synthesizable
• Parallelizable
• Tailor made memory hierarchy
• Automated analysis of heap-manipulating programs
  – Partition heap into private and shared regions
  – Preserve semantics with parallel access to shared regions
• Future work
  – Intelligent cache sizing
  – Detecting burst opportunities
Thank you for listening.

f.winterstein12@imperial.ac.uk

http://cas.ee.ic.ac.uk/people/fw1811/