Imperial College London

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.) …

```
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];
     }
    }
  …
```

Background

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

```
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++) {
```

Good HLS results

```
      }
   …
```

Background

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

```
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++) {
```

Good HLS results

```
     }
   …
```

```
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
```

Background

Imperial College London

esa

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

```
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++) {
```

Good HLS results

```
     }
   …
```

```
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
```

Background

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

```
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++) {
```

Good HLS results

```
    }
  …
```

```
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
```

Background

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

```
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++) {
```

Good HLS results

```
   }
  ...
```

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

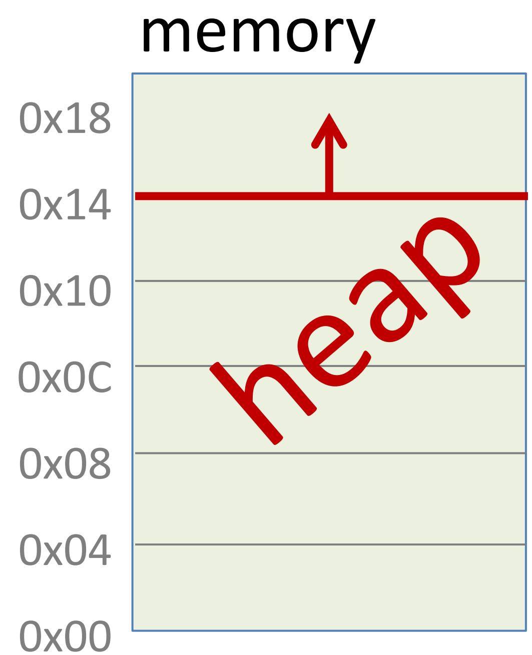
Doesn't synthesize

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

Background

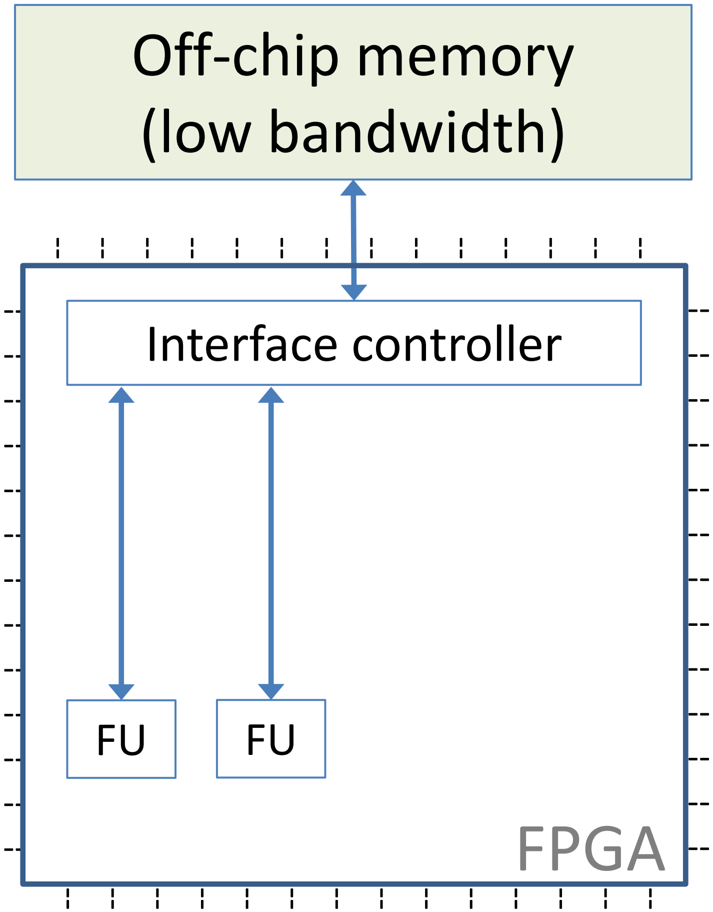
<u>Challenges</u>

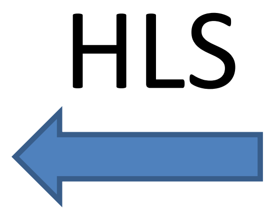
- Memory grows at run-time
- Parallelization: Determine data dependencies (pointer aliasing)

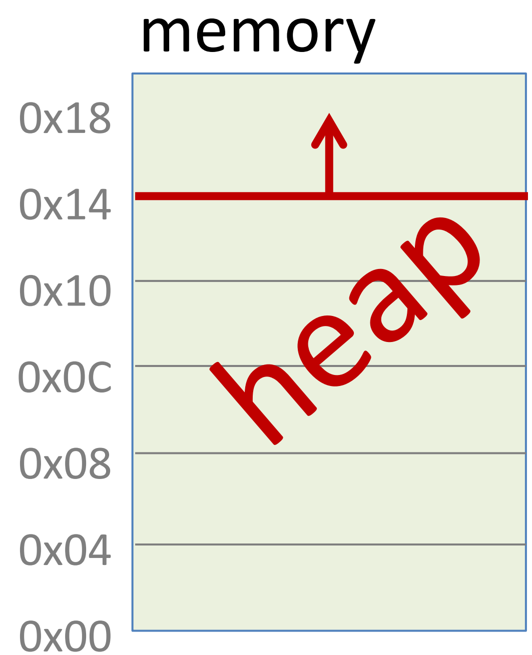



```
s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
   t = s;
   u = t->u;
   s = t->n;
```

Background

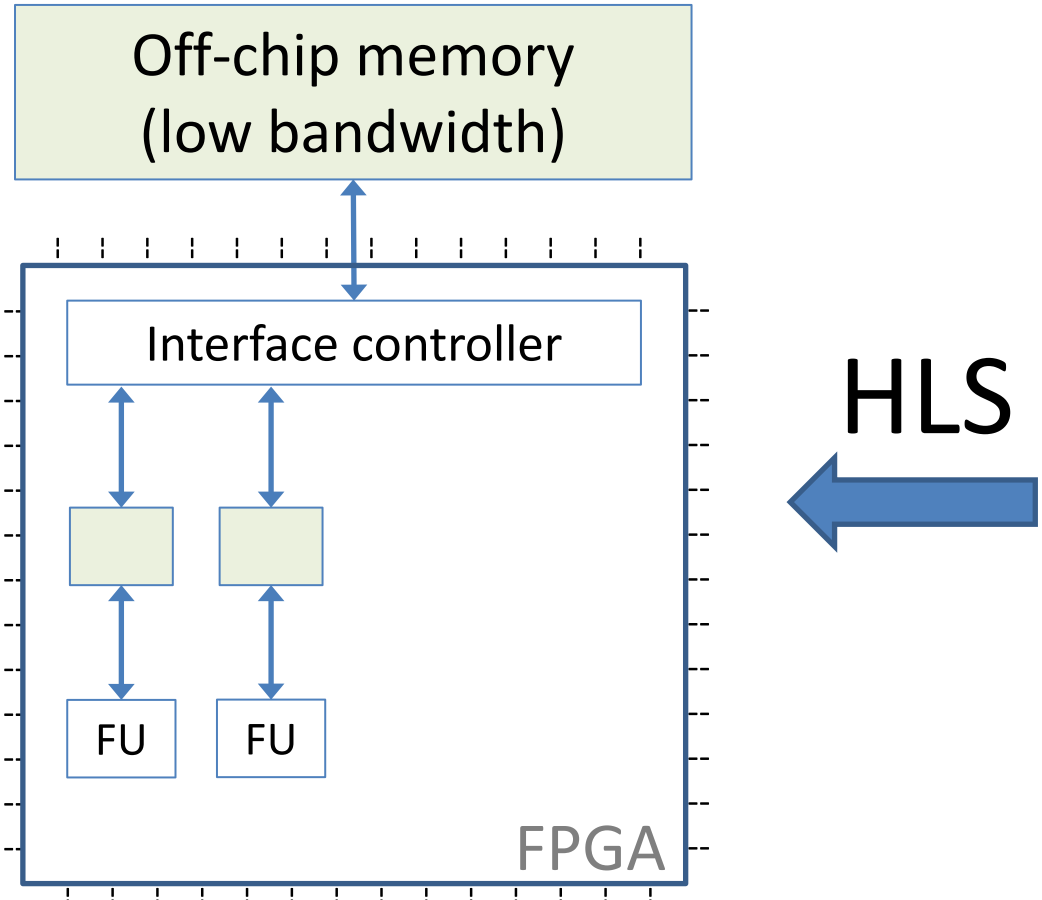


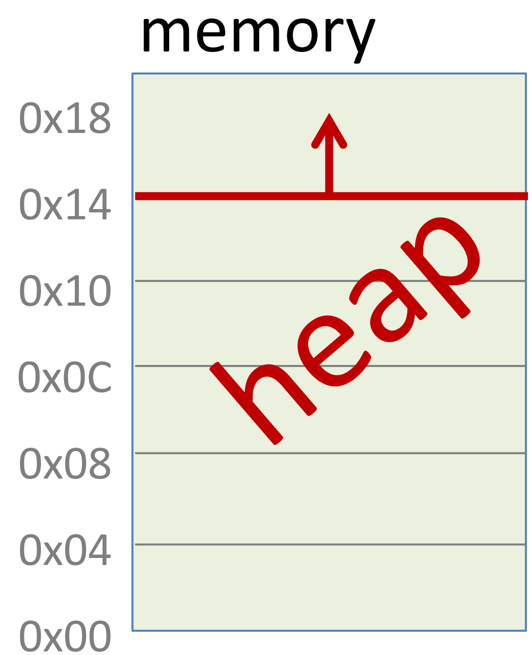





```
s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
    t = s;
    u = t->u;
    s = t->n;
```

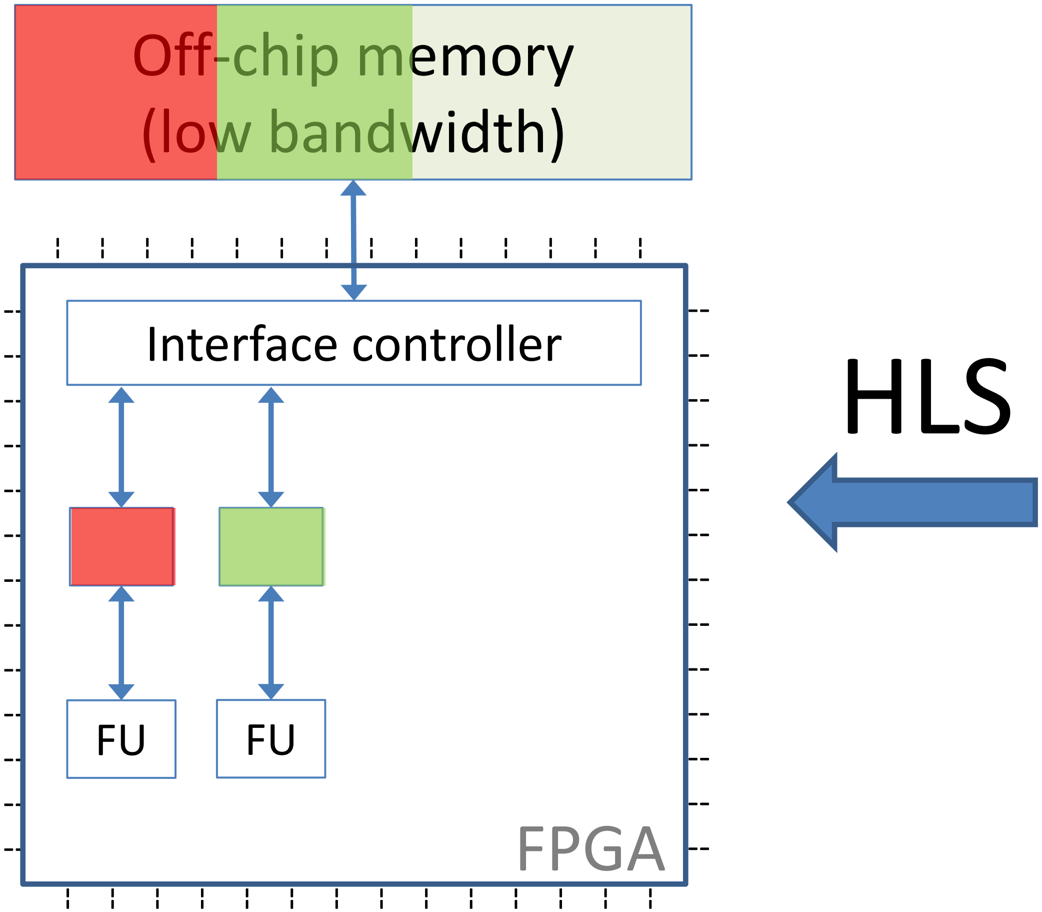
Background

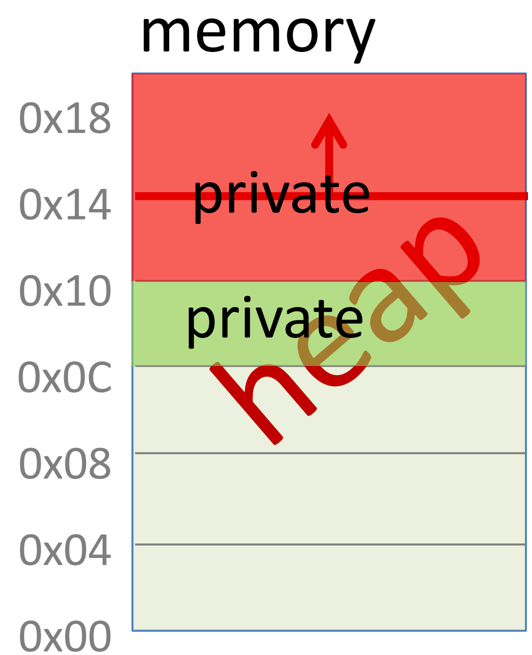





```
s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
   t = s;
   u = t->u;
   s = t->n;
```

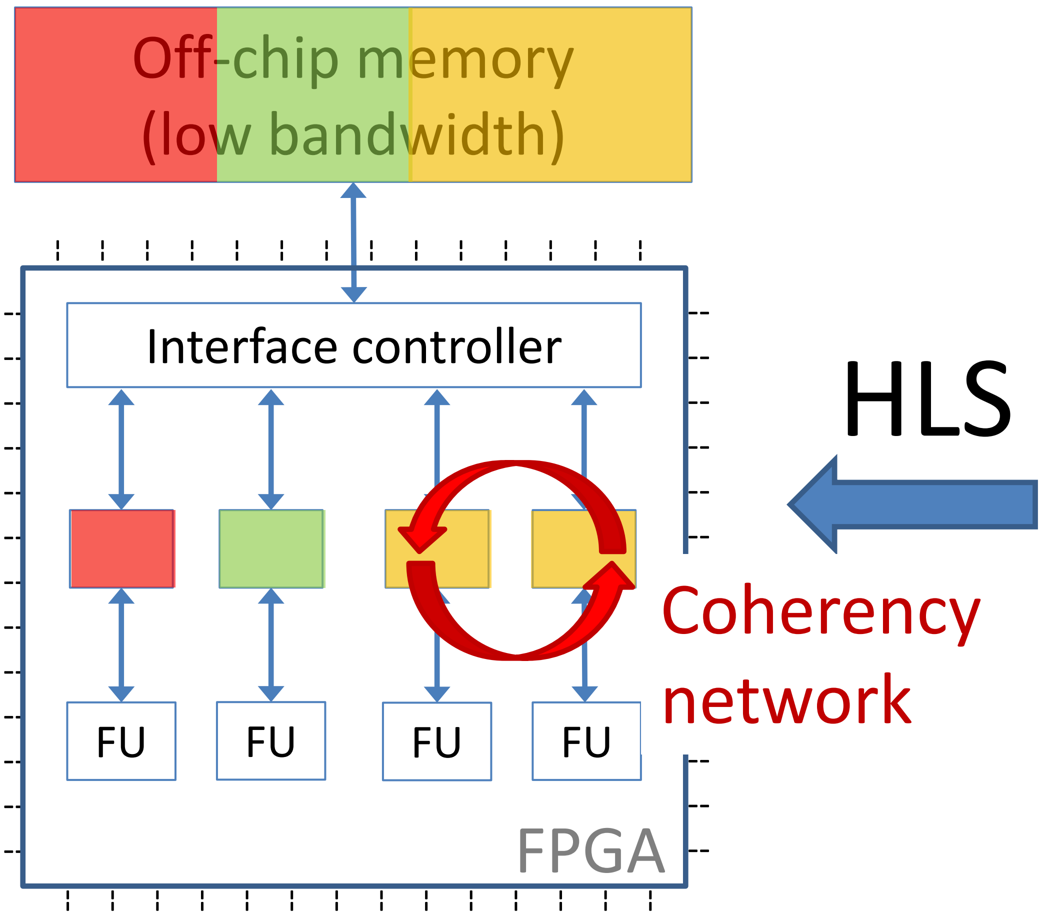
Background



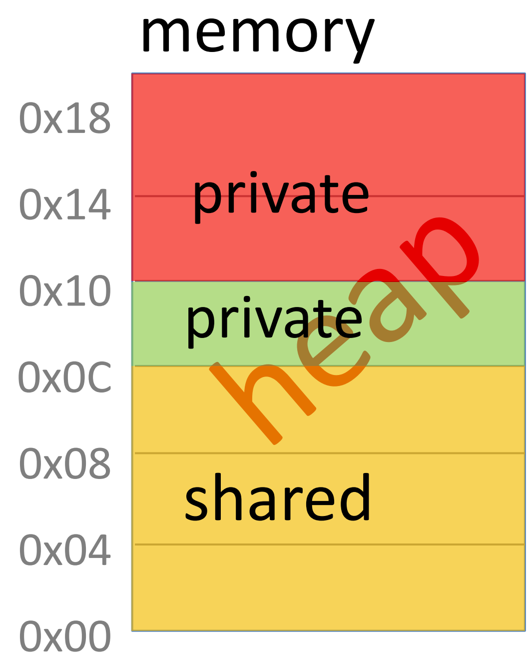



```
s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
    t = s;
    u = t->u;
    s = t->n;
```

Background



Tailor made memory system




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

Imperial College London

Executive summary

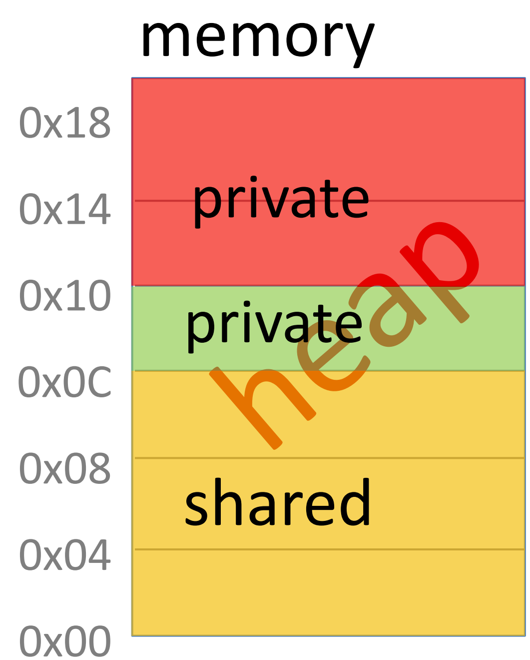
esa

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



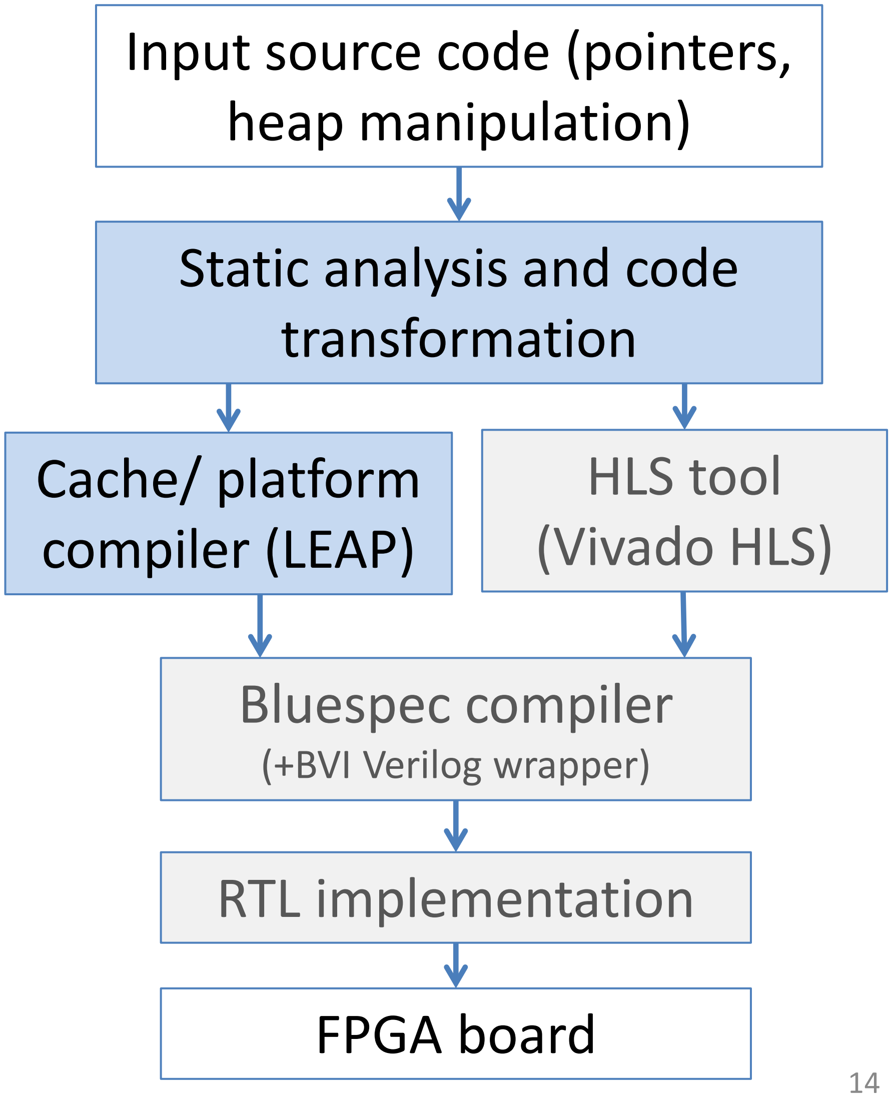

```
s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
   t = s;
   u = t->u;
   s = t->n;
```

Imperial College London

Executive summary

esa

- 8x speed-up from parallel caches (average)
- 49% area-time savings from application specificity (average)



Remainder of this talk

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?

```
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?

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

Independent?

Find private heap regions

- 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) &
      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?

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

Independent?

1st loop iteration

- **NO**

2nd loop iteration

- **YES**

All other iterations

- **YES**

Symbolic execution

Real execution (run time)

Heap layout

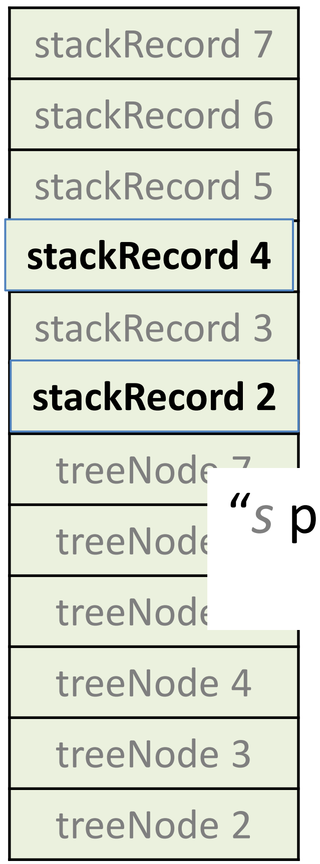
- stackRecord 7
- stackRecord 6
- stackRecord 5
- **stackRecord 4**
- stackRecord 3
- **stackRecord 2**
- treeNode 7
- treeNode 6
- treeNode 5
- treeNode 4
- treeNode 3
- treeNode 2

```
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) th
      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



Formal layout

$$s \rightarrow [u{:}\, u_2{}', n{:}\, s_3{}']$$

"*s* points to a record with fields *u* and *n*"

```
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
          left!= 0) && (u->right!=0) th
          USH(u->right, s);
          USH(u->left, s);
    end if
    delete u;
end while
```

Imperial College London

Symbolic execution

esa

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

$$s \rightarrow [u{:}\, {u_2}', n{:}\, {s_3}']$$

$$u \rightarrow [l{:}\, {u_4}', r{:}\, {u_5}']$$

```
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) th
      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

$s_7' \rightarrow [u: u_7', n: 0]$

$s_6' \rightarrow [u: u_6', n: s_7']$

$s_5' \rightarrow [u: u_5', n: 0]$

$s_4' \rightarrow [u: u_4', n: s_5']$

$s_3' \rightarrow [u: u_3', n: 0]$

$s \rightarrow [u: u_2', n: s_3']$

$u_7' \rightarrow [l: 0, r: 0]$

$u_6' \rightarrow [l: 0, r: 0]$

$u_5' \rightarrow [l: 0, r: 0]$

$u_4' \rightarrow [l: 0, r: 0]$

$u_3' \rightarrow [l: u_6', r: u_7']$

$u \rightarrow [l: u_4', r: u_5']$

```
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) th
      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) th
        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

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

Separation logic, see paper
Describes heap state and aliasing information

```
  s   s → [u: x', n: y']
  delete t;
  ... do something
  if (u->left!= 0) && (u->right!=0) th
     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
  s → [u: x', n: y']
  delete t;
  ... do something
  if (u->left!= 0) && (u->right!=0) th
      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) th
        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

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

Formal layout

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

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

Imperial College London

Symbolic execution

esa

Real execution (run time)

Heap layout

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

Symbolic execution (compile time)

Formal layout

$s_7{}' \to [u{:}\,u_7{}', n{:}\,0]\ *$

$s_6{}' \to [u{:}\,u_6{}', n{:}\,s_7{}']\ *$

$s_5{}' \to [u{:}\,u_5{}', n{:}\,0]\ *$

$s_4{}' \to [u{:}\,u_4{}', n{:}\,s_5{}']\ *$

$s_3{}' \to [u{:}\,u_3{}', n{:}\,0]\ *$

$s \to [u{:}\,u_2{}', n{:}\,s_3{}']\ *$

$u_7{}' \to [l{:}\,0, r{:}\,0]\ *$

$u_6{}' \to [l{:}\,0, r{:}\,0]\ *$

$u_5{}' \to [l{:}\,0, r{:}\,0]\ *$

$u_4{}' \to [l{:}\,0, r{:}\,0]\ *$

$u_3{}' \to [l{:}\,u_6{}', r{:}\,u_7{}']\ *$

$u \to [l{:}\,u_4{}', r{:}\,u_5{}']$

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

Symbolic execution

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

$s_7{}' \to [u{:}\,u_7{}', n{:}\,0]\ *$

$s_6{}' \to [u{:}\,u_6{}', n{:}\,s_7{}']\ *$

$s_5{}' \to [u{:}\,u_5{}', n{:}\,0]\ *$

$s_4{}' \to [u{:}\,u_4{}', n{:}\,s_5{}']\ *$

$s_3{}' \to [u{:}\,u_3{}', n{:}\,0]\ *$

$s \to [u{:}\,u_2{}', n{:}\,s_3{}']\ *$

$u_7{}' \to [l{:}\,0, r{:}\,0]\ *$

$u_6{}' \to [l{:}\,0, r{:}\,0]\ *$

$u_5{}' \to [l{:}\,0, r{:}\,0]\ *$

$u_4{}' \to [l{:}\,0, r{:}\,0]\ *$

$u_3{}' \to [l{:}\,u_6{}', r{:}\,u_7{}']\ *$

$u \to [l{:}\,u_4{}', r{:}\,u_5{}']$

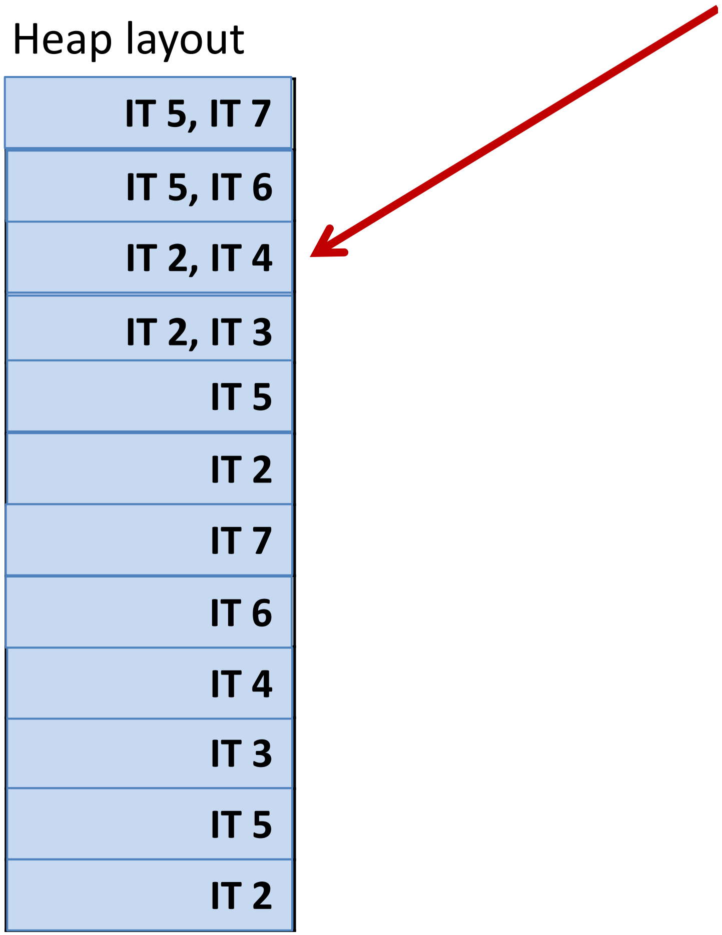
```
s = new stackRecord;
s->u = root;
s->n = 0;
while s!=0 do
   t = s;
   u = t->u;
   s = t->n;
   delete t;
   ... do something
   if (u->left!= 0) && (u->right!=0) th
      s = PUSH(u->right, s);
      s = PUSH(u->left, s);
   end if
   delete u;
end while
```

Heap footprint analysis

Real execution

Heap layout

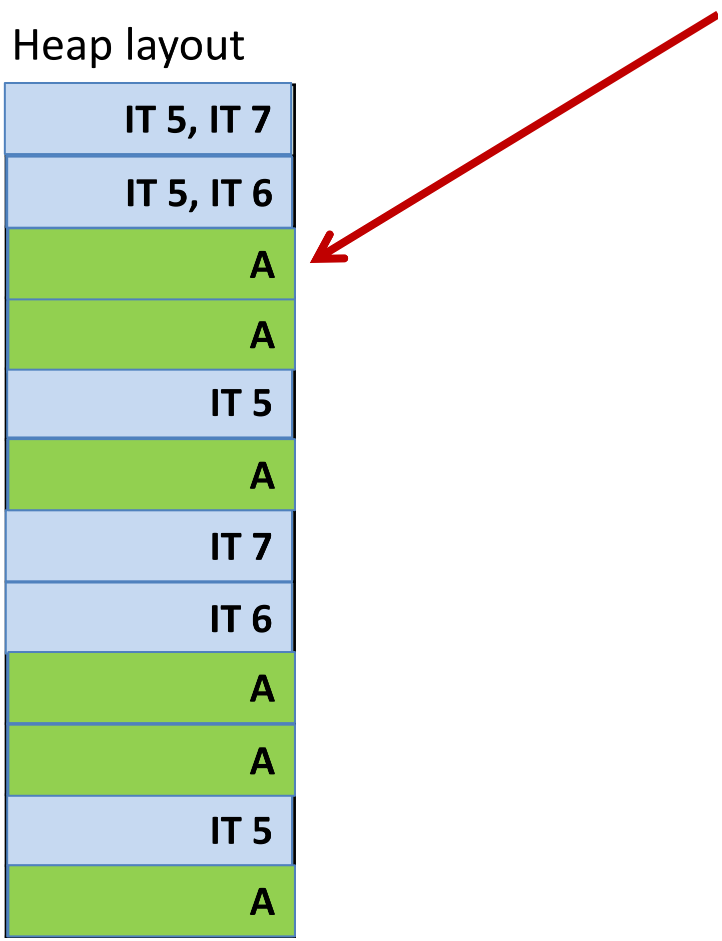
Dependency between iteration 2 and 4



Heap footprint analysis

Real execution

Heap layout



Dependency between iteration 2 and 4

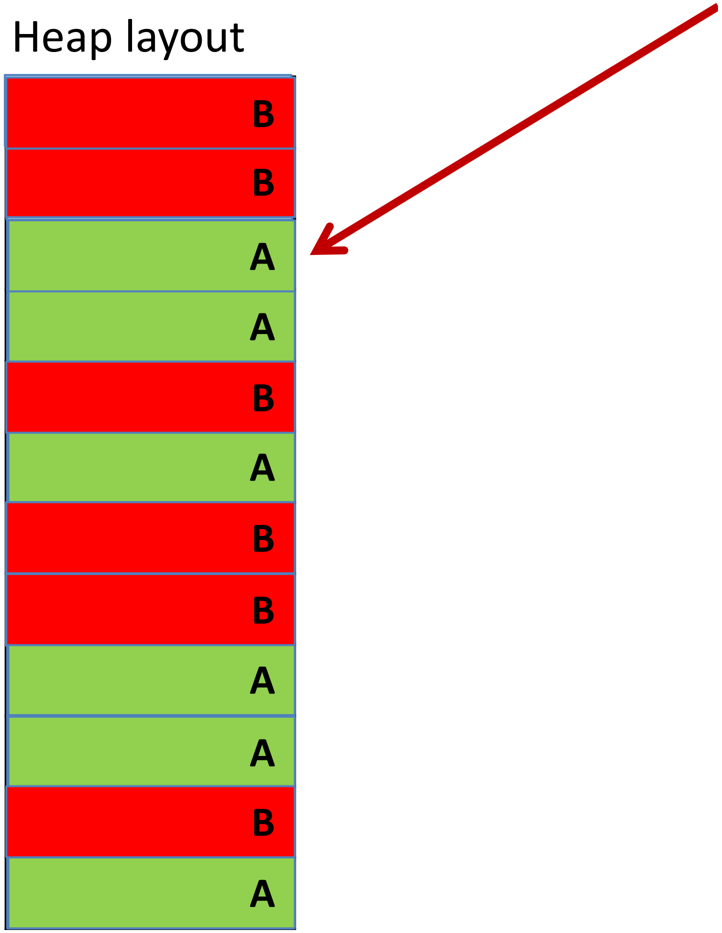
Dependency groups:

- 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

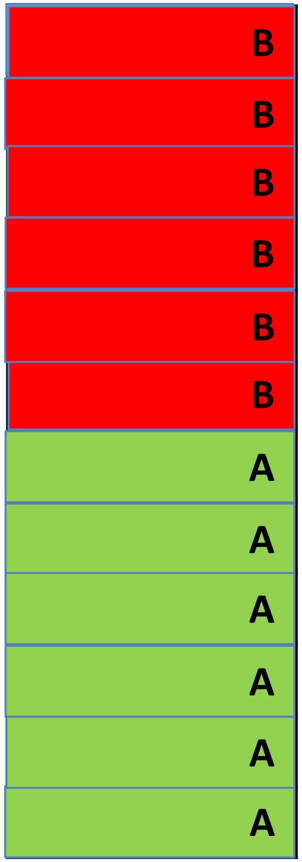
<u>Source transformation</u>

- Annotate new/delete commands

Heap footprint analysis

Real execution

Heap layout



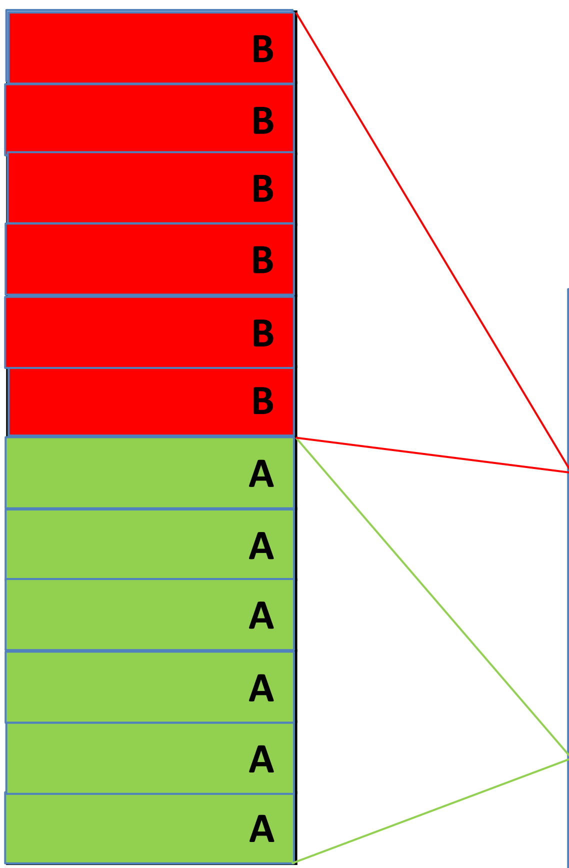
<u>Source transformation</u>

- Annotate new/delete commands

Heap footprint analysis

Real execution

Heap layout



<u>Source transformation</u>

- Annotate new/delete commands
- Parallelization: Split loop

...
while s_B!=0 **do**
 ... loop body (access memory partition B)
end while

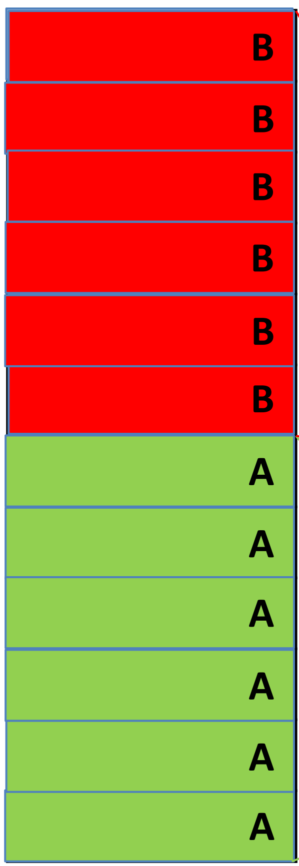
while s_A!=0 **do**
 ... loop body (access memory partition A)
end while

Imperial College London

Heap footprint analysis

Real execution

Heap layout



<u>Source transformation</u>

- Annotate new/delete commands
- Parallelization: Split loop

<u>Cache synthesis</u>

- Private cache for each loop kernel

...
while s_B!=0 **do**
... loop body (access memory partition B)
end while

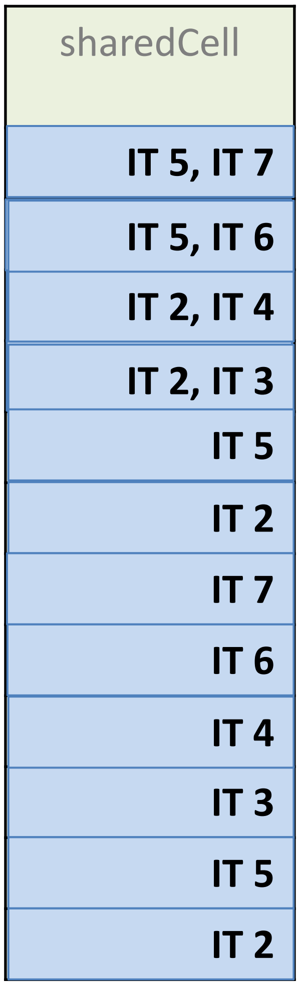
while s_A!=0 **do**
... loop body (access memory partition A)
end while

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

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

Heap layout

IT 1, 2, 3, 4, 5, 6, 7
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

```
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

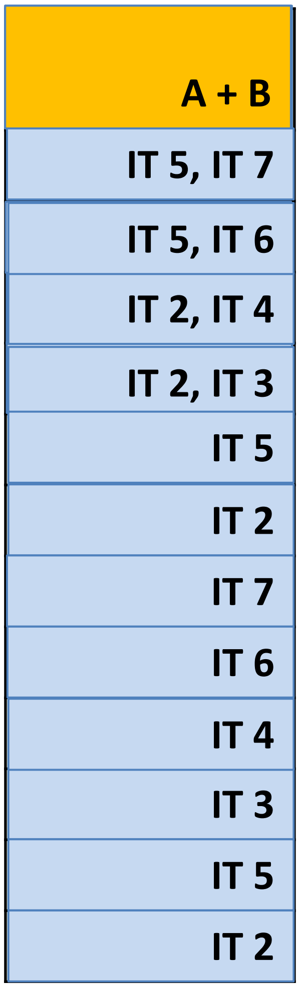
IT 1, 2, 3, 4, 5, 6, 7
IT 5, IT 7
IT 5, IT 6
IT 2, IT 4
IT 2, IT 3
IT 5
IT 2
IT 7
IT 6
IT 4
IT 3
IT 5
IT 2

- Run heap footprint analysis until depth K

```
s->u = root;
...
    ... 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

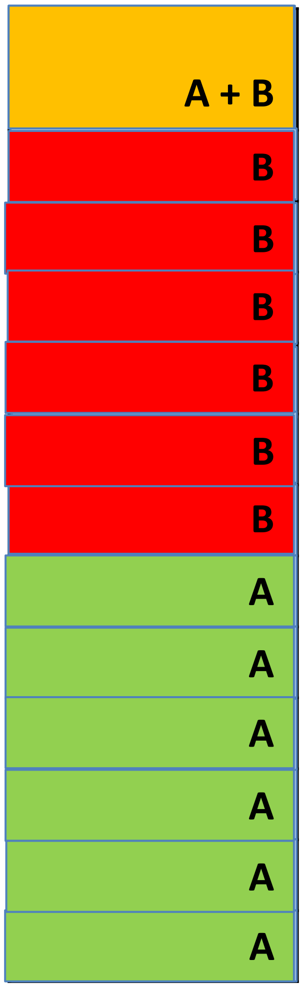


- Run heap footprint analysis until depth K
- Mark offending heap portions as shared

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

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

Imperial College London

Remainder of this talk

esa

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

```
…
z->w = w_prev + x;
```

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

Accessing shared memory

Assume:

Statement executes in IT 4 and IT 7

```
…
z->w = w_prev + x;
```

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

Accessing shared memory

Assume:
Statement executes in IT 4 and IT 7

```
…
z->w = w_prev + x;
```

- 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

Imperial College London

esa

Assume:

Statement executes in IT 4 and IT 7

```
…
z->w = w_prev + x;
```

- 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

- How can a tool decide this?

Commutativity analysis

- Idea: Offload verification to SMT solver

$$\exists\ x^{(IT\ 4)},\ y^{(IT\ 7)}$$

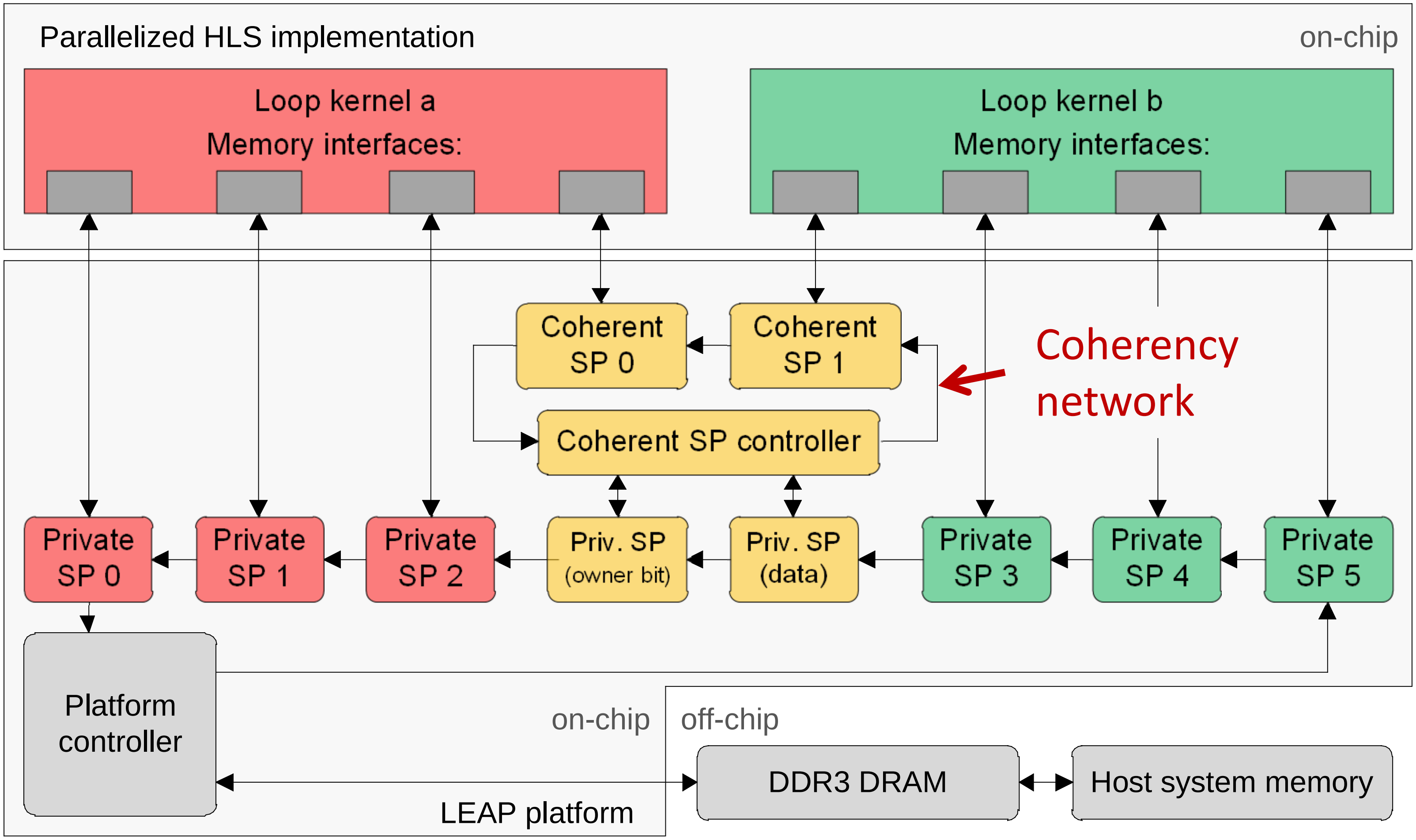
$$w_1 = w_{prev} + x^{(IT\ 4)} + y^{(IT\ 7)}$$

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

$$\wedge \quad w_1 \neq w_2$$

- Not satisfiable: Prove legality of parallelization

Implementation



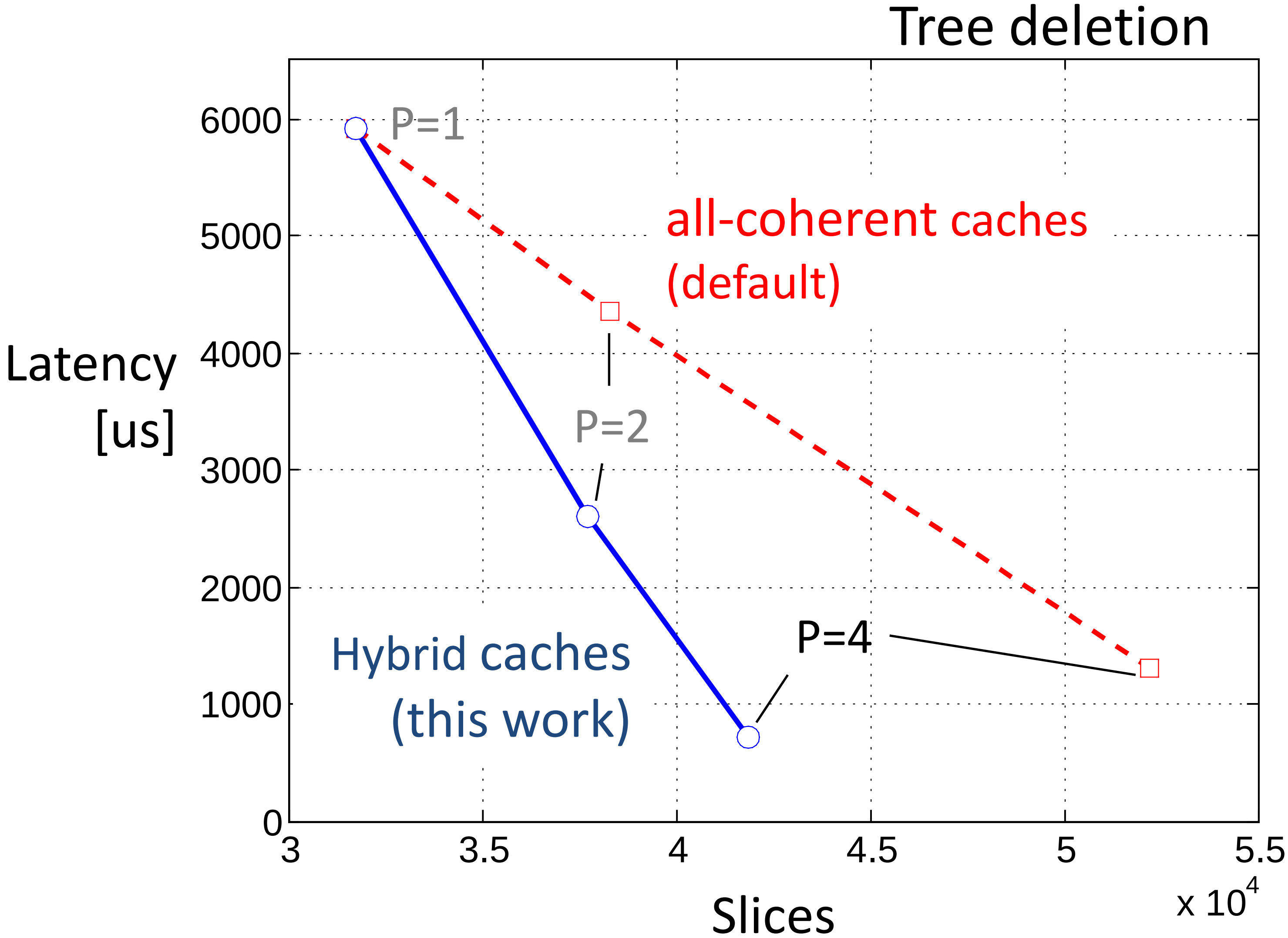
Imperial College London

Results

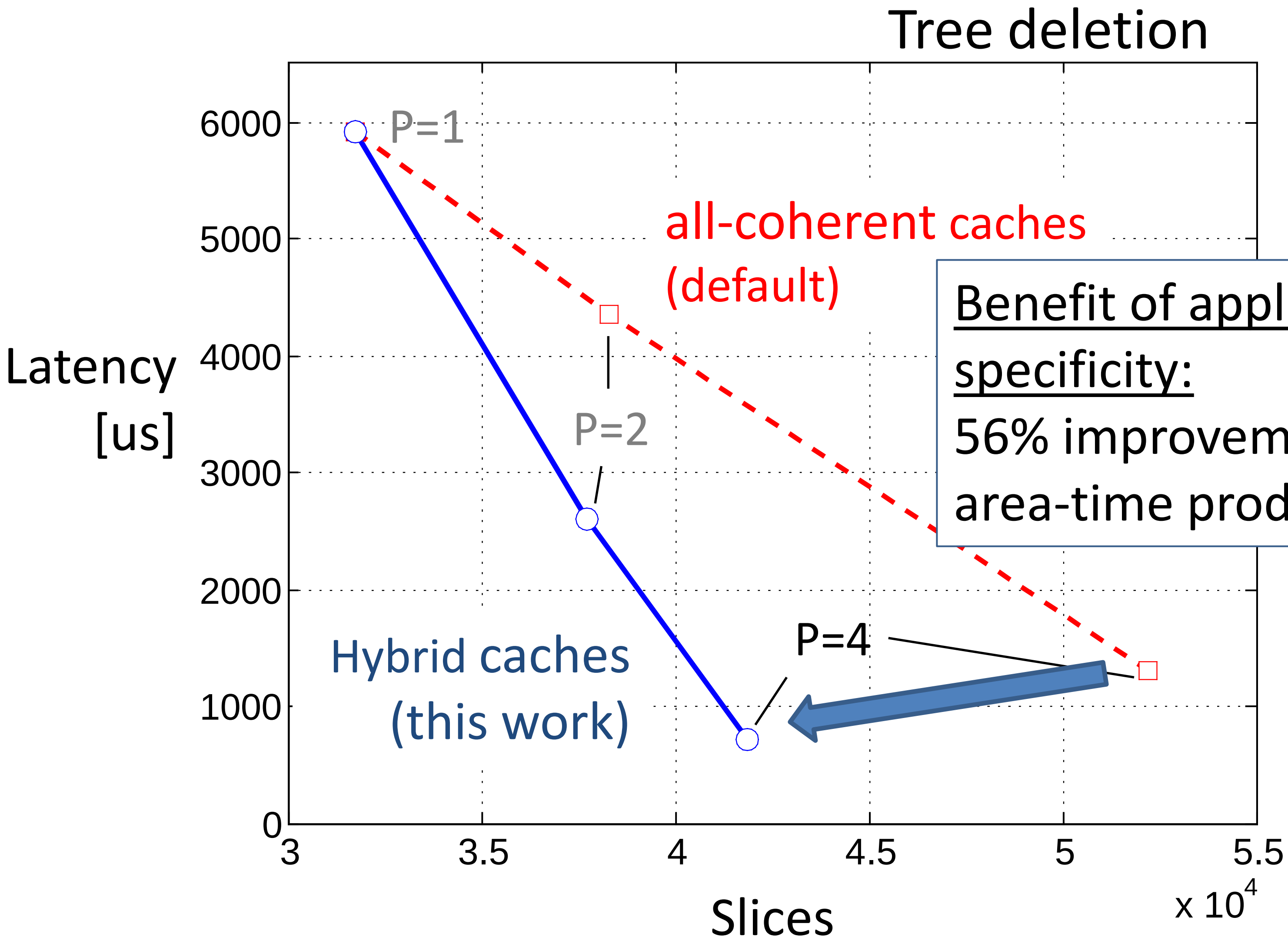
esa

		P	BRAM	Clock	Latency	
1	**Merger**					
	Baseline (no par., no caches)	1	42	10.0 ns	1258 ms	x10
	Parallelization (no caches)	4	62	10.0 ns	539 ms	
	Parallelization (with caches)	4	72	10.0 ns	115 ms	
2	**Tree deletion**					
	Baseline (no par., no caches)	1	52	10.0 ns	6575 us	x9
	Parallelization (no caches)	4	91	10.0 ns	2208 us	
	Parallelization (with caches)	4	202	10.5 ns	711 us	
3	***K*-means clustering**					
	Baseline (no par., no caches)	1	69	10.0 ns	136 ms	x3
	Parallelization (no caches)	4	125	10.0 ns	62 ms	
	Parallelization (with caches)	4	272	11.1 ns	42 ms	

Results


Tree deletion
Latency [us]
6000
5000
4000
3000
2000
1000
0
3
3.5
4
4.5
5
5.5
x 10^4
Slices
P=1
P=2
P=4
all-coherent caches (default)
Hybrid caches (this work)

Results



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;
```

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) {
```

- Not synthesizable
- Not parallelizable

```
    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;
```

Imperial College London

Conclusion

esa

```
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) {
```

- Not synthesizable
- Not parallelizable

```
    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;
```

MATCHUP

```
 new_pointerType_1 r =
malloc<new_pointerType_1>(freelist_1_0,&nextFreeLocatio
n_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_
```

- Synthesizable
- Parallelizable
- Tailor made memory hierarchy

```
 if
ma
);
ma
);
sta
//
ma
);
sta
   orig_pointerType_2 u__0_ptr;
   stackPointer_ptr =
make_pointer<orig_pointerType_1>(heap_1_0,stackPointer
);
```

Conclusion

- 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

Imperial College London

esa

Thank you for listening.

f.winterstein12@imperial.ac.uk

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