STeffiHLS: Separation Logic-Assisted Code Transformations for Efficient High-Level Synthesis

Felix Winterstein
14 May 2014
Contact: f.winterstein12@imperial.ac.uk
Executive summary

• HLS tools require manual source code refactoring...
  – ... to map pointer-manipulating programs efficiently into HW

• Static program analysis
  – Analyse pointer-based memory accesses and heap layout
  – Identify disjoint, independent regions in heap memory

• Source-to-source transformations
  – Partition heap across on-chip memory banks
  – Automatic loop parallelization
Executive summary

- HLS tools require manual source code refactoring...
  - ... to map pointer-manipulating programs efficiently into HW
- Static program analysis
  - Analyse pointer-based memory accesses and heap layout
  - Identify disjoint, independent regions in heap memory
- Source-to-source transformations
  - Partition heap across on-chip memory banks
  - Automatic loop parallelization

Original source code (heap-directed pointers, dynamic memory allocation)

STeffiHLS

Modified source code

Standard HLS tool (e.g. Vivado HLS)
FPGAs everywhere

Some examples ...

... of COTS FPGAs in ESA’s Ground Station Division
FPGAs everywhere

Some examples...

... of COTS FPGAs in ESA’s Ground Station Division

<table>
<thead>
<tr>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mono-static space surveillance radar</td>
</tr>
<tr>
<td>Bi-static space surveillance radar</td>
</tr>
<tr>
<td>Phased array radar demonstrator (in-house cross verification)</td>
</tr>
<tr>
<td>Ground station modem system</td>
</tr>
<tr>
<td>NextGen tracking, telemetry and command processor</td>
</tr>
<tr>
<td>Combined high-rate telemetry and ranging transceiver</td>
</tr>
<tr>
<td>OPS-SAT: experimental satellite in low Earth orbit</td>
</tr>
<tr>
<td>SARAS: direction finding for rapid signal acquisition</td>
</tr>
</tbody>
</table>
FPGAs everywhere

Some examples ...

... of COTS FPGAs in ESA’s Ground Station Division

<table>
<thead>
<tr>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mono-static space surveillance radar</td>
</tr>
<tr>
<td>Bi-static space surveillance radar</td>
</tr>
</tbody>
</table>

Modern FPGAs allow us to map increasingly complex applications to reconfigurable logic

If they weren’t so hard to program...

| SARAS: direction finding for rapid signal acquisition |
## HLS tools (examples)

<table>
<thead>
<tr>
<th>Tool</th>
<th>Input language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cadence C-to-Silicon</td>
<td>C</td>
</tr>
<tr>
<td>Synopsys Synphony C Compiler</td>
<td>C</td>
</tr>
<tr>
<td>Mentor Graphics Catapult C</td>
<td>C</td>
</tr>
<tr>
<td>Impulse CoDeveloper</td>
<td>C</td>
</tr>
<tr>
<td>Xilinx Vivado HLS</td>
<td>C</td>
</tr>
<tr>
<td>Bluespec</td>
<td>BSV</td>
</tr>
<tr>
<td>National Instruments LabVIEW FPGA</td>
<td>LabVIEW schematic</td>
</tr>
<tr>
<td>Xilinx System Generator for DSP</td>
<td>Matlab/Simulink</td>
</tr>
<tr>
<td>DEFACTO</td>
<td>C</td>
</tr>
<tr>
<td>ROCCC</td>
<td>C</td>
</tr>
<tr>
<td>LegUP</td>
<td>C</td>
</tr>
<tr>
<td>Chisel</td>
<td>Scala</td>
</tr>
</tbody>
</table>
HLS – Applications

Examples of HLS applications

<table>
<thead>
<tr>
<th>Author</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meeus et al. (2012)</td>
<td>Image processing</td>
</tr>
<tr>
<td>Sarkar et al. (2009)</td>
<td>Stream-based video processing</td>
</tr>
<tr>
<td>BDTI (2010)</td>
<td>- Stream-based video processing</td>
</tr>
<tr>
<td></td>
<td>- Stream-based signal processing</td>
</tr>
<tr>
<td>Hammani et al. (2008)</td>
<td>- Image and signal processing</td>
</tr>
<tr>
<td></td>
<td>- Ray casting</td>
</tr>
<tr>
<td>Cong et al. (2013)</td>
<td>Image processing</td>
</tr>
<tr>
<td>Cong et al. (2011)</td>
<td>Image processing</td>
</tr>
</tbody>
</table>

- Mainly regular control flow/ memory access
- Pointers, dynamic memory allocation, linked data structures?  Worth considering at all?
Outline

• **Case study: High-level synthesis of dynamic data structures**
  • Challenge
  • Motivating example
  • Leveraging recent advances in software verification
  • Implementation and results
  • Outlook
Case study: $K$-means clustering

- Compare computational properties of two algorithms for $K$-means clustering
- SW/ RTL/ HLS implementations
- Code available on GitHub (Vivado-KMeans)
$K$-means clustering

N data points
K-means clustering

- Automatic partitioning

N data points
**K-means clustering**

- Automatic partitioning
- \( K \) is a known parameter (e.g. \( K=4 \))
- Clusters represented by their centres
- Centre position determines cluster assignment
- Find optimal positioning

\[ \| x_i - z_1 \|^2 \]
**K-means clustering**

- Automatic partitioning
- $K$ is a known parameter (e.g. $K=4$)
- Clusters represented by their centres
- Centre position determines cluster assignment
- Find optimal positioning

![Diagram of N data points with clusters labeled 1, 2, 3, 4]
**K-means clustering**

- Automatic partitioning
- \( K \) is a known parameter (e.g. \( K=4 \))
- Clusters represented by their centres
- Centre position determines cluster assignment
- Find optimal positioning

\( \text{N data points} \)
**K-means clustering**

- Automatic partitioning
- *K* is a known parameter (e.g. *K*=4)
- Clusters represented by their centres
- Centre position determines cluster assignment
- Find optimal positioning

![Graph showing K-means clustering with 4 clusters and N data points]
K-means clustering

- Automatic partitioning
- \( K \) is a known parameter (e.g. \( K=4 \))
- Clusters represented by their centres
- Centre position determines cluster assignment
- Find optimal positioning

N data points
Brute-force algorithm

**Algorithm:**

\[
\text{for all } x_i \in \text{data points do } \\
\quad \text{for all } z_j \in \text{centers do } \\
\quad\quad \text{compute distance } \|x_i - z_j\|^2 \\
\quad\quad \text{pick & update closest center} \\
\quad \text{end for} \\
\text{end for}
\]

- For each data point ...
- search among $K$ candidates for the closest center
- Popular for HW implementation

Tree-based algorithm*

- Recursively split data set
- Build a pointer-linked tree data structure

* "The Filtering Algorithm", Kanungo et al., 2002
Tree-based algorithm*

- Recursively split data set
- Build a pointer-linked tree data structure
- Clustering: Recursive tree traversal
- Acceleration through
  - Considering ONLY PROMISING candidates in the search for the closest center
  - Search space pruning (sub-trees)

---

*“The Filtering Algorithm”, Kanungo et al., 2002
Tree-based algorithm*

- Recursively split data set
- Build a pointer-linked tree data structure
- Clustering: Recursive tree traversal
- Acceleration through
  - Considering ONLY PROMISING candidates in the search for the closest center
  - Search space pruning (sub-trees)
- Dynamically (de-)allocate memory to store intermediate results

* “The Filtering Algorithm”, Kanungo et al., 2002
Same result – two algorithms

Brute-force algorithm
• Computationally expensive
• Simple control flow
• Embarrassingly parallel

Tree-based algorithm
• Data-dependent control flow
• Pointer-based tree traversal
• Dynamic memory allocation
Results

The battlefield

Implementation

Time per clustering iteration [ms]

Identical area constraint for FPGA implementations: 6500 slices
Results

The battlefield

Implementation

Software
C++, GCC -O3
Intel i7, 3.4GHz

Time per clustering iteration [ms]

1/6.3

Identical area constraint for FPGA implementations: 6500 slices
Software
C++, GCC -O3
Intel i7, 3.4GHz

Hand-crafted RTL
Virtex7 FPGA

The battlefield

Time per clustering iteration [ms]

Identical area constraint for FPGA implementations: 6500 slices
Results

The battlefield

Implementation

Software
C++, GCC -O3
Intel i7, 3.4GHz

Hand-crafted RTL
Virtex7 FPGA

C++-based HLS
Virtex7 FPGA
(Vivado HLS)

Identical area constraint for FPGA implementations: 6500 slices

Time per clustering iteration [ms]
Results

The battlefield

Implementation

Software
C++, GCC -O3
Intel i7, 3.4GHz

Hand-crafted RTL
Virtex7 FPGA

C++-based HLS
Virtex7 FPGA
(Vivado HLS)

Time per clustering iteration [ms]

1/7.2
1/6.3
1/4.3

Identical area constraint for FPGA implementations: 6500 slices
Results

The battlefield

Implementation

- Software
  - C++, GCC -O3
  - Intel i7, 3.4GHz
- Hand-crafted RTL
  - Virtex7 FPGA
- C++-based HLS
  - Virtex7 FPGA
    (Vivado HLS)

Time per clustering iteration [ms]

- Tree-based
  - x 4.3
  - x 4.4
  - x 6.3
  - 7.2x improvement after extensive source code refactoring

Identical area constraint for FPGA implementations: 6500 slices
Brute-force algorithm
- Computationally expensive
- Simple control flow
- Embarrassingly parallel
- **Seamless C-to-FPGA implementation**

Tree-based algorithm
- Data-dependent control flow
- Pointer-based tree traversal
- Dynamic memory allocation
- **C-to-FPGA requires substantial code modifications**
Code transformations

Tree-based algorithm:

- Memory partitioning
- Parallelization
- Custom implementation of dynamic memory allocation
- Loop flattening
- Loop distribution
- Custom bit widths
- ...

Code transformations

Automate

• Memory partitioning
• Parallelization

• Custom implementation of dynamic memory allocation
• Loop flattening
• Loop distribution
• Custom bit widths
• ...

Outline

• Case study: High-level synthesis of dynamic data structures

• **Challenge**

• Motivating example

• Leveraging recent advances in software verification

• Implementation and results

• Outlook
Works well for ‘regular’ memory accesses (polyhedral model, ex. Cong, Pouchet..)
Lack of automated optimizations ...

- ... for programs using pointers
- ... because pointers are difficult to analyze
- ... and memory is allocated, disposed, and reused at run-time
- Yet widely used in SW

```c
int main() {
    x = A[i];
    p = new int;
    *p = 3;
    ...
}
```
Lack of automated optimizations ... 
- ... for programs using pointers
- ... because pointers are difficult to analyze
- ... and memory is allocated, disposed, and reused at run-time
- Yet widely used in SW

STeffiHLS takes a step towards closing this gap

```c
int main() {
    x = A[i];
    p = new int;
    *p = 3;
    ...
}
```
**Challenge**

**Our goal**
- Partition heap-allocated data structures (‘heaplets’)
- Synthesize a parallel implementation

\[
\text{heap}[N] \rightarrow \text{heap}_a[N/2] \quad \text{heap}_b[N/2]
\]

**SW memory model**

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x18</td>
<td>high address</td>
</tr>
<tr>
<td>0x14</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td></td>
</tr>
<tr>
<td>0x0C</td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td></td>
</tr>
<tr>
<td>0x04</td>
<td></td>
</tr>
<tr>
<td>0x00</td>
<td>low address</td>
</tr>
</tbody>
</table>
Our goal

• Partition heap-allocated data structures ('heaplets')

• Synthesize a parallel implementation

• Ensure that heap partitions are ‘private’
• Case study: High-level synthesis of dynamic data structures
• Challenge
• **Motivating example**
• Leveraging recent advances in software verification
• Implementation and results
• Outlook
Can we parallelize this loop?

```
s = PUSH(root, s);
while s!=0 do
  s = POP(&u, s);
  ... 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
```
Motivating example

\[
\begin{align*}
\text{s} &= \text{PUSH}(\text{root, s}); \\
\text{while } \text{s} \neq 0 \text{ do} \\
& \quad \text{s} = \text{POP}(&u, s); \\
& \quad \ldots \text{ do something} \\
& \quad \text{if } (u->\text{left} \neq 0) \& \&(u->\text{right} \neq 0) \text{ then} \\
& \quad & \quad \text{s} = \text{PUSH}(u->\text{right, s}); \\
& \quad & \quad \text{s} = \text{PUSH}(u->\text{left, s}); \\
& \quad & \quad \text{end if} \\
& \quad \text{delete } u; \\
& \text{end while}
\end{align*}
\]
Motivating example

```
s = PUSH(root, s);
while s!=0 do
  s = POP(&u, s);
  ... 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
```
Motivating example

```
s = PUSH(root, s);
while s!=0 do
    s = POP(&u, s);
    ... 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
```
Motivating example

```
s = PUSH(root, s);
while s!=0 do
s = POP(&u, s);
... 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
```
Motivating example

```
s = PUSH(root, s);
while s!=0 do
    s = POP(&u, s);
    \[\ldots\text{do something}\]
    if (u->left!= 0) && (u->right!=0) then
        s = PUSH(u->right, s);
        s = PUSH(u->left, s);
    end if
    delete u;
end while
```
Motivating example

```
s = PUSH(root, s);
while s!=0 do
    s = POP(&u, s);
    ... 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
```
Motivating example

\[ s = \text{PUSH}(\text{root}, s); \]

\textbf{while} \( s \neq 0 \) \textbf{do}

\[ s = \text{POP}(&u, s); \]

... do something

\textbf{if} (u->left\neq 0) \&\& (u->right\neq 0) \textbf{then}

\[ s = \text{PUSH}(u->right, s); \]
\[ s = \text{PUSH}(u->left, s); \]

\textbf{end if}

\textbf{delete} u;

\textbf{end while}
s = PUSH(root, s);
while s!=0 do
  s = POP(&u, s);
  ... 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
Motivating example

```
s = PUSH(root, s);
while s!=0 do
    s = POP(&u, s);
    ... 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
```
Motivating example

```
s = PUSH(root, s);
while s!=0 do
    s = POP(&u, s);
    ... 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
```
Motivating example

After two iterations...

```
s = PUSH(root, s);
while s!=0 do
    s = POP(&u, s);
    ... 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
```
Motivating example

- Partition linked list and tree

```c
s = PUSH(root, s);
while s!=0 do
    s = POP(&u, s);
    ... 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
```
Motivating example

- Partition linked list and tree

... preamble (accessing root node)

while $s_a != 0$ do
    ... loop body (access left sub-tree)
end while

while $s_b != 0$ do
    ... loop body (access right sub-tree)
end while
Motivating example

• Partition linked list and tree
• Will the red loop ever access data in the green partition?

... preamble (accessing root node)

while \( s_a \neq 0 \) do
    ... loop body (access left sub-tree)
end while

while \( s_b \neq 0 \) do
    ... loop body (access right sub-tree)
end while
Motivating example

- Partition linked list and tree
- Will the red loop ever access data in the green partition? No!

... preamble (accessing root node)

while $s_a 
eq 0$ do
  ... loop body (access left sub-tree)
end while

while $s_b 
eq 0$ do
  ... loop body (access right sub-tree)
end while
Motivating example

- Partition linked list and tree
- Will the **red loop** ever access data in the **green partition**? No!
- Parallelization is legal (does not violate data dependencies)

```
while sa!=0 do
    ... loop body (access left sub-tree)
end while

while sb!=0 do
    ... loop body (access right sub-tree)
end while
```

... preamble (accessing root node)
Motivating example

- Partition linked list and tree
- Will the red loop ever access data in the green partition? No!
- Parallelization is legal (does not violate data dependencies)
- Why is it hard for a tool to figure this out?

... preamble (accessing root node)

while $s_a != 0$ do
  ... loop body (access left sub-tree)
end while

while $s_b != 0$ do
  ... loop body (access right sub-tree)
end while
The problem

\[ n \quad u \]
\[ n \quad u \]
\[ n \quad u \]

s

\[ \cdots \quad \cdots \]

\[ \cdots \quad \cdots \quad \cdots \quad \cdots \]
Heap accessed in the next iteration
The problem

Heap accessed in the next iteration

Heap accessed in some iteration in the future
The problem

- Do these iterations access the same memory cell?
• Do these iterations access the same memory cell? 
  heap[s]
• Do these iterations access the same memory cell?

heap[heap[s].u]
• Do these iterations access the same memory cell?

heap[heap[s].u]  heap[s]
The problem

- Do these iterations access the same memory cell?
  
  ```latex
  \text{heap[heap[s].u]} \quad \text{heap[heap[s].n]}
  ```

```
The problem

- Do these iterations access the same memory cell?

heap[heap[s].u]  heap[heap[heap[heap[s].n].n]]
The problem

Do these iterations access the same memory cell?

heap[heap[s].u]  heap[heap[heap[heap[s].n].n].u]

...
The problem

- Do these iterations access the same memory cell?

\[ \text{heap[heap[s].u]} = \text{heap[heap[heap[heap[s].n].n].u]} \]
The problem

- Do these iterations access the same memory cell?

heap[heap[s].u] \(=\) heap[heap[heap[heap[s].n].n].u]
The problem

• Do these iterations access the same memory cell?

heap[heap[s].u] = ?

heap[heap[heap[heap[s].n].n].u]

... which has links to sub-trees
The problem

- Do these iterations access the same memory cell?
  \[
  \text{heap[heap[s].u]} = \text{heap[heap[heap[heap[s].n].n].u]}
  \]
- Need to reason about structure, heap layout and disjointness
The problem

- Do these iterations access the same memory cell?
  - \[\text{heap[heap[heap[s].n].n].u} = \text{heap[heap[s].u]}\]

- Need to reason about structure, heap layout and disjointness
- None of this is explicit in the above representation
Outline

• Case study: High-level synthesis of dynamic data structures
• Challenge
• Motivating example
• Leveraging recent advances in software verification
• Implementation and results
• Outlook
Describe heap layout with formulae
Classical first order logic

Describe heap layout with formulae

\[ s \rightarrow [u: u_1', n: s_1'] \]
Classical first order logic

Describe heap layout with formulae

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

"s points to"
Describe heap layout with formulae

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

“\( s \) points to a record with fields \( u \) and \( n \)”
Describe heap layout with formulae

\[ s \rightarrow [u: u_1', n: s_1'] \land s_1' \rightarrow [u: u_2', n: s_2'] \]
Describe heap layout with formulae

$s \rightarrow [u: u_1', n: s_1'] \land s_1' \rightarrow [u: u_2', n: s_2'] \land s_2' \rightarrow [u: u_3', n: 0]$
Classical first order logic

Describe heap layout with formulae

\[ s \rightarrow [u: u'_1, n: s'_1] \land s'_1 \rightarrow [u: u'_2, n: s'_2] \land s'_2 \rightarrow [u: u'_3, n: 0] \land u'_1 \rightarrow [l: u'_4, r: u'_5] \]
Describe heap layout with formulae

\[ s \rightarrow [u: u_1', n: s_1'] \land s_1' \rightarrow [u: u_2', n: s_2'] \land s_2' \rightarrow [u: u_3', n: 0] \land u_1' \rightarrow [l: u_4', r: u_5'] \land u_3' \rightarrow [l: u_8', r: u_9'] \]
Describe heap layout with formulae

\[ s \rightarrow [u: u'_1, n: s'_1] \land s'_1 \rightarrow [u: u'_2, n: s'_2] \land s'_2 \rightarrow [u: u'_3, n: 0] \land u'_1 \rightarrow [l: u'_4, r: u'_5] \land u'_3 \rightarrow [l: u'_8, r: u'_9] \land u'_2 \rightarrow [l: u'_6, r: u'_7] \]
Describe heap layout with formulae

\[ s \rightarrow [u: u_1', n: s_1'] \land s_1' \rightarrow [u: u_2', n: s_2'] \land s_2' \rightarrow [u: u_3', n: 0] \]
\[ \land u_1' \rightarrow [l: u_4', r: u_5'] \land u_3' \rightarrow [l: u_8', r: u_9'] \land u_2' \rightarrow [l: u_6', r: u_7'] \]
\[ \land \ldots \]
Describe heap layout with formulae

\[ s \rightarrow [u: u'_1, n: s'_1] \bigwedge s'_1 \rightarrow [u: u'_2, n: s'_2] \bigwedge s'_2 \rightarrow [u: u'_3, n: 0] \bigwedge u'_1 \rightarrow [l: u'_4, r: u'_5] \bigwedge u'_3 \rightarrow [l: u'_8, r: u'_9] \bigwedge u'_2 \rightarrow [l: u'_6, r: u'_7] \bigwedge \ldots \]
Classical first order logic

Describe heap layout with formulae

Formula below can also mean this

Conjunction ‘∧’ does not rule out aliasing!

\[ s \rightarrow [u: u'_1, n: s'_1] \land s'_1 \rightarrow [u: u'_2, n: s'_2] \land s'_2 \rightarrow [u: u'_3, n: 0] \land u'_1 \rightarrow [l: u'_4, r: u'_5] \land u'_3 \rightarrow [l: u'_8, r: u'_9] \land u'_2 \rightarrow [l: u'_6, r: u'_7] \land ... \]
Describe heap layout with formulae

Formula below can also mean this:

All $u$-pointers alias:

\[ u'_1 = u'_2 = u'_3 = u'_4 = \]
\[ u'_5 = u'_6 = u'_7 = u'_8 = u'_9 = \ldots \]

\[
\begin{align*}
    s & \to [u: u'_1, n: s'_1] \land s'_1 \to [u: u'_2, n: s'_2] \land s'_2 \to [u: u'_3, n: 0] \\
    \land u'_1 & \to [l: u'_4, r: u'_5] \land u'_3 \to [l: u'_8, r: u'_9] \land u'_2 \to [l: u'_6, r: u'_7] \\
    \land \ldots
\end{align*}
\]
Classical first order logic

Describe heap layout with formulae

Conjunction ‘∧’ does not rule out aliasing!

\[ s \rightarrow [u: u'_1, n: s'_1] \land s'_1 \rightarrow [u: u'_2, n: s'_2] \land s'_2 \rightarrow [u: u'_3, n: 0] \land u'_1 \rightarrow [l: u'_4, r: u'_5] \land u'_3 \rightarrow [l: u'_8, r: u'_9] \land u'_2 \rightarrow [l: u'_6, r: u'_7] \land \ldots \]
Describe heap layout with formulae

Could add loads of constraints

Conjunction ‘\(\land\)’ does not rule out aliasing!

\[
s \rightarrow [u: u'_1, n: s'_1] \land s'_1 \rightarrow [u: u'_2, n: s'_2] \land s'_2 \rightarrow [u: u'_3, n: 0]
\land u'_1 \rightarrow [l: u'_4, r: u'_5] \land u'_3 \rightarrow [l: u'_8, r: u'_9] \land u'_2 \rightarrow [l: u'_6, r: u'_7]
\land \ldots \land u'_1 \neq u'_2 \land u_1 \neq u'_3 \land u'_1 \neq u'_4 \land u'_1 \neq u'_5 \land u'_1 \neq u'_6 \land u'_1 \neq u'_7
\land u'_1 \neq u'_8 \land u_1 \neq u'_9 \land u'_3 \neq u'_4 \land u'_3 \neq u'_5 \land u'_3 \neq u'_6 \land u'_3 \neq u'_7
\land u'_3 \neq u'_8 \land u_2 \neq u'_3 \land u'_2 \neq u'_4 \land u'_2 \neq u'_5 \land u'_2 \neq u'_6 \land \ldots
\]
Describe heap layout with formulae

\[
\begin{align*}
  s & \rightarrow [u: u^\prime_1, n: s^\prime_1] \land s^\prime_1 \rightarrow [u: u^\prime_2, n: s^\prime_2] \land s^\prime_2 \rightarrow [u: u^\prime_3, n: 0] \\
  \land u^\prime_1 & \rightarrow [l: u^\prime_4, r: u^\prime_5] \land u^\prime_3 \rightarrow [l: u^\prime_8, r: u^\prime_9] \land u^\prime_2 \rightarrow [l: u^\prime_6, r: u^\prime_7] \\
  \land \ldots & \land u^\prime_1 \neq u^\prime_2 \land u_1 \neq u^\prime_3 \land u^\prime_1 \neq u^\prime_4 \land u^\prime_1 \neq u^\prime_5 \land u^\prime_1 \neq u^\prime_6 \land u^\prime_1 \neq u^\prime_7 \\
  \land u^\prime_1 \neq u^\prime_8 \land u_1 \neq u^\prime_9 \land u^\prime_3 \neq u^\prime_4 \land u^\prime_3 \neq u^\prime_5 \land u^\prime_3 \neq u^\prime_6 \land u^\prime_3 \neq u^\prime_7 \\
  \land u^\prime_3 \neq u^\prime_8 \land u_2 \neq u^\prime_3 \land u^\prime_2 \neq u^\prime_4 \land u^\prime_2 \neq u^\prime_5 \land u^\prime_2 \neq u^\prime_6 \land \ldots
\end{align*}
\]

Separating conjunction ‘*’

O’Hearn, Reynolds, Ishtiaq, Yang:
Describe heap layout with formulae

\[ s \rightarrow [u: u'_1, n: s'_1] \land s'_1 \rightarrow [u: u'_2, n: s'_2] \land s'_2 \rightarrow [u: u'_3, n: 0] \land u'_1 \rightarrow [l: u'_4, r: u'_5] \land u'_3 \rightarrow [l: u'_8, r: u'_9] \land u'_2 \rightarrow [l: u'_6, r: u'_7] \land \ldots \land u'_1 \not= u'_2 \land u_1 \not= u'_3 \land u'_1 \not= u'_4 \land u'_1 \not= u'_5 \land u'_1 \not= u'_6 \land u'_1 \not= u'_7 \land u'_1 \not= u'_8 \land u_1 \not= u'_9 \land u'_3 \not= u'_4 \land u'_3 \not= u'_5 \land u'_3 \not= u'_6 \land u'_3 \not= u'_7 \land u'_3 \not= u'_8 \land u_2 \not= u'_3 \land u'_2 \not= u'_4 \land u'_2 \not= u'_5 \land u'_2 \not= u'_6 \land \ldots \]
Describe heap layout with formulae

\[ s' \rightarrow [u: u'_1, n: s'_1] \quad \ast \quad s'_1 \rightarrow [u: u'_2, n: s'_2] \quad \ast \quad s'_2 \rightarrow [u: u'_3, n: 0] \]

\[ \ast \quad u'_1 \rightarrow [l: u'_4, r: u'_5] \quad \ast \quad u'_3 \rightarrow [l: u'_8, r: u'_9] \quad \ast \quad u'_2 \rightarrow [l: u'_6, r: u'_7] \]

\[ \land \quad \ldots \land u'_1 \neq u'_2 \land u_1 \neq u'_3 \land u'_1 \neq u'_4 \land u'_1 \neq u'_5 \land u'_1 \neq u'_6 \land u'_1 \neq u'_7 \land u'_1 \neq u'_8 \land u_1 \neq u'_9 \land u'_3 \neq u'_4 \land u'_3 \neq u'_5 \land u'_3 \neq u'_6 \land u'_3 \neq u'_7 \land u'_3 \neq u'_8 \land u_2 \neq u'_3 \land u'_2 \neq u'_4 \land u'_2 \neq u'_5 \land u'_2 \neq u'_6 \land \ldots \]
Describe heap layout with formulae

\[ s \rightarrow [u: u_1', n: s_1'] \ * \ s_1' \rightarrow [u: u_2', n: s_2'] \ * \ s_2' \rightarrow [u: u_3', n: 0] \]

\[ u_1' \rightarrow [l: u_4', r: u_5'] \ * \ u_3' \rightarrow [l: u_8', r: u_9'] \ * \ u_2' \rightarrow [l: u_6', r: u_7'] \]

\[ \land \ldots \land u_1' \neq u_2' \land u_1 \neq u_3' \land u_1' \neq u_4' \land u_1' \neq u_5' \land u_1' \neq u_6' \land u_1' \neq u_7' \land u_1' \neq u_8' \land u_1 \neq u_9' \land u_2' \neq u_4' \land u_3' \neq u_5' \land u_3' \neq u_6' \land u_3' \neq u_7' \land u_3' \neq u_8' \land u_2 \neq u_3' \land u_2' \neq u_4' \land u_2' \neq u_5' \land u_2' \neq u_6' \land \ldots \]

O’Hearn, Reynolds, Ishtiaq, Yang:
Separating conjunction ‘∗’ rules out aliasing!
Describe heap layout with formulae

\[ s \rightarrow [u: u_1', n: s_1'] \ * \ s_1' \rightarrow [u: u_2', n: s_2'] \ * \ s_2' \rightarrow [u: u_3', n: 0] \]
\[ u_1' \rightarrow [l: u_4', r: u_5'] \ * \ u_3' \rightarrow [l: u_8', r: u_9'] \ * \ u_2' \rightarrow [l: u_6', r: u_7'] \]

- Tractable heap analysis – very popular in SW verification

O’Hearn, Reynolds, Ishtiaq, Yang:
Separating conjunction ‘∗’ rules out aliasing!
Separation logic

Describe heap layout with formulae

\[
s \rightarrow [u: u'_1, n: s'_1] \quad \star \quad s'_1 \rightarrow [u: u'_2, n: s'_2] \quad \star \quad s'_2 \rightarrow [u: u'_3, n: 0]
\]
\[
\star \quad u'_1 \rightarrow [l: u'_4, r: u'_5] \quad \star \quad u'_3 \rightarrow [l: u'_8, r: u'_9] \quad \star \quad u'_2 \rightarrow [l: u'_6, r: u'_7]
\]

- Tractable heap analysis – very popular in SW verification
- We use it to prove disjointness of heap regions

O’Hearn, Reynolds, Ishtiaq, Yang:
Separating conjunction ‘\(\star\)’ rules out aliasing!
Back to our partitioning task

\[ s \rightarrow [u: u'_1, n: s'_1] \quad \ast s'_1 \rightarrow [u: u'_2, n: s'_2] \quad \ast s'_2 \rightarrow [u: u'_3, n: 0] \]

\[ * u'_1 \rightarrow [l: u'_4, r: u'_5] \quad * u'_3 \rightarrow [l: u'_8, r: u'_9] \quad * u'_2 \rightarrow [l: u'_6, r: u'_7] \]
Back to our partitioning task

• Partitioning the heap
  = partitioning the formula describing it

\[s \rightarrow [u: u_1', n: s_1'] \quad * \quad s_1' \rightarrow [u: u_2', n: s_2'] \quad * \quad s_2' \rightarrow [u: u_3', n: 0] \]

\[u_1' \rightarrow [l: u_4', r: u_5'] \quad * \quad u_3' \rightarrow [l: u_8', r: u_9'] \quad * \quad u_2' \rightarrow [l: u_6', r: u_7']\]
Back to our partitioning task

- Partitioning the heap = partitioning the formula describing it
- Add a second ‘hook’ into the data structure

\[ s \rightarrow [u: u_1', n: s_1'] \hspace{1em} s'_1 \rightarrow [u: u_2', n: s_2'] \hspace{1em} s'_2 \rightarrow [u: u_3', n: 0] \]

* \( u_1' \rightarrow [l: u_4', r: u_5'] \) * \( u_3' \rightarrow [l: u_8', r: u_9'] \) * \( u_2' \rightarrow [l: u_6', r: u_7'] \)
Back to our partitioning task

- Partitioning the heap = partitioning the formula describing it
- Add a second ‘hook’ into the data structure
- ‘Symbolically’ step through loop iterations

\[ s \to [u: u_1', n: s_1'] \times s_1' \to [u: u_2', n: s_2'] \times s_2' \to [u: u_3', n: 0] \]
\[ u_1' \to [l: u_4', r: u_5'] \times u_3' \to [l: u_8', r: u_9'] \times u_2' \to [l: u_6', r: u_7'] \]
Back to our partitioning task

- Partitioning the heap = partitioning the formula describing it
- Add a second ‘hook’ into the data structure
- ‘Symbolically’ step through loop iterations

\[
s \rightarrow [u: u_1', n: s_1'] \quad * \quad s_1' \rightarrow [u: u_2', n: s_2'] \quad * \quad s_2' \rightarrow [u: u_3', n: 0]
\]
\[
* \quad u_1' \rightarrow [l: u_4', r: u_5'] \quad * \quad u_3' \rightarrow [l: u_8', r: u_9'] \quad * \quad u_2' \rightarrow [l: u_6', r: u_7']
\]
Back to our partitioning task

- Partitioning the heap = partitioning the formula describing it
- Add a second ‘hook’ into the data structure
- ‘Symbolically’ step through loop iterations

\[ s \rightarrow [u: u'_1, n: s'_1] \]
\[ s'_1 \rightarrow [u: u'_2, n: s'_2] \]
\[ s'_2 \rightarrow [u: u'_3, n: 0] \]
\[ u'_1 \rightarrow [l: u'_4, r: u'_5] \]
\[ u'_3 \rightarrow [l: u'_8, r: u'_9] \]
\[ u'_2 \rightarrow [l: u'_6, r: u'_7] \]
Back to our partitioning task

- Partitioning the heap = partitioning the formula describing it
- Add a second ‘hook’ into the data structure
- ‘Symbolically’ step through loop iterations
- Attach labels to heaplets

\[
s \rightarrow [u: u'_1, n: s'_1] \quad \ast \quad s'_1 \rightarrow [u: u'_2, n: s'_2] \quad \ast \quad s'_2 \rightarrow [u: u'_3, n: 0] \\
\ast \quad u'_1 \rightarrow [l: u'_4, r: u'_5] \quad \ast \quad u'_3 \rightarrow [l: u'_8, r: u'_9] \quad \ast \quad u'_2 \rightarrow [l: u'_6, r: u'_7]
\]
Back to our partitioning task

- Partitioning the heap = partitioning the formula describing it
- Add a second ‘hook’ into the data structure
- ‘Symbolically’ step through loop iterations
- Attach labels to heaplets

\[
\begin{align*}
s &\rightarrow [u: u'_1, n: s'_1] \\
* \ u'_1 &\rightarrow [l: u'_4, r: u'_5]
\end{align*}
\]

\[
\begin{align*}
s'_1 &\rightarrow [u: u'_2, n: s'_2] \\
* \ u'_2 &\rightarrow [l: u'_6, r: u'_7]
\end{align*}
\]

\[
\begin{align*}
s'_2 &\rightarrow [u: u'_3, n: 0] \\
* \ u'_3 &\rightarrow [l: u'_8, r: u'_9]
\end{align*}
\]
Back to our partitioning task

- Partitioning the heap = partitioning the formula describing it
- Add a second ‘hook’ into the data structure
- ‘Symbolically’ step through loop iterations
- Attach labels to heaplets

\[
s \rightarrow [u: u'_1, n: s'_1] \quad * \quad s'_1 \rightarrow [u: u'_2, n: s'_2] \quad * \quad s'_2 \rightarrow [u: u'_3, n: 0]
\]

\[
* \quad u'_1 \rightarrow [l: u'_4, r: u'_5] \quad * \quad u'_3 \rightarrow [l: u'_8, r: u'_9] \quad * \quad u'_2 \rightarrow [l: u'_6, r: u'_7]
\]
Back to our partitioning task

- Partitioning the heap = partitioning the formula describing it
- Add a second ‘hook’ into the data structure
- ‘Symbolically’ step through loop iterations
- Attach labels to heaplets

\[
\begin{align*}
  s & \rightarrow [u: u'_1, n: s'_1] \\
  s'_1 & \rightarrow [u: u'_2, n: s'_2] \\
  u'_1 & \rightarrow [l: u'_4, r: u'_5] \\
  u'_3 & \rightarrow [l: u'_8, r: u'_9] \\
  s'_2 & \rightarrow [u: u'_3, n: 0] \\
  u'_2 & \rightarrow [l: u'_6, r: u'_7]
\end{align*}
\]
Back to our partitioning task

• Partitioning the heap
  = partitioning the formula describing it
• Add a second ‘hook’ into the data structure
• ‘Symbolically’ step through loop iterations
• Attach labels to heaplets

\[
\begin{align*}
  s & \rightarrow [u: u'_1, n: s'_1] \\
  u'_1 & \rightarrow [l: u'_4, r: u'_5] \\
  s'_1 & \rightarrow [u: u'_2, n: s'_2] \\
  u'_2 & \rightarrow [l: u'_6, r: u'_7] \\
  s'_2 & \rightarrow [u: u'_3, n: 0] \\
  u'_3 & \rightarrow [l: u'_8, r: u'_9]
\end{align*}
\]
Back to our partitioning task

- Partitioning the heap = partitioning the formula describing it
- Add a second ‘hook’ into the data structure
- ‘Symbolically’ step through loop iterations
- Attach labels to heaplets

\[
s \rightarrow [u: u_1', n: s_1'] \quad \ast \quad s_1' \rightarrow [u: u_2', n: s_2'] \quad \ast \quad s_2' \rightarrow [u: u_3', n: 0]
\]
\[
\ast \quad u_1' \rightarrow [l: u_4', r: u_5'] \quad \ast \quad u_3' \rightarrow [l: u_8', r: u_9'] \quad \ast \quad u_2' \rightarrow [l: u_6', r: u_7']
\]

Communication-free parallelism: Never ...
Symbolic execution

- Capture the semantics of a program in terms of heap access
- Specify semantics of program commands
- Examples:
  \[
  \{x = y'\} \ x := 3 \ {x = 3} \quad \text{assignment}
  \]
  \[
  \{x \rightarrow y'\} \ [x] := 3 \ {x \rightarrow 3} \quad \text{heap assignment}
  \]
  \[
  \{\text{emp}\} \ \text{new}(x) \ {x \rightarrow y'} \quad \text{allocation}
  \]
Symbolic execution

Propagate formulae through CFG:

\[
\begin{align*}
\{\text{emp}\} \\
\text{l1: } & \text{ new(x); } \\
& \{x \rightarrow y'\} \\
\text{l2: } & \text{ [x] = 3; } \\
& \{x \rightarrow 3\} \\
\text{l3: } & \ldots
\end{align*}
\]
Loops

\{pre_0\}

l1: while \(b\) do

\{pre_0 \land b\}

l1: new(x);

l2: \([x] = 3\);

l3: ...

\{post_0\}

l4: end while
• Can’t unroll ALL iterations during analysis
  data-dependent loop condition $b$

\begin{align*}
\text{l1:} & \quad \text{while } b \text{ do} \\
& \quad \begin{cases} 
\{pre_0\} \\
\{pre_1 \land b\} \\
\{post_1\} \\
\end{cases} \\
\text{l1:} & \quad \text{new(x);} \\
\text{l2:} & \quad [x] = 3; \\
\text{l3:} & \quad \ldots \\
\text{l4:} & \quad \text{end while}
\end{align*}
Loops

- Can’t unroll ALL iterations during analysis data-dependent loop condition \( b \)
- Instead:
  Symb. execute iterations until a fix-point can be established (Magill 2006)

\[
\text{post}_i \Rightarrow \text{post}_{i-1}
\]

- Theorem prover to decide

\[
\{\text{pre}_0\}
\]

\[
\begin{align*}
\text{l1:} & \quad \text{while } b \text{ do} \\
\{\text{pre}_1 \land b\} & \\
\text{l1:} & \quad \text{new}(x); \\
\text{l2:} & \quad [x] = 3; \\
\text{l3:} & \quad \ldots \\
\{\text{post}_1\} & \\
\text{l4:} & \quad \text{end while}
\end{align*}
\]
Outline

• Case study: High-level synthesis of dynamic data structures
• Challenge
• Motivating example
• Leveraging recent advances in software verification
• Implementation and results
• Outlook
Implementation

Automated source-to-source compiler
LLNL ROSE Compiler Infrastructure (C++)

Parse into AST

C/C++ code

Analysis interface

Substitute dynamic memory allocation

Heap splitting / Loop splitting

Heap analysis / Theorem proving

Automated proof engine (OCaml)

Unparse AST

C/C++ code

Vivado HLS

RTL impl.
Tree-based $K$-means clustering

Latency [clock cycles]

<table>
<thead>
<tr>
<th>Slices</th>
<th>Latency $\times 10^5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>2</td>
</tr>
<tr>
<td>4000</td>
<td>4</td>
</tr>
<tr>
<td>6000</td>
<td>6</td>
</tr>
<tr>
<td>8000</td>
<td>8</td>
</tr>
<tr>
<td>10000</td>
<td>10</td>
</tr>
</tbody>
</table>

- no parallelization, only ensuring synthesizability
- parallelization $p=2$
- parallelization $p=4$
## Case Studies (II)

<table>
<thead>
<tr>
<th>Case Study</th>
<th>P</th>
<th>Slices</th>
<th>Clock</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1. Merger (linked lists)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baseline (no par.)</td>
<td>1</td>
<td>574</td>
<td>9.0 ns</td>
<td>21167k</td>
</tr>
<tr>
<td>Autom. Parallelization</td>
<td>4</td>
<td>965</td>
<td>8.7 ns</td>
<td>5483k</td>
</tr>
<tr>
<td><strong>2. Tree deletion (tree, linked list)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baseline (no par.)</td>
<td>1</td>
<td>1521</td>
<td>5.2 ns</td>
<td>901k</td>
</tr>
<tr>
<td>Autom. Parallelization</td>
<td>2</td>
<td>4069</td>
<td>6.0 ns</td>
<td>487k</td>
</tr>
<tr>
<td><strong>3. K-means (tree, linked list, single heap records)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baseline (no par.)</td>
<td>1</td>
<td>2694</td>
<td>6.1 ns</td>
<td>1120k</td>
</tr>
<tr>
<td>Autom. Parallelization</td>
<td>2</td>
<td>5618</td>
<td>7.0 ns</td>
<td>606k</td>
</tr>
</tbody>
</table>
## Case Studies (II)

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>Slices</th>
<th>Clock</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong></td>
<td><strong>Merger (linked lists)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baseline (no par.)</td>
<td>1</td>
<td>574</td>
<td>9.0 ns</td>
<td>21167k</td>
</tr>
<tr>
<td>Autom. Parallelization</td>
<td>4</td>
<td>965</td>
<td>8.7 ns</td>
<td>5483k</td>
</tr>
<tr>
<td><strong>2</strong></td>
<td><strong>Tree deletion (tree, linked list)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baseline (no par.)</td>
<td>1</td>
<td>1521</td>
<td>5.2 ns</td>
<td>901k</td>
</tr>
<tr>
<td>Autom. Parallelization</td>
<td>2</td>
<td>4069</td>
<td>6.0 ns</td>
<td>487k</td>
</tr>
<tr>
<td><strong>3</strong></td>
<td><strong>K-means (tree, linked list, single heap records)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baseline (no par.)</td>
<td>1</td>
<td>2694</td>
<td>6.1 ns</td>
<td>1120k</td>
</tr>
<tr>
<td>Autom. Parallelization</td>
<td>2</td>
<td>5618</td>
<td>7.0 ns</td>
<td>606k</td>
</tr>
</tbody>
</table>
## Case Studies (II)

<table>
<thead>
<tr>
<th>P</th>
<th>Slices</th>
<th>Clock</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Merger (linked lists)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Baseline (no par.)</td>
<td>1</td>
<td>574</td>
</tr>
<tr>
<td></td>
<td>Autom. Parallelization</td>
<td>4</td>
<td>965</td>
</tr>
<tr>
<td>2</td>
<td><strong>Tree deletion (tree, linked list)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Baseline (no par.)</td>
<td>1</td>
<td>1521</td>
</tr>
<tr>
<td></td>
<td>Autom. Parallelization</td>
<td>2</td>
<td>4069</td>
</tr>
<tr>
<td>3</td>
<td><strong>K-means (tree, linked list, single heap records)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Baseline (no par.)</td>
<td>1</td>
<td>2694</td>
</tr>
<tr>
<td></td>
<td>Autom. Parallelization</td>
<td>2</td>
<td>5618</td>
</tr>
<tr>
<td></td>
<td>Hand-optimized HLS</td>
<td>2</td>
<td>5492</td>
</tr>
</tbody>
</table>

Manual loop flattening, pipelining, custom bit widths, data streaming directives, data packing, ...
### Case Studies (II)

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>Slices</th>
<th>Clock</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Merger (linked lists)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baseline (no par.)</td>
<td>1</td>
<td>574</td>
<td>9.0 ns</td>
<td>21167k</td>
</tr>
<tr>
<td>Autom. Parallelization</td>
<td>4</td>
<td>965</td>
<td>8.7 ns</td>
<td>5483k</td>
</tr>
<tr>
<td><strong>2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Tree deletion (tree, linked list)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baseline (no par.)</td>
<td>1</td>
<td>1521</td>
<td>5.2 ns</td>
<td>901k</td>
</tr>
<tr>
<td>Autom. Parallelization</td>
<td>2</td>
<td>4069</td>
<td>6.0 ns</td>
<td>487k</td>
</tr>
<tr>
<td><strong>3</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>K-means (tree, linked list, single heap records)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Baseline (no par.)</td>
<td>1</td>
<td>2694</td>
<td>6.1 ns</td>
<td>1120k</td>
</tr>
<tr>
<td>Autom. Parallelization</td>
<td>2</td>
<td>5618</td>
<td>7.0 ns</td>
<td>606k</td>
</tr>
<tr>
<td>Hand-optimized HLS</td>
<td>2</td>
<td>5492</td>
<td>5.5 ns</td>
<td>165k</td>
</tr>
</tbody>
</table>

Manual loop flattening, pipelining, custom bit widths, data streaming directives, data...
• Case study: High-level synthesis of dynamic data structures
• Challenge
• Motivating example
• Leveraging recent advances in software verification
• Implementation and results
• **Outlook**
What else?

1. Generating application-specific multi-cache architectures
2. Loop transformations for pipelining
3. Worst-case/ average case bounds on heap usage
Generate multi-cache arch

Off-chip memory

arb

FPGA

FU1

FU2
Generate multi-cache arch

Off-chip memory

arb

cache1

FU1

cache2

FU2

FPGA
Generate multi-cache arch

Off-chip memory

Cache coherence

arb

cache1

FU1

FU2

cache2

FPGA

Communication-free parallelism:
Caches are truly private
What else?

1. Generating application-specific multi-cache architectures
2. Loop transformations for pipelining
3. Worst-case/average case bounds on heap usage
s = PUSH(root, s);
while s!=0 do
    s = POP(&u, s);
    ... 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

Change loop schedule to increase distance between dependent iterations.
Pipelining

Loop-carried dependency

\begin{verbatim}
s = PUSH(root, s);
while s!=0 do
    s = POP(&u, s);
    ... 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
\end{verbatim}
Loop-carried dependency

```
s = PUSH(root, s);
while s!=0 do
    s = POP(&u, s);
    ... 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
```
1. Generating application-specific multi-cache architectures
2. Loop transformations for pipelining
3. Worst-case/ average case bounds on heap usage
Bounds on heap usage

Profiling the tree-based clustering app

Bound on memory size [number of centre sets]

Run-time [node-centre pairs x100k]

N=16384, K=128, \( \sigma = 0.2 \)
Bounds on heap usage

Profiling the tree-based clustering app

Run-time (node-centre pairs x100k)

Bound on memory size [number of centre sets]

N=16384, K=128, $\sigma=0.2$

512 BRAMs

Cook et al., FMCAD 2009?
Bounds on heap usage

Profiling the tree-based clustering app

- Bound on memory size [number of centre sets]
- Run-time [node-centre pairs x100k]

- N=16384, K=128, σ=0.2

- 8 BRAMs
- 512 BRAMs

- How?

- How? [Profiling the tree-based clustering app]

Cook et al., FMCAD 2009?
Bounds on heap usage

Profiling the tree-based clustering app

Run-time

[node-centre pairs x100k]

Bound on memory size [number of centre sets]

On-chip

Off-chip

N=16384, K=128, \( \sigma = 0.2 \)

[8 BRAMs x 22]

8 BRAMs

512 BRAMs

How?

Cook et al., FMCAD 2009?
Conclusion

• Limited HLS support for heap-manipulating programs

• Static analysis of heap-manipulating programs
  – Leveraging recent advances in separation logic
  – Distribute heap across on-chip memory banks
  – Loop parallelization

• STeffiHLS tool implementation
  – Automated heap analyzer
  – Source-to-source transformations (synthesizability and parallelization)
  – Successful parallelization using standard HLS tool

• Future work
  – Generating application-specific multi-cache architectures
  – Loop transformations for pipelining
  – Compute worst-case/average-case bounds on heap usage
Thank you for listening.

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