Improving High-Level Synthesis with Decoupled Data Structure Optimization

Ritchie Zhao, Gai Liu, Shreesha Srinath, Christopher Batten, Zhiru Zhang

Computer Systems Lab
Electrical and Computer Engineering
Cornell University
Vision for Future HLS

Current HLS tools
Limited usage by hardware designers in a few specialized domains

Many challenges still remain

Our contribution: Extend HLS to efficiently handle new programs

Future HLS tools
Wide usage by mainstream programmers who are not hardware experts
Traditional HLS Programs

- program = algorithm + data structure
  - Interface = data structure methods

### Algorithms

- Simple Data Structures
  - e.g., arrays, queues
  - Fixed latency methods

### CRC Error Detection

```c
unsigned crc( msg[32], len ) {
    R = 0;
    for (i = 0; i < len; ++i) {
        R ^= msg[i] << (3*8);
        for (bit = 8; bit > 0; --bit) {
            if (R & (1 << 31))
                R ^= 0xD8;
            R = R << 1;
        }
    }
    return R;
}
```
HLS with Complex Data Structures

- **program = algorithm + data structure**
  - Interface = data structure methods

### Dijkstra's Algorithm

```cpp
void priority_queue::push(val) {
    data[size] = val;
    unsigned curr = size;
    ++size;

    while (curr != 0) {
        prev = (curr-1) >> 1;
        if (data[curr] > data[prev])
            swap(&data[curr], &data[prev]);
        else
            break;
        curr = prev;
    }
}
```
Many programs base their efficiency on complex data structures, which are poorly handled by existing HLS tools.

**Definition:** A data structure is complex if its **key methods** exhibit **variable latency**

### Example complex data structures from the C++ STL containers library

<table>
<thead>
<tr>
<th>Container</th>
<th>Underlying Data Structure</th>
<th>Key Methods</th>
<th>Variable Latency Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>map, set</td>
<td>Red-black tree</td>
<td>insert, delete</td>
<td>Tree traversal and rotations</td>
</tr>
<tr>
<td>unordered_map, unordered_set</td>
<td>Hash table</td>
<td>insert, delete</td>
<td>Collision chain traversal</td>
</tr>
<tr>
<td>priority_queue</td>
<td>Heap</td>
<td>push, pop</td>
<td>Maintaining heap condition</td>
</tr>
</tbody>
</table>
Dijkstra’s Algorithm

```cpp
s = u.begin_neighbors();
e = u.end_neighbors();
// inner loop
for (v = s; v < e; ++v) {
  alt = dist[u] + edge[u][v];
  if (dist[v] > alt) {
    dist[v] = alt;
    // priority queue push
    Q.push(v, dist[v]);
  }
}
```

Conventional HLS Flow

1. **Static schedule** for the entire program
2. Monolithic hardware executes **in lockstep** adhering to the schedule

Generated Hardware
Dijkstra’s Algorithm

s = u.begin_neighbors();
e = u.end_neighbors();  
// inner loop
for (v = s; v < e; ++v) {
    alt = dist[u] + edge[u][v];
    if (dist[v] > alt) {
        dist[v] = alt;
        // priority queue push
        Q.push(v, dist[v]);
    }
}

Algorithm
Complex Method

Baseline Execution

Pipelined Execution

Initiation Interval (II)
Decoupled Data Structure Synthesis

1. **Decouple** complex methods from the algorithm using a latency-insensitive interface
   - Separation of concerns, eliminate lockstep execution

2. **Map** the complex data structure to a **specialized container unit (SCU)**
   - Potential for parallel and out-of-order method execution
Previous Work

- Individual data structure accelerators [Xu et al., CISP’08] [Huang et al., FPL’14] [Oberg et al., FPL’12]
  - Complementary to our approach

- Memory operation decoupling [Cheng & Wawrzynek, FPT’14]
  - We decouple entire methods, which may contain many loads and/or stores
Specialized Container Unit

- Architectural template
- Arrows indicate latency-insensitive interfaces (e.g., val-ready)
- Complex method calls $\rightarrow$ request/response messages
Specialized Method Units

- Complex method code is removed from the program and synthesized into **specialized method units** (SMUs)
  - Accessor-SMUs (A-SMUs)
  - Mutator-SMUs (M-SMUs)

**Accessor SMUs** contain multiple lanes for parallel and out-of-order execution

**Mutator SMUs** may only execute when all other SMUs are idle (conservative assumption)
Other SCU Blocks

- **Dispatcher**: handles requests and safely invokes the SMUs
- **Collector**: gathers results and returns them in calling order
- **Crossbar & Arbiter**: allows SMUs to share memory ports
SCU Execution (Mutator)

**Dijkstra’s Algorithm**

```c
s = u.begin_neighbors();
e = u.end_neighbors();
// inner loop
for (v = s; v < e; ++v) {
    alt = dist[u] + edge[u][v];
    if (dist[v] > alt) {
        dist[v] = alt;
        // priority queue push
        Q.push(v, dist[v]);
    }
}
```

**Static Pipeline Execution**

Initiation Interval (II)

```
i=0  i=1  i=2  i=3  ...
```

```
i=0  i=1  i=2  ...
```

**Decoupled Execution**

```
i=0  i=1  i=2  ...
```

- Decoupling enables continuous execution without a static schedule and dynamically exploits parallelism
SCU Execution (Accessor)

KeySearch Kernel

```c
void key_search() {
    for (i = ...)
        // hash find
        n = table.find(k);
        // algorithm
        if (n != NULL)
            vals[i] = n.val;
}
```

Decoupled Execution

- Multiple lanes enable parallel and out-of-order method execution to greatly improve performance.
Experimental Setup

- **Baseline**
  - HLS program written in synthesizable C++

- **SCU Flow**
  - Extract the complex method code and use it to synthesize SMUs
  - Synthesize dispatcher, collector, etc. from C++ templates

- **Tools**
  - Vivado HLS as the HLS tool
  - Vivado 2015.3 to implement the generated HDL
Performance and Area Comparison

- Target device is Virtex-7, target clock period is 5ns
- **Average Speedup**: 1.6x
- **Average Area Overhead**: 30% LUT, 20% FF, 10% BRAM
Scalability

Area overhead remains fairly constant while speedup continues to improve with more lanes.
Current and future HLS applications will contain both fixed and variable-latency code.

Decoupling the two parts and separately optimizing them is a promising approach.

Where to decouple and what optimizations to apply are key questions to address.

Future work
  - Intelligent prefetching in the SCU
  - Overlapped mutator-SMU execution
Example Execution (Mutator)

a) Baseline, non-pipelined
b) Baseline, pipelined
c) Decoupled, algorithm unit pipelined
Example Execution (Mutator)

a) Baseline, non-pipelined
b) Baseline, pipelined
c) Decoupled
d) Decoupled, algorithm pipelined
Specialized Method Units (SMUs)

- Complex method code is synthesized into **specialized method units** (SMUs)
- SMUs can be **accessor** or **mutator** SMUs