Towards Loop Pipelining for a Verified HLS Tool

Yann Herklotz, John Wickerson

Imperial College London

The Need For Formally Verified HLS

- HLS cannot be used for **critical applications**.

The Need For Formally Verified HLS

- HLS cannot be used for **critical applications**.
    - Even simple programs can produce bugs in HLS tools.

```c
unsigned int x = 0x1194D7FF;
int arr[6]={1,1,1,1,1,1};
int main() {
  for (int i = 0; i < 2; i++) x = x >> arr[i];
  return x;
}
```

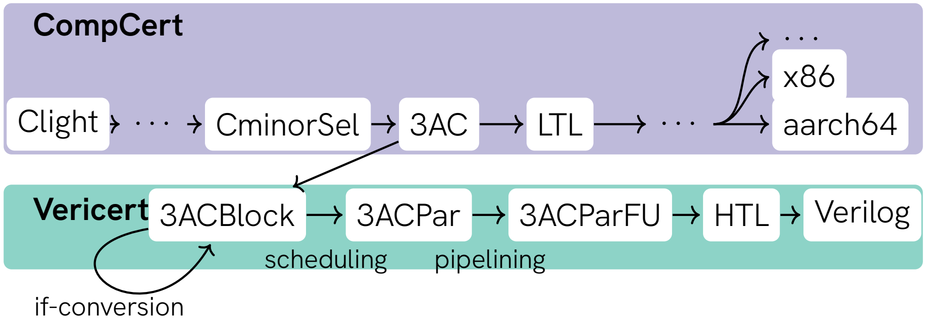
The Need For Formally Verified HLS

- HLS cannot be used for **critical applications**.
    - Even simple programs can produce bugs in HLS tools.
- **Functional testing** of hardware has to be **redone**.

The Need For Formally Verified HLS

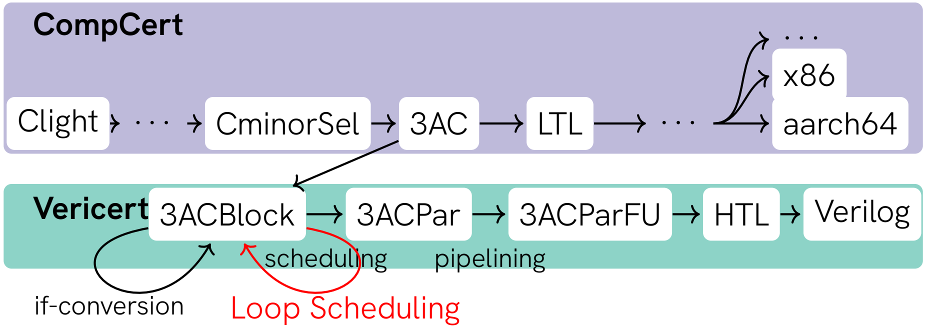
- HLS cannot be used for **critical applications**.
    - Even simple programs can produce bugs in HLS tools.
- **Functional testing** of hardware has to be **redone**.
- **Goal**: Create a practical, formally verified HLS tool in Coq.

Current Status of Vericert



Current work adds **hyperblock scheduling** to **Vericert**.

Current Status of Vericert



We argue we can add **hardware loop pipelining** as a **source-to-source transformation** doing **software loop pipelining**, which is easier to verify in **Coq**.

Outline

The Need for Loop Pipelining

- Main difficulty with having hardware as a target is the need to pipeline loops.

```
for (int i = 3; i < N; i++)
    acc[i] = acc[i-3]*c+x[i]*y[i];
```

The Need for Loop Pipelining

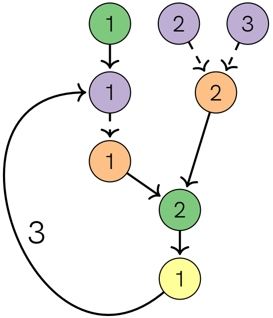
- Main difficulty with having hardware as a target is the need to pipeline loops.

```
for (int i = 3; i < N; i++) {
① x18 = i - 3
① x16 = load[1, x18]
① x8 = x16 * x1
② x12 = load[3, i]
③ x13 = load[2, i]
② x7 = x12 * x13
② x11 = x8 + x7
① store[1, i] = x11
  i = i + 1
}
```

The Need for Loop Pipelining

- Main difficulty with having hardware as a target is the need to pipeline loops.

```
for (int i = 3; i < N; i++) {
```

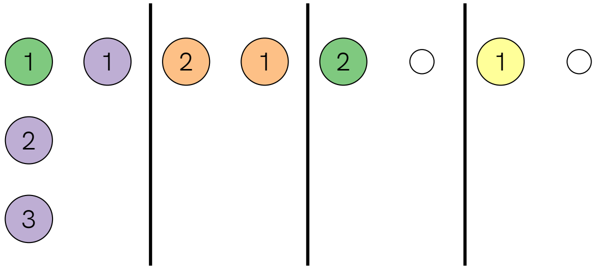


```
}
```

Ideas Behind Static Hardware Loop Pipelining

One Possible Workflow

- Generate scheduling constraints for linear code as well as loops.
- Solve for a scheduling using an ILP solver.
- Place the instructions into the cycle that it was assigned to.



Verifying Hardware Pipelining is Difficult

- Normally part of the scheduling step.

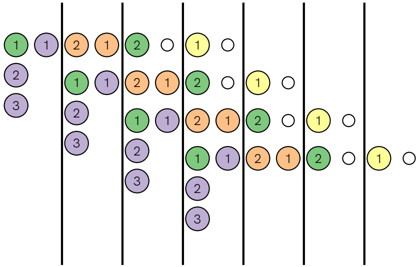
Verifying Hardware Pipelining is Difficult

- Normally part of the scheduling step.
- **Lose control** about how the loops are translated, the fundamental **structure** of the loop could change and would be difficult to identify again.

Outline

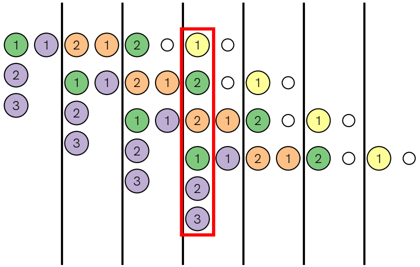
Ideas Behind Software Loop Pipelining

In software pipelining we represent a vertical slice of the pipeline.



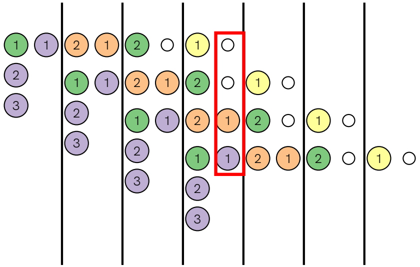
Ideas Behind Software Loop Pipelining

In software pipelining we represent a vertical slice of the pipeline.



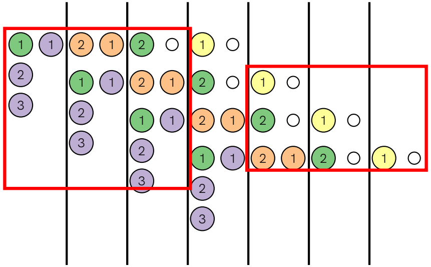
Ideas Behind Software Loop Pipelining

In software pipelining we represent a vertical slice of the pipeline.



Ideas Behind Software Loop Pipelining

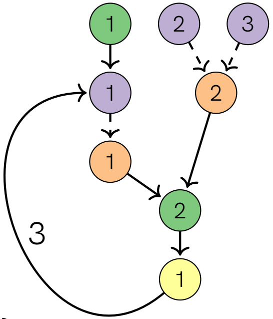
In software pipelining we represent a vertical slice of the pipeline.



Ideas Behind Software Loop Pipelining

- Source-to-source transformation to generate a **pipeline** in **software**.
- Use **rotating register file** to avoid unrolling due to **modulo variable expansion**.

```
for (int i = 3; i < N; i++) {
```



```
}
```

```
for (int i = 3; i < N; i++) {
    ①[i]
    ②[i]
    ③[i]
    ②[i-1]
    ②[i-2]
    ①[i-3]

    ①[i]
    ①[i-1]
    i = i + 1
}
```

Ideas Behind Software Loop Pipelining

- Use **predicated execution** to avoid adding explicit prologue and epilogue.

```
if (i < N): p0 = true
|| if p0: p1 = true
|| if p1: p2 = true
|| if p2: p3 = true
if (i >= N): p0 = false
|| if !p0: p1 = false
|| if !p1: p2 = false
|| if !p2: p3 = false
```

```
for (int i = 3; i < N+4; i++) {
    ...
    if p0: ①[i]
    if p0: ②[i]
    if p0: ③[i]
    if p1: ②[i-1]
    if p2: ②[i-2]
    if p3: ①[i-3]

    if p0: ①[i]
    if p1: ①[i-1]
    i = i + 1
}
```

Use Abstract Interpretation to Verify the Transformation

Abstract Interpretation

Define an α, such that $\alpha(\mathcal{C})$ evaluates some code $\mathcal{C}$ and returns **symbolic states** for all registers.

Use Abstract Interpretation to Verify the Transformation

Abstract Interpretation

Define an α, such that $\alpha(\mathcal{C})$ evaluates some code $\mathcal{C}$ and returns **symbolic states** for all registers.

Example

Executing the following code will evaluate to the following symbolic code:

$$\alpha\begin{pmatrix}\texttt{x = 2} \\ \texttt{y = x + z}\end{pmatrix} \quad = \quad \begin{array}{l}\mathtt{x} \mapsto 2 \\ \mathtt{y} \mapsto 2 + \mathtt{z}^0\end{array}$$

Using Abstract Interpretation to Verify Software Pipelining

$$\begin{tikzcd}\boxed{\text{3ACBlock}} \arrow[r, "\text{loop pipelining}"] & \boxed{\text{3ACBlock}}\end{tikzcd}$$

- Difficult to prove the loop pipelining algorithm correct directly.

Using Abstract Interpretation to Verify Software Pipelining

$$\begin{tikzcd}
\boxed{\text{3ACBlock}} \arrow[r, "\text{loop pipelining}"] \arrow[r, bend right, dashed, "\sim"'] & \boxed{\text{3ACBlock}}
\end{tikzcd}$$

- Difficult to prove the loop pipelining algorithm correct directly.

Using Abstract Interpretation to Verify Software Pipelining

$$\begin{tikzcd}
\boxed{\text{3ACBlock}} \arrow[r, "\text{loop pipelining}"] \arrow[d, "\alpha"'] & \boxed{\text{3ACBlock}} \arrow[d, "\alpha"'] \\
\boxed{\text{Abstr}} \arrow[r, dashed, no head, "\text{equivalence check}"'] & \boxed{\text{Abstr}}
\end{tikzcd}$$

- Instead, use a verified validator to check the schedule.

Using Abstract Interpretation to Verify Software Pipelining

$$\begin{tikzcd}
\boxed{\text{3ACBlock}} \arrow[r, "\text{loop pipelining}"] \arrow[d, "\alpha"'] \arrow[d, dashed, bend left, "\sim"] & \boxed{\text{3ACBlock}} \arrow[d, "\alpha"] \\
\boxed{\text{Abstr}} \arrow[r, dash, dashed, "\text{equivalence check}"'] \arrow[r, leftrightarrow, dashed, bend left, "\sim"] & \boxed{\text{Abstr}} \arrow[u, dashed, bend left, "\sim"]
\end{tikzcd}$$

- Instead, use a verified validator to check the schedule.

Verifying Software Loop Pipelining

For a loop $\mathcal{L}_1$ and a pipelined loop $\mathcal{L}_2$, we want to prove:

$$\forall N, \alpha(\mathcal{L}_1^N) = \alpha(\mathcal{L}_2^N)$$

- This is not feasible as N is often not known statically

Verifying Software Loop Pipelining

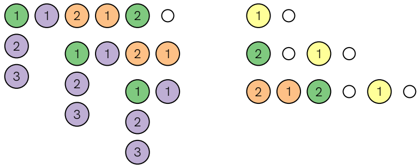
For a loop $\mathcal{L}_1$ and a pipelined loop $\mathcal{L}_2$, we want to prove:

$$\forall N, \alpha(\mathcal{L}_1^N) = \alpha(\mathcal{L}_2^N)$$

- This is not feasible as N is often not known statically

It is enough to prove various static properties:

$$\alpha\Bigg($$



$$\Bigg) = \alpha(\mathcal{L}^3)$$

Outline

Comparing Pipelines in Hardware and Software

Representation

Hardware pipelining each instruction is put into a state and it is filled with data at the correct II.

Software pipelining the code represents the kernel of the pipeline, expressing each repeating instruction.

Comparing Pipelines in Hardware and Software

Representation

Hardware pipelining each instruction is put into a state and it is filled with data at the correct II.

Software pipelining the code represents the kernel of the pipeline, expressing each repeating instruction.

Pipelines themselvs are identical

- In terms of **expressivity**, both hardware and software pipelining can express the **same loop pipelines**.

Outline

Conclusion

- Verifying **hardware pipelining** together with scheduling is difficult.
  - Too many instructions move around and their positions need to be recovered.
- By doing **software pipelining** followed by **instruction scheduling** and **hardware generation**, hardware pipelines can be approximated.
  - Same pipeline but with **slightly higher resource usage**.