EECS150 - Digital Design
Lecture 23 - High-level Design and Optimization 3, Parallelism and Pipelining

Nov 12, 2002
John Wawrzynek
Parallelism

Parallelism is the act of **doing more than one thing at a time**. Optimization in hardware design often involves using parallelism to trade between cost and performance.

- Example, Student final grade calculation:
  ```plaintext
  read mt1, ht2, mt3, project;
  grade = 0.2 \times mt1 + 0.2 \times mt2
  + 0.2 \times mt3 + 0.4 \times project;
  write grade;
  ```
- High performance hardware implementation:

As many operations as possible are done in parallel.
Parallelism

- Is there a lower cost hardware implementation? Different tree organization?
- Can factor out multiply by 0.2:

\[ \text{mt1} \quad \text{mt2} \quad 0.4 \quad \text{proj} \]

\[ \begin{array}{c}
\text{mt3} \\
\times \\
\text{X} \\
+ \\
\text{grade}
\end{array} \]

- How about sharing operators (multipliers and adders)?
Parallelism

- *Time multiplex* single ALU for all adds and multiplies:
  - Attempts to minimize cost at the expense of time.
    - Need to add extra register, muxes, control.

- If we adopt this approach, we can then consider the combinational hardware circuit diagram as an *abstract computation-graph*.

  Using other primitives, other coverings are possible.

- This technique “covers” the computation graph by performing the action of each node one at a time. (*Sort of emulates* it.)
HW versus SW

• **This time-multiplexed ALU** approach is very similar to what a conventional software version would accomplish:
  
  ```
  add r2, r1, r3
  add r2, r2, r4
  mult r2, r4, r5
  ...  
  ```

• CPUs time-multiplex function units (ALUs, etc.)

• This model matches our tendency to express computation sequentially - even though many naturally contain parallelism.

• Our programming languages also strengthen this tendency.

• In hardware we have the ability to exploit problem parallelism - gives us a “knob” on performance/cost.

• Maybe best to express computations as abstract computations graphs (rather than “programs”) - should lead to wider range of implementations.

• **Note:** modern processors spend much of their cost budget attempting to restore execution parallelism: “super-scalar execution”.
Optimizing Iterative Computations

• Hardware implementations of computations almost always involves looping. Why?
• Is this true with software?
• Are there programs without loops?
  – Maybe in “through away” code.
• We probably would not bother building such a thing into hardware, would we?
  – (FPGA may change this.)
• Fact is, our computations are closely tied to loops. Almost all our HW includes some looping mechanism.
• What do we use looping for?
Optimizing Iterative Computations

Types of loops:

1) Looping over input data (streaming):
   - ex: MP3 player, video compressor, music synthesizer.

2) Looping over memory data
   - ex: vector inner product, matrix multiply, list-processing

   These two are really very similar. 1) is often turned into 2) by buffering up input data, and processing “offline”. Even for “online” processing, buffers are used to smooth out temporary rate mismatches.

3) CPUs are one big loop.
   - Instruction fetch ⇒ execute ⇒ Instruction fetch ⇒ execute ⇒ …
   - but change their personality with each iteration.

4) Others?

Loops offer opportunity for parallelism

by executing more than one iteration at once,

through parallel iteration execution &/or pipelining
Pipelining

• With looping usually we are less interested in the latency of one iteration and more in the loop execution rate, or throughput.
• These can be different due to parallel iteration execution &/or pipelining.
• Pipelining review from CS61C:

Analog to washing clothes:

step 1: wash (20 minutes)
step 2: dry (20 minutes)
step 3: fold (20 minutes)

60 minutes x 4 loads ⇒ 4 hours

```
wash  | load1 | load2 | load3 | load4 |
dry   | load1 | load2 | load3 | load4 |
fold  | load1 | load2 | load3 | load4 |
```

20 min

overlapped ⇒ 2 hours
Pipelining

<table>
<thead>
<tr>
<th></th>
<th>wash</th>
<th>load1</th>
<th>load2</th>
<th>load3</th>
<th>load4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>dry</td>
<td>load1</td>
<td>load2</td>
<td>load3</td>
<td>load4</td>
</tr>
<tr>
<td></td>
<td>fold</td>
<td>load1</td>
<td>load2</td>
<td>load3</td>
<td>load4</td>
</tr>
</tbody>
</table>

- In the limit, as we increase the number of loads, the average time per load approaches 20 minutes.
- The latency (time from start to end) for one load = 60 min.
- The throughput = 3 loads/hour
- The pipelined throughput $\approx$ # of pipe stages x un-pipelined throughput.
Pipelining

- General principle:

  - Cut the CL block into pieces (stages) and separate with registers:

    \[
    T' = 4\text{ns} + 1\text{ns} + 4\text{ns} + 1\text{ns} = 10\text{ns}
    \]

    \[
    F = \frac{1}{4\text{ns} + 1\text{ns}} = 200\text{MHz}
    \]

  - CL block produces a new result every 5ns instead of every 9ns.

- Cut the CL block into pieces (stages) and separate with registers:

  \[
  T = 8\text{ns}
  \]

  \[
  T_{FF}(\text{setup } + \text{clk} \rightarrow q) = 1\text{ns}
  \]

  \[
  F = \frac{1}{9\text{ns}} = 111\text{MHz}
  \]

  Assume \(T_1 = T_2 = 4\text{ns}\)

  \[
  T' = 4\text{ns} + 1\text{ns} + 4\text{ns} + 1\text{ns} = 10\text{ns}
  \]

  \[
  F = \frac{1}{(4\text{ns} + 1\text{ns})} = 200\text{MHz}
  \]
Limits on Pipelining

- Without FF overhead, throughput improvement $\alpha$ # of stages.
- After many stages are added, FF overhead begins to dominate:

Other limiters:
- clock skew contributes to clock overhead
- unequal stages
- FFs dominate cost
- clock distribution power consumption
- feedback (dependencies between loop iterations)
Example

• $F(x) = y_i = ax_i^2 + bx_i + c$

• $x$ and $y$ are assumed to be "streams"

• Divide into 3 (nearly) equal stages.
  • Insert pipeline registers at dashed lines.

• Can we pipeline basic operators?

• Computation graph:
Example: Pipelined Adder

\[ \begin{align*}
\text{FA} & \quad \text{FA} & \quad \text{FA} & \quad \text{FA} \\
\text{FF} & \quad \text{FF} & \quad \text{FF} & \quad \text{FF}
\end{align*} \]

Input: \( b_3 a_3 b_2 a_2 b_1 a_1 b_0 a_0 \)

Output: \( c_0 \)

States: \( s_3 s_2 s_1 s_0 \)
Pipelining Loops with Feedback

“Loop carry dependency”

- Example 1: \( y_i = y_{i-1} + x_i + a \)

  unpipelined version:

<table>
<thead>
<tr>
<th>add1</th>
<th>( x_i + y_{i-1} )</th>
<th>( x_{i+1} + y_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>add2</td>
<td>( y_i )</td>
<td>( y_{i+1} )</td>
</tr>
</tbody>
</table>

  Can we “cut” the feedback and overlap iterations?

  Try putting a register after add1:

<table>
<thead>
<tr>
<th>add1</th>
<th>( x_i + y_{i-1} )</th>
<th>( x_{i+1} + y_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>add2</td>
<td>( y_i )</td>
<td>( y_{i+1} )</td>
</tr>
</tbody>
</table>

  - Can’t overlap the iterations because of the dependency.
  - The extra register doesn’t help the situation (actually hurts).
  - In general, can’t pipeline feedback loops.
Pipelining Loops with Feedback

“Loop carry dependency”

However, we can overlap the “non-feedback” part of the iterations:

Add is associative and commutitive. Therefore we can reorder the computation to shorten the delay of the feedback path:

\[ y_i = (y_{i-1} + x_i) + a = (a + x_i) + y_{i-1} \]

\begin{array}{|c|c|c|}
\hline
\text{add}_1 & x_i + a & x_{i+1} + a & x_{i+2} + a \\
\hline
\text{add}_2 & y_i & y_{i+1} & y_{i+2} \\
\hline
\end{array}

“Shorten” the feedback path.

- Pipelining is limited to 2 stages.
Pipelining Loops with Feedback

- Example 2:
  \[ y_i = a \cdot y_{i-1} + x_i + b \]

Reorder to shorten the feedback loop and try putting register after multiply:

- Still need 2 cycles/iteration
**Pipelining Loops with Feedback**

- **Example 2:**
  \[ y_i = a \cdot y_{i-1} + x_i + b \]

- Once again, adding register doesn’t help. Best solution is to overlap non-feedback part with feedback part.

- Therefore critical path includes a multiply in series with add.

- Can overlap first add with multiply/add operation.

- Only 1 cycle/iteration. Higher performance solution.

<table>
<thead>
<tr>
<th>add(_1)</th>
<th>(x_i+b)</th>
<th>(x_{i+1}+b)</th>
<th>(x_{i+2}+b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult</td>
<td>(a\cdot y_{i-1})</td>
<td>(a\cdot y_i)</td>
<td>(a\cdot y_{i+1})</td>
</tr>
<tr>
<td>add(_2)</td>
<td>(y_i)</td>
<td>(y_{i+1})</td>
<td>(y_{i+2})</td>
</tr>
</tbody>
</table>

- Alternative is to move flip-flop to after multiple, but same critical path.
“C-slow” Technique

- Another approach to increasing throughput in the presence of feedback: try to fill in “holes” in the chart with another (independent) computation:

<table>
<thead>
<tr>
<th></th>
<th>( x_i + b )</th>
<th>( x_{i+1} + b )</th>
<th>( x_{i+2} + b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{add}_1 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{mult} )</td>
<td>( ay_{i-1} )</td>
<td>( ay_i )</td>
<td>( ay_{i+1} )</td>
</tr>
<tr>
<td>( \text{add}_2 )</td>
<td></td>
<td>( y_i )</td>
<td>( y_{i+1} )</td>
</tr>
</tbody>
</table>

If we have a second similar computation, can interleave it with the first:

\[
x^1 \rightarrow F^1 \rightarrow y^1 = a^1 y^1_{i-1} + x^1_i + b^1
\]

\[
x^2 \rightarrow F^2 \rightarrow y^2 = a^2 y^2_{i-1} + x^2_i + b^2
\]

- Here the feedback depth=2 cycles (we say \( C=2 \)).
- Each loop has throughput of \( F/C \). But the aggregate throughput is \( F \).
- With this technique we could pipeline even deeper, assuming we could supply \( C \) independent streams.

Use muxes to direct each stream. Each produces 1 result / 2 cycles.
“C-slow” Technique

- Essentially this means we go ahead and cut feedback path:

- This makes operations in adjacent pipeline stages independent and allows full cycle for each:

- C computations (in this case C=2) can use the pipeline simultaneously.
- Must be independent.
- Input MUX interleaves input streams.
- Each stream runs at have the pipeline frequency.
- Pipeline achieves full throughput.

<table>
<thead>
<tr>
<th>add₁</th>
<th>x+b</th>
<th>x+b</th>
<th>x+b</th>
<th>x+b</th>
<th>x+b</th>
<th>x+b</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult</td>
<td>ay</td>
<td>ay</td>
<td>ay</td>
<td>ay</td>
<td>ay</td>
<td>ay</td>
</tr>
<tr>
<td>add₂</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
</tbody>
</table>
Beyond Pipelining - SIMD Parallelism

- An obvious way to exploit more parallelism from loops is to make multiple instances of the loop execution data-path and run them in parallel sharing the same controller.
- For P instances, throughput improves by a factor of P.
- Example: \( y_i = f(x_i) \)
- Assumes the next 4 \( x \) values available at once. The validity of this assumption depends on the ratio of \( f \) repeat rate to input rate (or memory bandwidth).
- Cost \( \alpha P \). Usually, much higher than for pipelining. However, potentially provides a high speedup. Often applied after pipelining.
- Limited, once again, by loop carry dependencies. Feedback translates to dependencies between parallel data-paths.

Usually called SIMD parallelism. Single Instruction Multiple Data.
SIMD Parallelism with Feedback

- Example, from earlier:
  \[ y_i = a \cdot y_{i-1} + x_i + b \]

- In this example end up with “carry ripple” situation.
- Could employ look-ahead / parallel-prefix optimization techniques to speed up propagation.
- As with pipelining, this technique is most effective in the absence of a loop carry dependence.