Compiling LLVM to Custom Hardware (via Bluespec)

Ali Mustafa Zaidi

University of Cambridge
Computer Laboratory
The Dark Silicon Problem

Amdahl's Law

\[ S \text{ speedup} = \frac{1}{(1-f)S + \frac{f}{N*S}} \]

Utilization Wall

= Dark Silicon

45nm → 8nm (32x resources)
- CPU: 3.5x, GPU 2.4x (Conservative)
- CPU: 7.9x, GPU 2.7x (ITRS Scaling)

Solution: Custom Computing?

Energy Efficiency improvements over CPU:

- Fixed Function (ASIC) Hardware: 100-1000x
- Field Programmable Gate Arrays: 10-100x
  - 100-1000x for domain-specific FPGAs
Major Issues with Custom Computing

- Poor Productivity & Application Design Complexity
  - Specialized languages, or restricted subsets of existing languages
  - Requires H/W design expertise (timing closure, I/O scheduling)
  - Prohibitive Compilation & verification times
  - Lack of portability/standardization

- Limited Amenability
  - Not all applications see performance benefit from Custom Computing

Resolving these issues would mean:
1. Easier to develop Energy Efficient SOCs
2. Better Utilization of Dark Silicon via Heterogeneity
3. Increased utilization of FPGAs as co-processor accelerators
Need to resolve issues with Custom Computing

- **Poor Productivity & Application Design Complexity**
  - Specialized languages, or restricted subsets of existing languages
  - Requires H/W design expertise (timing closure, I/O scheduling)
  - Prohibitive Compilation & verification times
  - Lack of portability/standardization

  > Use LLVM as Input Language

  > Switch to 'Dynamic Scheduling' execution model

  > New Dataflow Compiler IR (that maps directly to hardware)

- **Limited Amenability**
  - Not all applications see performance benefit from Custom Computing
#1: 'Dynamic Scheduling' Execution Model

- Supports conventional memory hierarchy
  - Full Support for C language features
- Hardware more decentralized
- Compilation flow simplified: No Scheduling, Easier P&R(?), Easier Timing Closure (?)

#2: Hardware Oriented Dataflow IR

\[ \%1 = \text{mul} \ i32 \ \%x, \ \%y \quad ; \quad <i32> \]

\[ \%2 = \text{srem} \ i32 \ \%1, \ \%z \quad ; \quad <i32> \]

\[ \%3 = \text{icmp} \ \text{slt} \ i32 \ \%2, \ \%1 \quad ; \quad <i1> \]

**LLVM IR**

---

**Dataflow IR with Petri Net based Semantics**

**Equivalent Hardware Datapath**
%1 = mul i32 %x, %y ; <i32>
%2 = srem i32 %1, %z ; <i32>
%3 = icmp slt i32 %2, %1 ; <i1>

LLVM IR

Equivalent Bluespec Code

FIFOF(int) x ← mkFIFO;
FIFOF(int) y ← mkFIFO;
FIFOF(int) z ← mkFIFO;
FIFOF(int) srem_1 ← mkFIFO;
FIFOF(int) icmp_1 ← mkFIFO;
FIFOF(int) icmp_2 ← mkFIFO;

rule mul_inst;
let val1 = x.first; x.deq;
let val2 = y.first; y.deq;
let rslt = val1 * val2;
srem_1.enq (rslt);
icmp_1.enq (rslt);
endrule

rule srem_inst;
let val1 = srem_1.first; srem_1.deq;
let val2 = z.first; z.deq;
let rslt = val1 * val2;
icmp_2.enq (rslt);
endrule
Putting it All Together

Problem:

Dark Silicon + Productivity + Amenability

Solution:

Application Specific Hardware + Dynamic Execution Scheduling + Hardware Oriented Dataflow IR

High Performance, High Energy Efficiency Hardware Implementation of legacy software & sequential programming models!

Implementation:

Any High Level Language → LLVM → DF IR → Bluespec SystemVerilog → ASIC / FPGA

Coarse-Grained, Reconfigurable Architecture (Asynchronous)
internal_int_transpose()
- From 'epic' Mediabench benchmark
- 2 level nested loop
- Compared with:
  - In-Order MIPS-64
  - CDFG SC from CMU,
  - 4-way OOO Superscalar

\[ \text{Power } P = f(R, A) \]
\[ R = \text{Resource Utilization} \]
\[ A = \text{Activity Ratios} \]

Assumed: \( A_{\text{SpatialComp}} = A_{\text{MIPS64}} \)

Reality: \( A_{\text{SpatialComp}} \ll A_{\text{MIPS64}} \)

\[ \text{Energy } E = P \times T \]
\[ T = \text{Time} = \frac{\text{Cycles}}{\text{Iteration}} \times \text{Num Iterations} \times \left( \frac{1}{F} \right) \]
Thank you
Additional Slides
Energy Efficiency of Custom Computing

Why is custom computing energy efficient?

\[
EPI_{CPU} = \frac{E_{I-Cache\_Warmup}}{\text{InstructionCount}_{Static}} + \left( \frac{E_{Execute} + E_{CPU\_Overheads}}{\text{InstructionCount}_{Dynamic}} \right)
\]

\[
E_{CPU\_Overheads} = E_{I-Cache} + E_{Branch\_Pred} + E_{Decode} + E_{Reg\_Rename} + E_{Scheduling} + E_{Forwarding} + E_{Reg\_File} + E_{ROB}
\]
Energy Efficiency of Custom Computing

Why is custom computing energy efficient?

\[ EPI_{CPU} = \frac{E_{I-Cache\text{Warmup}}}{\text{InstructionCount}_{\text{Static}}} + \left( \frac{E_{\text{Execute}} + E_{\text{CPUOverheads}}}{\text{InstructionCount}_{\text{Dynamic}}} \right) \]

\[ E_{\text{CPUOverheads}} = E_{I-Cache} + E_{\text{BranchPred}} + E_{\text{Decode}} + E_{\text{RegRename}} + E_{\text{Scheduling}} + E_{\text{Forwarding}} + E_{\text{RegFile}} + E_{\text{ROB}} \]
Why is custom computing energy efficient?

\[
EPI_{CPU} = \frac{E_{I - \text{CacheWarmup}}}{\text{InstructionCount}_{\text{Static}}} + \left( \frac{E_{\text{Execute}} + E_{\text{CPUOverheads}}}{\text{InstructionCount}_{\text{Dynamic}}} \right)
\]

\[
E_{\text{CPUOverheads}} = E_{I - \text{Cache}} + E_{\text{BranchPred}} + E_{\text{Decode}} + E_{\text{RegRename}} + E_{\text{Scheduling}} + E_{\text{Forwarding}} + E_{\text{RegFile}} + E_{\text{ROB}}
\]

\[
EPI_{FPGA} = \frac{E_{\text{Reconfiguration}}}{\text{InstructionCount}_{\text{Static}}} + \left( \frac{E_{\text{Execute}} + E_{\text{Multiplexing}} + E_{\text{Control}} + E_{\text{RC - Overhead}}}{\text{InstructionCount}_{\text{Dynamic}}} \right)
\]

\[
E_{I - \text{CacheWarmup}} \leq E_{\text{Reconfiguration}}
\]

\[
E_{\text{CPUOverheads}} \gg E_{\text{Multiplexing}} + E_{\text{Control}} + E_{\text{RC - Overhead}}
\]
Energy Efficiency of Custom Computing

Why is custom computing energy efficient?

\[
EPI_{CPU} = \frac{E_{CacheWarmup}}{InstructionCount_{Static}} + \left( \frac{E_{Execute} + E_{CPUOverheads}}{InstructionCount_{Dynamic}} \right)
\]

\[
E_{CPUOverheads} = E_{Cache} + E_{BranchPred} + E_{Decode} + E_{RegRename} + E_{Scheduling} + E_{Forwarding} + E_{RegFile} + E_{ROB}
\]

\[
EPI_{FPGA} = \frac{E_{Reconfiguration}}{InstructionCount_{Static}} + \left( \frac{E_{Execute} + E_{Multiplexing} + E_{Control} + E_{RCOverhead}}{InstructionCount_{Dynamic}} \right)
\]

\[
EPI_{ASIC} = \frac{(E_{Execute} + E_{Multiplexing} + E_{Control})}{InstructionCount_{Dynamic}}
\]

\[
E_{CacheWarmup} \leq E_{Reconfiguration}
\]

\[
E_{CPUOverheads} \gg E_{Multiplexing} + E_{Control} + E_{RCOverhead}
\]
Dynamic Scheduling?

<table>
<thead>
<tr>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPSE</td>
<td>DPSE</td>
</tr>
<tr>
<td>VLIW</td>
<td>EPIC</td>
</tr>
</tbody>
</table>

Placement

Execution Scheduling

- Custom Computation
- Spatial Computation
- Superscalar
- In Order
- Wavescalar
Dynamic Scheduling

Execution Scheduling

- DPSE
- DPDE
- EPIC
- VLIW
- Custom Computation
- Wavescalar
- Superscalar
- In Order
- Spatial Computation

Placement

Increasing Compiler Complexity

Increasing Hardware Complexity
Why Dynamic Scheduling?

Increasing Compiler Complexity

Placement

Dynamic

Static

Execution Scheduling

Static

Dynamic

DPSE

EPIC

VLIW

SPSE

Custom Computation

Wavescalar

Superscalar

In Order

Spatial Computation

But also increasing performance on irregular code!

Increasing Hardware Complexity
internal_int_transpose()

- kernel from 'epic' benchmark (Mediabench)

- Nested loop with low ILP
  - Poor performance on Spatial Computation by Budiu et al.

- Hand-converted:
  - LLVM-IR → VSDG → 'VSFG' → Bluespec SystemVerilog
A Little more Perspective on ILP

Multiple Flows of Control

```c
for (i = 0; i < 100; i++)
    if (A[i] > 0) foo();
bar();
```
A Little more Perspective on ILP

Multiple Flows of Control

```plaintext
for (i = 0; i < 100; i++)
    if (A[i] > 0) foo();
bar();
```

Conventional Superscalar

- Single Flow of Control only
- 7-12 ILP Max
  - With Best case BP, MD, RR

A Little more Perspective on ILP

Multiple Flows of Control

```c
for (i = 0; i < 100; i++)
    if (A[i] > 0) foo();
    bar();
```

Lam, Wilson (1993)

- Control-flow primary limitation on ILP
- "Multiple Flows of Control"
  - 5-20x ILP, even on 'sequential' code
A Little more Perspective on ILP

Mak, Mycroft (2009)

- 10x ILP by excluding Control dependencies
  - e.g. via Speculation & Multiple Flows of Control
- Additional 10x ILP with perfect Memory Disambiguation!
#2: Fixing Amenability with Dataflow IR

Control-Data Flow Graph

- Control flow is the primary constraint on ILP

- Conventional processors use aggressive branch prediction
  - 95+% accuracy
  - Single flow of control

- Custom hardware has very limited speculation
  - No Branch prediction
  - Single flow of control

```c
for (i = 0; i < 100; i++)
    if (A[i] > 0) foo();
    bar();
```
#2: Fixing Amenability with Dataflow IR

- Data-dependences are explicit
- Control-flow is implicit
- Speculation explicit in IR
- 'Multiple Flows of Control'

```c
for (i = 0; i < 100; i++)
    if (A[i] > 0) foo();
bar();
```
Why the VSDG instead of the CDFG?

CDFG

```plaintext
for (i = 0; i < 100; i++)
    if (A[i] > 0) foo();
    bar();
```

VSDG

```
A = A[i]
> 0
STATE_IN

= A[i]
> 0
foo()

FPST

STATE_OUT
```

Start → End
Why the VSDG instead of the CDFG?

CDFG

for (i = 0; i < 100; i++)
    if (A[i] > 0) foo();
    bar();

VSDG

Next iteration of 'for' loop
Why the VSDG instead of the CDFG?

CDFG:

```c
for (i = 0; i < 100; i++)
    if (A[i] > 0) foo();
    bar();
```

VSDG:

```
i = 0
A
STATE_IN

= A[i]
>
foo()

i++
< 100
F  P  T

Next iteration of 'for' loop

STATE_OUT

bar()
```

Next iteration of 'for' loop
Why the VSDG instead of the CDFG?

**CDFG**

```
for (i = 0; i < 100; i++)
    if (A[i] > 0) foo();
    bar();
```

**VSDG**

```
i = 0
STATE_IN

= A[i] > 0
foo() foo()

i++
< 100
Next iteration of 'for' loop
FFFF

bar() STATE_OUT

```