Spatial Computation

Mihai Budiu
CMU CS

Thesis committee:
Seth Goldstein
Peter Lee
Todd Mowry
Babak Falsafi
Nevin Heintze

Ph.D. Thesis defense, December 8, 2003

sky image courtesy of NASA
A model of general-purpose computation based on Application-Specific Hardware.

Definition of Spatial Computation.
Thesis Statement

Application-Specific Hardware (ASH):

• can be synthesized by adapting software compilation for predicated architectures,

• provides high-performance for programs with high ILP, with very low power consumption,

• is a more scalable and efficient computation substrate than monolithic processors.
Outline

- Introduction
- Compiling for ASH
- Media processing on ASH
- ASH vs. superscalar processors
- Conclusions

Handshake image from www.247mail.com/why.html
CPU Problems

• Complexity
• Power
• Global Signals
• Limited ILP

This research addresses some limitations of the (superscalar) microprocessor structure.
Complexity = of design, verification, manufacturing.
Design complexity outpaces productivity increase. A slide from Michael Flynn’s FCRC plenary presentation.
Communication vs. Computation

Gates are cheap and wires expensive at very high clock speeds.
Our Approach: ASH

Application-Specific Hardware

Volcanic ash Image from
http://www.geology.sdsu.edu/how_volcanoes_work/Tephra.html
Resource Binding Time

We take advantage of the fact that we create hardware *after* the program is known.
We escape the “tyranny of the ISA”. This makes compilation more difficult.
The dataflow machine generated is very close semantically to the internal compiler representation.
This research makes most contributions in the area of optimizing compilation, but brings new insights into other areas as well.
Outline

• Introduction
• CASH: Compiling for ASH
• Media processing on ASH
• ASH vs. superscalar processors
• Conclusions
We do not have the disadvantages of the traditional dataflow machine model, because we implement the machines directly in hardware.
Both the internal representation and the synthesized circuit are built from such constructs.
Asynchronous computation is triggered by data availability; it is completely dynamically scheduled.
There is no centralized FSM controlling the behavior of the circuit; each producer-consumer pair has some control logic for handshaking, and that is all.

(There is however also logic for memory access, which features some global structures and arbitration.)
Forward Branches

if (x > 0)
y = -x;
else
  y = b*x;

Conditionals $\Rightarrow$ Speculation

Conditionals are compiled into circuits executing simultaneously all branches.
Control Flow $\Rightarrow$ Data Flow

<table>
<thead>
<tr>
<th>Control Flow</th>
<th>Data Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge (label)</td>
<td>Gateway</td>
</tr>
<tr>
<td>Split (branch)</td>
<td></td>
</tr>
</tbody>
</table>

Note: in reality the merge operator is slightly more complicated than described here, since multiple merge operators sometimes need to cooperate in order to correctly execute the program.

The merge is also called “mu”, while the gateway is also called “eta”.
Loops can be built using merges and gateways as building blocks. Some gateways direct the data to the beginning of the loop while other direct the data out of the loop (red). They are controlled by complementary predicates.
Side-effect operations are predicated (never speculated).
The lack of a PC, i.e., program order, mandates the use of another mechanism to preserve correctness; this is the flow of tokens.
Tokens are both a compile-time and run-time construct. This is one instance where we do things an ISA would not allow.
Thesis Statement

Application-Specific Hardware:

• can be synthesized by adapting software compilation for predicated architectures,
Outline

- Introduction
- CASH: Compiling for ASH
  - An optimization on the SIDE
- Media processing on ASH
- ASH vs. superscalar processors
- Conclusions

Salad image from www.amgmedia.com/freephotos/salad.jpg
Replace expressions already computed with scalar values.
Dataflow Analysis Is Conservative

Conservativeness must assume that the expression is not available.
Static Instantiation, Dynamic Evaluation

```java
flag = false;
if (x) {
    ...
    y = a*b;
    flag = true;
}
...
... = flag ? y : a*b;
```

Solution: maintain dynamically the availability state of an expression; predicate its re-computation.
We applied SIDE to memory access removal through register promotion; it can be very effective.
Outline

• Introduction
• CASH: Compiling for ASH
• Media processing on ASH

• ASH vs. superscalar processors
• Conclusions

Photo taken and processed by me.
Assumption: all operations have the same latency.

We compare ASH and a 4-way superscalar by connecting them to the same memory hierarchy.

We also assume (for most of the evaluations) that the operation latencies are identical.

Ideally we’d like to compare ASH and a VLIW, but we didn’t have access handy to a VLIW simulator.
This data is just for the kernels.

Kernels were selected manually to be the hottest 1, 2, or 3 functions in each application.

See the thesis for details on these kernels.

Only one kernel exhibits a slowdown.
ASH can sustain very high parallelism for these kernels.
This graph compares speed-up and parallelism increase; i.e., how much parallelism we waste for getting the speed-up.

Note that the parallelism includes speculative execution.

epic_e is a very bad case: lots of useless speculation.

rasta is a great case; it results from the register promotion generating much better code than for the processor.
We compare the truly asynchronous circuits generated with a 600Mhz 4-way OOO superscalar based on SimpleScalar + Wattch.

This is preliminary data; this back-end is still under development.

The Verilog back-end was written entirely by Girish Venkataramani.
These values are for slightly different kernels: for each program exactly one function was selected.

This data includes aggressive loop unrolling.
These values are not percents!
Bigger is better.
Thesis Statement

Application-Specific Hardware:

• provides high-performance for programs with high ILP, with very low power consumption,
Outline

- Introduction
- CASH: Compiling for ASH
- Media processing on ASH
  - dataflow pipelining
- ASH vs. superscalar processors
- Conclusions

What is the source of parallelism in ASH?
Pipeline image from www.bfi.org/Trimtab/winter02/ brittleTimes.htm
Let us look how the sum-of-squares program behaves at run-time.

```c
int sum=0, i;
for (i=0; i < 100; i++)
    sum += i*i;
return sum;
```
Pipelining

cycle=2
Pipelining

\[
\text{cycle} = 3
\]
Pipelining

cycle=4
Pipelining occurs naturally, due to the asynchronous nature of the computation.

This phenomenon was exploited by the dataflow machines.
Now we perform a limit-study to see the potential of ASH on whole-programs.
Karate fighters image from http://www.enighet.se/karate.htm
This Is Obvious!

This is true at full dataflow speed, so CPU cannot do any better (if compilers equally good).

There was already a hint in the epic_e kernel which was doing worse than a 4-way OOO.
For most of SpecInt95 ASH underperforms OOO.
This data is again with the high-level simulation model in which everything is assumed equal.
(Successful) branch prediction in processor renders some computations irrelevant.

Since we transformed control-flow into dataflow, the gateway predicate is on the critical path for ASH, but not for processor!

In the thesis we suggest a way to add a limited form of branch-prediction to ASH without destroying its distributed structure.

The hard part is not prediction, but stopping runaway speculation which crosses hyperblock boundaries.

```
for (i=0; i < N; i++) {
    ...

    if (exception) break;
}
```

branch prediction diagram
Unfortunately even branch prediction does not redeem ASH.
ASH Problems

- Both branch and join not free
- Static dataflow
  - *(no re-issue of same instr)*
- Memory is “far”
- Fully static
  - No branch prediction
  - No dynamic unrolling
  - No register renaming
- Calls/returns not lenient
- ...
Thesis Statement

Application-Specific Hardware:

• is a more scalable and efficient computation substrate than monolithic processors.
Outline

Introduction
+ CASH: Compiling for ASH
+ Media processing on ASH
+ ASH vs. superscalar processors

= Conclusions
<table>
<thead>
<tr>
<th>Strengths</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>low power</td>
<td>economies of scale</td>
</tr>
<tr>
<td>simple verification?</td>
<td>highly optimized</td>
</tr>
<tr>
<td>specialized to app.</td>
<td>branch prediction</td>
</tr>
<tr>
<td>unlimited ILP</td>
<td>control speculation</td>
</tr>
<tr>
<td>simple hardware</td>
<td>full-dataflow</td>
</tr>
<tr>
<td>no fixed window</td>
<td>global signals/decision</td>
</tr>
</tbody>
</table>
Conclusions

• Compiling “around the ISA” is a fruitful research approach.

• Distributed computation structures require more synchronization overhead.

• Spatial Computation efficiently implements high-ILP computation with very low power.
Backup Slides

- Control logic
- Pipeline balancing
- Lenient execution
- Dynamic Critical Path
- Memory PRE
- Critical path analysis
- CPU + ASH
Typical micropipelines.
The rdy and ack signals may have fan-out/fan-in.
Last-Arrival Events

- Event enabling the generation of a result
- May be an ack
- Critical path=collection of last-arrival edges
Dynamic Critical Path

1. Start from last node
2. Trace back along last-arrival edges
3. Some edges may repeat

See Fields & Bodik’s seminal ISCA 2001 paper.
Speculation introduces the problem of long critical paths. The solution is on the next slide.
Lenient Operations

if (x > 0)
    y = -x;
else
    y = b*x;

Solve the problem of unbalanced paths

A lenient operation can generate the result before all of its inputs are known. Leniency reduces the dynamic critical path.
Pipelining may suffer from stalls due to unbalanced lengths of paths. The thesis presents a general solution called “pipeline balancing”.
The i loop is coupled to the sum loop through the predicate. The acknowledgment for the predicate prevents i’s loop from making progress.
The critical path of this circuit actually goes through an acknowledgment edge.
By inserting FIFO stages on the predicate path we can decouple the two loops.
The critical path is minimal for the resulting circuit, implying maximal performance.
Register Promotion

Load is executed only if store is not

The load predicate is weakened.
As a particular case of the previous optimization, if whenever the load predicate is true the store predicate is as well, the load can be completely removed. It is removed by the dead-code optimization, since its predicate becomes false. This optimization can be generalized for the case of multiple stores preceding the load.
This corresponds in the CFG to lifting the load to a basic block dominating the original loads.

The same optimization can be applied to loads as well.

The code is basically the same as for common-subexpression elimination in an SSA form.

(Technicality: to enable this transformation no predicate must depend on the other operation. Easily checked with a reachability computation.)
Store-store (1)

- When \( p_1 \Rightarrow p_2 \) the first store becomes dead...
- ...i.e., when second store post-dominales first in CFG

In this case the predicate of the first store is weakened, so that it is never executed when the second one will overwrite it.
Since the two predicates are now disjoint, the two stores can never occur simultaneously, so there is no need for a token edge between them. Notice that the removal of the token edge is not trivial.
A Code Fragment

```c
for(i = 0; i < 64; i++)
    {
        for (j = 0; X[j].r != 0xF; j++)
            if (X[j].r == i)
                break;

        Y[i] = X[j].q;
    }
```

SpecINT95:124.m88ksim:init_processor, stylized

A sample analysis of bottlenecks in ASH.
Critical path contains control-flow (i.e., eta) predicate.
MIPS gcc Code

LOOP:
L1: beq $v0,$a1,EXIT  ; X[j].r == i
L2: addiu $v1,$v1,20  ; &X[j+1].r
L3: lw $v0,0($v1)    ; X[j+1].r
L4: addiu $a0,$a0,1  ; j++
L5: bne $v0,$a3,LOOP  ; X[j+1].r == 0xF

EXIT:

for (j = 0; X[j].r != 0xF; j++)
  if (X[j].r == i)
    break;

L1 L2 L3 L5 L1
4-instructions loop-carried dependence

Processor critical path also spans two branches.
If Branch Prediction Correct

LOOP:
L1: beq $v0,$a1,EXIT ; X[j].r == i
L2: addiu $v1,$v1,20 ; &X[j+1].r
L3: lw $v0,0($v1) ; X[j+1].r
L4: addiu $a0,$a0,1 ; j++
L5: bne $v0,$a3,LOOP ; X[j+1].r == 0xF
EXIT:

for (j = 0; X[j].r != 0xF; j++)
    if (X[j].r == i)
        break;

L1=L2=L3=L5>L1

Superscalar is issue-limited!
2 cycles/iteration sustained

But not if branch prediction guesses their outcome. If it does, the critical path
is 1 instruction long!
When predicting the load-controlling predicate is on the critical path.
Prediction + Load Speculation

~4 cycles!
Load not pipelined
(self-anti-dependence)

\[
\text{\begin{verbatim}
for (j = 0; X[i].r != 0xF; j++)
    if (X[i].r == i)
        break;
\end{verbatim}}
\]

When we allow loads to issue speculatively (as a processor also does), the critical path contains an ack edge, because the latency of the load is 3 cycles. There is a dependence between the output register of the load in different iterations.
A processor however can have multiple instances of the load simultaneously active, since each uses a different physical register, due to register renaming.
Unrolling?

```c
for(i = 0; i < 64; i++) {
    for (j = 0; X[j].r != 0xF; j+=2) {
        if (X[j].r == i)
            break;
        if (X[j+1].r == 0xF)
            break;
        if (X[j+1].r == i)
            break;
    }
    Y[i] = X[j].q;
}
```

Unrolling does not help when the number of iterations may be small.
Overhead projector image from staples.com
CPUs are very convenient; they will probably be around forever, but they will not evolve as fast as in the past.

We can envision delegating to them “fringe” jobs, while we run the core computation on the ASH fabrics.