Idempotent Processor Architecture

Marc de Kruijf
Karthikeyan Sankaralingam
Vertical Research Group
UW-Madison

MICRO 2011, Porto Alegre
Idempotent Processor Architecture

idempotent regions

idempotence: “live state”
re-execution has no side-effects (inputs are preserved)

region entry point:
implicit checkpoint in the live state of the program

applications naturally decompose into idempotent regions
Idempotent Processor Architecture

Idempotent recovery

**CHECKPOINT**

Conventional recovery

Idempotent recovery

Recovery without checkpoints – just re-execute
Idempotent Processor Architecture

Idempotent processors

conventional processor

Fetch | Decode | RF | Issue | Exec | WB

idempotent processor

Fetch | Decode | RF | Issue | Exec | WB

simpler decode, issue, execute, and writeback
Presentation Overview

1. Idempotent Recovery

2. Idempotent Processors

3. Evaluation
Idempotent Recovery
example

1. \( R2 = \text{add } R0, R1 \)
2. \( R3 = \text{ld } [R2 + 4] \)
3. \( R2 = \text{add } R2, R4 \)
4. \( \text{beqz } R2, \text{NEXT} \)

“Something went wrong executing instruction 2!”
(page fault, corrupted write, etc.)

“No problem – just re-execute from instruction 1!”
Idempotent Recovery
idempotent regions

idempotent “regions”: freely re-executable program regions

sizes inhibited by *clobber* antidependences (WAR after no RAW)

normal compiler: 

custom compiler:

adds some runtime overhead (typically 2-10%)
Idempotent Recovery
average idempotent region sizes

- frequent clobber antidependences – can be removed, but requires large restructuring effort
- limited aliasing information
  - with func params marked using C/C++ “restrict”

![Graph showing ARM micro-ops for different benchmark suites and select benchmarks.]

- SPEC INT
- SPEC FP
- PARSEC
- Parboil
- 401.bzip2
- 456.hmmer
- OVERALL

- benchmark suites (geo-mean)
- select benchmarks (geo-mean)

- custom compiler
Presentation Overview

1. Idempotent Recovery

2. Idempotent Processors

3. Evaluation
Idempotent Processors
exploring the opportunity

Exceptions & out-of-order retirement

branch misprediction

hardware faults

Vdd

In

Out

branch

issue execute retire

in-order out-of-order multi-core
Idempotent Processors
steps one, two, and three

Step 1: construct a high-performance in-order processor

ARM Cortex-A8 (‘05)    IBM Cell SPE (‘05)    Intel Atom (‘08)

Step 2: prune out unnecessary parts

↓ power, area, & complexity

Step 3: optimize for energy efficiency

↑ performance at low cost
Step 1: Construction

Fetch

Decode & Issue

Add

Ld

Integer

Integer

Branch

Multiply

Exception!

FP

RF
Step 1: Construction
v1.0

Decoded, Rename, & Issue

Fetch

Bypass

Integer

Integer

Branch

Multiply

Load/Store

FP

RF

Flush?

Staged instruction completion
Step 1: Construction

\textit{v1.0 v1.1}

FP exceptions handled in hardware.
Separate FP unit implements full IEEE 754.

Fetch → Decode, Rename, & Issue → Bypass → Integer → Integer → Branch → Multiply → Load/Store → RF → Flush?
Step 1: Construction
\(v1.0\ v1.1\ v1.2\)

Load miss?
Have to flush!

Fetch → Decode, Rename, & Issue → Bypass → Integer → Integer → Branch → Multiply → Load/Store → RF

Replay queue

Flush?
Flush?
Replay?
Step 1: Construction

v1.0 v1.1 v1.2 v1.3

TOO COMPLICATED!
Step 2: Simplification
idempotent edition (simple)

WHAT IS GONE?
• staging register file (6-entries)
• replay queue (8-entries)
• entire rename pipeline stage
• IEEE-compliant floating point unit
• pipeline flush for exceptions and replays
• all associated control logic
Step 3: Optimization
idempotent edition (fast)

Fetch → Decode & Issue

Integer
Integer
Branch
Multiply
Load/Store
FP

*SDB* (Slice Data Buffer)

Continual Flow Pipelines. ASPLOS ‘04

details in paper...

* Slice Data Buffer (SDB) – *Continual Flow Pipelines.* ASPLOS ‘04
Presentation Overview

1. Idempotent Recovery

2. Idempotent Processors

3. Evaluation
Evaluation
idempotent processor performance

![Chart showing performance speed-up over in-order for different benchmarks and benchmark suites.]

- Simple Idem
- Fast Idem
- OoO

**Benchmark suites (geo-mean)**

**Select benchmarks (geo-mean)**
## Evaluation summary

<table>
<thead>
<tr>
<th>Processor Type</th>
<th>Performance (vs. In-Order)</th>
<th>Power/Complexity (vs. In-Order)</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple idempotent</td>
<td>worse by ~5% (compilation &amp; serialization overheads)</td>
<td>better</td>
</tr>
<tr>
<td>fast idempotent</td>
<td>better by ~5% (modest OoO execution)</td>
<td>same or better</td>
</tr>
<tr>
<td>out-of-order</td>
<td>better by ~30%</td>
<td>worse</td>
</tr>
</tbody>
</table>
Presentation Overview

1. Idempotent Recovery

2. Idempotent Processors

3. Evaluation
Future Work

– quantify **power/complexity** benefits
  (build real hardware prototype)

– more general **error conditions**
  (hardware faults, branch prediction, etc.)

– impact on **multithreading/multiprocessors**
  (re-execution currently assumes no interference)

– region **overlapping** (“region pipelining”)
  (analagous to overlapping checkpoints)
Conclusions

recovery using idempotence
– recovery without checkpoints

multiple uses and multiple designs
– uses: exception, speculation, fault recovery, and more
– designs: in-order, out-of-order, multi-core, GPU, and more

in this work: exception recovery + in-order design
– simplified out-of-order execution
– better performance at equal or lower power/complexity
Back-up Slides
Optimal Idempotent Region Size?

(rough sketch – graph not to scale)
Optimal Processor Design?

Compiler overheads dominate? ~ 250mW

Re-execution overheads dominate? ~ 2.5W

---

Single-issue in-order

Dual-issue OoO

Best potential?

Dual-issue in-order

Quad-issue OoO

~ 2.5W
Out-of-Order Issue Processors?

Some additional challenges:

- Re-execution overhead high if mis-speculation frequent
  - cannot restart from point of mis-speculation, and hence...
  - re-execution overhead on average ≈ half the region

Example: branch misprediction

- With in-order issue, simple to flush/drain pipeline
- With out-of-order issue, we can use idempotence but...
  - 5 branches/region @ 90% confidence ≈ 41% re-execution rate

1. Speculate only high-confidence branches
2. Hybrid checkpointing/idempotence
3. ...?