Compiler Construction of Idempotent Regions and Applications in Architecture Design

Marc de Kruijf

Advisor: Karthikeyan Sankaralingam

PhD Defense 07/20/2012
Example
source code

```c
int sum(int *array, int len) {
    int x = 0;
    for (int i = 0; i < len; ++i)
        x += array[i];
    return x;
}
```
Example
assembly code

R2 = load [R1]
R3 = 0
LOOP:
R4 = load [R0 + R2]
R3 = add R3, R4
R2 = sub R2, 1
bnez R2, LOOP
EXIT:
return R3
Example
assembly code

R2 = load [R1]
R3 = 0

LOOP:
R2 = sub R2, 1
bnez R2, LOOP

EXIT:
return R3
Example
assembly code

R2 = load [R1]
R3 = 0

LOOP:
R4 = load [R0 + R2]
R3 = add R3, R4
R2 = sub R2, 1
bnez R2, LOOP

EXIT:
return R3

R0 and R1 are unmodified

Example assembly code just re-execute!

convention:
use checkpoints/buffers
It’s Idempotent!

idempoh... what...?

```c
int sum(int *data, int len) {
    int x = 0;
    for (int i = 0; i < len; ++i)
        x += data[i];
    return x;
}
```
Thesis
the thing that I am defending

idempotent regions
All The Time

specifically...
– using compiler analysis (intra-procedural)
  – transparent; no programmer intervention
– hardware/software co-design
  – software analysis ➔ hardware execution
Thesis
prelim.pptx ➔ defense.pptx

preliminary exam (11/2010)
– idempotence: concept and simple empirical analysis
– compiler: preliminary design & partial implementation
– architecture: some area and power savings...?

defense (07/2012)
– idempotence: formalization and detailed empirical analysis
– compiler: complete design, source code release*
– architecture: compelling benefits (various)

* http://research.cs.wisc.edu/vertical/iCompiler
Contributions & Findings
a summary

contribution areas
– idempotence: models and analysis framework
– compiler: design, implementation, and evaluation
– architecture: design and evaluation

findings
– potentially large idempotent regions exist in applications
– for compilation, larger is better
  – small regions (5-15 instructions), 10-15% overheads
  – large regions (50+ instructions), 0-2% overheads
– enables efficient exception and hardware fault recovery
Overview

1. Idempotence Models in Architecture

2. Compiler Design & Evaluation

3. Architecture Design & Evaluation
Idempotence Models
idempotence: what does it mean?

**DEFINITION**

1. A region is *idempotent* iff its re-execution has no side-effects.
2. A region is *idempotent* iff it preserves its inputs.

*OK, but what does it mean to preserve an input?*

four models (next slides): A, B, C, & D

**MODEL A**

An input is a variable that is *live-in* to the region. A region preserves an input if the input is *not overwritten*.
Idempotence Model A
a starting point

\[
\begin{align*}
R1 &= R2 + R3 \\
\text{if } R1 &> 0 \\
\text{true} &
\end{align*}
\]

\[
\begin{align*}
\text{false} &
\end{align*}
\]

\[
\begin{align*}
\text{mem}[R4] &= R1 \\
\text{STOP} &
\end{align*}
\]

\[
\begin{align*}
SP &= SP - 16 \\
\text{STOP} &
\end{align*}
\]

\[
\begin{align*}
\vdots &
\end{align*}
\]

\[
\begin{align*}
?? &= \text{mem}[R4] \\
? &
\end{align*}
\]

Live-ins:
\{all registers\} \setminus \{R1\}
\{all memory\}
Idempotence Model A

a starting point

R1 = R2 + R3
if R1 > 0

false

true

mem[R4] = R1

SP = SP - 16

Live-ins:

{all registers}

{all memory}
Idempotence Model A
a starting point

\[
R1 = R2 + R3 \\
\text{if } R1 > 0
\]

\[\text{true}\]

\[\text{false}\]

\[\text{mem}[R4] = R1\]

\[\text{SP} = \text{SP} - 16\]

Live-ins:
\{all registers\}
\{all memory\} \setminus \{\text{mem}[R4]\}

\[\ldots\]

\text{TWO REGIONS. SOME STUFF NOT COVERED.}\]
Idempotence Model B
varying control flow assumptions

\[ R_1 = R_2 + R_3 \]
if \( R_1 > 0 \)

\[ \text{true} \]
\[ \text{false} \]

\[ \text{mem}[R4] = R1 \]

\[ SP = SP - 16 \]

live-in but *dynamically dead* at time of write
– OK to overwrite if control flow invariable

**ONE REGION. SOME STUFF STILL NOT COVERED.**
Idempotence Model C

varying *sequencing* assumptions

\[
R_1 = R_2 + R_3 \\
\text{if } R_1 > 0 \\
\text{true} \quad \text{false} \\
\text{mem}[R_4] = R_1 \\
SP = SP - 16
\]

allow final instruction to overwrite input (to include otherwise ineligible instructions)

**ONE REGION. EVERYTHING COVERED.**
Idempotence Model D

varying *isolation* assumptions

\[
R_1 = R_2 + R_3 \\
\text{if } R_1 > 0
\]

\[\text{mem}[R_4] = R_1\]

\[SP = SP - 16\]

may be *concurrently read* in another thread.

**TWO REGIONS. EVERYTHING COVERED.**
Idempotence Models
an idempotence taxonomy

(Model C)
sequencing axis

(Model B)
control axis

(Model D)
isolation axis
what are the implications?
Empirical Analysis
methodology

measurement
– *dynamic* region size (*path length*)
  – subject to axis constraints
– **x86** dynamic instruction count (using PIN)

benchmarks
– SPEC 2006, PARSEC, and Parboil suites

experimental configurations
– **unconstrained**: ideal upper bound (Model C)
– **oblivious**: actual in normal compiled code (Model C)
– **X-constrained**: ideal upper bound constrained by axis *X*
Empirical Analysis

Oblivious vs. unconstrained

Average region size geometrically averaged across suites

30x improvement achievable*
Empirical Analysis
control axis sensitivity

average region size

SPEC INT SPEC FP PARSEC Parboil OVERALL

160.2

5.2

40.1

4x DROP IF VARIABLE CONTROL FLOW

geometrically averaged across suites
Empirical Analysis

isolation axis sensitivity

average region size*

SPEC INT SPEC FP PARSEC Parboil OVERALL

160.2
27.4
40.1
5.2

*geometrically averaged across suites

1.5x DROP IF NO ISOLATION
Empirical Analysis
sequencing axis sensitivity

INHERENTLY NON-IDEMPOTENT INSTRUCTIONS: RARE AND OF LIMITED TYPE

generically averaged across suites
Idempotence Models
a summary

A spectrum of idempotence models
- significant opportunity: 100+ sizes possible
- 4x reduction constraining control axis
- 1.5x reduction constraining isolation axis

two models going forward
- architectural idempotence & contextual idempotence
- both are effectively the ideal case (Model C)
  - architectural idempotence: invariable control always
  - contextual idempotence: variable control w.r.t. locals
Overview

1. Idempotence Models in Architecture

2. Compiler Design & Evaluation

3. Architecture Design & Evaluation
Compiler Design
choose your own adventure

**ANALYZE:** identify *semantic clobber antidependences*

**PARTITION:** *cut* semantic clobber antidependences

**CODE GEN:** *preserve* semantic idempotence
WHAT DO YOU MEAN: PERFORMANCE OVERHEADS?
Compiler Evaluation

preamble

- preserve input values in registers
- spill other values (if needed)

- spill input values to stack
- allocate other values to registers
Compiler Evaluation
methodology

compiler implementation
– LLVM, support for both x86 and ARM

benchmarks
– SPEC 2006, PARSEC, and Parboil suites

measurements
– performance overhead: \textit{dynamic instruction count}
  – for x86, using PIN
  – for ARM, using \texttt{gem5} (just for ISA comparison at end)
– region size: \textit{instructions between boundaries} (path length)
  – x86 only, using PIN
Results, Take 1/3
initial results – overhead

**NON-TRIVIAL OVERHEAD**

percentage increase in x86 dynamic instruction count
geometrically averaged across suites
Results, Take 1/3

analysis of trade-offs

YOU ARE HERE
(typically 10-30 instructions)
Results, Take 1/3
analysis of trade-offs

\[ l = \frac{d}{r + d} \]
Results, Take 2/3
minimizing register pressure

Before

After

SPEC INT
SPEC FP
PARSEC
Parboil
OVERALL

STILL NON-TRIVIAL OVERHEAD
Results, Take 2/3
analysis of trade-offs

$\frac{d}{r+d}$

$e \propto \left(\frac{1}{1-p}\right)^r$
Big Regions
how do we get there?

**Problem #1:** aliasing analysis
  – no flow-sensitive analysis in LLVM; really hurts loops

**Problem #2:** loop optimizations
  – *boundaries in loops are bad for everyone*
  – loop blocking, fission/fusion, inter-change, peeling, unrolling, scalarization, etc. can all help

**Problem #3:** large array structures
  – awareness of *array access patterns* can help

**Problem #4:** intra-procedural scope
  – limited scope *aggravates all effects* listed above
Big Regions
how do we get there?

solutions can be automated
– a lot of work... what would be the gain?

ad hoc for now
– consider PARSEC and Parboil suites as a case study
  – aliasing annotations
  – manual loop refactoring, scalarization, etc.
  – partitioning algorithm refinements (application-specific)
  – inlining annotations
Results Take 3/3
big regions

PARSEC  | Before | After | Gain
--- | --- | --- | ---
Parboil | 13.1% | 0.06% | Trivial Overhead

Gain = BIG; Trivial Overhead
Results Take 3/3

50+ instructions is good enough
ISA Sensitivity
you might be curious

ISA matters?
(1) two-address (e.g. x86) vs. three-address (e.g. ARM)
(2) register-memory (e.g. x86) vs. register-register (e.g. ARM)
(3) number of available registers

the short version (          )
– impact of (1) & (2) not significant (+/- 2% overall)
  – even less significant as regions grow larger
– impact of (3): to get same performance w/ idempotence
  – increase registers by 0% (large) to ~60% (small regions)
design and implementation

– static analysis algorithms, modular and perform well
– code-gen algorithms, modular and perform well
– LLVM implementation source code available*

findings

– pressure-related performance overheads range from:
  – 0% (large regions) to ~15% (small regions)
– greatest opportunity: loop-intensive applications
– ISA effects are insignificant

* [http://research.cs.wisc.edu/vertical/iCompiler](http://research.cs.wisc.edu/vertical/iCompiler)
Overview

1. Idempotence Models in Architecture

2. Compiler Design & Evaluation

3. Architecture Design & Evaluation
Architecture Recovery: It’s Real
safety-first  speed first  (safety second)
Architecture Recovery: It’s Real
lots of sharp turns

closer to the truth
Architecture Recovery: It’s Real

lots of interaction

1. Fetch
2. Decode
3. Execute
4. Write-back

too late!
Architecture Recovery: It’s Real
bad stuff can happen

mis-speculation
(a) branch mis-prediction,
(b) memory re-ordering,
(c) transaction violation,
   etc.

hardware faults
(d) wear-out fault,
(e) particle strike,
(f) voltage spike,
   etc.

exceptions
(g) page fault,
(h) divide-by-zero,
(i) mis-aligned access,
   etc.
Architecture Recovery: It’s Real

bad stuff can happen

mis-speculation

- branch mis-prediction,
- memory re-ordering,
- transaction violation,
- etc.

hardware faults

- wear-out fault,
- particle strike,
- voltage spike,
- etc.

exceptions

- page fault,
- divide-by-zero,
- mis-aligned access,
- etc.

register pressure

detection latency

register pressure

re-execution time

detection latency

register pressure

re-execution time

etc.

etc.

etc.
Architecture Recovery: It’s Real
bad stuff can happen

mis-speculation
(a) branch mis-prediction,
(b) memory re-ordering,
(c) transaction violation,

etc.

hardware faults
(d) wear-out fault,
(e) particle strike,
(f) voltage spike,

etc.

exceptions
(g) page fault,
(h) divide-by-zero,
(i) mis-aligned access,

etc.
Architecture Recovery: It’s Real

bad stuff can happen

- hardware faults
  - (d) wear-out fault,
  - (e) particle strike,
  - (f) voltage spike,
  - etc.

- exceptions
  - (g) page fault,
  - (h) divide-by-zero,
  - (i) mis-aligned access,
  - etc.

integrated GPU

low-power CPU

high-reliability systems
GPU Exception Support
GPU Exception Support
why would we want it?

GPU/CPU integration
– unified address space: support for demand paging
– numerous secondary benefits as well...
GPU Exception Support
why is it hard?

the CPU solution

pipeline  buffers  registers
GPU Exception Support
why is it hard?

CPU: 10s of registers/core
GPU: 10s of registers/thread
32 threads/warp
48 warps per “core”
10,000s of registers/core
GPU Exception Support
idempotence on GPUs

GPUs hit the sweet spot
(1) extractably large regions (low compiler overheads)
(2) detection latencies long or hard to bound (large is good)
(3) exceptions are infrequent (low re-execution overheads)
GPU Exception Support
idempotence on GPUs

GPUs hit the sweet spot
(1) extractably large regions (low compiler overheads)
(2) detection latencies long or hard to bound (large is good)
(3) exceptions are infrequent (low re-execution overheads)

GPU design topics
– compiler flow
– hardware support
– exception live-lock
– bonus: fast context switching
GPU Exception Support
evaluation methodology

**compiler**
- LLVM targeting ARM

**simulation**
- gem5 for ARM: simple dual-issue in-order (e.g. Fermi)
- 10-cycle page fault detection latency

**benchmarks**
- Parboil GPU benchmarks for CPUs, modified

**measurement**
- performance overhead in execution cycles
GPU Exception Support
evaluation results

NEGLIGIBLE OVERHEADS

cutcp
fft
histo
mri-q
sad
tpacf
gmean

performance overhead

0.0% 0.5% 1.0% 1.5%
CPU Exception Support
CPU Exception Support
why is it a problem?

the CPU solution

pipeline buffers registers
CPU Exception Support
why is it a problem?

Before

Fetch -> Decode, 
Renamed, 
& Issue

Bypass

Integer

Integer

Branch

Multiply

Load/Store

FP

IEEE FP

Flush?

Replay?

Replay queue

...
CPU Exception Support
why is it a problem?

After

Fetch → Decode & Issue

Integer

Integer

Branch

Multiply

Load/Store

FP

...
CPU Exception Support
idempotence on CPUs

CPU design simplification
– in ARM Cortex-A8 (dual-issue in-order) can remove:
  – bypass / staging register file, replay queue
  – rename pipeline stage
  – IEEE-compliant floating point unit
  – pipeline flush for exceptions and replays
  – all associated control logic

leaner hardware
– bonus: cheap (but modest) OoO issue
CPU Exception Support
evaluation methodology

compiler
– LLVM targeting ARM, minimize pressure (take 2/3)

simulation
– gem5 for ARM: aggressive dual-issue in-order (e.g. A8)
– stall on potential in-flight exception

benchmarks
– SPEC 2006 & PARSEC suites (unmodified)

measurement
– performance overhead in execution cycles
CPU Exception Support
evaluation results

LESS THAN 10%... BUT NOT AS GOOD AS GPU (REGIONS TOO SMALL)
Hardware Fault Tolerance
Hardware Fault Tolerance

what is the opportunity?

**reliability trends**
- CMOS reliability is a growing problem
- future CMOS alternatives are no better

**architecture trends**
- hardware power and complexity are premium
- desire for **simple** hardware + **efficient** recovery

**application trends**
- emerging workloads consist of **large idempotent regions**
- increasing levels of **software abstraction**
Hardware Fault Tolerance

design topics

fault detection capability
- fine-grained in hardware (e.g. Argus, MICRO ‘07) or
- fine-grained in software (e.g. instruction/region DMR)

hardware organizations
- *homogenous*: idempotence everywhere
- *statically heterogeneous*: e.g. accelerators
- *dynamically heterogeneous*: adaptive cores

fault model (aka ISA semantics)
- similar to pipeline-based (e.g. ROB) recovery
Hardware Fault Tolerance
evaluation methodology

compiler
– LLVM targeting ARM (compiled to minimize pressure)

simulation
– gem5 for ARM: simple dual-issue in-order
– DMR detection; compare against checkpoint/log and TMR

benchmarks
– SPEC 2006, PARSEC, and Parboil suites (unmodified)

measurement
– performance overhead in execution cycles
Hardware Fault Tolerance

evaluation results

IDEMPOTENCE: BEATS THE COMPETITION
Overview

1. Idempotence Models in Architecture

2. Compiler Design & Evaluation

3. Architecture Design & Evaluation
RELATED WORK

CONCLUSIONS
Conclusions

**idempotence: not good for everything**
- small regions are expensive
  - preserving register state is difficult with limited flexibility
- large regions are cheap
  - preserving register state is easy with amortization effect
  - preserving memory state is mostly “for free”

**idempotence: synergistic with modern trends**
- programmability (for GPUs)
- low power (for everyone)
- high-level software ➔ efficient recovery (for everyone)
The End
MapReduce for CELL

SELSE '09: Synergy

ISCA '10: Relax

DSN '10: TS model

MICRO '11: Idempotent Processors

ISCA '12: iGPU

PLDI '12: Static Analysis and Compiler Design

CGO ??: Code Gen

TACO ??: Models

Back-Up: Chronology
Choose Your Own Adventure Slides
Idempotence Analysis
is this idempotent?

\[
\left( \begin{array}{c}
\text{Image 1} \\
\text{Image 2}
\end{array} \right)^2 = \text{Image 3}
\]

Yes
Idempotence Analysis

how about this?

$\left( \begin{array}{c} \text{No} \\ \text{No} \end{array} \right)^2 = \text{No}$
Idempotence Analysis
maybe this?

\[
\left( \begin{array}{c}
\text{(a)}
\end{array} \right) \rightarrow \left( \begin{array}{c}
\text{(b)}
\end{array} \right) \rightarrow \left( \begin{array}{c}
\text{(c)}
\end{array} \right)^2
\]

= 

Yes
## Idempotence Analysis

It’s all about the data dependences

<table>
<thead>
<tr>
<th>operation sequence</th>
<th>dependence chain</th>
<th>idempotent?</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Operation 1" /> → <img src="image2" alt="Result 1" /></td>
<td>write</td>
<td>Yes</td>
</tr>
<tr>
<td><img src="image3" alt="Operation 2" /> → <img src="image4" alt="Result 2" /> → <img src="image5" alt="Result 3" /></td>
<td>read, write</td>
<td>No</td>
</tr>
<tr>
<td><img src="image6" alt="Operation 3" /> → <img src="image7" alt="Result 4" /> → <img src="image8" alt="Result 5" /></td>
<td>write, read, write</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Idempotence Analysis
it’s all about the data dependences

<table>
<thead>
<tr>
<th>Operation sequence</th>
<th>Dependence chain</th>
<th>Idempotent?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clobber Antidependence</td>
<td>Antidependence with an exposed read</td>
<td></td>
</tr>
<tr>
<td>read, write</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>write, read, write</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
Semantic Idempotence

two types of program state

(1) local ("pseudoregister") state:
can be renamed to remove clobber antidependences*
\textit{does not semantically constrain idempotence}

(2) non-local ("memory") state:
cannot “rename” to avoid clobber antidependences
\textit{semantically constrains idempotence}

\textbf{semantic idempotence} = no \textit{non-local} clobber antidep.
preserve \textit{local} state by renaming and careful allocation
Region Partitioning Algorithm
steps one, two, and three

Step 1: transform function
remove artificial dependences, remove non-clobbers

Step 2: construct regions around antidependences
cut all non-local antidependences in the CFG

Step 3: refine for correctness & performance
account for loops, optimize for dynamic behavior
Step 1: Transform
not one, but two transformations

Transformation 1: SSA for pseudoregister antidependences

But we still have a problem:

- region identification
- depends on
- region boundaries
- clobber antidependences
Step 1: Transform
not one, but two transformations

Transformation 1: SSA for pseudoregister antidependences

Transformation 2: Scalar replacement of memory variables

\[
\begin{align*}
\text{[x]} &= a; \\
b &= \text{[x]}; \\
\text{[x]} &= c;
\end{align*}
\]

\[
\begin{align*}
\text{[x]} &= a; \\
b &= a; \\
\text{[x]} &= c;
\end{align*}
\]

non-clobber antidependences... GONE!
Step 1: Transform
not one, but two transformations

**Transformation 1:** SSA for pseudoregister antidependences

**Transformation 2:** Scalar replacement of memory variables
Region Partitioning Algorithm
steps one, two, and three

Step 1: transform function
remove artificial dependences, remove non-clobbers

Step 2: construct regions around antidependences
cut all non-local antidependences in the CFG

Step 3: refine for correctness & performance
account for loops, optimize for dynamic behavior
Step 2: Cut the CFG

cut, cut, cut...

construct regions by “cutting” non-local antidependences
larger is (generally) better: large regions amortize the cost of input preservation

Step 2: Cut the CFG
but where to cut...?

optimal region size?

rough sketch

overhead

region size

sources of overhead
Step 2: Cut the CFG
but where to cut...?

goal: the *minimum* set of cuts that cuts all antidependence paths

intuition: minimum cuts $\rightarrow$ fewest regions $\rightarrow$ large regions

approach: a series of reductions:

- *minimum vertex multi-cut* (NP-complete) $\rightarrow$
- *minimum hitting set* among paths $\rightarrow$
- *minimum hitting set* among “dominating nodes”

*details omitted*
Region Partitioning Algorithm
steps one, two, and three

**Step 1:** transform function
remove artificial dependences, remove non-clobbers

**Step 2:** construct regions around antidependences
cut all non-local antidependences in the CFG

**Step 3:** refine for correctness & performance
account for loops, optimize for dynamic behavior
Step 3: Loop-Related Refinements
loops affect correctness and performance

**correctness:** Not *all* local antidependences removed by SSA...

Loop-carried antidependences may clobber
depends on boundary placement; handled as a post-pass

**performance:** Loops tend to execute multiple times...

to maximize region size, place cuts outside of loop
algorithm modified to prefer cuts outside of loops

details omitted
Code Generation Algorithms
idempotence preservation

background & concepts:

*live intervals, region intervals, and shadow intervals*

compiling for *contextual idempotence*:

*potentially variable control flow upon re-execution*

compiling for *architectural idempotence*:

*invariable control flow upon re-execution*
Code Generation Algorithms
live intervals and region intervals

\[ x = \ldots \]
\[ \ldots = f(x) \]
\[ y = \ldots \]

region boundaries

\( x \)'s live interval

region interval
Code Generation Algorithms
shadow intervals

shadow interval
the interval over which a variable must not be overwritten specifically to preserve idempotence

different for architectural and contextual idempotence
Code Generation Algorithms
for contextual idempotence

\[ x = \ldots \]

\[ \ldots = f(x) \]

\[ y = \ldots \]

region boundaries

\( x \)'s live interval

\( x \)'s shadow interval
Code Generation Algorithms
for architectural idempotence
Code Generation Algorithms
for architectural idempotence
Big Regions
Re: Problem #2 (cut in loops are bad)

C code

```c
for (i = 0; i < X; i++) {
    ...
}
```

CFG + SSA

```
i_0 = \phi(0, i_1)
```

```
i_1 = i_0 + 1
if (i_1 < X)
```

Big Regions
Re: Problem #2 (cut in loops are bad)

C code

```c
for (i = 0; i < X; )
    ...
```

machine code

```
R0 = 0
R0 = R0 + 1
if (R0 < X)
```

**NO BOUNDARIES = NO PROBLEM**
Re: Problem #2 (cut in loops are bad)

C code

```c
for (i = 0; i < X; i++) {
    ...
}
```

machine code

```
R0 = 0

R0 = R0 + 1 if (R0 < X)
```
Big Regions
Re: Problem #2 (cut in loops are bad)

C code

```c
for (i = 0; 
    i < X; 
    i++) {
    ...
}
```

- “redundant” copy
- extra boundary (pressure)

machine code

```
R1 = 0
R0 = R1
R1 = R0 + 1
if (R1 < X)
```

Big Regions
Re: Problem #3 (array access patterns)

Algorithm makes this simplifying assumption:

\[
\begin{align*}
\text{[x]} &= a; \\
b &= \text{[x]}; \\
\text{[x]} &= c;
\end{align*}
\]

\[
\begin{align*}
\text{[x]} &= a; \\
b &= a; \\
\text{[x]} &= c;
\end{align*}
\]

Non-clobber antidependences... GONE!

Cheap for scalars, expensive for arrays
Big Regions
Re: Problem #3 (array access patterns)

not really practical for large arrays
but if we don’t do it, non-clobberer antidependences remain

```c
// initialize:
int[100] array;
memset(&array, 100*4, 0);
// accumulate:
for (...) array[i] += foo(i);
```

solution: handle potential non-clobbers in a post-pass
(same way we deal with loop clobbers in static analysis)
## Big Regions

results: sizes

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Problems</th>
<th>Size Before</th>
<th>Size After</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>ALIASING, SCOPE</td>
<td>78.9</td>
<td>&gt;10,000,000</td>
</tr>
<tr>
<td>canneal</td>
<td>SCOPE</td>
<td>35.3</td>
<td>187.3</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>ARRAYS, LOOPS, SCOPE</td>
<td>9.4</td>
<td>&gt;10,000,000</td>
</tr>
<tr>
<td>streamcluster</td>
<td>ALIASING</td>
<td>120.7</td>
<td>4,928</td>
</tr>
<tr>
<td>swaptions</td>
<td>ALIASING, ARRAYS</td>
<td>10.8</td>
<td>211,000</td>
</tr>
<tr>
<td>cutcp</td>
<td>LOOPS</td>
<td>21.9</td>
<td>612.4</td>
</tr>
<tr>
<td>fft</td>
<td>ALIASING</td>
<td>24.7</td>
<td>2,450</td>
</tr>
<tr>
<td>histo</td>
<td>ARRAYS, SCOPE</td>
<td>4.4</td>
<td>4,640,000</td>
</tr>
<tr>
<td>mri-q</td>
<td>–</td>
<td>22,100</td>
<td>22,100</td>
</tr>
<tr>
<td>sad</td>
<td>ALIASING</td>
<td>51.3</td>
<td>90,000</td>
</tr>
<tr>
<td>tpacdf</td>
<td>ARRAYS, SCOPE</td>
<td>30.2</td>
<td>107,000</td>
</tr>
</tbody>
</table>
## Big Regions

results: overheads

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Problems</th>
<th>Overhead Before</th>
<th>Overhead After</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>ALIASING, SCOPE</td>
<td>-2.93%</td>
<td>-0.05%</td>
</tr>
<tr>
<td>canneal</td>
<td>SCOPE</td>
<td>5.31%</td>
<td>1.33%</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>ARRAYS, LOOPS, SCOPE</td>
<td>26.67%</td>
<td>-0.62%</td>
</tr>
<tr>
<td>streamcluster</td>
<td>ALIASING</td>
<td>13.62%</td>
<td>0.00%</td>
</tr>
<tr>
<td>swaptions</td>
<td>ALIASING, ARRAYS</td>
<td>17.67%</td>
<td>0.00%</td>
</tr>
<tr>
<td>cutcp</td>
<td>LOOPS</td>
<td>6.344%</td>
<td>-0.01%</td>
</tr>
<tr>
<td>fft</td>
<td>ALIASING</td>
<td>11.12%</td>
<td>0.00%</td>
</tr>
<tr>
<td>histo</td>
<td>ARRAYS, SCOPE</td>
<td>23.53%</td>
<td>0.00%</td>
</tr>
<tr>
<td>mri-q</td>
<td>–</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>sad</td>
<td>ALIASING</td>
<td>4.17%</td>
<td>0.00%</td>
</tr>
<tr>
<td>tpacpf</td>
<td>ARRAYS, SCOPE</td>
<td>12.36%</td>
<td>-0.02%</td>
</tr>
</tbody>
</table>
Big Regions
problem labels

Problem #1: aliasing analysis (ALIASING)
– no flow-sensitive analysis in LLVM; really hurts loops

Problem #2: loop optimizations (LOOPS)
– boundaries in loops are bad for everyone
– loop blocking, fission/fusion, inter-change, peeling, blocking, unrolling, scalarization, etc. can all help

Problem #3: large array structures (ARRAYS)
– awareness of array access patterns can help

Problem #4: intra-procedural scope (SCOPE)
– limited scope aggravates all effects listed above
ISA Sensitivity
x86-64 vs. ARMv7

percentage overhead

SPEC INT  SPEC FP  PARSEC  Parboil  OVERALL

same configuration as *take 1/3*
ISA Sensitivity

general purpose register (GPR) sensitivity

ARMv7, 16 GPR baseline; data as geometric mean across SPEC INT
ISA Sensitivity
more registers isn’t always enough

C code

```c
x = 0;
if (y > 0)
  x = 1;
z = x + y;
```

R0 = 0

if (R1 > 0)
  R0 = 1

R2 = R0 + R1
ISA Sensitivity
more registers isn’t always enough

C code

```c
x = 0;
if (y > 0)
    x = 1;
z = x + y;
```

```
R0 = 0
if (R1 > 0)
R3 = R0
R3 = 1
R2 = R3 + R1
```
GPU Exception Support
compiler flow & hardware support

compiler

hardware
GPU Exception Support
exception live-lock and fast context switching

exception live-lock
  – multiple recurring exceptions can cause live-lock
  – detection: save PC and compare
  – recovery: single-stepped re-execution or re-compilation

bonus: fast context switching
  – boundary locations are configurable at compile time
  – observation 1: save/restore only live state
  – observation 2: place boundaries to minimize liveness
CPU Exception Support

design simplification

idempotence $\rightarrow$ OoO retirement $\rightarrow$
  – simplify result bypassing
  – simplifies exception support for long latency instructions
  – simplifies scheduling of variable-latency instructions

OoO issue?
CPU Exception Support
design simplification

what about branch prediction, etc.?
high re-execution costs; live-lock issues

region placement to minimize re-execution...?
CPU Exception Support

minimizing branch re-execution cost

- take 2/3
- cut at branch

SPEC INT: 115%
SPEC FP: 18.1%
PARSEC: 9.1%
OVERALL: 18.1%
Hardware Fault Tolerance
fault semantics

hardware fault model (fault semantics)
– side-effects are *temporally* contained to region execution
– side-effects are *spatially* contained to target resources
– control flow is *legal* (follows static CFG edges)
## Related Work

on idempotence

<table>
<thead>
<tr>
<th>Very Related</th>
<th>Year</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sentinel Scheduling</td>
<td>1992</td>
<td>Speculative memory re-ordering</td>
</tr>
<tr>
<td>Reference Idempotency</td>
<td>2006</td>
<td>Reducing speculative storage</td>
</tr>
<tr>
<td>Restart Markers</td>
<td>2006</td>
<td>Virtual memory in vector machines</td>
</tr>
<tr>
<td>Encore</td>
<td>2011</td>
<td>Hardware fault recovery</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Somewhat Related</th>
<th>Year</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-Instruction Retry</td>
<td>1995</td>
<td>Branch and hardware fault recovery</td>
</tr>
<tr>
<td>Atomic Heap Transactions</td>
<td>1999</td>
<td>Atomic memory allocation</td>
</tr>
</tbody>
</table>
Related Work
on idempotence

what’s new?

– idempotence model classification and analysis
– first work to decompose entire programs
– static analysis in terms of clobber (anti-)dependences
– static analysis and code generation algorithms
– overhead analysis: detection, pressure, re-execution
– comprehensive (and general) compiler implementation
– comprehensive compiler evaluation
– a spectrum of architecture designs & applications