**idempotent** (ī-dəm-pō-tənt) adj. 1 of, relating to, or being a mathematical quantity which when applied to itself equals itself; 2 of, relating to, or being an operation under which a mathematical quantity is idempotent.

**idempotent processing** (ī-dəm-pō-tənt prə-ses-iŋ) n. the application of only idempotent operations in sequence; said of the execution of computer programs in units of only idempotent computations, typically, to achieve restartable behavior.
Static Analysis and Compiler Design for Idempotent Processing

Marc de Kruijf
Karthikeyan Sankaralingam
Somesh Jha

PLDI 2012, Beijing
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

exceptions
mis-speculations
faults
Example
assembly code

\[ R2 = \text{load} [R1] \]
\[ R3 = 0 \]

**BAD STUFF HAPPENS!**

\[ R2 = \text{sub} R2, 1 \]
\[ \text{bnez} R2, \text{LOOP} \]

**EXIT:**
\[ \text{return} \ R3 \]
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

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;
}
```
Idempotent Processing

(idempotent regions)

All The Time
Idempotent Processing

executive summary

how?

idempotence inhibited by *clobber antidependences*

cut *semantic* clobber antidependences

normal compiler:

custom compiler:

low runtime overhead (typically 2-12%)
Presentation Overview

1. Idempotence

2. Algorithm

3. Results
What is Idempotence?

Is this idempotent?

\[
\left( \begin{array}{c}
\text{Hand stirring coffee} \\
\text{Coffee} \\
\end{array} \right)^2
\rightarrow
\left( \begin{array}{c}
\text{Coffee} \\
\end{array} \right)
\]

\[
= \\
\text{Yes}
\]
What is Idempotence?
how about this?

(1)  

=  

2  

No
What is Idempotence?
maybe this?

(\text{\includegraphics{image.png}} \rightarrow \text{\includegraphics{image.png}} \rightarrow \text{\includegraphics{image.png}})^2 \Rightarrow \text{\includegraphics{image.png}}

= \text{Yes}
What is Idempotence?
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="Image" /> → <img src="image2" alt="Image" /></td>
<td>write</td>
<td>Yes</td>
</tr>
<tr>
<td><img src="image3" alt="Image" /> → <img src="image4" alt="Image" /></td>
<td>read, write</td>
<td>No</td>
</tr>
<tr>
<td><img src="image5" alt="Image" /> → <img src="image6" alt="Image" /> → <img src="image7" alt="Image" /></td>
<td>write, read, write</td>
<td>Yes</td>
</tr>
</tbody>
</table>
What is Idempotence?

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>Write, read</td>
<td>Clobber Antidependence</td>
<td>Yes</td>
</tr>
<tr>
<td>Read, write</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>Write, read, write</td>
<td></td>
<td>Yes</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 non-local clobber antidep.

preserve local state by renaming and careful allocation
Presentation Overview

1. Idempotence

2. Algorithm

3. Results
Region Construction 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
- region boundaries
- clobber antidependences

depends on
Step 1: Transform
not one, but two transformations

Transformation 1: SSA for pseudoregister antidependences
Transformation 2: Scalar replacement of memory variables

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

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 Construction 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
Step 2: Cut the CFG
but where to cut…?

optimal region size?

rough sketch

larger is (generally) better:
large regions amortize the cost of input preservation
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 in paper...*
Region Construction 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 in paper...
Presentation Overview

1. Idempotence

2. Algorithm

3. Results
Results

compiler implementation
  – Paper compiler implementation in LLVM v2.9
  – LLVM v3.1 source code release in July timeframe

experimental data
  (1) runtime overhead
  (2) region size
  (3) use case
Runtime Overhead
as a percentage

<table>
<thead>
<tr>
<th>benchmark suites (gmean)</th>
<th>(gmean)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPEC INT</td>
<td></td>
</tr>
<tr>
<td>SPEC FP</td>
<td></td>
</tr>
<tr>
<td>PARSEC</td>
<td></td>
</tr>
<tr>
<td>OVERALL</td>
<td></td>
</tr>
</tbody>
</table>

percent overhead

- SPEC INT
- SPEC FP
- PARSEC
- OVERALL

instruction count
execution time

7.6
7.7

benchmark suites (gmean)
Region Size
average number of instructions

dynamic region size

SPEC INT  SPEC FP  PARSEC  OVERALL

benchmark suites (gmean)  (gmean)

28

(gmean)

compiler-generated
Use Case

hardware fault recovery

percent overhead

SPEC INT  SPEC FP  PARSEC  OVERALL

benchmark suites (gmean)

(idempotence)
(checkpoint/log)
(instruction TMR)

gmean

30.5
24.0
8.2
Presentation Overview

1. Idempotence

2. Algorithm

3. Results
Summary & Conclusions

idempotent processing
  – large (low-overhead) idempotent regions all the time

static analysis, compiler algorithm
  – (a) remove artifacts (b) partition (c) compile

low overhead
  – 2-12% runtime overhead typical
Summary & Conclusions

conclusions

several applications already demonstrated
– CPU hardware simplification (MICRO ’11)
– GPU exceptions and speculation (ISCA ’12)
– hardware fault recovery (*this paper*)

future work
– more applications, hybrid techniques
– optimal region size?
– enabling even larger region sizes
Back-up Slides
Error recovery
dealing with side-effects

exceptions
– generally no side-effects beyond out-of-order-ness
– fairly easy to handle

mis-speculation (e.g. branch misprediction)
– compiler handles for pseudoregister state
– for non-local memory, store buffer assumed

arbitrary failure (e.g. hardware fault)
– ECC and other verification assumed
– variety of existing techniques; details in paper
Optimal Region Size?
it depends... (rough sketch not to scale)

![Diagram showing the relationship between overhead, region size, detection latency, register pressure, and re-execution time.](image)
## Prior Work relating to idempotence

<table>
<thead>
<tr>
<th>Technique</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>Fast Mutual Exclusion</td>
<td>1992</td>
<td>Uniprocessor mutual exclusion</td>
</tr>
<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>
<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>Data-Triggered Threads</td>
<td>2011</td>
<td>Data-triggered multi-threading</td>
</tr>
<tr>
<td>Idempotent Processors</td>
<td>2011</td>
<td>Hardware simplification for exceptions</td>
</tr>
<tr>
<td>Encore</td>
<td>2011</td>
<td>Hardware fault recovery</td>
</tr>
<tr>
<td>iGPU</td>
<td>2012</td>
<td>GPU exception/speculation support</td>
</tr>
</tbody>
</table>
Detailed Runtime Overhead

as a percentage

percent overhead

<table>
<thead>
<tr>
<th></th>
<th>SPEC INT</th>
<th>SPEC FP</th>
<th>PARSEC</th>
<th>gcc</th>
<th>namd</th>
<th>OVERALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>suites (gmean)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>outliers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(gmean)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7.6 non-idempotent inner loops + high register pressure
Detailed Region Size
average number of instructions

suites (gmean)  outliers  (gmean)

limited aliasing information

SPEC INT  SPEC FP  PARSEC  hmmmer  ibm  OVERALL

compiler  ideal  Ideal w/o outliers

>1,000,000

28  116
45

40