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

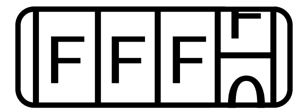
```c
int sum(int *array, int len) {
  int x = 0;
  for (int i = 0; i < len; ++i)
    x += array[i];
  return x;
}
```

Example

assembly code

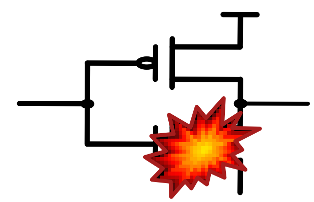
```
  R2 = load [R1]
  R3 = 0
LOOP:
  R4 = load [R0 + R2]
  R3 = add R3, R4
  R2 = sub R2, 1
  bnez R2, LOOP
EXIT:
  return R3
```

load --> ?



exceptions

mis-speculations



faults

Example

assembly code

```
  R2 = load [R1]
  R3 = 0
LOOP:
```

BAD STUFF HAPPENS!

```
  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
```

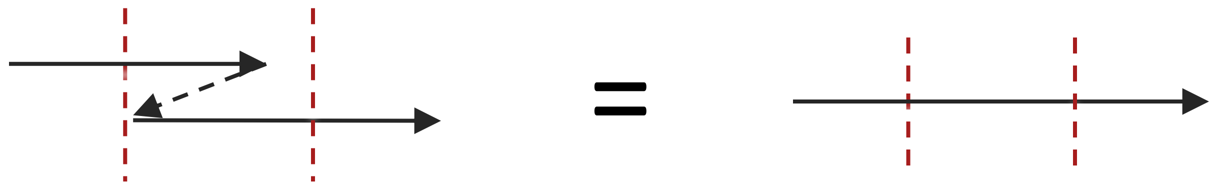
R0 and R1 are unmodified

just re-execute!

~~***convention:***~~
~~use checkpoints/buffers~~

It’s Idempotent!

idempoh… what…?



```
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

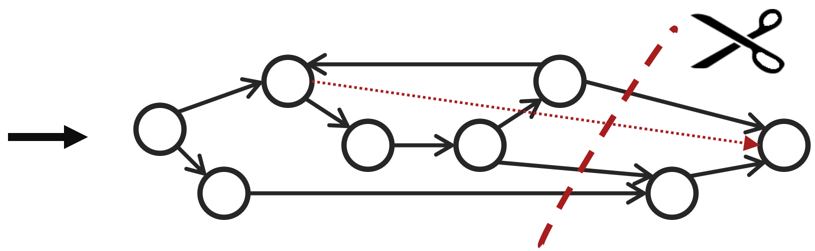
ALL THE TIME

Idempotent Processing
executive summary

how?

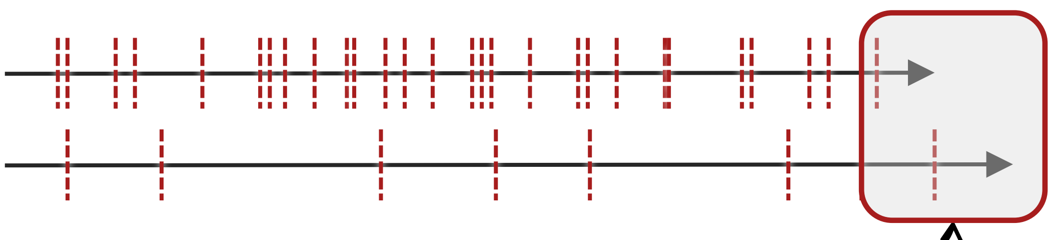
idempotence inhibited by *clobber antidependences*

cut *semantic* clobber antidependences



normal compiler:

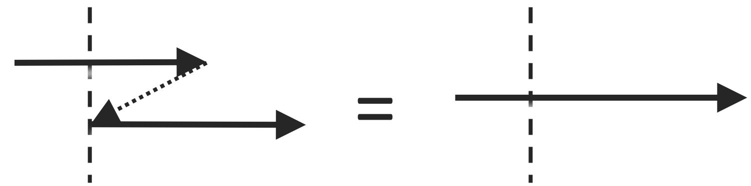
custom compiler:



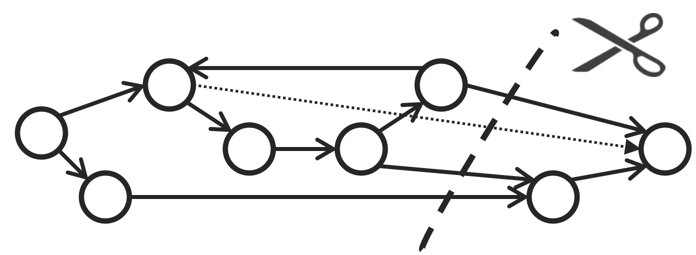
low runtime overhead (typically 2-12%)

Presentation Overview

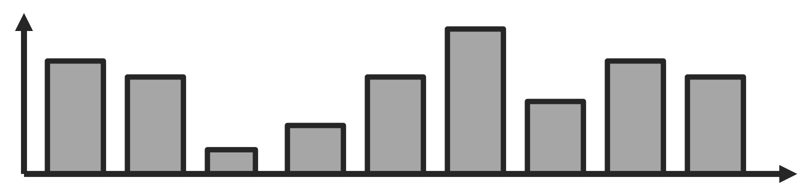
❶ Idempotence



❷ Algorithm



❸ Results



What is Idempotence?

is this idempotent?

( → )^2

= 

Yes

What is Idempotence?

how about this?

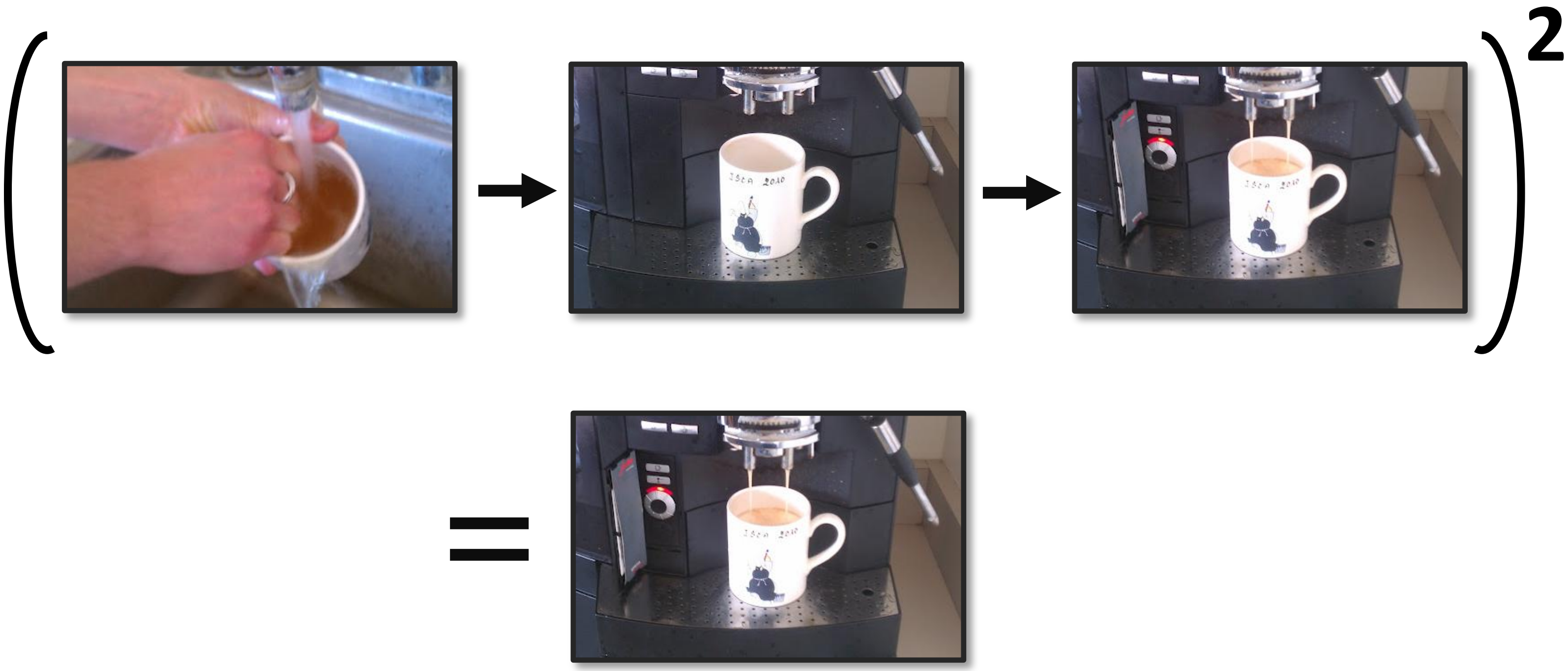
( → )^2

= 

No

What is Idempotence?

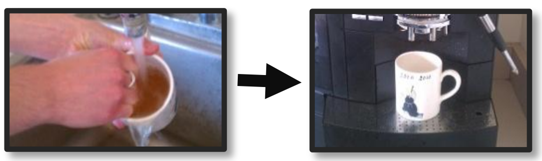
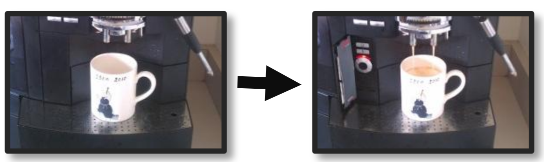
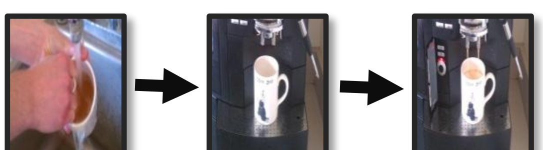
maybe this?



Yes

What is Idempotence?

it’s all about the data dependences

operation sequence	dependence chain	idempotent?
	write	Yes
	read, write	No
	write, read, write	Yes

What is Idempotence?

it’s all about the data dependences

CLOBBER ANTIDEPENDENCE

antidependence with an exposed read

operation sequence	dependence chain	idempotent?
	read, write	No
	write, read, write	Yes

Semantic Idempotence

two types of program state

(1) local (“pseudoregister”) state:

can be renamed to remove clobber antidependences*

does not semantically constrain idempotence

(2) non-local (“memory”) state:

cannot “rename” to avoid clobber antidependences

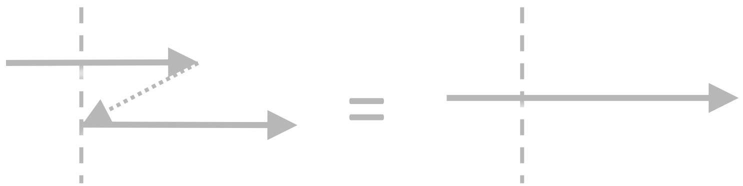
semantically constrains idempotence

semantic idempotence = no *non-local* clobber antidep.

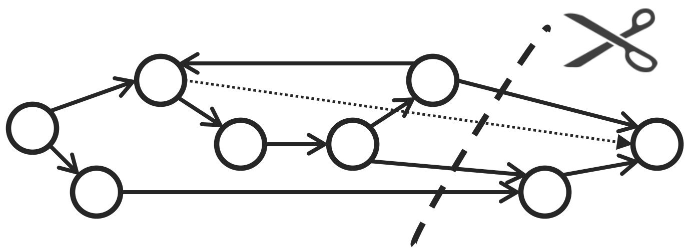
preserve *local* state by renaming and careful allocation

Presentation Overview

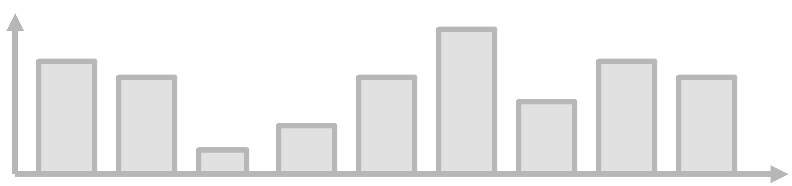
❶ Idempotence



❷ Algorithm



❸ 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

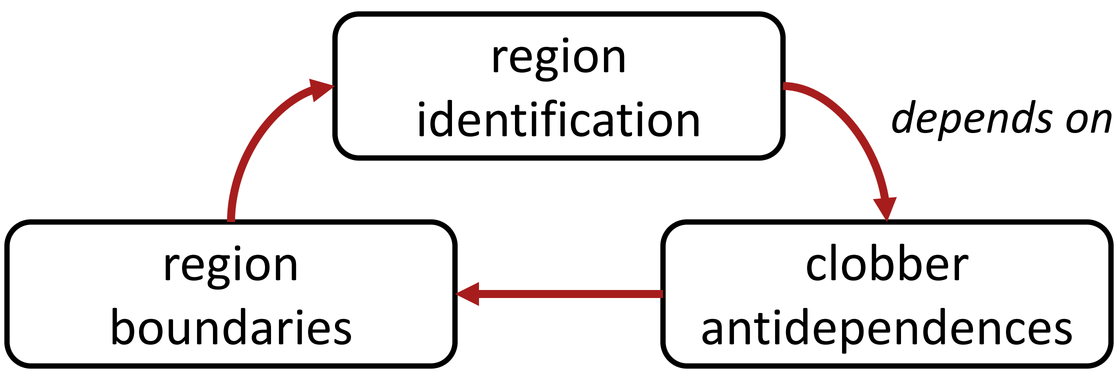
Step 3: refine for correctness & performance
account for loops, optimize for dynamic behavior

Step 1: Transform

not one, but two transformations

Transformation 1: SSA for pseudoregister antidependences

But we still have a problem:



Step 1: Transform

not one, but two transformations

Transformation 1: SSA for pseudoregister antidependences

Transformation 2: Scalar replacement of memory variables

```
[x] = a;
b   = [x];
[x] = c;
```

→

```
[x] = a;
b   = a;
[x] = c;
```

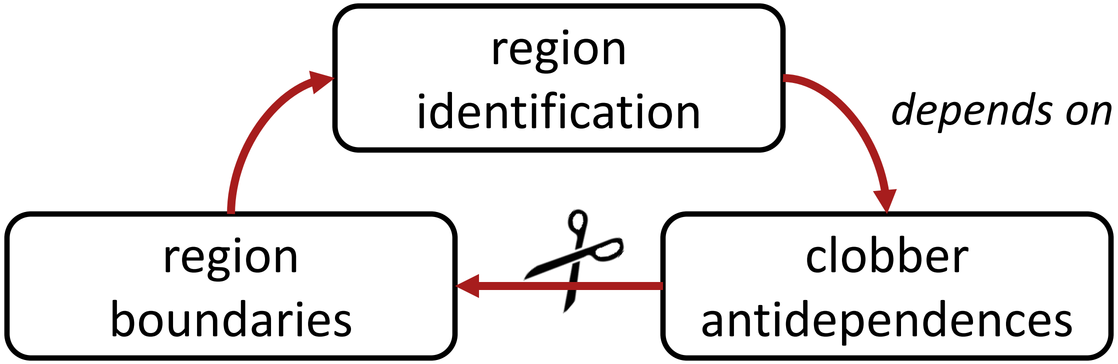
non-clobber antidependences… GONE!

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

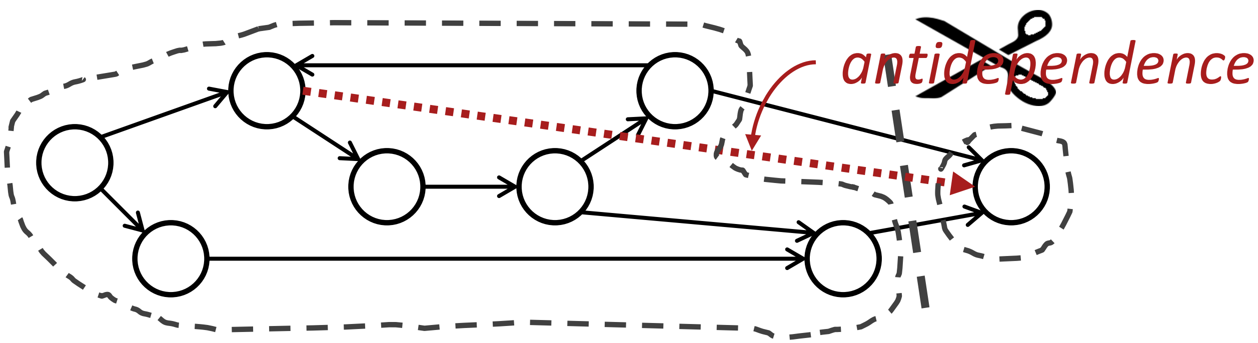
Step 2: construct regions around antidependences
cut all non-local antidependences in the CFG

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

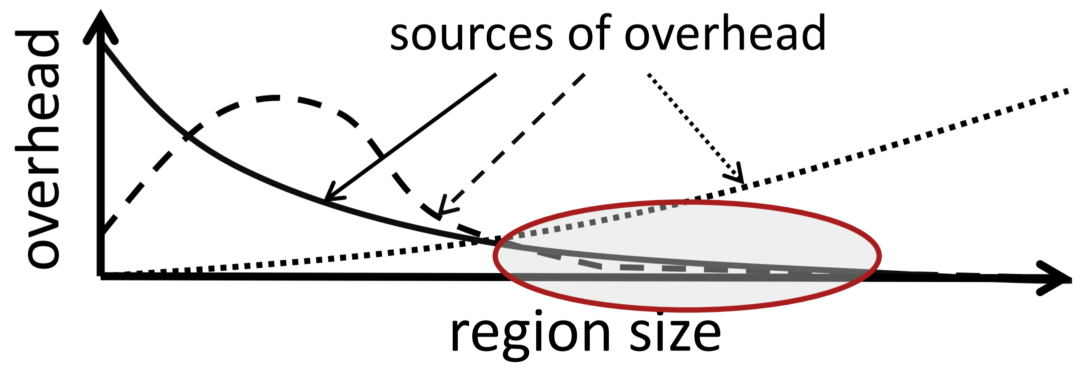


Step 2: Cut the CFG

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 → fewest regions → large regions

approach: a series of reductions:
minimum vertex multi-cut (NP-complete) →
minimum hitting set among paths →
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…

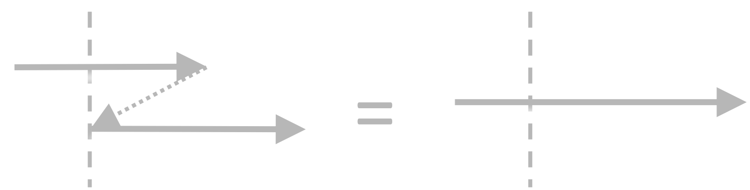
to maximize region size, place cuts outside of loop

algorithm modified to prefer cuts outside of loops

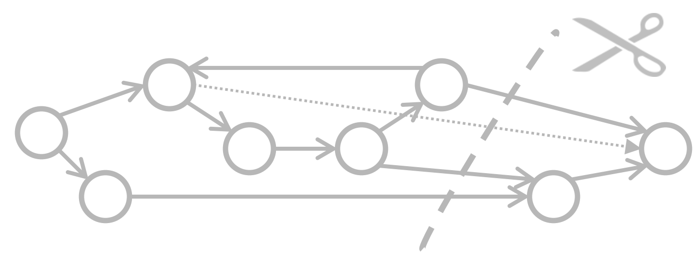
details in paper…

Presentation Overview

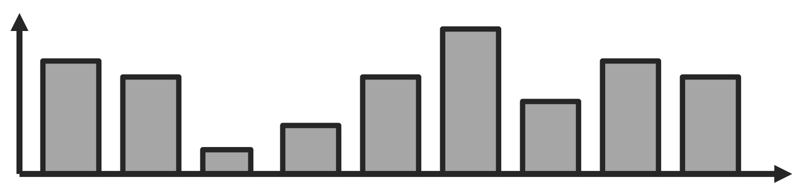
❶ Idempotence



❷ Algorithm



❸ Results



Results

compiler implementation

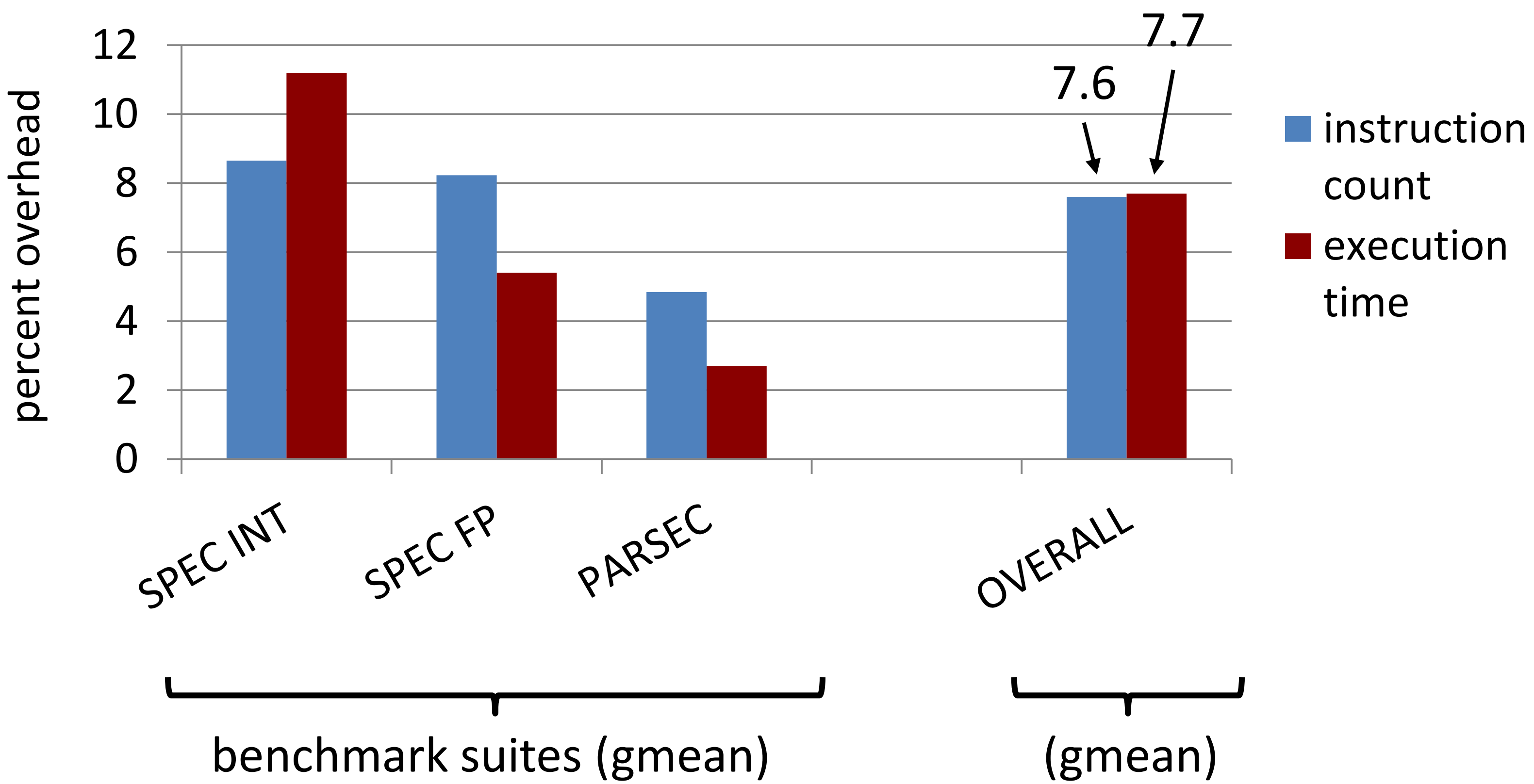
– Paper compiler implementation in LLVM v2.9
– LLVM v3.1 source code release in July timeframe

experimental data

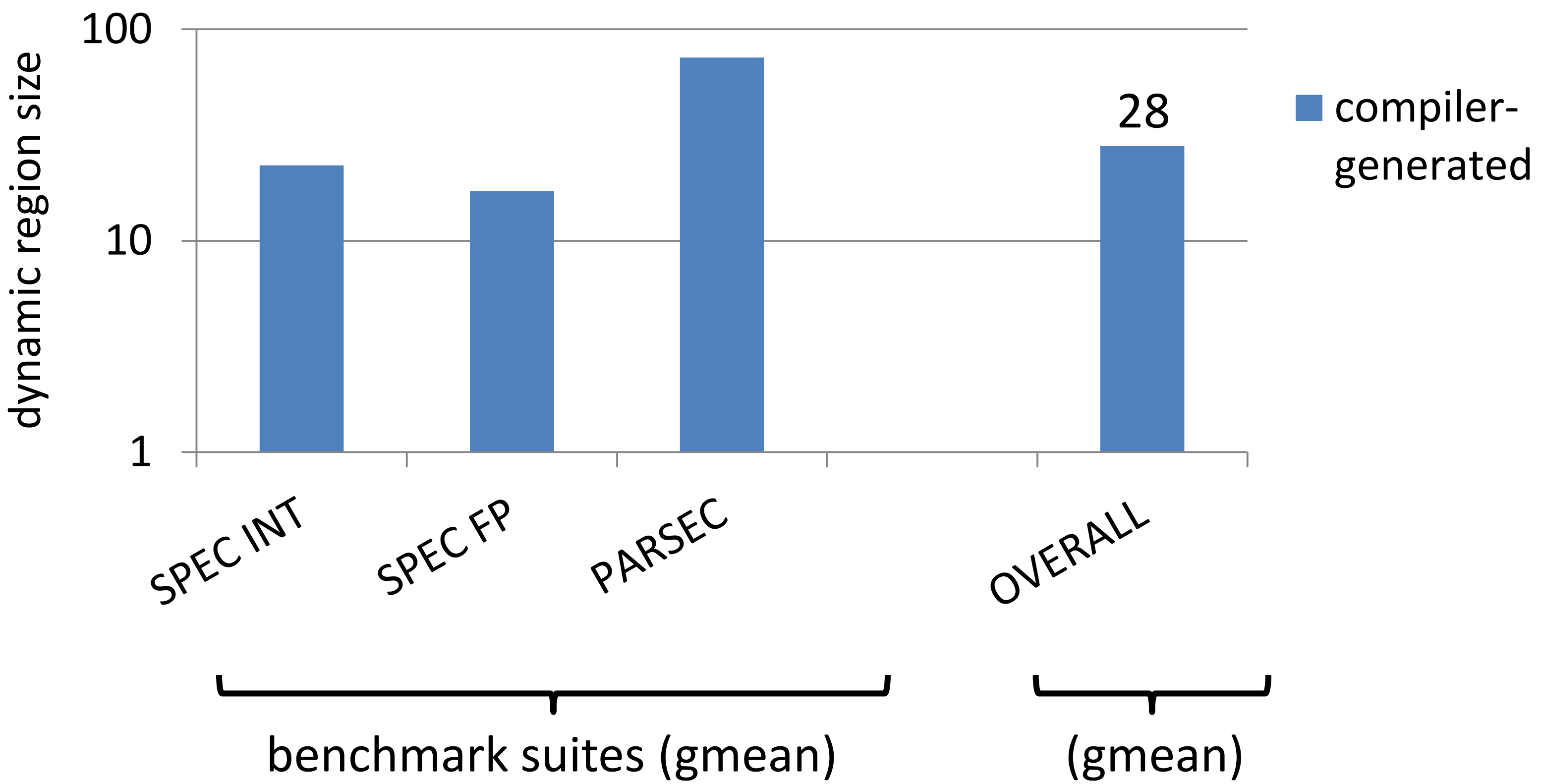
(1) runtime overhead
(2) region size
(3) use case

Runtime Overhead

as a percentage

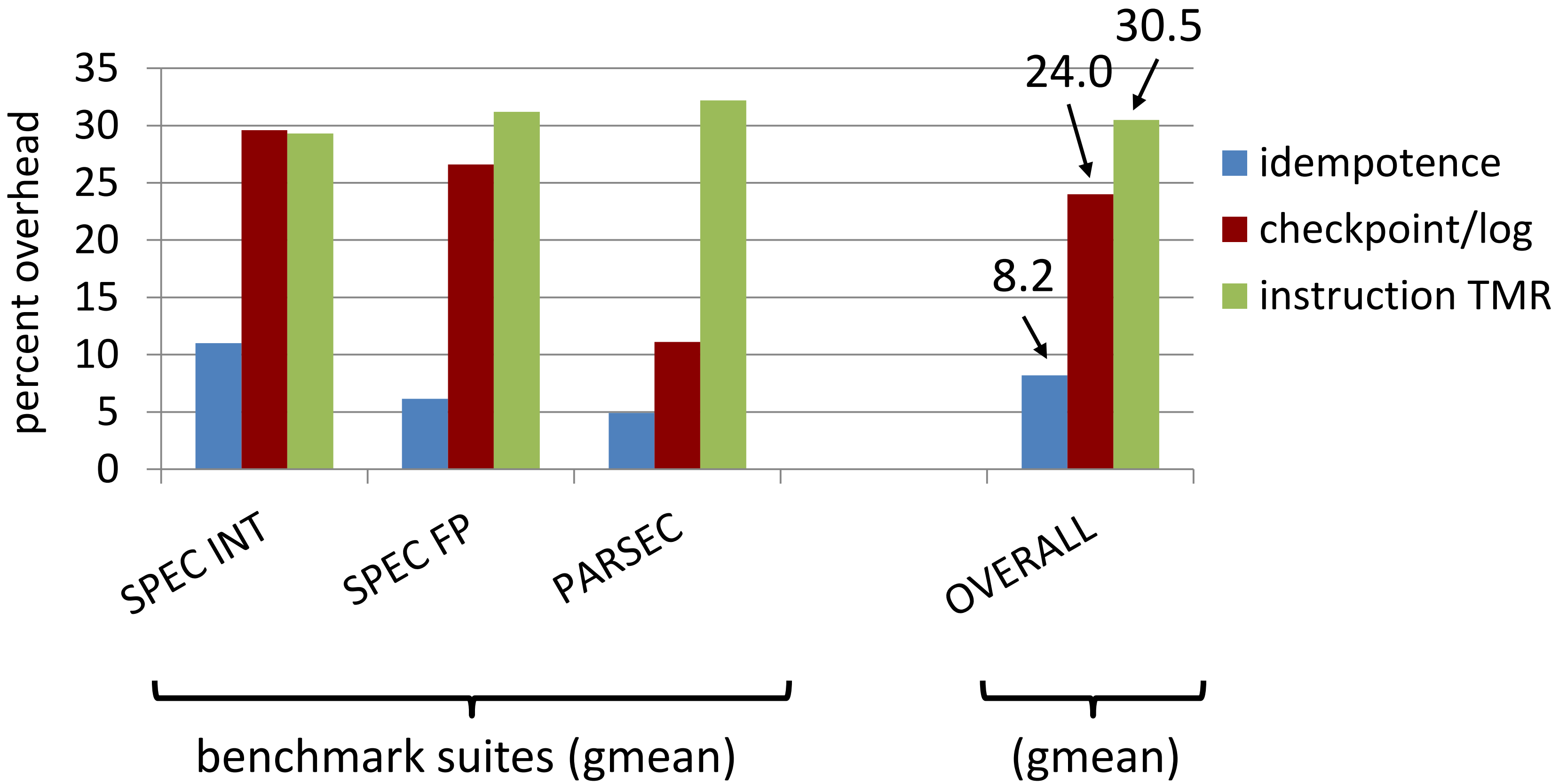


Region Size
average number of instructions



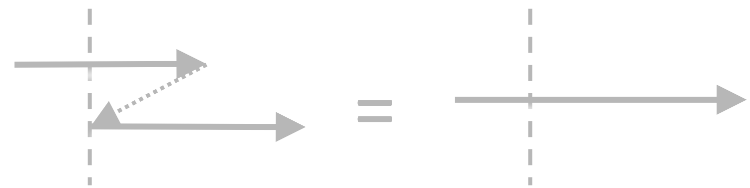
Use Case

hardware fault recovery

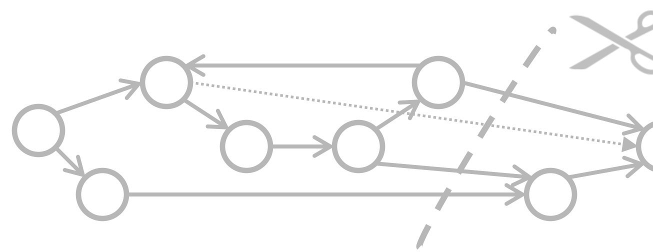


Presentation Overview

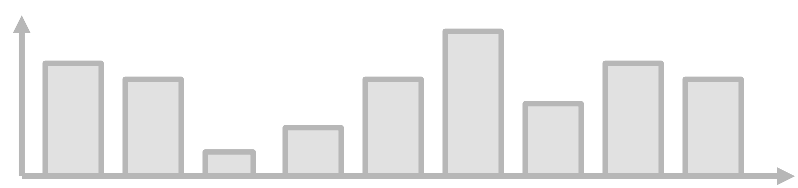
1 Idempotence



2 Algorithm



3 Results



Summary & Conclusions
summary

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)

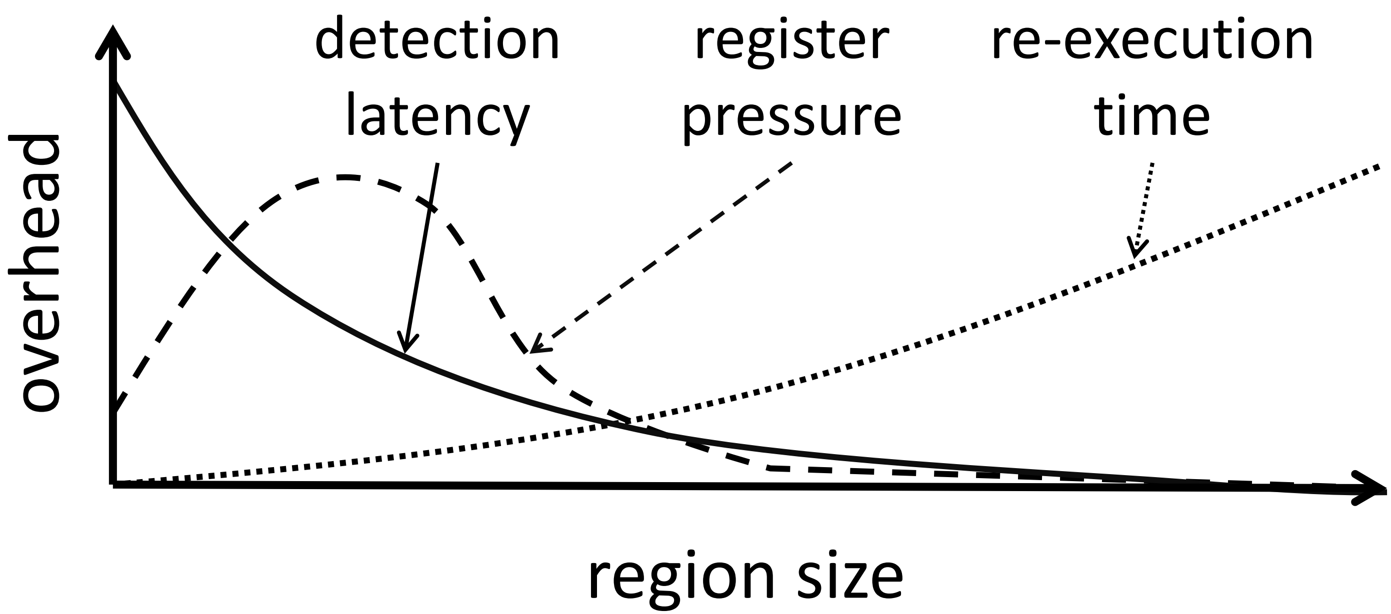
– compiler handles for pseudoregister state
– for non-local memory, store buffer assumed

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)

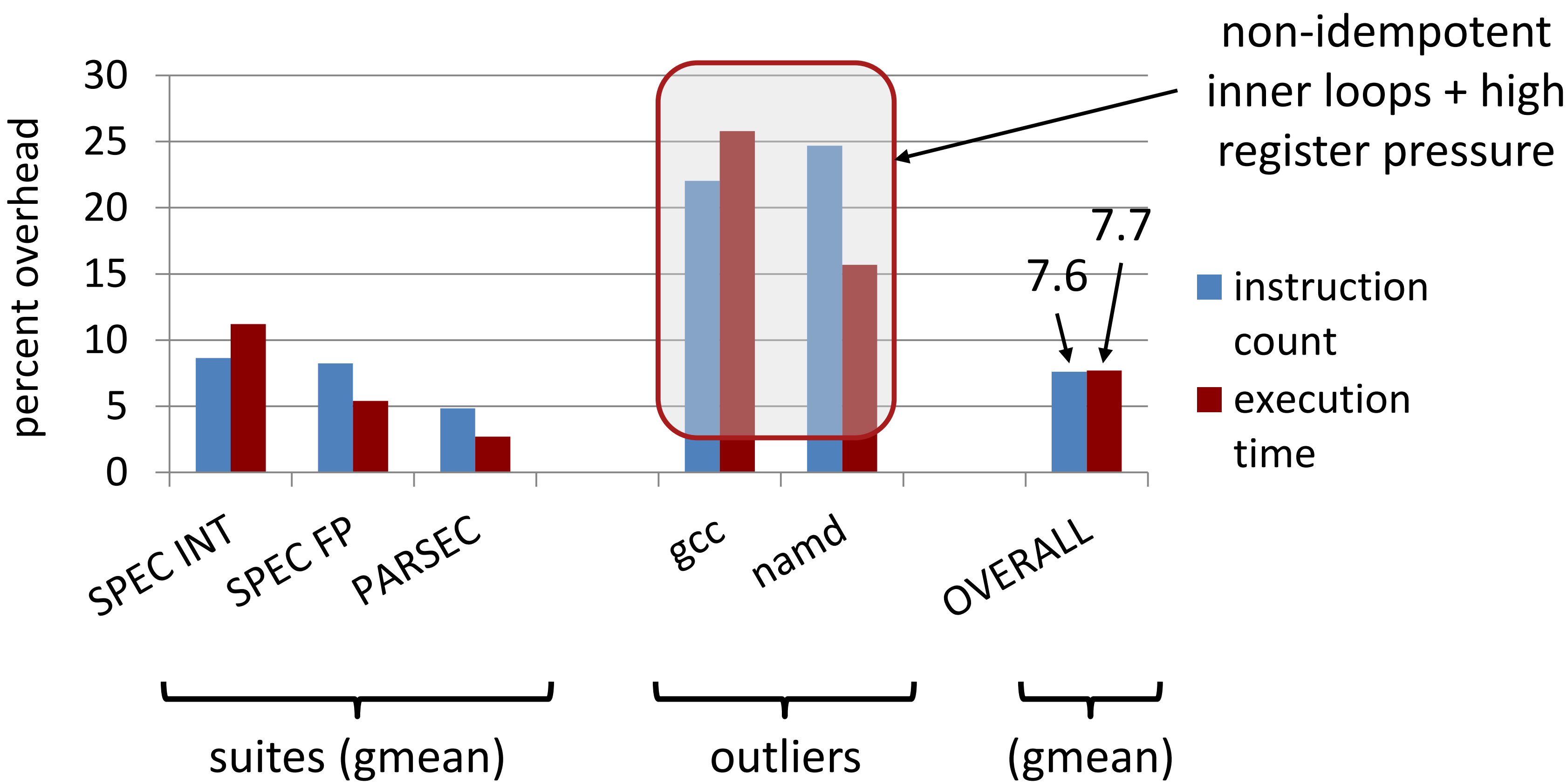


Prior Work
relating to idempotence

Technique	Year	Domain
Sentinel Scheduling	1992	Speculative memory re-ordering
Fast Mutual Exclusion	1992	Uniprocessor mutual exclusion
Multi-Instruction Retry	1995	Branch and hardware fault recovery
Atomic Heap Transactions	1999	Atomic memory allocation
Reference Idempotency	2006	Reducing speculative storage
Restart Markers	2006	Virtual memory in vector machines
Data-Triggered Threads	2011	Data-triggered multi-threading
Idempotent Processors	2011	Hardware simplification for exceptions
Encore	2011	Hardware fault recovery
iGPU	2012	GPU exception/speculation support

Detailed Runtime Overhead

as a percentage



Detailed Region Size

average number of instructions