Efficient and Effective **Handling of Exceptions** in Java Points-To Analysis

George Kastrinis ~ Yannis Smaragdakis
University of Athens

Executive Summary

- Huge amount of analysis time spent on exceptions
- They mainly affect control-flow
- Significant speedup from coarsening exceptions
- Type-based merging as an effective coarsening
- No trade-off in precision (in “normal” code)
- Datalog formalism makes changes clear
- Also excellent implementation platform

Efficient and Effective Handling of Exceptions in Java Points-To Analysis

Efficient and Effective Handling of Exceptions in Java Points-To Analysis

What **objects** may a **variable** point to?
(statically, object = allocation site)

Efficient and Effective Handling of **Exceptions** in Java Points-To Analysis

Refresher on Exceptions

```
void foo (...) throws AnException {
    try {
        ...
        throw new MyException();
        ...
    }
    catch (OtherException e) { ... }
}
```

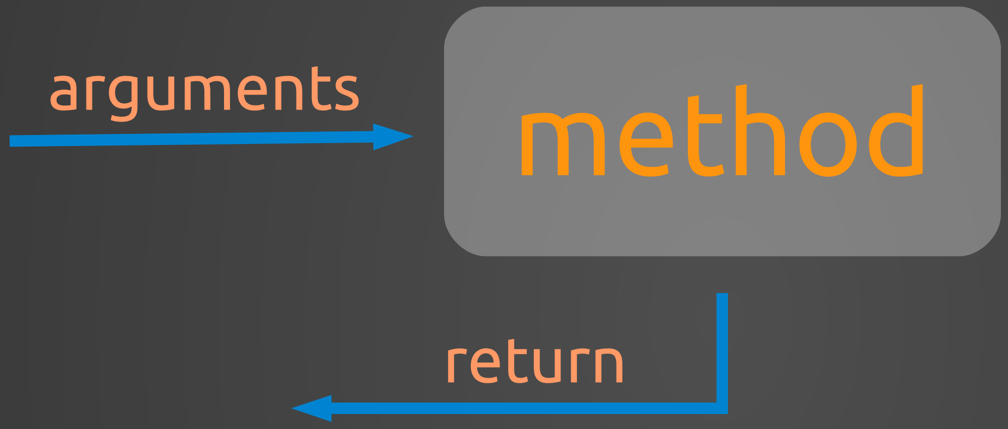
Significance of Exceptions

- Exceptions are non-local control flow
- They are also regular objects with data fields
- How significant is the data-flow of exceptions?
  - Our research indirectly answers this

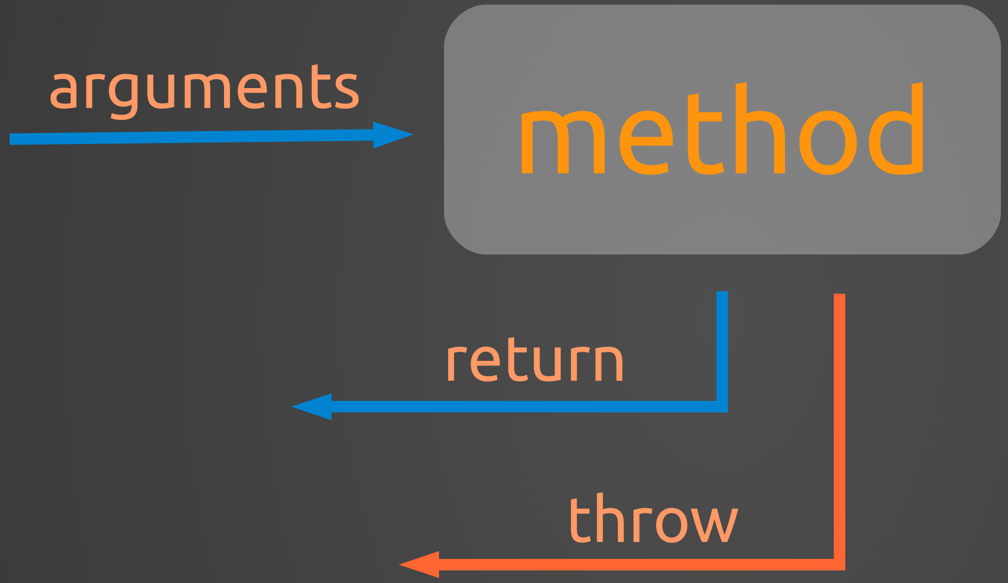
FLOW OF OBJECTS

method

Normal Flow Of Objects



Exceptional Flow Of Objects



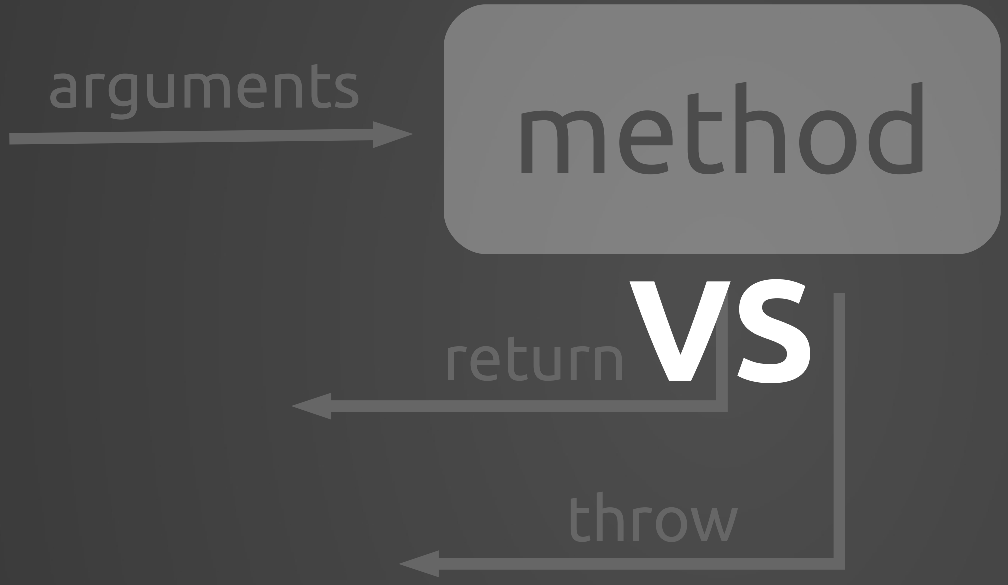
arguments

method

return

throw

Normal Flow



arguments

method

return

throw

VS

Exceptional Flow

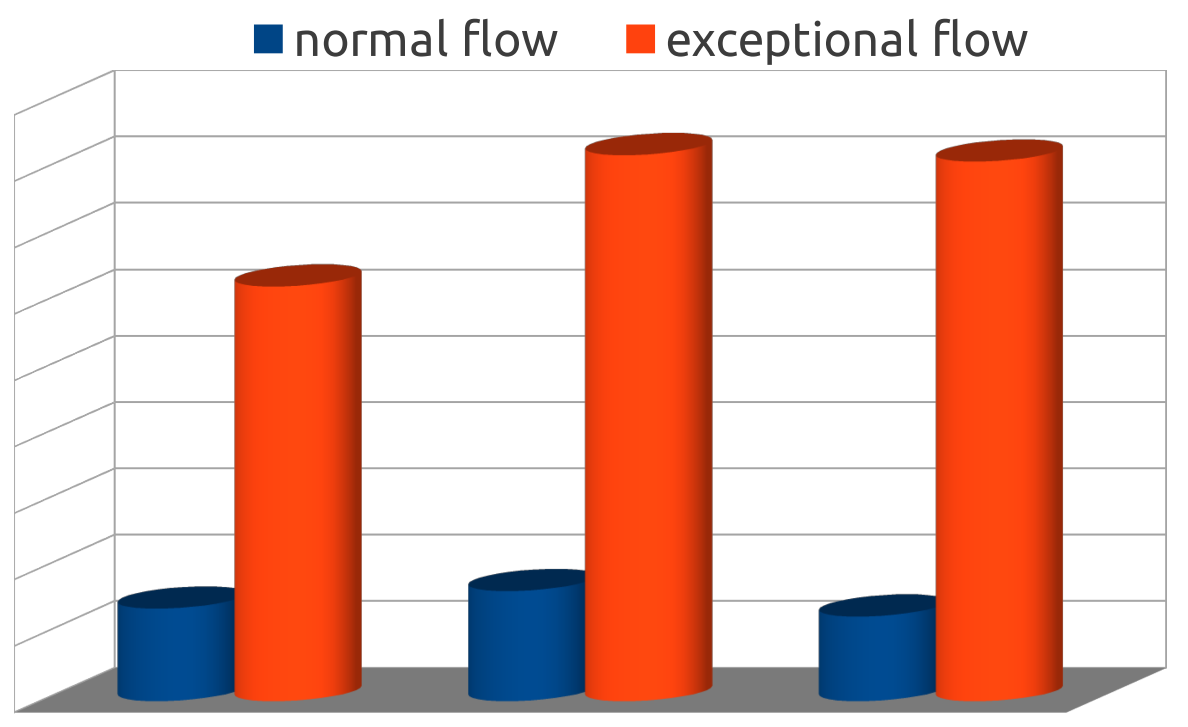
Normal Flow **vs** Exceptional Flow

Which one do **you** think dominates?

i.e., for the average method are there more objects that may be thrown out of it or that may be passed into/out of it as args/returns?

Normal Flow vs Exceptional Flow

Which one do **you** think dominates?



NEED PRECISE EXCEPTIONS?

NEED PRECISE EXCEPTIONS?

Not per se

NEED PRECISE EXCEPTIONS?

Not per se

Overall analysis effect

COARSEN EXCEPTIONS

A. Context Insensitive

Coarsen Exceptions

A. Context Insensitive

B. Type-based Merging

Efficient and Effective Handling of Exceptions in Java Points-To Analysis

Efficient and Effective Handling of Exceptions in Java Points-To Analysis

Doop

INPUT

JAVA CODE

```
v = new A();


to = from;

to = base.fld;
base.fld = from;


void meth(..., A arg, ...) {
   ...
   return ret;
}


base.sig(...);
```

Java Code as Tables

```java
v = new A();
```

Alloc (*var*, *obj*, *meth*)
ObjType (*obj*, *type*)

```java
to = from;
```

Move (*to*, *from*)

```java
to = base.fld;
base.fld = from;
```

Load (*to*, *base*, *fld*)
Store (*base*, *fld*, *from*)

```java
void meth(..., A arg, ...) {
   ...
   return ret;
}
```

FormalArg (*meth*, *i*, *arg*)

FormalReturn (*meth*, *ret*)

```java
base.sig(...);
```

VCall (*base*, *sig*, *invo*)

Java Code as Tables

`v = new A();`	Alloc (*var, obj, meth*) ObjType (*obj, type*)
`to = from;`	Move (*to, from*)

`to = base.fld;`
`base.fld = from;`
	Load (*to, base, fld*)
	Store (*base, fld, from*)

`void meth(..., A arg, ...) {`	FormalArg (*meth, i, arg*)

`...`
`return ret;`
	FormalReturn (*meth, ret*)

`}`	
`base.sig(...);`	VCall (*base, sig, invo*)

Java Code as Tables

`v = new A();`	Alloc (*var*, *obj*, *meth*) ObjType (*obj*, *type*)
`to = from;`	Move (*to*, *from*)

`to = base.fld;`
`base.fld = from;`
	Load (*to*, *base*, *fld*)
	Store (*base*, *fld*, *from*)

`void meth(..., A arg, ...) {`	FormalArg (*meth*, *i*, *arg*)

`    ...`
`    return ret;`
`}`
	FormalReturn (*meth*, *ret*)

`base.sig(...);`	VCall (*base*, *sig*, *invo*)

and many more...

Java Code as Tables

`v = new A();`	Alloc (*var*, *obj*, *meth*) ObjType (*obj*, *type*)
`to = from;`	Move (*to*, *from*)

`to = base.fld;`
`base.fld = from;`
	Load (*to*, *base*, *fld*)
	Store (*base*, *fld*, *from*)

`void meth(..., A arg, ...) {`
`   ...`
`   return ret;`
`}`
	FormalArg (*meth*, *i*, *arg*)
	FormalReturn (*meth*, *ret*)

`base.sig(...);`	VCall (*base*, *sig*, *invo*)

Blue is Input

and many more...

OUTPUT

OUTPUT

most important...

VARPOINTSTO (*var*, *ctx*, *obj*, *objCtx*)

Output

most important…

VarPointsTo (*var*, *ctx*, *obj*, *objCtx*)

Reachable (*meth*, *ctx*)
CallGraph (*invo*, *callerCtx*, *meth*, *calleeCtx*)

“On the fly” construction

OUTPUT

most important...

VARPOINTSTO (*var, ctx, obj, objCtx*)

Orange is Output

REACHABLE (*meth, ctx*)

CALLGRAPH (*invo, callerCtx, meth, calleeCtx*)

“On the fly” construction

CONTEXTS : BLACK BOX

VARPOINTSTO (*var*, *ctx*, *obj*, *objCtx*)

REACHABLE (*meth*, *ctx*)
CALLGRAPH (*invo*, *callerCtx*, *meth*, *calleeCtx*)

CONTEXTS CONSTRUCTORS

VARPOINTSTO (*var*, *ctx*, *obj*, *objCtx*)

REACHABLE (*meth*, *ctx*)
CALLGRAPH (*invo*, *callerCtx*, *meth*, *calleeCtx*)

RECORD (...) = ***newObjCtx***

MERGE (...) = ***newCtx***

Pick Your Contexts Well:
Understanding Object-Sensitivity
Smaragdakis – Bravenboer – Lhotak

POPL'11

RULES

Example Rule

$$\mathbf{P}\ (x),\ \mathbf{Q}\ (x,\ z) \leftarrow \mathbf{R}\ (x,\ y,\ w),\ \mathbf{S}\ (y,\ z).$$

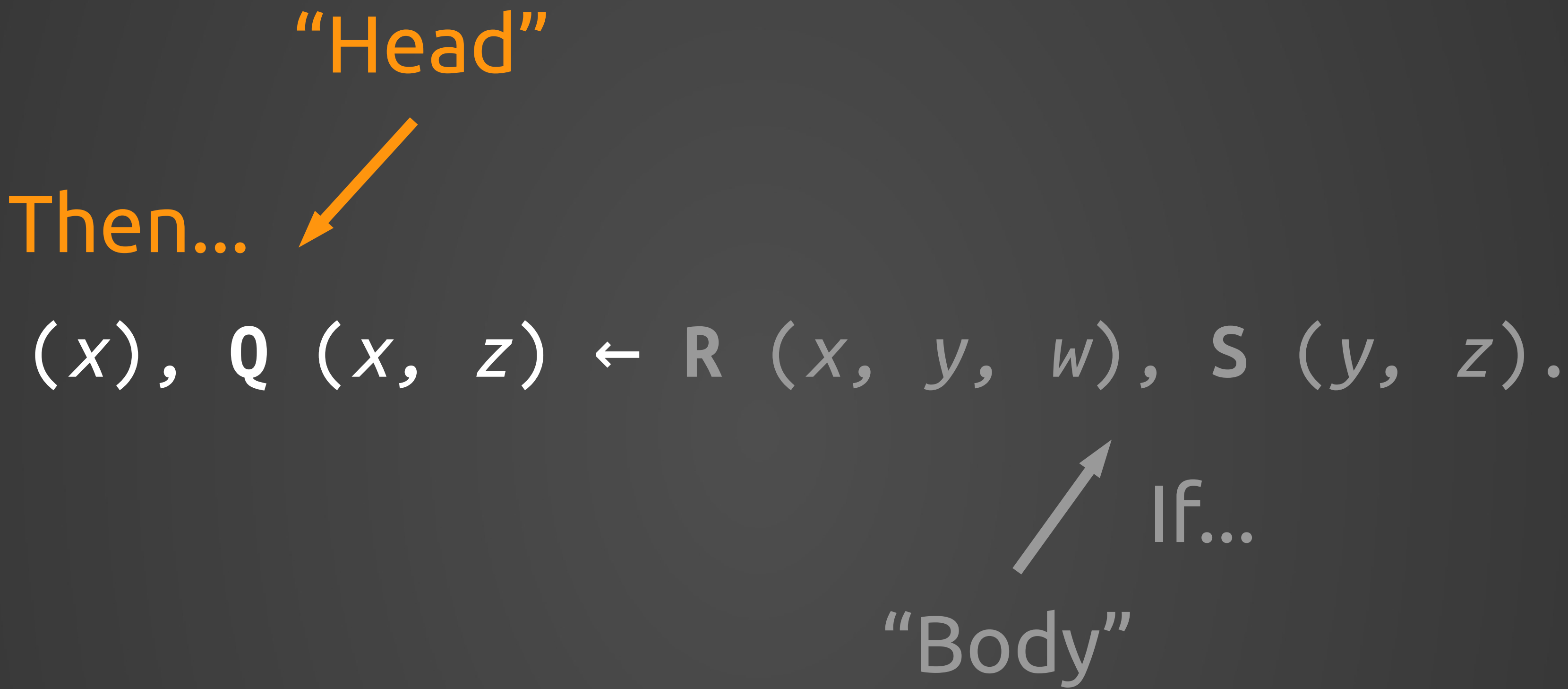
Example Rule

```
P (x), Q (x, z) ← R (x, y, w), S (y, z).
```

If…

“Body”

EXAMPLE RULE



“Head”

Then…

$$\mathbf{P}\ (x),\ \mathbf{Q}\ (x,\ z) \leftarrow \mathbf{R}\ (x,\ y,\ w),\ \mathbf{S}\ (y,\ z).$$

If…

“Body”

OBJECT ALLOCATION

```
var = new ...
```

REACHABLE (*meth*, *ctx*), ALLOC (*var*, *obj*, *meth*).

Object Allocation

```
var = new ...
```

```
VarPointsTo (var,       obj          ) ←
   Reachable (meth, ctx), Alloc (var, obj, meth).
```

OBJECT ALLOCATION

```
var = new ...
```

```
VARPOINTSTO (var, ctx, obj        ) ←
   REACHABLE (meth, ctx), ALLOC (var, obj, meth).
```

Variables share context with their methods

OBJECT ALLOCATION

```
var = new ...
```

Construct a new object context

```
RECORD (...) = objCtx,
VARPOINTSTO (var, ctx, obj, objCtx) ←
   REACHABLE (meth, ctx), ALLOC (var, obj, meth).
```

Local Assignment

```
to = from
```

```
MOVE (to, from), VARPOINTSTO (from, ctx, obj, objCtx).
```

Local Assignment

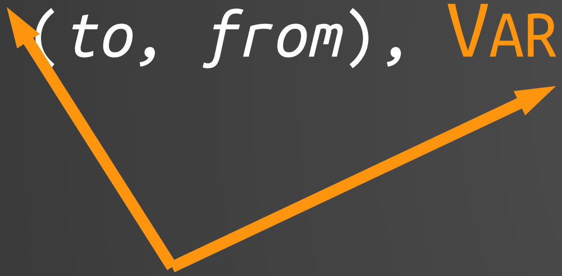
```
to = from
```

```
VarPointsTo (to, ctx, obj, objCtx) ←
   Move (to, from), VarPointsTo (from, ctx, obj, objCtx).
```

Local Assignment

```
to = from
```

```
VarPointsTo (to, ctx, obj, objCtx) ←
   Move (to, from), VarPointsTo (from, ctx, obj, objCtx).
```



Recursion

Efficient and Effective
Handling of Exceptions
in Java Points-To Analysis

INPUT

THROW (*instr*, *e*)
CATCH (*objT*, *instr*, *arg*)

Input

Throw (*instr*, *e*)
Catch (*objT*, *instr*, *arg*)

Output

ThrowPointsTo (*meth*, *ctx*, *obj*, *objCtx*)

INPUT

```
THROW (instr, e)
CATCH (objT, instr, arg)
```

OUTPUT

```
THROWPOINTSTO (meth, ctx, obj, objCtx)
```

Exception Analysis and Points-to Analysis:Better Together
Bravenboer – Smaragdakis

ISSTA'09

"On the fly" handling of exceptions

EXCEPTION RULES

```
void meth() {
   ...
   throw e;
   ...
}
```

THROW (*instr*, *e*)

Exception Rules

```
void meth() {
    ...
    throw e;
    ...
}
```

Throw (*instr*, *e*), VarPointsTo (*e*, *ctx*, *obj*, *objCtx*), ObjType (*obj*, *objT*)

EXCEPTION RULES

```
void meth() {
    ...
    throw e;
    ...
}
```

THROW (*instr*, *e*), VARPOINTSTO (*e*, *ctx*, *obj*, *objCtx*), OBJTYPE (*obj*, *objT*), ¬CATCH (*objT*, *instr*, _)

Exception Rules

```
void meth() {
   ...
   throw e;
   ...
}
```

```
ThrowPointsTo (meth, ctx, obj, objCtx) ←
   Throw (instr, e), VarPointsTo (e, ctx, obj, objCtx),
   ObjType (obj, objT), ¬Catch (objT, instr, _),
   InMethod (instr, meth).
```

Exception Rules

```
void meth() {
    try {
        throw e;
    }
    catch (objT arg) {...}
}
```

```
Throw (instr, e), VarPointsTo (e, ctx, obj, objCtx),
ObjType (obj, objT), Catch (objT, instr, arg).
```

Exception Rules

```
void meth() {
    try {
        throw e;
    }
    catch (objT arg) {...}
}
```

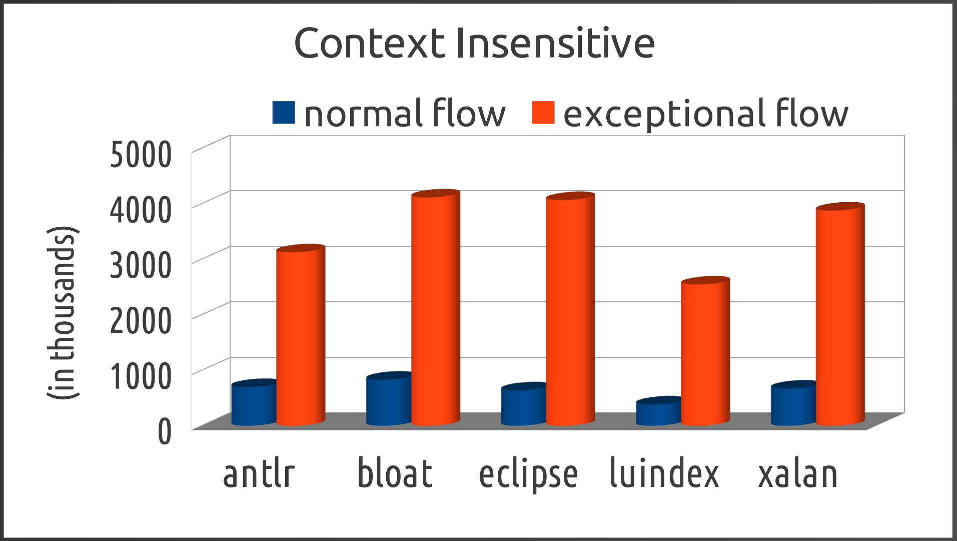
VarPointsTo (*arg*, *ctx*, *obj*, *objCtx*) ←
    Throw (*instr*, *e*), VarPointsTo (*e*, *ctx*, *obj*, *objCtx*),
    ObjType (*obj*, *objT*), Catch (*objT*, *instr*, *arg*).

Exception Rules

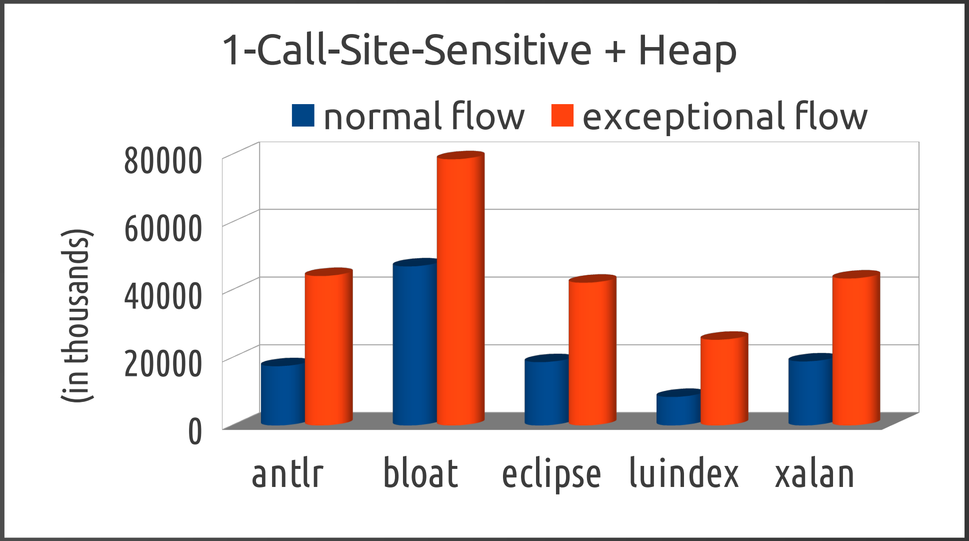
Same logic for method invocation

Is It Enough?

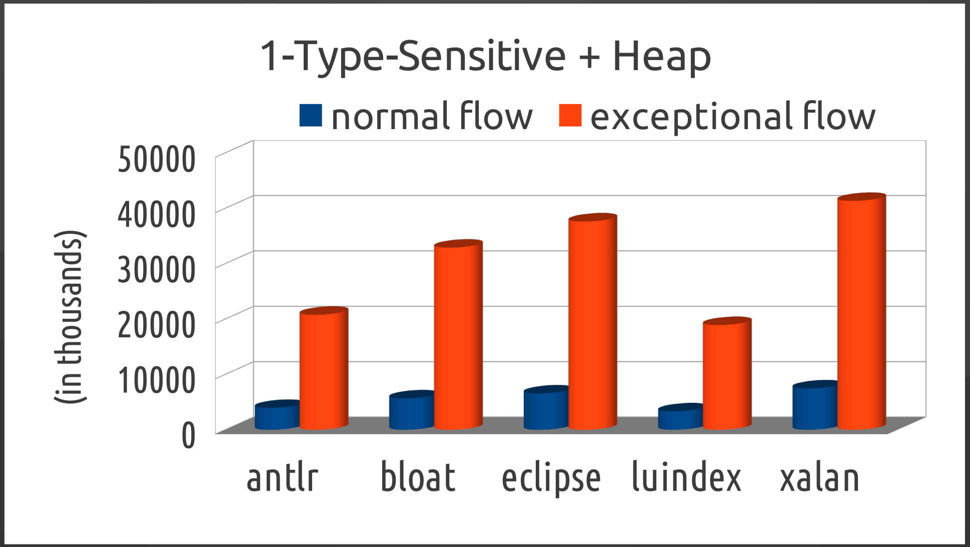
Normal Flow vs Exceptional Flow



Context Insensitive



1-Call-Site-Sensitive + Heap



1-Type-Sensitive + Heap

Efficient and Effective Handling of Exceptions in Java Points-To Analysis

Efficient and Effective Handling of Exceptions in Java Points-To Analysis

Coarsen Exceptions

Object Allocation

Recall...

```
Record (...) = objCtx,
VarPointsTo (var, ctx, obj, objCtx) ←
   Reachable (meth, ctx), Alloc (var, obj, meth).
```

Filter Out Exceptions

Change to...

```
Record (...) = objCtx,
VarPointsTo (var, ctx, obj, objCtx) ←
   Reachable (meth, ctx), Alloc (var, obj, meth),
   ObjType (obj, objT), ¬ExceptionType (objT).
```

HANDLING EXCEPTIONS

REACHABLE (*meth*, *ctx*), ALLOC (*var*, *obj*, *meth*),
OBJTYPE (*obj*, *objT*), EXCEPTIONTYPE (*objT*).

Handling Exceptions

```
VarPointsTo (var, ctx, obj   ) ←
   Reachable (meth, ctx), Alloc (var, obj, meth),
   ObjType (obj, objT), ExceptionType (objT).
```

A. Context Insensitive Exceptions

```
VarPointsTo (var, ctx, obj, ?) ←
    Reachable (meth, ctx), Alloc (var, obj, meth),
    ObjType (obj, objT), ExceptionType (objT).
```

A. Context Insensitive Exceptions

Single Context

```
VarPointsTo (var, ctx, obj, “ConstantObjCtx”) ←
    Reachable (meth, ctx), Alloc (var, obj, meth),
    ObjType (obj, objT), ExceptionType (objT).
```

B. Merge Exceptions

Not enough
Get more aggressive

```
VarPointsTo (var, ctx, obj, “ConstantObjCtx”) ←
   Reachable (meth, ctx), Alloc (var, obj, meth),
   ObjType (obj, objT), ExceptionType (objT).
```

B. Merge Exceptions

VarPointsTo (*var*, *ctx*, ~~*obj*~~, *“ConstantObjCtx”*) ←
Reachable (*meth*, *ctx*), Alloc (*var*, *obj*, *meth*),
ObjType (*obj*, *objT*), ExceptionType (*objT*).

B. Merge Exceptions

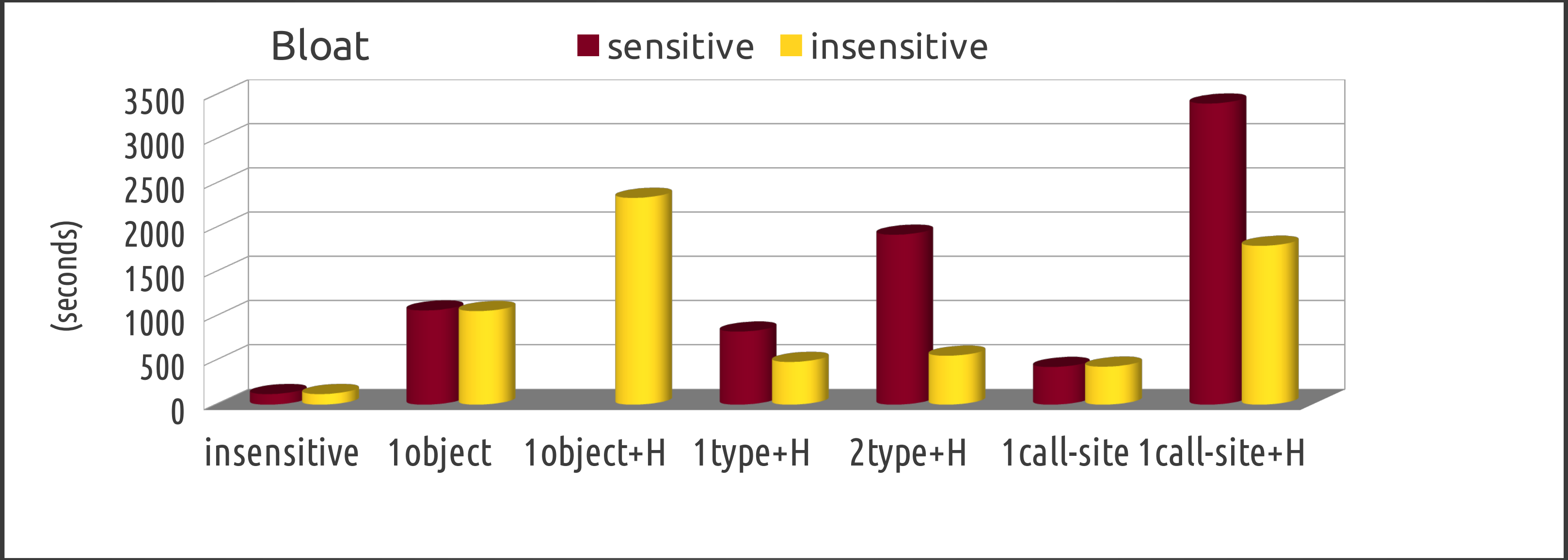
```
VarPointsTo (var, ctx, reprObj, “ConstantObjCtx”) ←
   Reachable (meth, ctx), Alloc (var, obj, meth),
   ObjType (obj, objT), ExceptionType (objT),
   Representative (obj, reprObj).
```

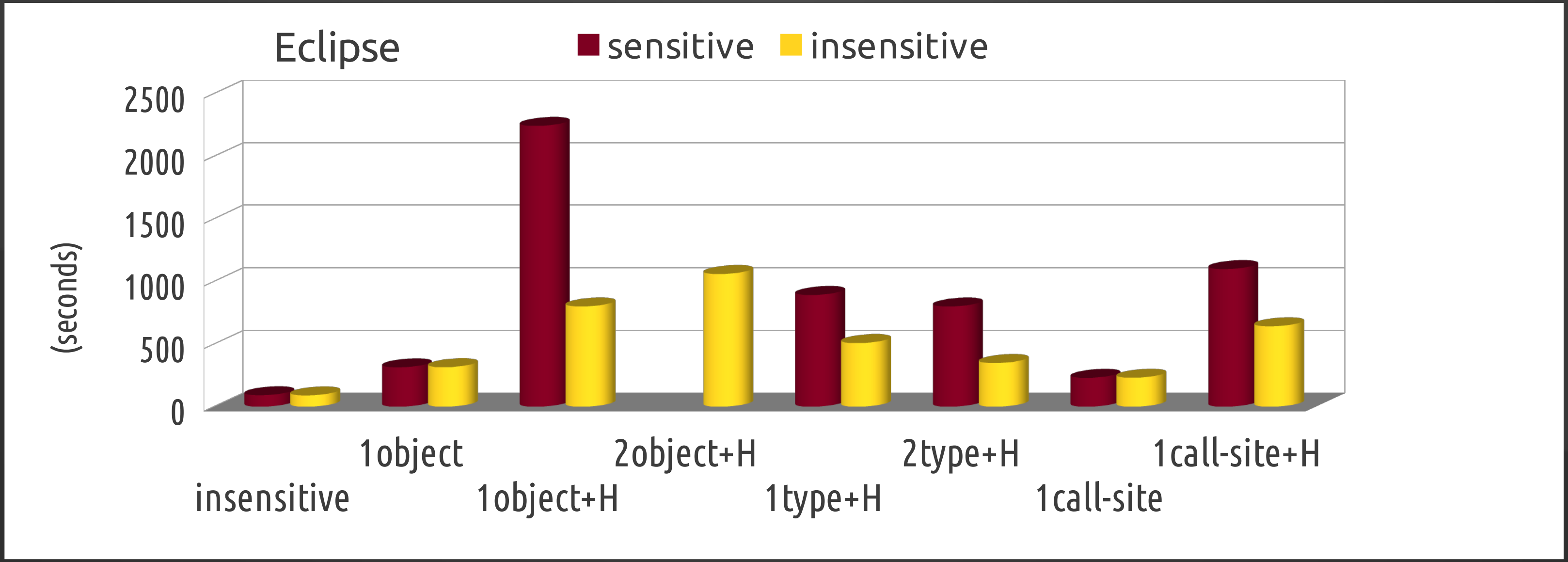
Experiments

EXPERIMENTS

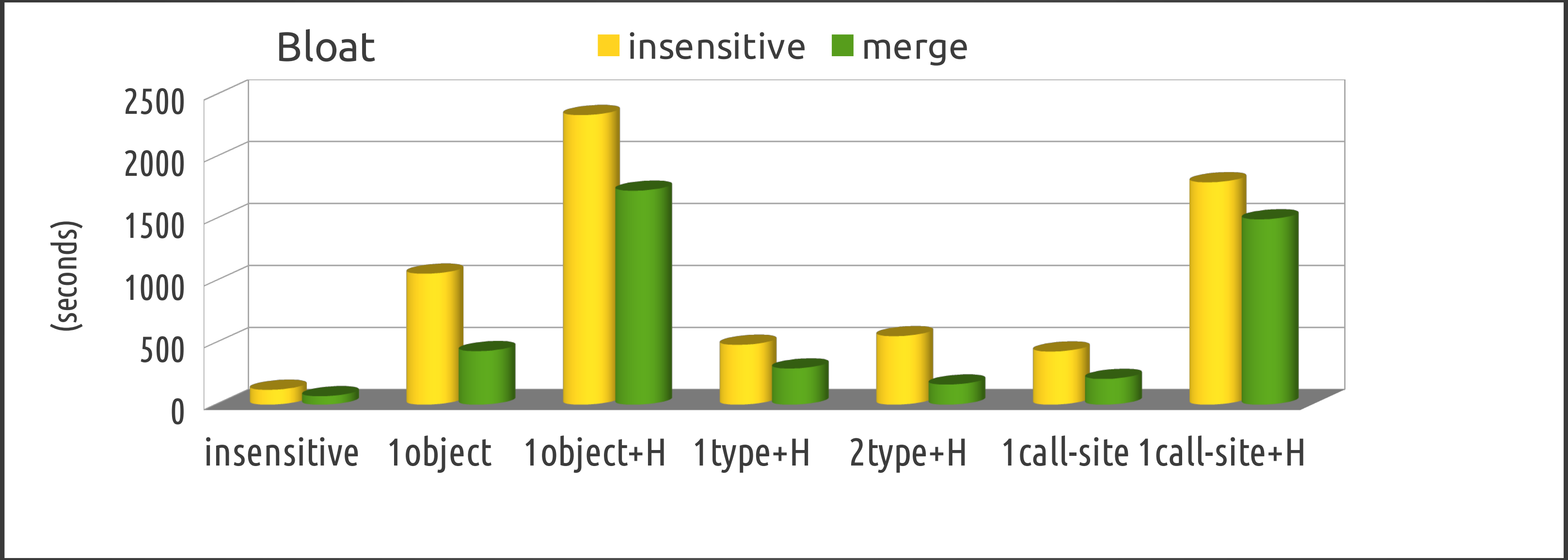
JDK 1.6

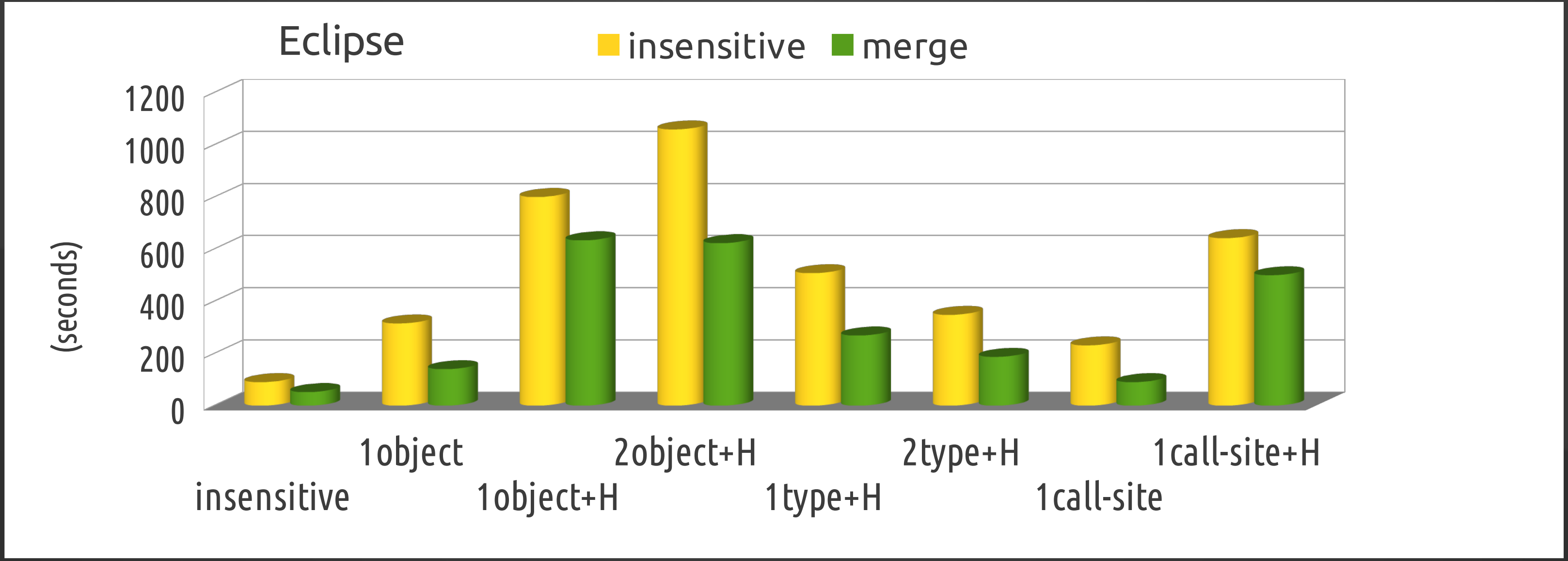
Context Insensitive Exceptions



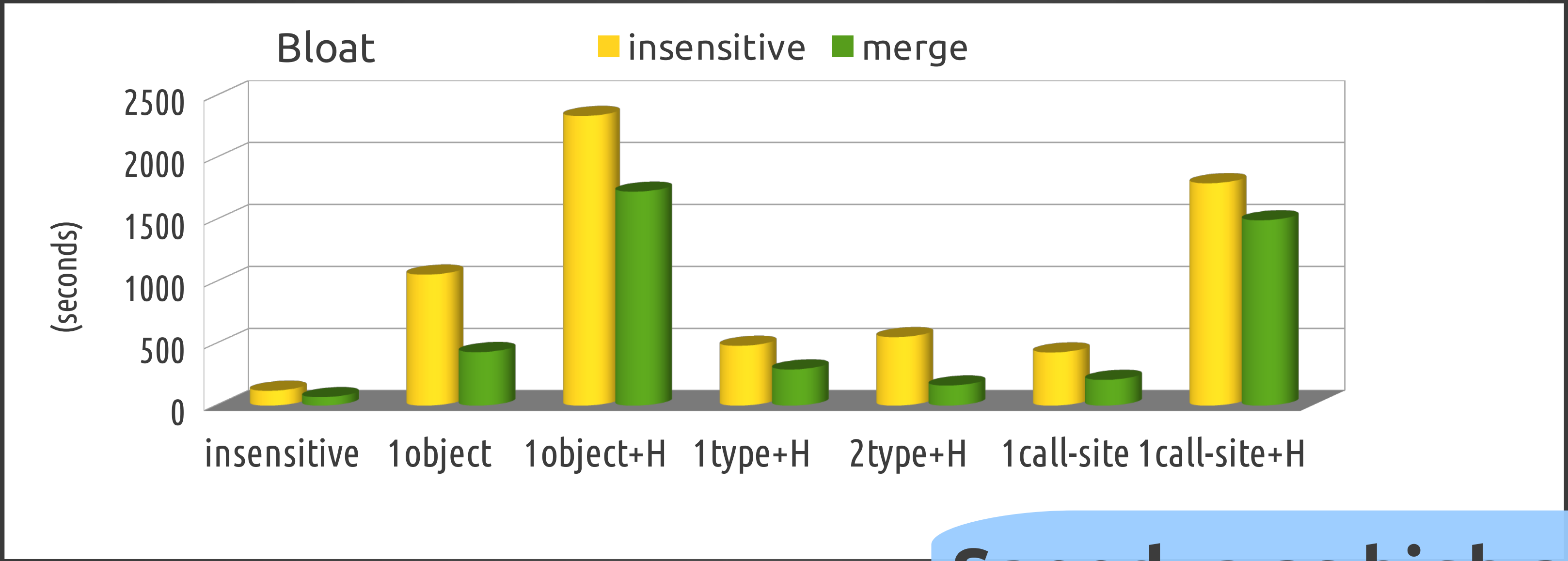


MERGE EXCEPTIONS

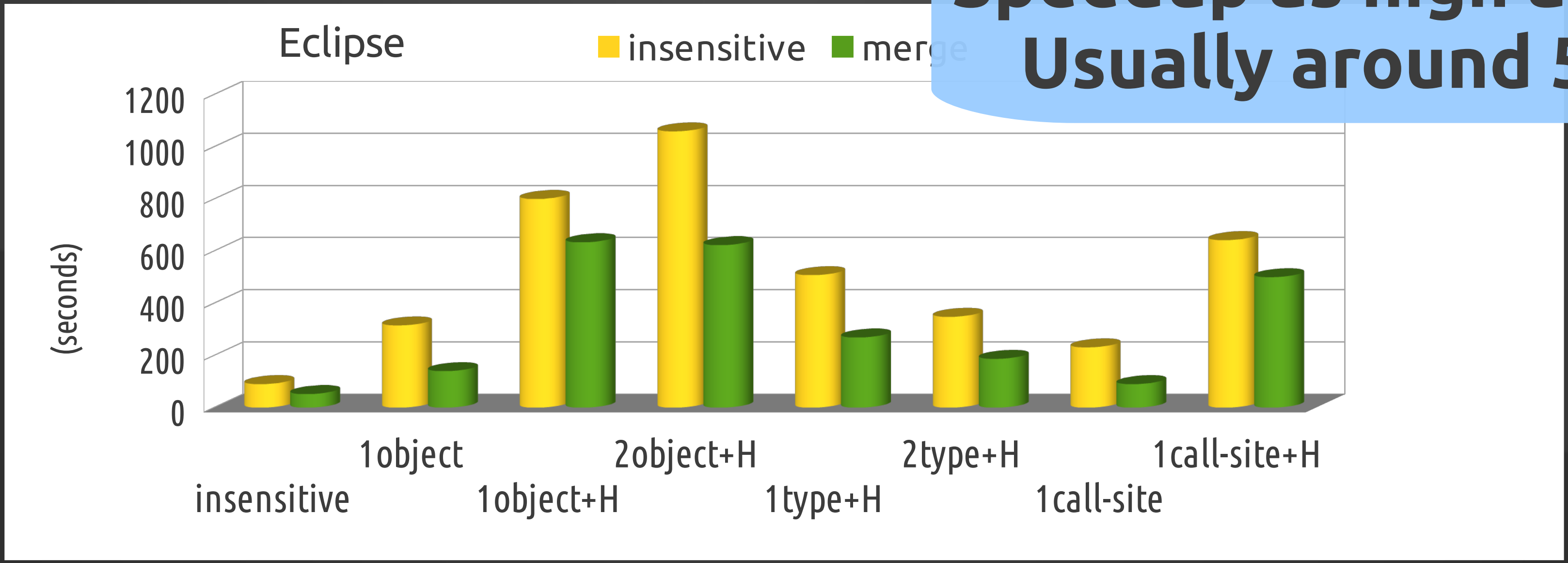




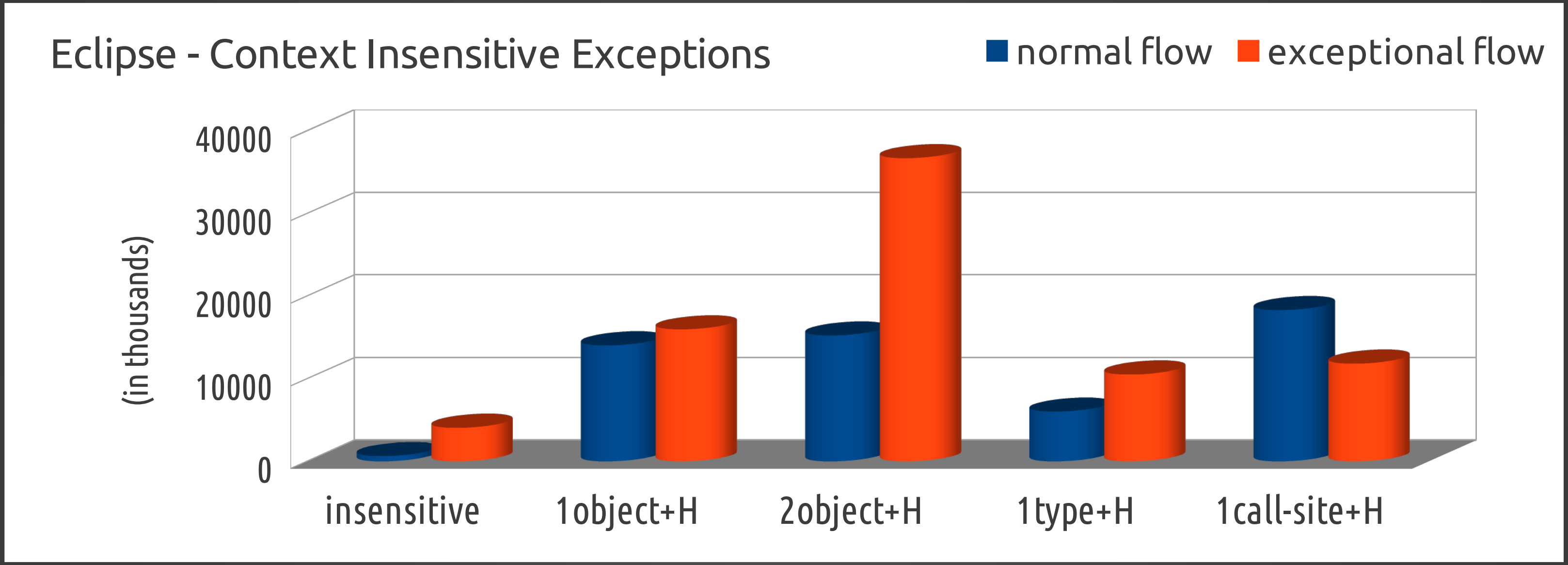
MERGE EXCEPTIONS



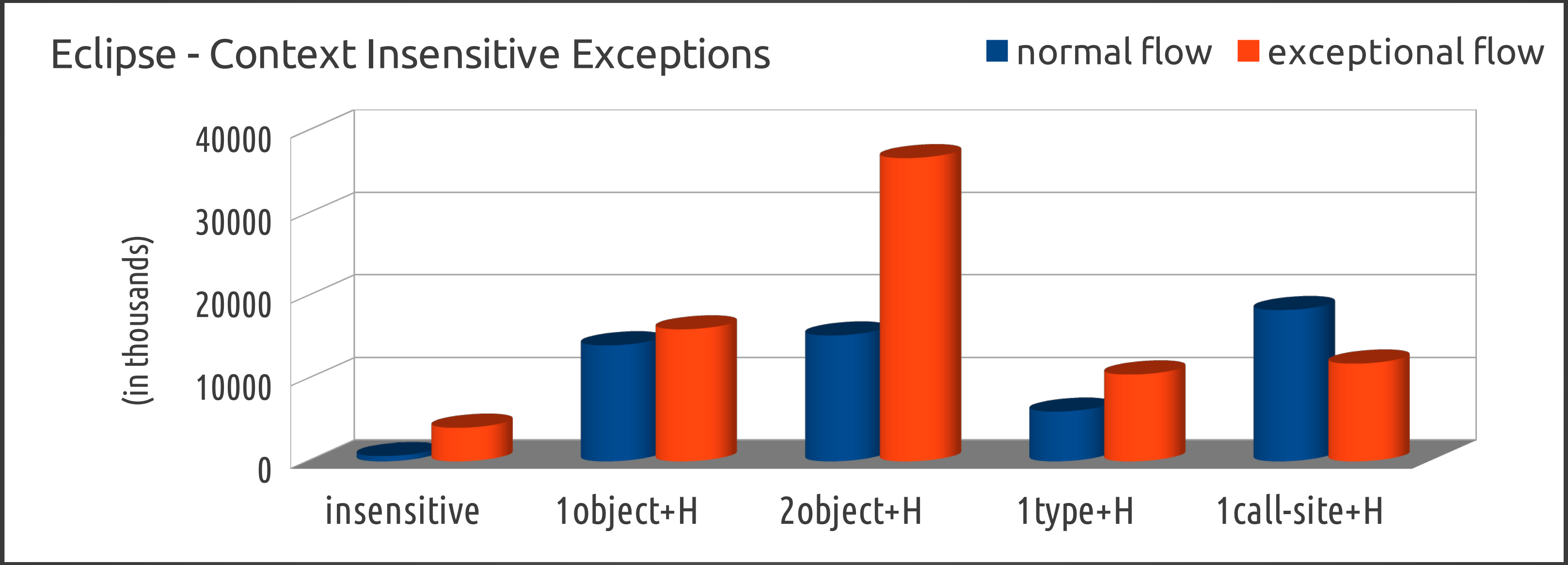
Speedup as high as 70%
Usually around 50%

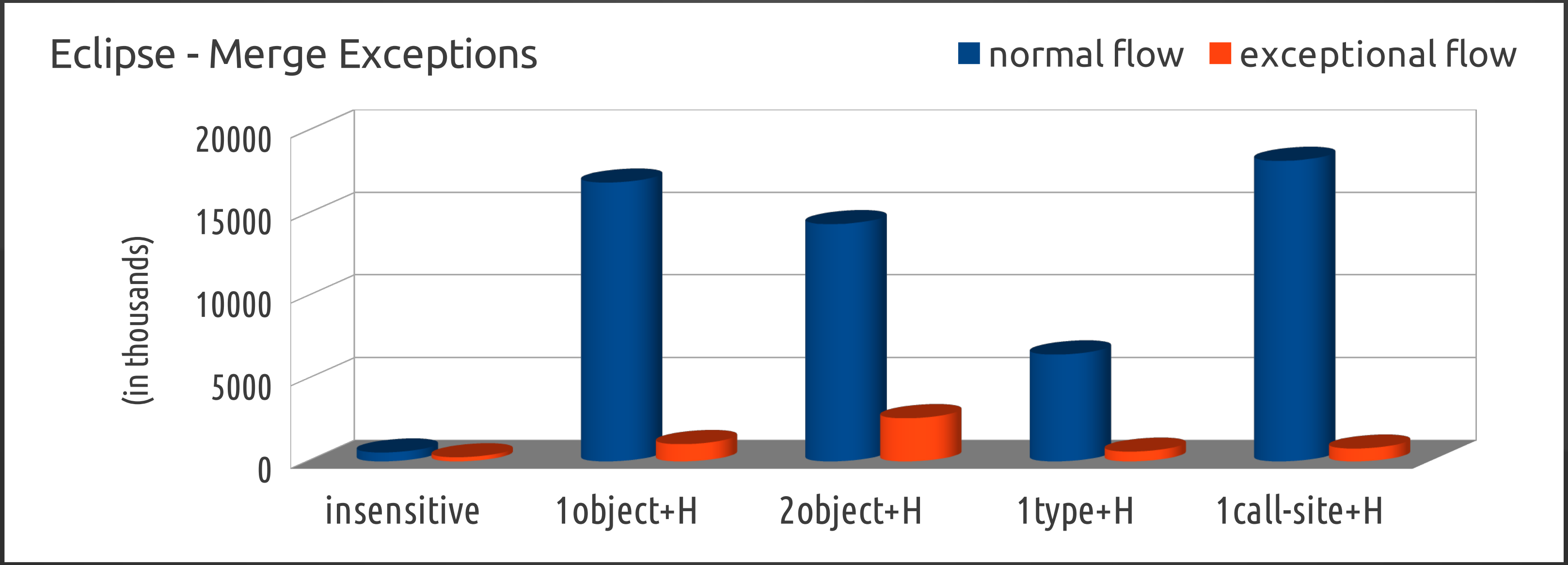


NORMAL FLOW vs EXCEPTIONAL FLOW



NORMAL FLOW vs EXCEPTIONAL FLOW





pre·ci·sion

RECAP

- Huge amount of analysis time spent on exceptions

Recap

- Huge amount of analysis time spent on exceptions
- They mainly affect control-flow

RECAP

- Huge amount of analysis time spent on exceptions
- They mainly affect control-flow
- Significant speedup from coarsening exceptions

Recap

- Huge amount of analysis time spent on exceptions
- They mainly affect control-flow
- Significant speedup from coarsening exceptions
- Type-based merging as an effective coarsening

RECAP

- Huge amount of analysis time spent on exceptions
- They mainly affect control-flow
- Significant speedup from coarsening exceptions
- Type-based merging as an effective coarsening
- No trade-off in precision (in “normal” code)

RECAP

- Huge amount of analysis time spent on exceptions
- They mainly affect control-flow
- Significant speedup from coarsening exceptions
- Type-based merging as an effective coarsening
- No trade-off in precision (in “normal” code)
- Datalog formalism makes changes clear

RECAP

- Huge amount of analysis time spent on exceptions
- They mainly affect control-flow
- Significant speedup from coarsening exceptions
- Type-based merging as an effective coarsening
- No trade-off in precision (in “normal” code)
- Datalog formalism makes changes clear
- Also excellent implementation platform

Hope you enjoyed!

George Kastrinis • http://gkastrinis.info