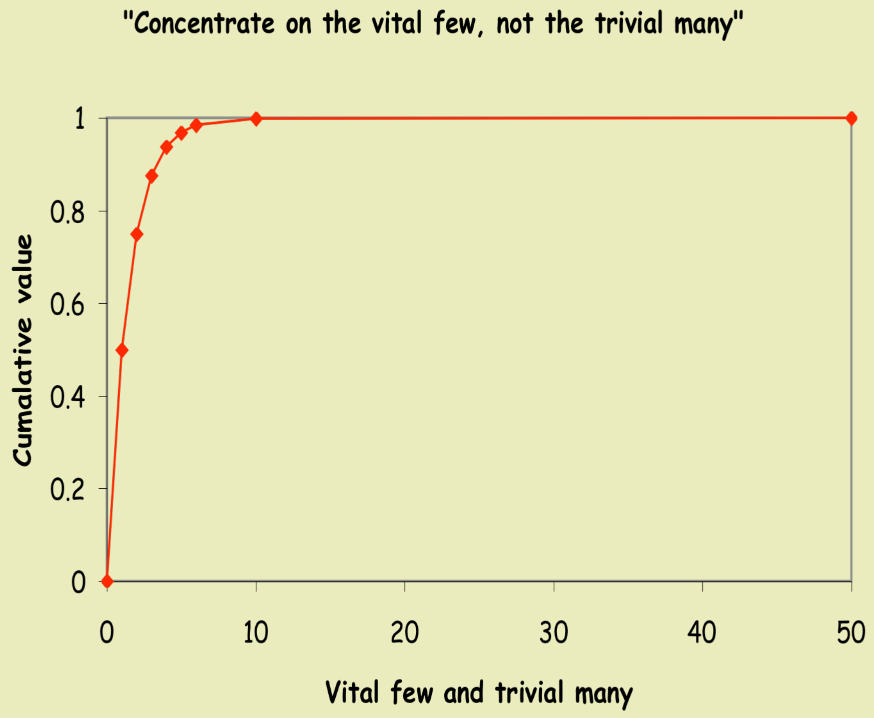
Finding and Removing Performance Bottlenecks in Large Systems

Glenn Ammons, IBM

Jong-Deok Choi, IBM

Manish Gupta, IBM

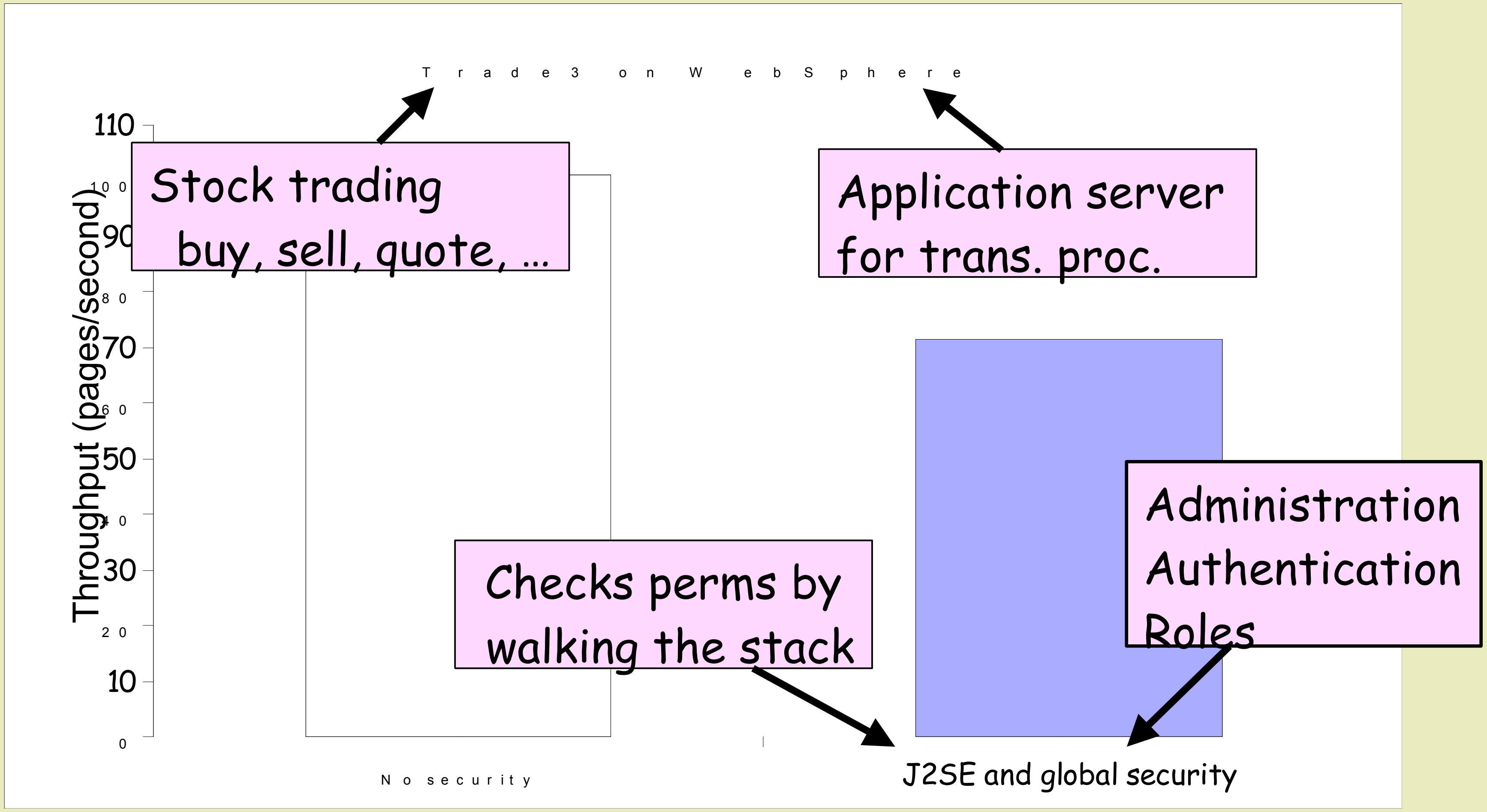
Nikhil Swamy, UMD



Background: "security" (in this talk)

- Permission checking (J2SE)
  - "May this code perform this operation?"
  - Keywords: checkPermission, doPrivileged, was.policy
- Global security (includes J2EE)
  - Authentication; administration
  - "May this user call this method?"
    - What roles are permitted to call this method?
    - What roles does the user play?
  - Keywords: isGrantedAnyRole, credentials, subject, role
- NOT web services, ssh, firewalls, …

Background: overhead of security



Preview of results

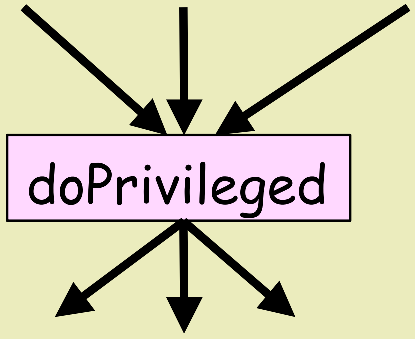
- Found bottlenecks (14 for 83% of 30% overhead)
  - Tough: 2 million LOC, 10000 classes, 800 J2SE security call sites; deep call stacks (avg. 64 methods)
- Path-based approach to finding bottlenecks
  - Bottleneck == path
  - General: interface separates analysis from profile
  - Overlap --> accurate speedup estimates
  - Interactive, extensible tool; also helps validate opts
- Four simple optimizations (down to 15% overhead)
  - Exploit redundancy
    - Temporal (data) 📬📬✇ caching
    - Spatial (code) 📬📬✇ specialization

Outline of the talk

- Finding bottlenecks
- Security optimizations
- Wrap it up
  - Related work
  - Future work

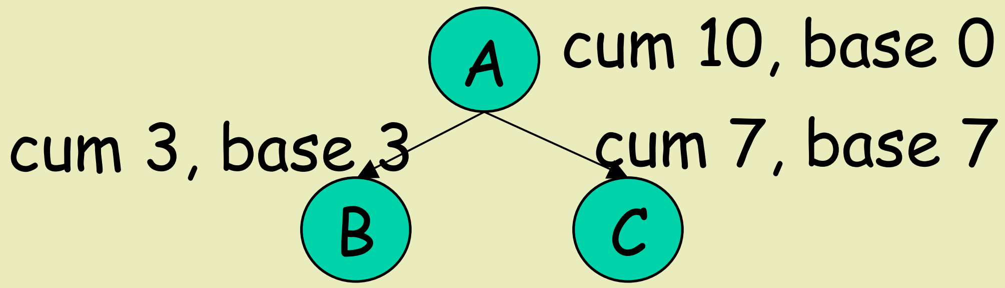
Finding bottlenecks

- Tough problem
    - 2 million LOC, 10000 classes, 800 J2SE security call sites; deep call stacks (avg. 64 methods)
- Need
    - Repeatable deployment
    - Accuracy
        - Many small slowdowns, spread throughout the code
    - Coverage
        - Short executions were inconsistent
        - Didn't know where to look
    - Context sensitivity



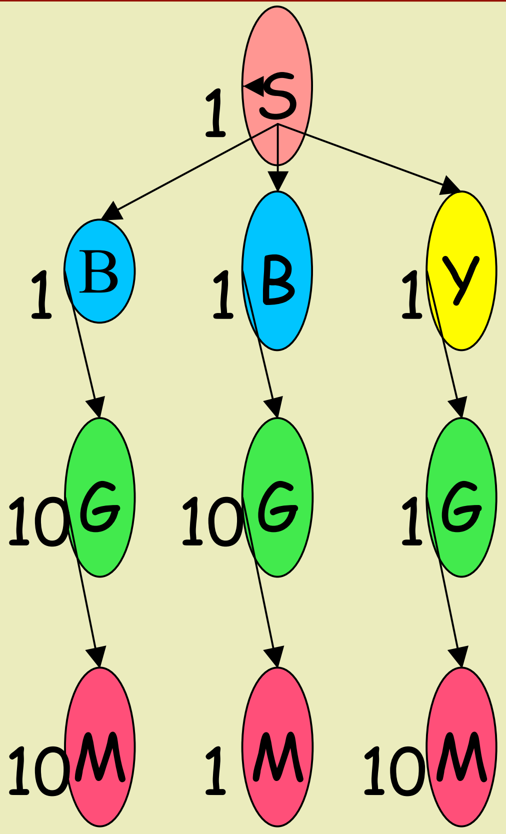
Measurement tools

- Repeatable deployment: wrote scripts
- Used ArcFlow to collect call-tree profiles

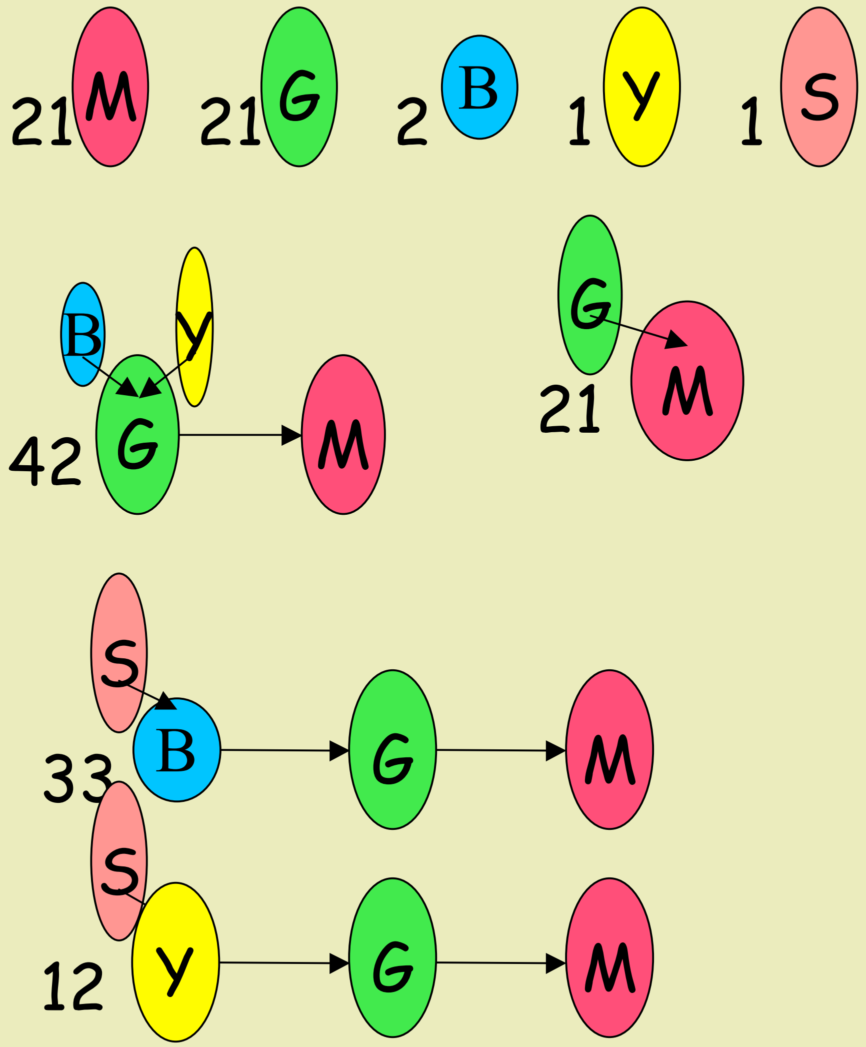


  - Accurate (instruction counts, not time)
  - Records context
  - Can profile over long execution intervals
  - Some flaws: no time, no edge profile

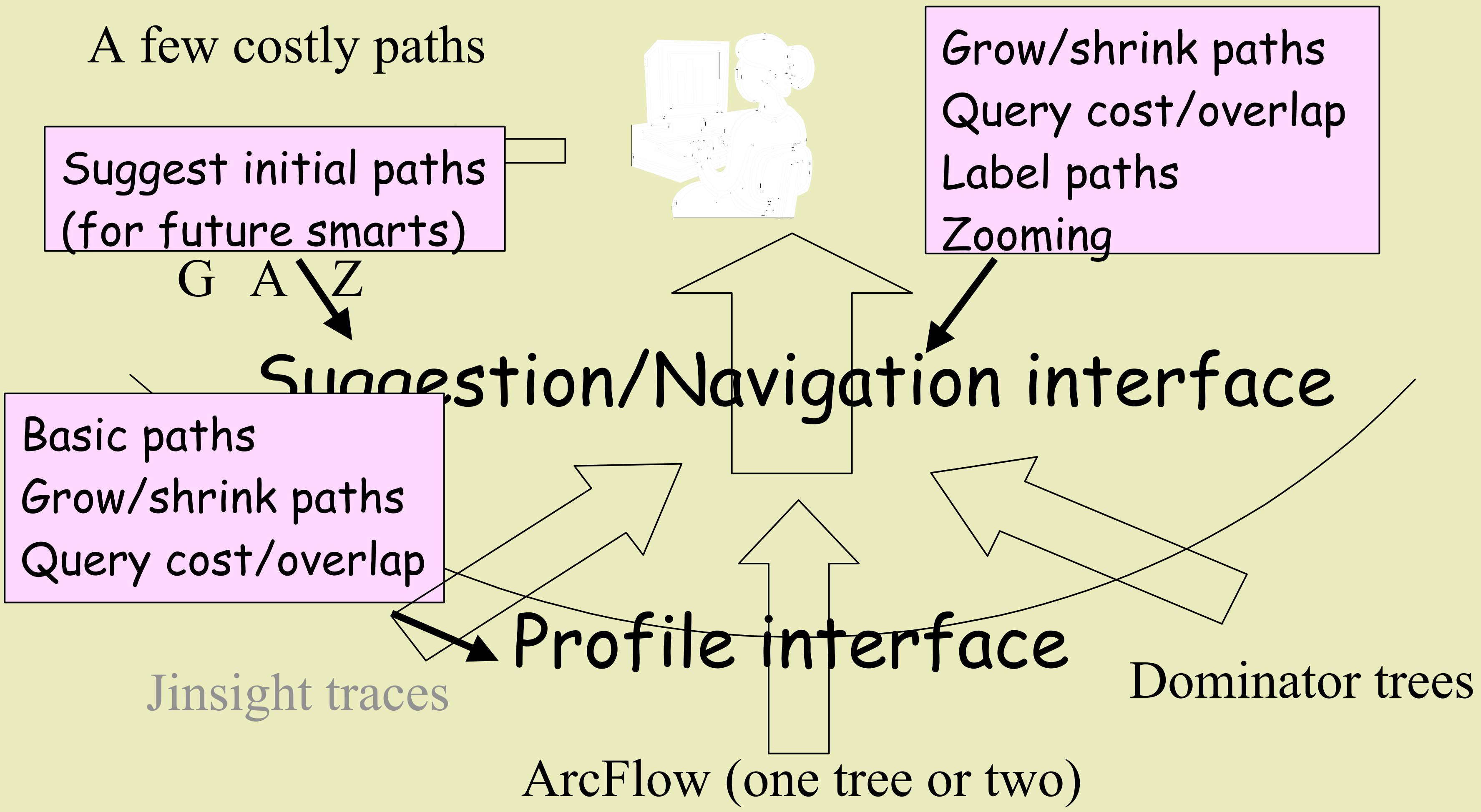
Analyzing a call-tree with paths



Expensive sum, but cheap parts!



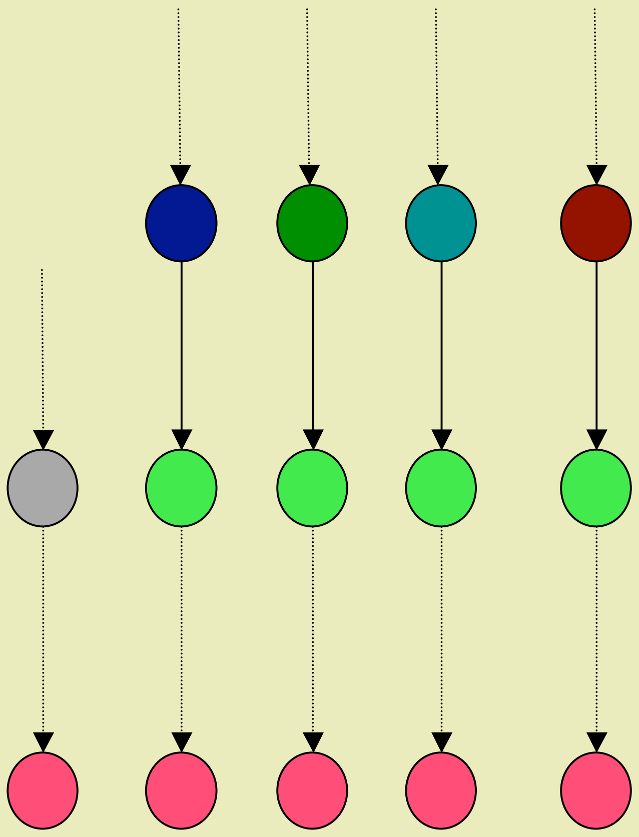
The path-based approach



An example

Set suggester to "by base"
Ask for first five suggestions

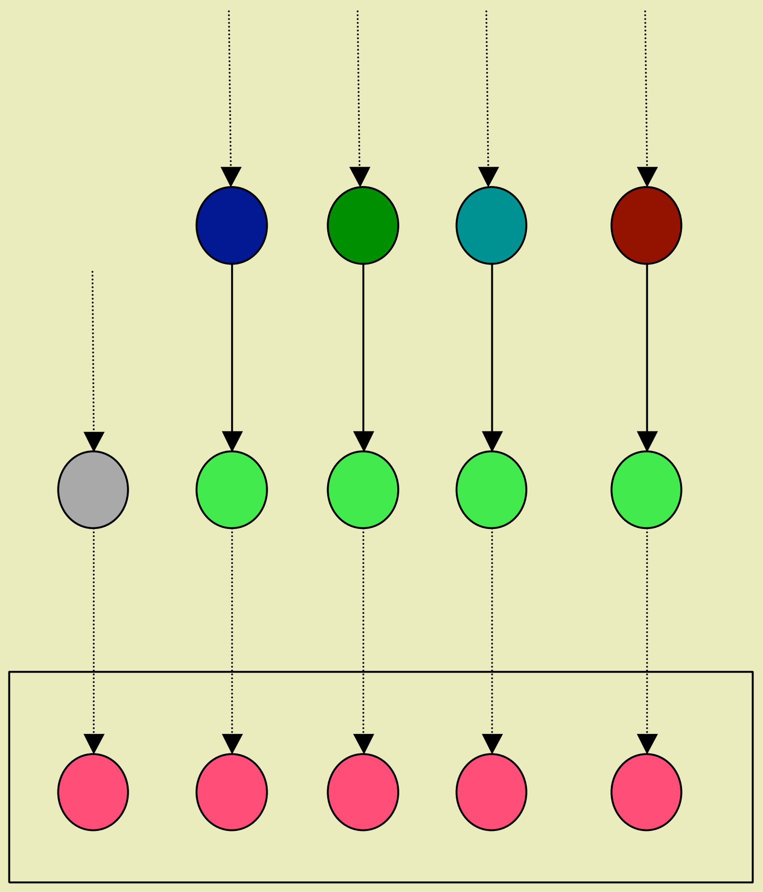
Result:
[0] getClassContext (22% of cost)
…



An example

After selecting getClassContext:

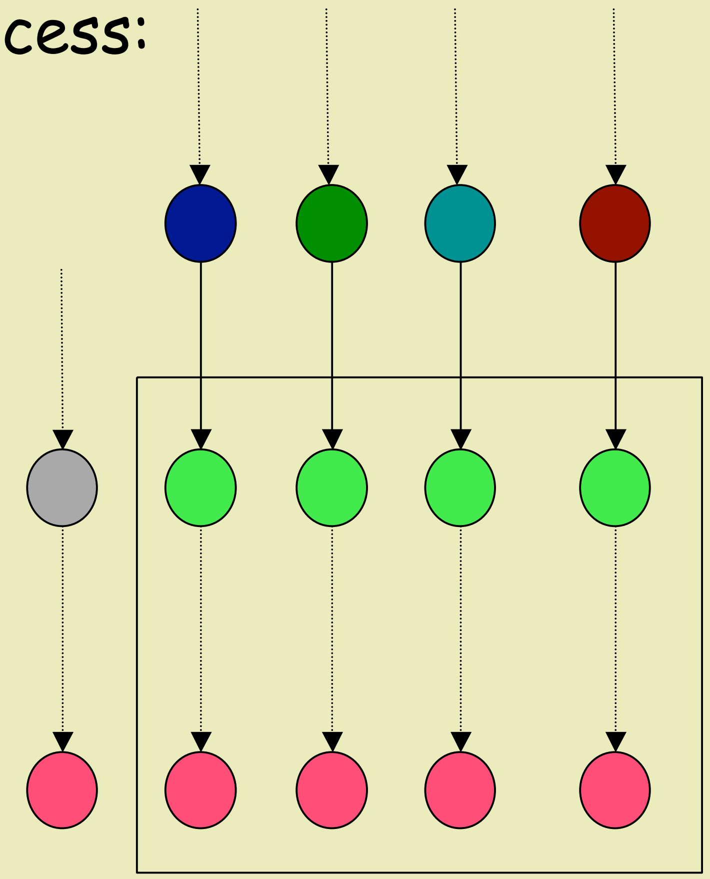
- Current:
    - getClassContext (22% of cost)
- Upwards extensions:
    - [0] execute (22% of cost)
        - <passes through 5 boring calls>
- Downwards extensions:



An example

After selecting checkMemberAccess:

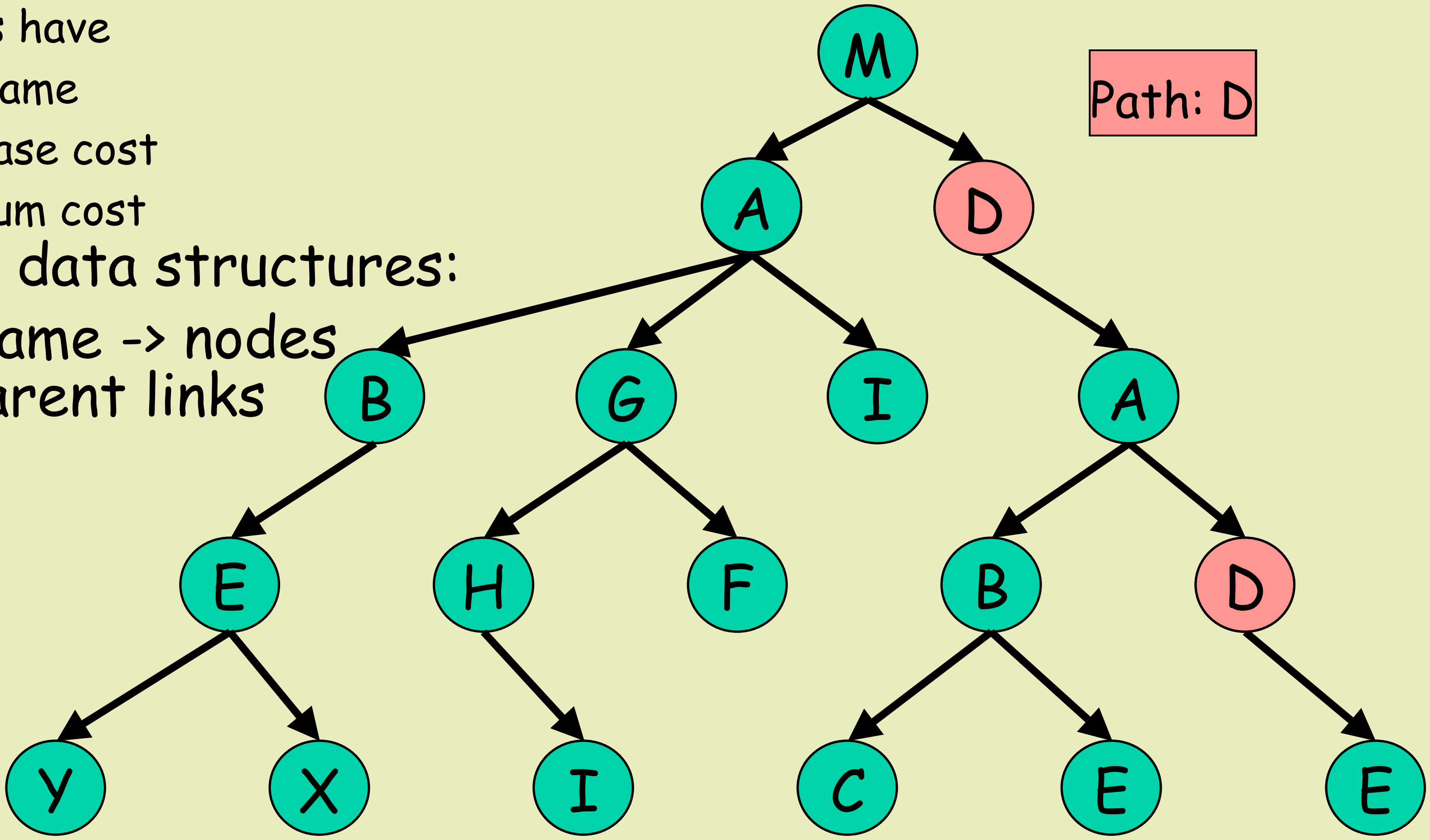
- Current:
  - execute (22% of cost)
  - setOutputProperties
  - doPrivileged
  - run
  - A.checkMemberAccess
  - B.checkMemberAccess
  - getClassContext
- Upwards extensions:
  - (Four Trade3 actions)
- Downwards extensions:
- Trimming the top: discards 6 of 7
- Trimming the bottom: discards 1 of 7



Cumulative costs in a call-tree profile

Nodes have
- Name
- Base cost
- Cum cost

Aux. data structures:
- Name -> nodes
- Parent links



Path: D

Experiments with Bottlenecks

App	Nodes	Bottlenecks			Cost to find	
	(1000s)	#	% Cost	Avg. len	Minutes	Cmds
Trade3 sec.	895	14	83%	14	32	151
SPECjAS20(	1096	13	36%	8.7	50	251
XML app.	24	13	89%	6.2	30	143

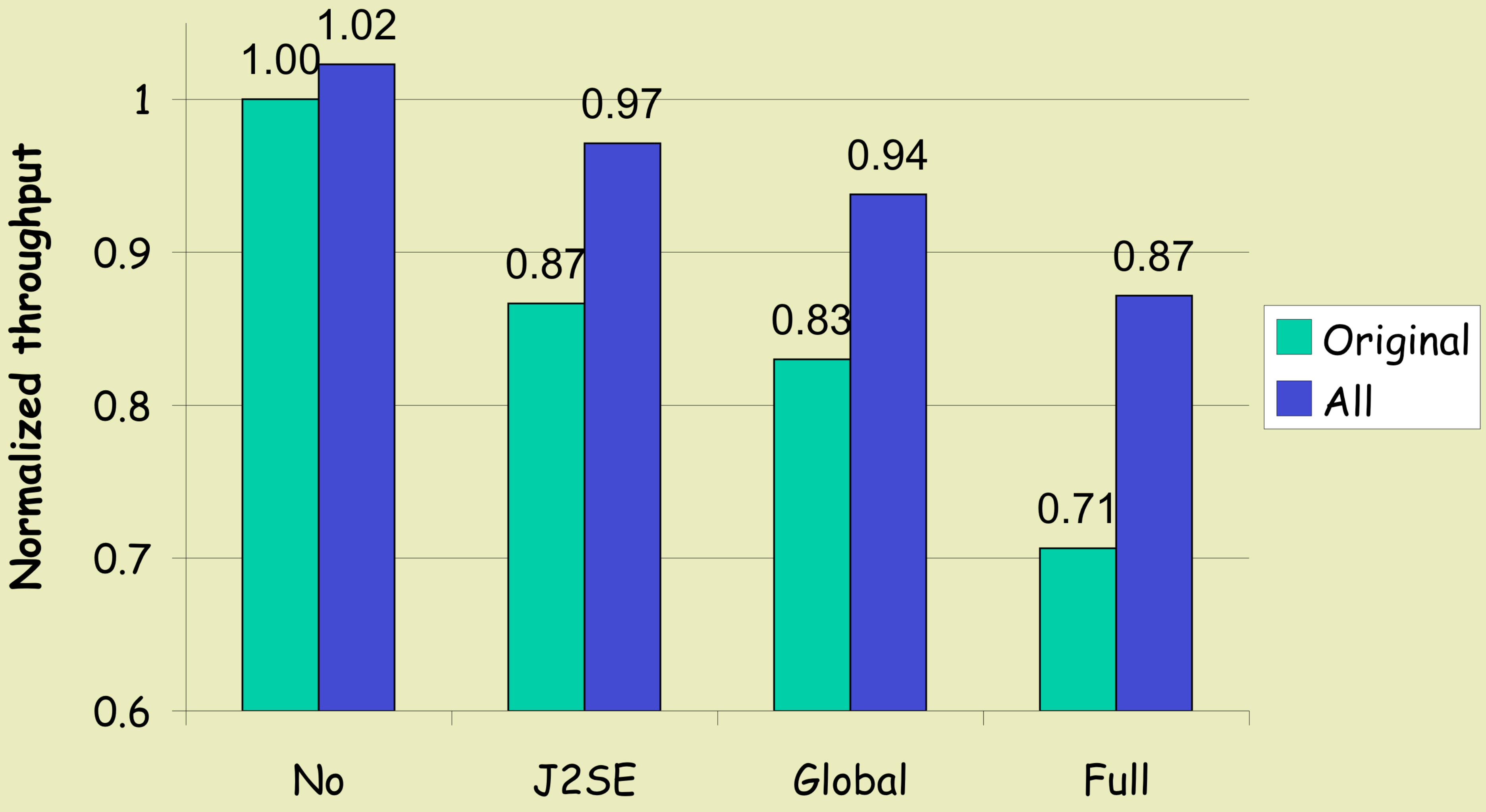
Notes:

- Time includes my think time (I am very fast).
- I knew biggest Trade3 bottlenecks already.
- SPECjAS2002 and XML app. were new to me.

Outline of the talk

- Finding bottlenecks
- Security optimizations
- Wrap it up
  - Related work
  - Future work

Throughput improvements



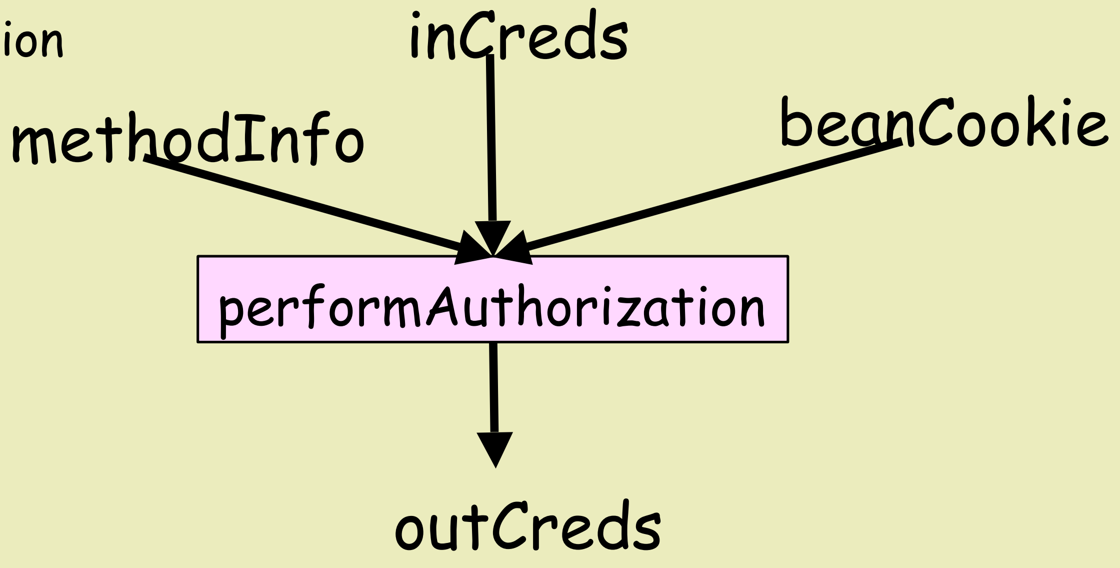
CheckRole (temporal)

Cost: 16% of 30% instruction-count overhead

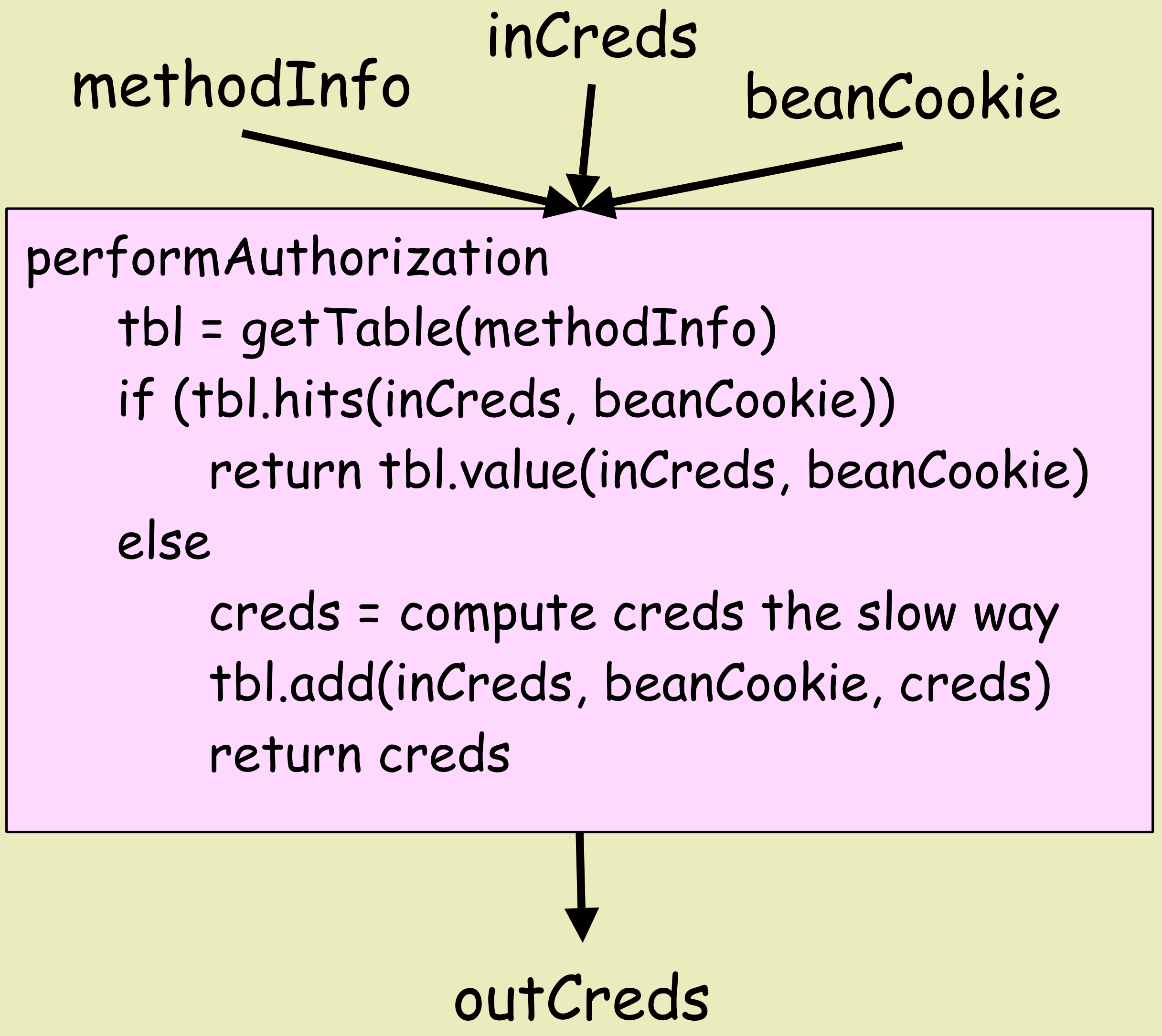
Path:

preInvoke
calls performAuthorization
calls ejbCheckAuthorization
calls checkAccess

Observation: decision is (almost) deterministic



CheckRole, optimized with cache



```
performAuthorization
    tbl = getTable(methodInfo)
    if (tbl.hits(inCreds, beanCookie))
        return tbl.value(inCreds, beanCookie)
    else
        creds = compute creds the slow way
        tbl.add(inCreds, beanCookie, creds)
        return creds
```

GetCred (spatial)

Cost: 13% of 30% instruction-count overhead

Path:

- preInvoke
- calls doPrivileged
- calls run
- calls get_credentials

```
preInvoke
    doPrivileged // Expensive!
        creds = get_credentials()
    if (!isGrantedAnyRole(roles, creds))
        Scream!

// In another class, far far away
public get_credentials()
    // Expensive!
    stack = getAccessContext()
    checkPermission(stack, canReadCreds)
    return creds
```

GetCred, optimized by specializing

```
private static boolean ok = false;
preInvoke
    if (ok) creds = get_credentialsQuickly()
    else doPrivileged       // Expensive!
                creds = get_credentials()
    if (!isGrantedAnyRole(roles, creds))
        Scream!

// In another class, far far away
public get_credentials()
    // Expensive!
    stack = getAccessContext()
    checkPermission(stack, canReadCreds)
    ok = true
    return creds
public get_credentialsQuickly()
    return creds
```

- Wide application
- Proposed for IBM JIT

Outline of the talk

- Finding bottlenecks
- Security optimizations
- Wrap it up
  - Related work
  - Future work

Related work

- Path profiling [Ball+Larus;Ammons,Ball,Larus;Larus]
- Call-path refinement profiles [Hall]
  - No overlap, call trees only, no comparison, nicer interface
- Hot-path browser [Ball,Larus,Rosay]
  - Visualizer for Ball-Larus;union/intersection/difference
- Interaction cost [Fields,Bodik,Hill,Newburn]
  - Microarch. bottlenecks;overlap=-1*interaction cost
- Paradyn/DeepStart
  - For big, parallel systems; on-line; automated search

Future work

- Other users/problems
  - look for natural experiments, like scaling problems
- Extensions to the tool
  - More operations on profiles, including global ops.
    - grouping methods by context (inserting holes in the tree)
    - grouping methods by package/purpose/etc.
  - More smarts
    - a programmatic interface
    - more expressive paths
- Other profilers
  - For time
  - For usability (for example, no kernel patches)
  - For other domains (network logs, process trees, ...)

Backup Slides

DBReuse

Cost: 10% of 30% instruction-count overhead

Paths:

getConnection
calls <5 methods>
calls doPrivileged
calls <2 methods>
calls Subject.equals

getConnection
calls <2 methods>
calls getSubject
calls doPrivileged

getConnection
calls <7 methods>
calls Subject.hashCode

Optimization:

Cache results of equals(), hashCode(), getSubject()

Reflection

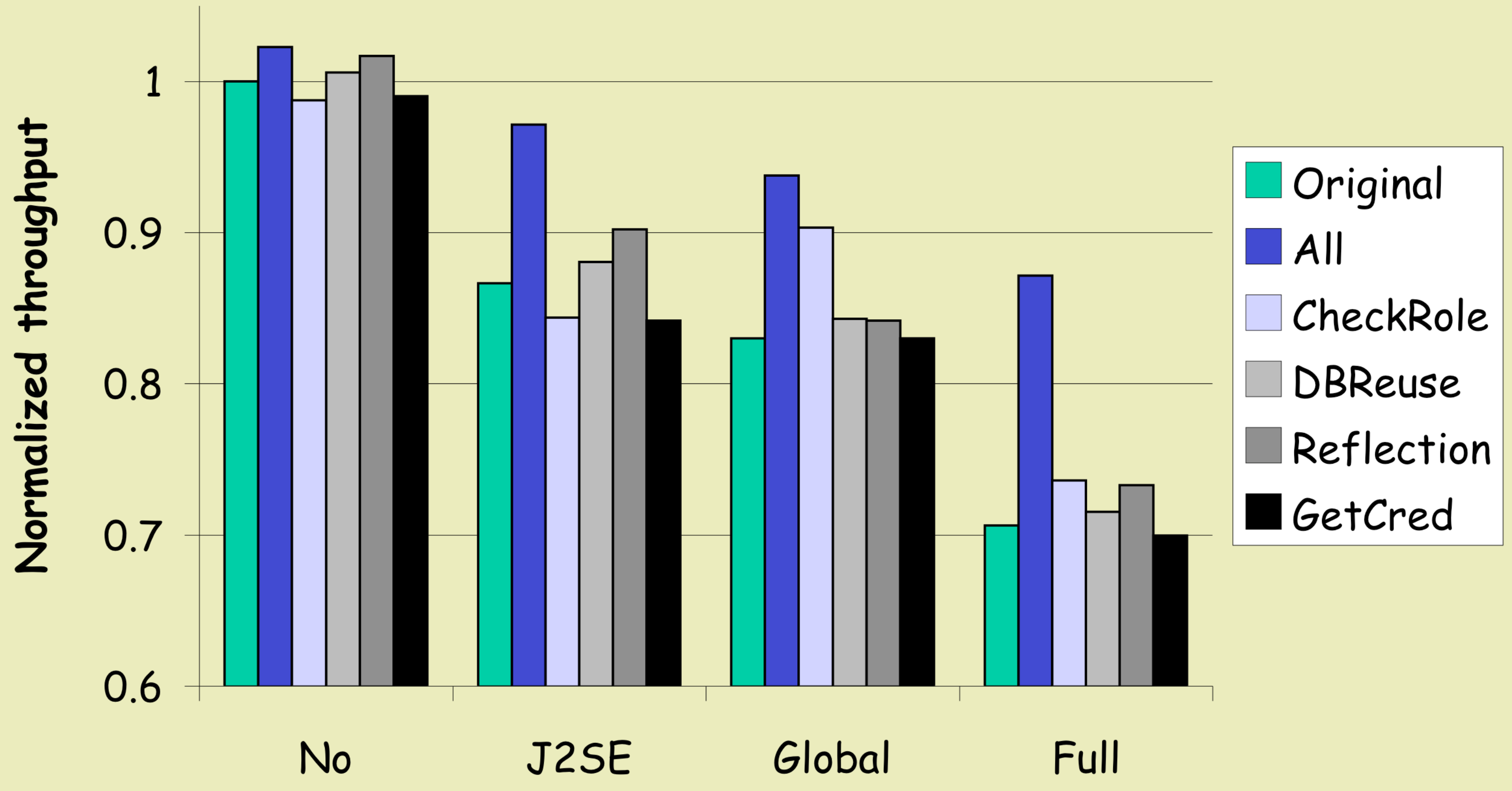
Cost: 25% of 30% instruction-count overhead

Path:

CacheableCommandImpl.execute
calls CacheableCommandImpl.setOutputProperties // uses reflection!
calls doPrivileged
calls run
calls checkMemberAccess

Optimization: avoid reflection by overriding setOutputProperties

Throughput improvements, by optimization


Normalized throughput
No
J2SE
Global
Full
Original
All
CheckRole
DBReuse
Reflection
GetCred