Deadlock Immunity

Enabling Systems to Defend Against Deadlocks

Anastasia Marinakou sdi2400120

Original research and work by: Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, George Candea,
Dependable Systems Laboratory, EPFL, Switzerland

The Problem

- Modern code uses concurrency
- *Synchronization* is complex
- Complex code = bug-prone

The Problem

- Modern code uses concurrency
- *Synchronization* is complex
- Complex code = bug-prone

- Suppose 2 concurrent threads, A and B
    - A needs a lock that B is holding
    - B needs a lock that A is holding
    - Both depend on each other and cannot continue or stop waiting
    - Result: no progress AKA **Deadlock** (unrecoverable)

Solutions?

1) Use type-strict languages

- ✓ simpler concurrent-based code
- ✓ no runtime overhead
- ✓ no deadlocks
- ✗ much harder for programmers to use

Solutions?

2) Transactional Memory (TM)

✓ locking order problem → thread scheduling problem

✓ programmer burden → dealt with at runtime

✗ problem with TM semantics; cannot solve all concurrency problems

Solutions?

3) Static scheduling

✓ pre-synchronization

✓ constraints in form of thread priority

✓ safety and liveliness

✗ not enough information ahead of time to do it efficiently

Solutions?

4) Static analysis tools

✓ find deadlock bugs at compile time

✓ fast

✗ can generate false positives

Solutions?

5) Model checkers (automated formal verification)

✓ check all thread interleavings

✓ high coverage

✗ does not scale well for general purpose systems

Solutions?

6) Dynamic approaches

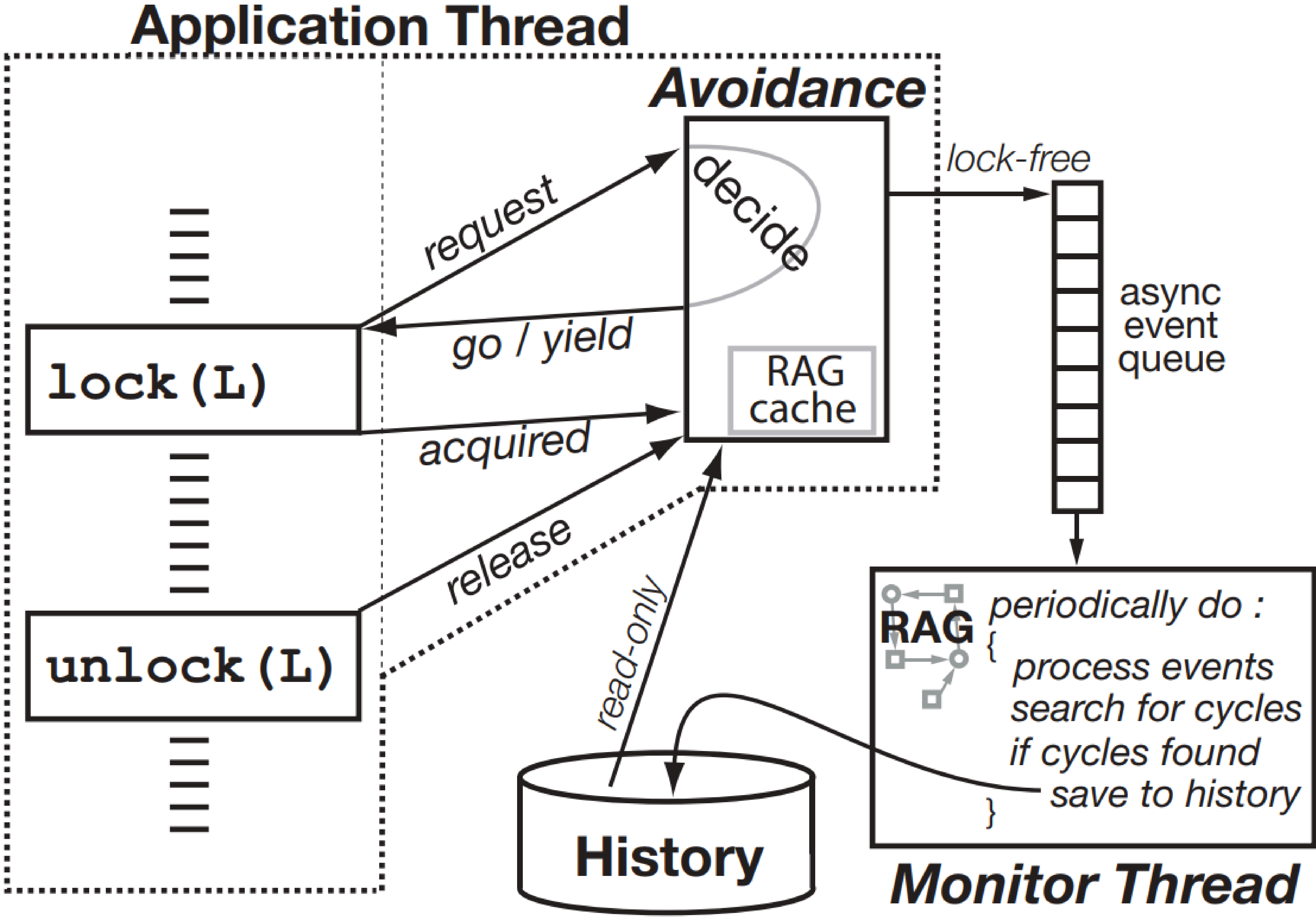
- ✓ either acquire a “gate lock” before a deadlock
- ✓ or when one occurs, roll back to a checkpoint
- ✗ large overhead

Combination of static elements with dynamic approaches...

What is Dimmunix - Overview

- Automatic resistance towards deadlocks/starvation
- Keeps track of shared resources using a Resource Allocation Graph (RAG) (periodically updated)
- Thread monitoring to save signatures (call stack labels) and avoid previously encountered ones (lock interception)
- Not thread/lock specific
- POSIX and Java threads

System Architecture



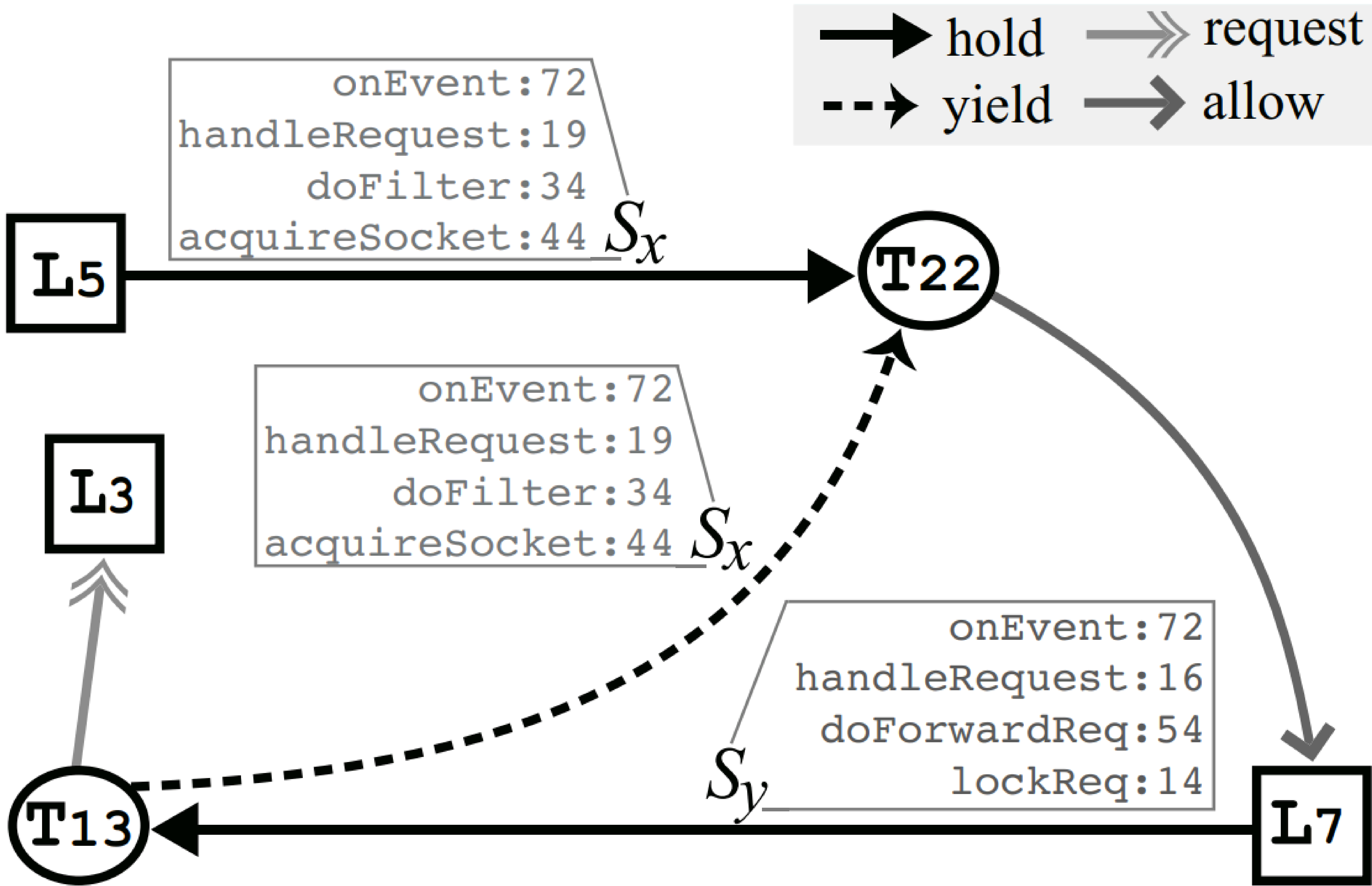
Synchronization Monitoring

Use of a Resource Allocation Graph (RAG), where:

- Vertices: *T* (thread) and *L* (lock)
- Edges

Edge	Connects	Meaning
Request	thread w/ lock	*T* requests *L*
Allow	thread w/ lock	*T* is allowed to (wait to) acquire *L*
Hold	thread w/ lock	*L* is currently held by *T*
Yield	thread w/ thread	*T* is yield because of *T'*

Synchronization Monitoring: example



Detecting Deadlocks and Starvation

- monitor thread wakes up after τ milliseconds and updates RAG based on recent lock-free events

Detecting Deadlocks and Starvation

- monitor thread wakes up after τ milliseconds and updates RAG based on recent lock-free events
- look for cycles not made out of completely old edges. Two types of cycles looking for:

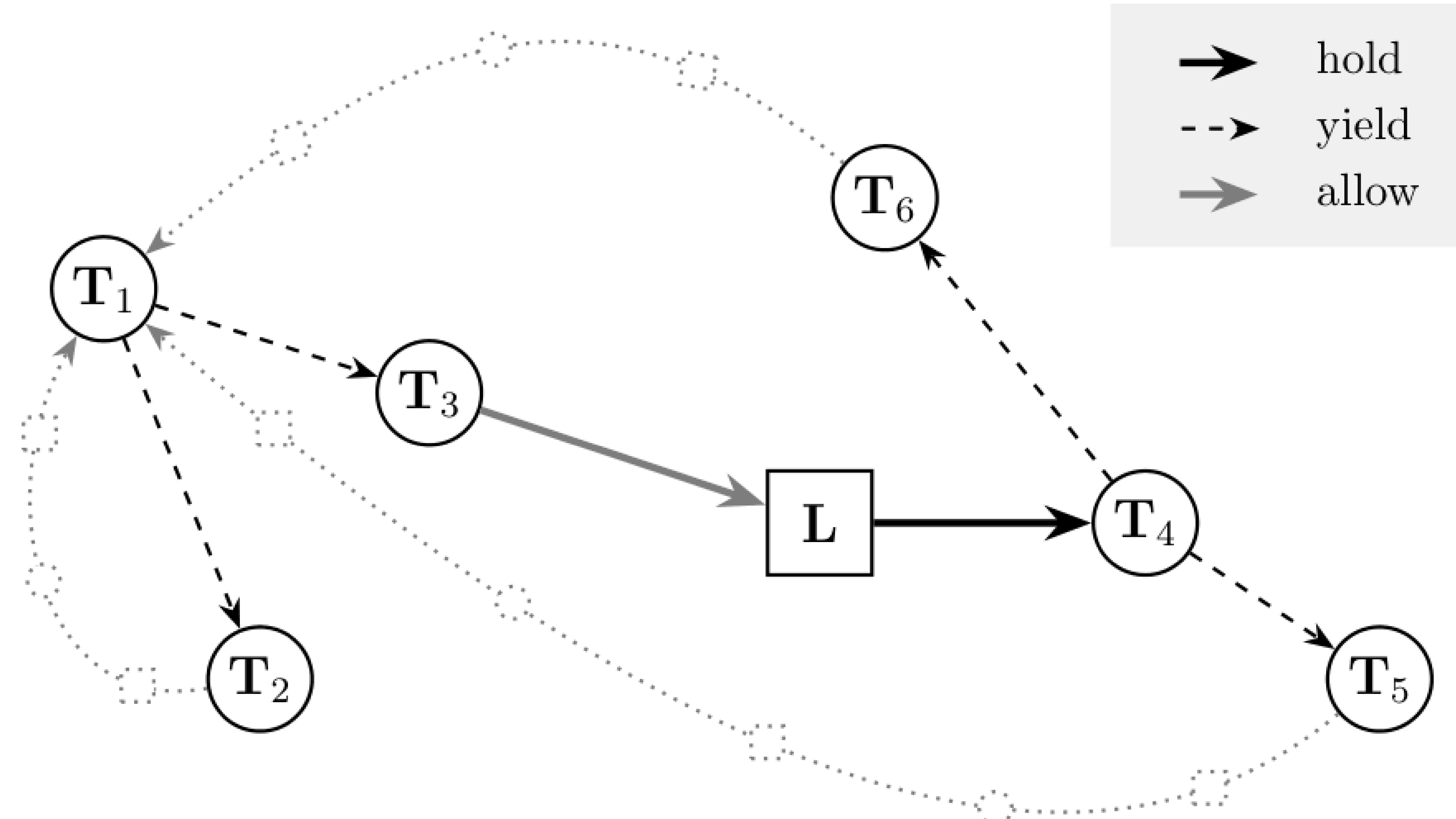
Detecting Deadlocks and Starvation

- monitor thread wakes up after τ milliseconds and updates RAG based on recent lock-free events
- look for cycles not made out of completely old edges. Two types of cycles looking for:
    - deadlock cycle = cycle made up exclusively of *hold, allow* and *request* edges
    - yield cycle

Detecting Deadlocks and Starvation

- monitor thread wakes up after τ milliseconds and updates RAG based on recent lock-free events
- look for cycles not made out of completely old edges. Two types of cycles looking for:
  - deadlock cycle = cycle made up exclusively of *hold, allow* and *request* edges
  - yield cycle
- if either cycle is detected, their signature is saved as a deadlock one

Detecting Deadlocks and Starvation: example



*graph recreation credit to Deathwish

From Cycles to Signatures

- signature format: {Sx, Sy, ...} + matching depth (later...)

From Cycles to Signatures

- signature format: {Sx, Sy, ...} + matching depth (later...)
- deadlock signature captured → restart system externally or let Dimmunix handle

From Cycles to Signatures

- signature format: {Sx, Sy, ...} + matching depth (later...)
- deadlock signature captured → restart system externally or let Dimmunix handle
- starvation signature captured (as a deadlock one) → Dimmunix breaks starvation

Avoiding Previously Seen Patterns

when thread requests lock:

Avoiding Previously Seen Patterns

when thread requests lock:

- add *allow* edge in RAG cache

Avoiding Previously Seen Patterns

when thread requests lock:

- add *allow* edge in RAG cache
- search for instantiation {(*T1*, *L1*, *Sx*), (*T2, L2, Sy), ...*} mathcing a signature

Avoiding Previously Seen Patterns

when thread requests lock:

- add *allow* edge in RAG cache
- search for instantiation {(*T1*, *L1*, *Sx*), (*T2, L2, Sy),* ...} mathcing a signature
- potential deadlock instance found:
  - make *allow* a *request* edge
  - add *yield* edges

Avoiding Previously Seen Patterns

when thread requests lock:

- add *allow* edge in RAG cache
- search for instantiation {(*T1*, *L1*, *Sx*), (*T2, L2, Sy), ...*} mathcing a signature
- potential deadlock instance found:
    - make *allow* a *request* edge
    - add *yield* edges
- not found
    - change *allow* to *hold*
    - remove any *yield edges*

Levels of Immunity

- Weak Immunity (default):
  - induced starvation automatically broken/ program continues
  - can lead to reoccurences
- Strong Immunity:
  - restart program always
  - guarantee that pattern doesn’t reoccur

Calibrating Matching Precision

- Signatures + matching depth (4 default)

Calibrating Matching Precision

- Signatures + matching depth (4 default)
- Make it dynamic with heuristics:
  - start at depth 1
  - after each avoiding decision, evaluate if it was correct (no false positives)
  - do that for N (default 20) levels
  - keep smallest depth d that had lowest false positive rate (FPmin)
  - Due to other parameters, FPmin can be non-zero

Evaluation

1. **Does Dimmunix work for real systems that do I/O, use system libraries and interact with users and other systems?**
2. **What performance overhead does Dimmunix introduce, and how does this overhead vary as parameters change?**
3. **What is the impact of false positives on performance?**
4. **What overheads does Dimmunix introduce in terms of resource consumption?**

Evaluation done on MySQL, SQLite, Apache ActiveMQ, JBoss, Limewire, Java JDK and HawkNL with strong immunity, $\tau = 100$ msec, $d = 4$.

Does Dimmunix work for real systems that do I/O, use system libraries and interact with users and other systems?

Does Dimmunix work for real systems that do I/O, use system libraries and interact with users and other systems?

System	Bug #	Deadlock Between ...	# Yields per Trial			Dlk Patterns	
			Min	Avg	Max	#	Depth
MySQL 6.0.4	37080	INSERT and TRUNCATE in two different threads	1	1	4	1	4
SQLite 3.3.0	1672	Deadlock in the custom recursive lock implementation	1	1	1	1	3
HawkNL 1.6b3	n/a	nlShutdown() called concurrently with nlClose()	10	10	10	1	2
MySQL 5.0 JDBC	2147	PreparedStatement.getWarnings() and Connection.close()	1	1	1	1	3
MySQL 5.0 JDBC	14972	Connection.prepareStatement() and Statement.close()	1	1	1	1	4
MySQL 5.0 JDBC	31136	PreparedStatement.executeQuery() and Connection.close()	1	1	1	1	3
MySQL 5.0 JDBC	17709	Statement.executeQuery() and Connection.prepareStatement()	1	1	1	1	3
Limewire 4.17.9	1449	HsqlDB TaskQueue cancel and shutdown()	15	15	15	2	10,10
ActiveMQ 3.1	336	Listener creation and active dispatching of messages to consumer	1	181079	221292	1	2
ActiveMQ 4.0	575	Queue.dropEvent() and PrefetchSubscription.add()	11252	80387	113652	3	2,2,2

Does Dimmunix work for real systems that do I/O, use system libraries and interact with users and other systems?

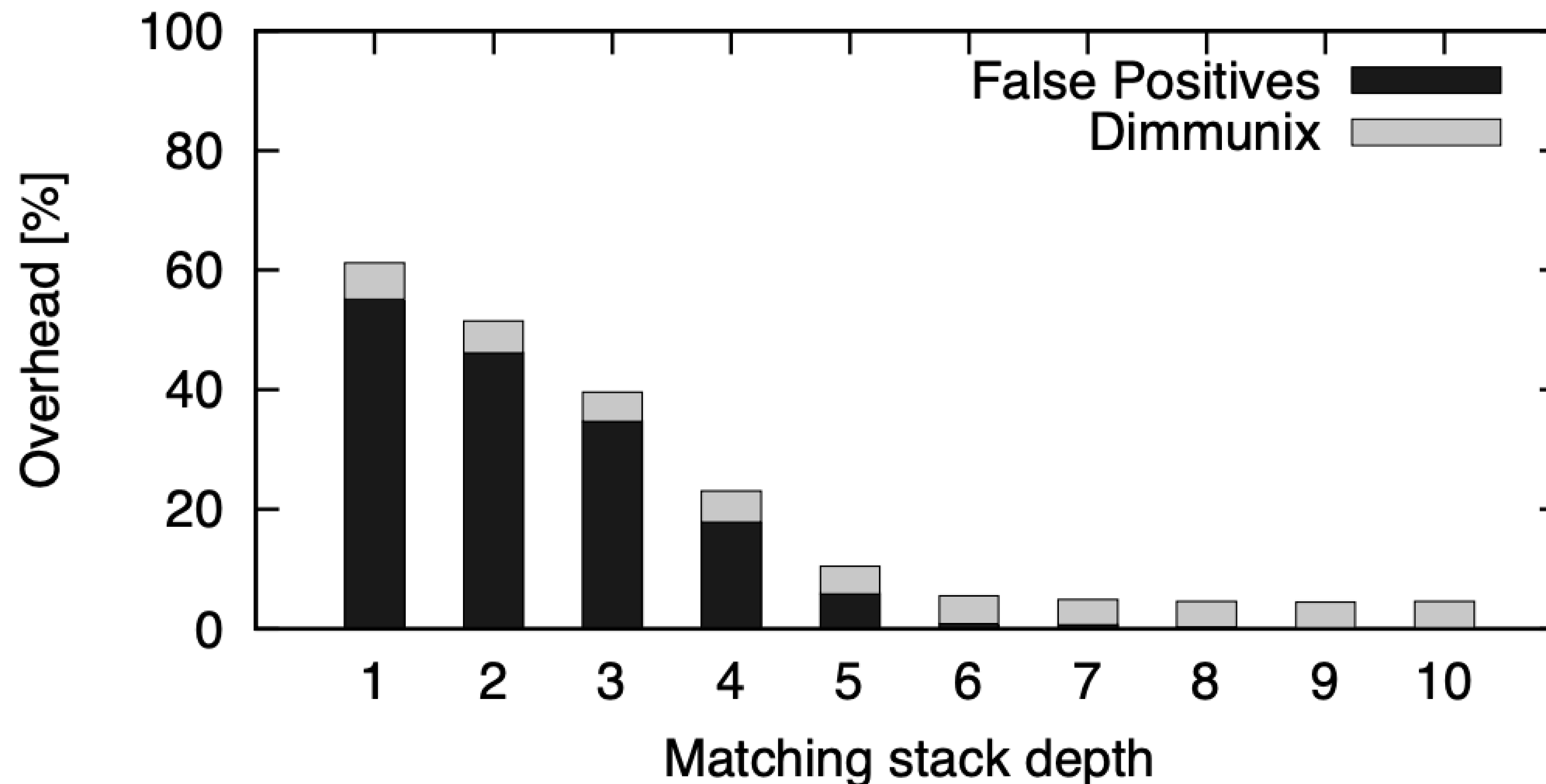
System	Bug #	Deadlock Between ...	# Yields per Trial			Dlk Patterns	
			Min	Avg	Max	#	Depth
MySQL 6.0.4	37080	INSERT and TRUNCATE in two different threads	1	1	4	1	4
SQLite 3.3.0	1672	Deadlock in the custom recursive lock implementation	1	1	1	1	3
HawkNL 1.6b3	n/a	nlShutdown() called concurrently with nlClose()	10	10	10	1	2
MySQL 5.0 JDBC	2147	PreparedStatement.getWarnings() and Connection.close()	1	1	1	1	3
MySQL 5.0 JDBC	14972	Connection.prepareStatement() and Statement.close()	1	1	1	1	4
MySQL 5.0 JDBC	31136	PreparedStatement.executeQuery() and Connection.close()	1	1	1	1	3
MySQL 5.0 JDBC	17709	Statement.executeQuery() and Connection.prepareStatement()	1	1	1	1	3
Limewire 4.17.9	1449	HsqlDB TaskQueue cancel and shutdown()	15	15	15	2	10,10
ActiveMQ 3.1	336	Listener creation and active dispatching of messages to consumer	1	181079	221292	1	2
ActiveMQ 4.0	575	Queue.dropEvent() and PrefetchSubscription.add()	11252	80387	113652	3	2,2,2

What performance overhead does Dimmunix introduce, and how does this overhead vary as parameters change?

- Real Applications
  - JBoss/RUBiS → 2.6%
  - MySQL/JDBCBench → **7.17%**
- Overhead as a function of the number of threads
  - 0.6% to **4.5%** for FreeBSD pthreads
  - **6.5%** to **17.5%** for Java
- Overhead as a function of number of signatures in history
  - relatively constant/negligible

What is the impact of false positives on performance?

- The larger the matching depth, the lower the amount of FPs



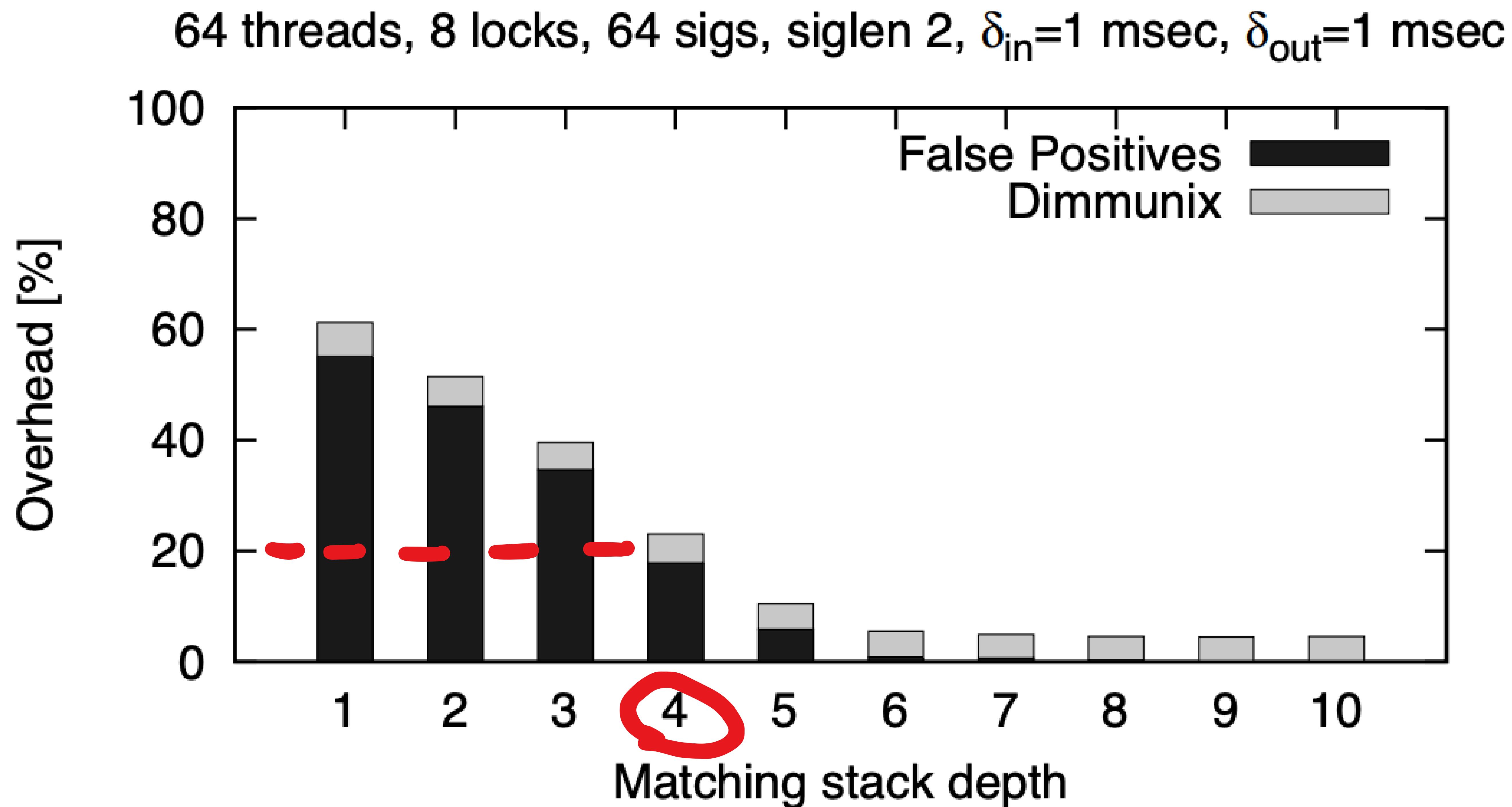
What is the impact of false positives on performance?

- The larger the matching depth, the lower the amount of FPs

64 threads, 8 locks, 64 sigs, siglen 2, δ_{in}=1 msec, δ_{out}=1 msec

What is the impact of false positives on performance?

- The larger the matching depth, the lower the amount of FPs



What overheads does Dimmunix introduce in terms of resource consumption?

- Disk: 200-1000 bytes/signature
- CPU: virtually zero - negative
- Memory: 6-25 MB (POSIX) , 79-127 MB (Java)

Limitations

- Some times all executions lead to deadlocks
- Other synchronization mechanisms need to be specified

Limitations

- Some times all executions lead to deadlocks
- Other synchronization mechanisms need to be specified
- Large overhead for a general purpose system
- Just lock-specific

A quick demo…

or not

Thank you!

Any questions?