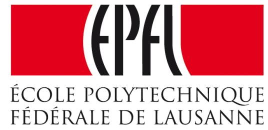


Hints for Writing About Your Research

George Candea

School of Computer & Communication Sciences

Good Writing

… so I wait for you like a lonely house
till you will see me again and live in me.
Till then my windows ache.

(Pablo Neruda)

The dopamine signaling in the nucleus accumbens of my basal forebrain is lower than normal due to your physical absence.

The performance of our cache becomes tremendously small when the data is accessed in a very adversarial manner.

The hit rate of the CPU cache drops by up to 95% if programs consistently write to the least-recently read memory address.

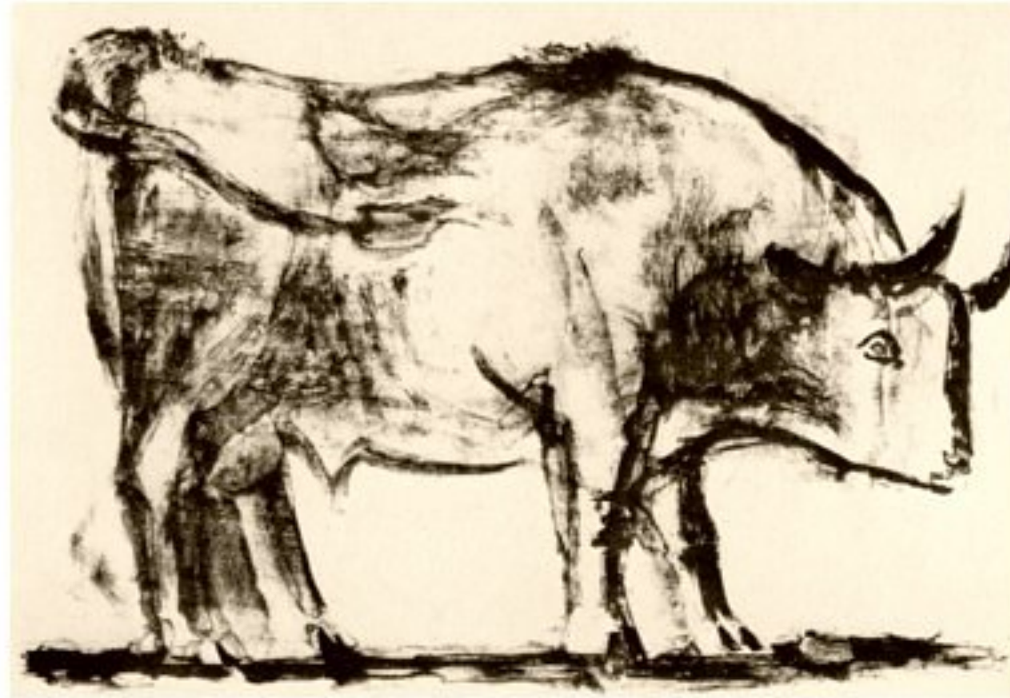
Lyrical writing

Technical writing

The Writing Process

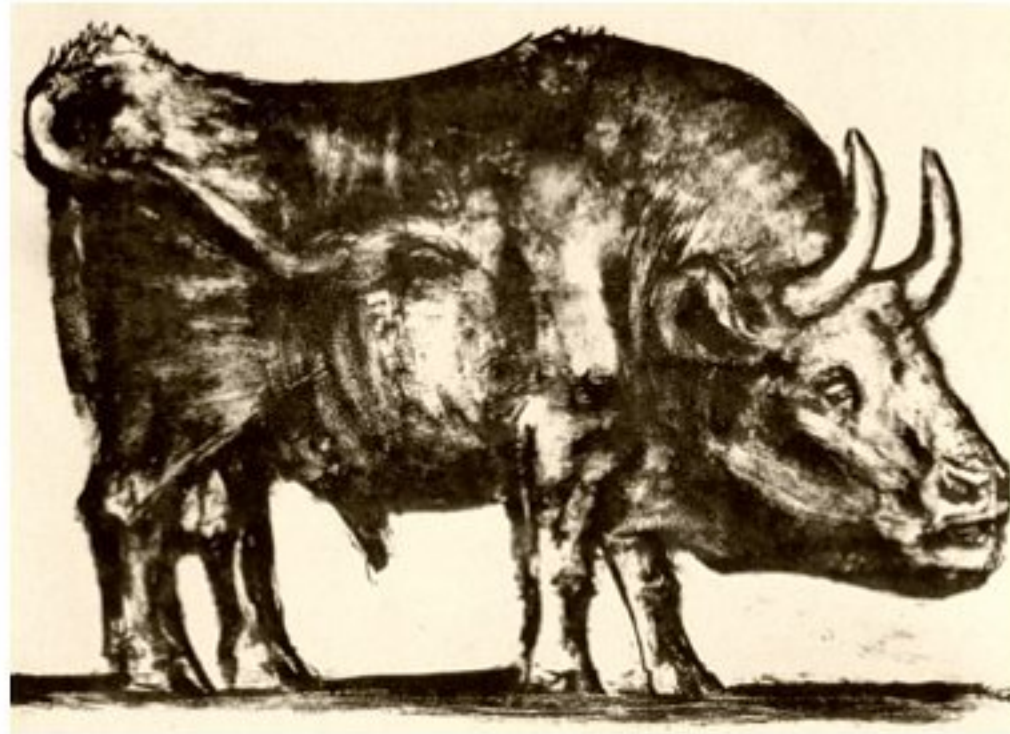
Perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away.

(Antoine de Saint-Exupéry, “L’Avion”, Ch. III)



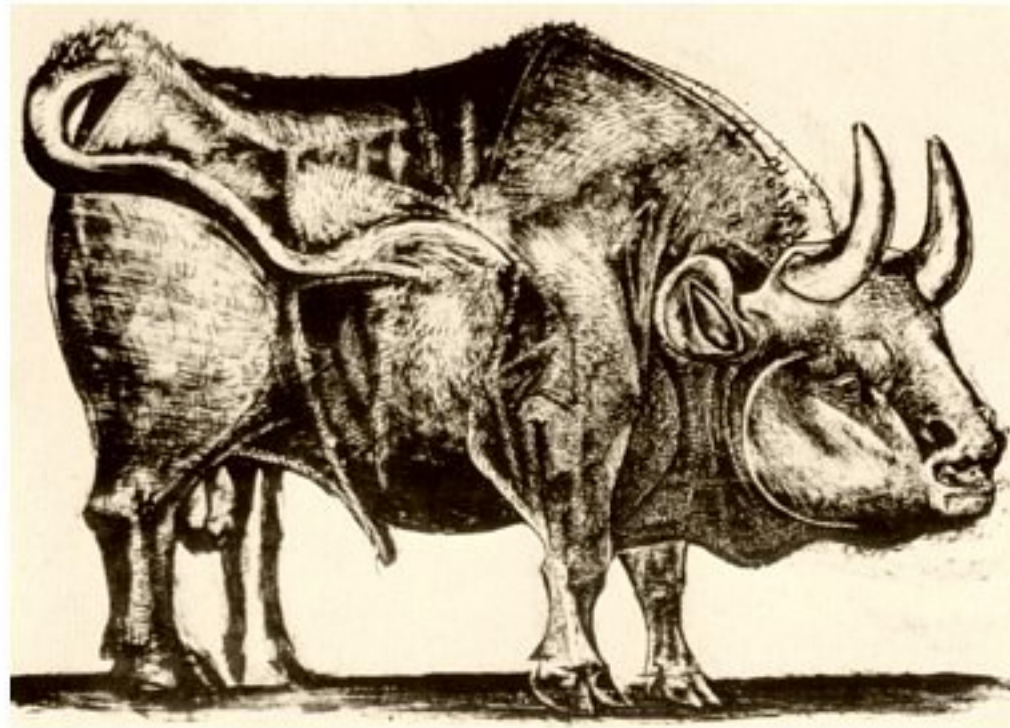
December 5, 1945

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm



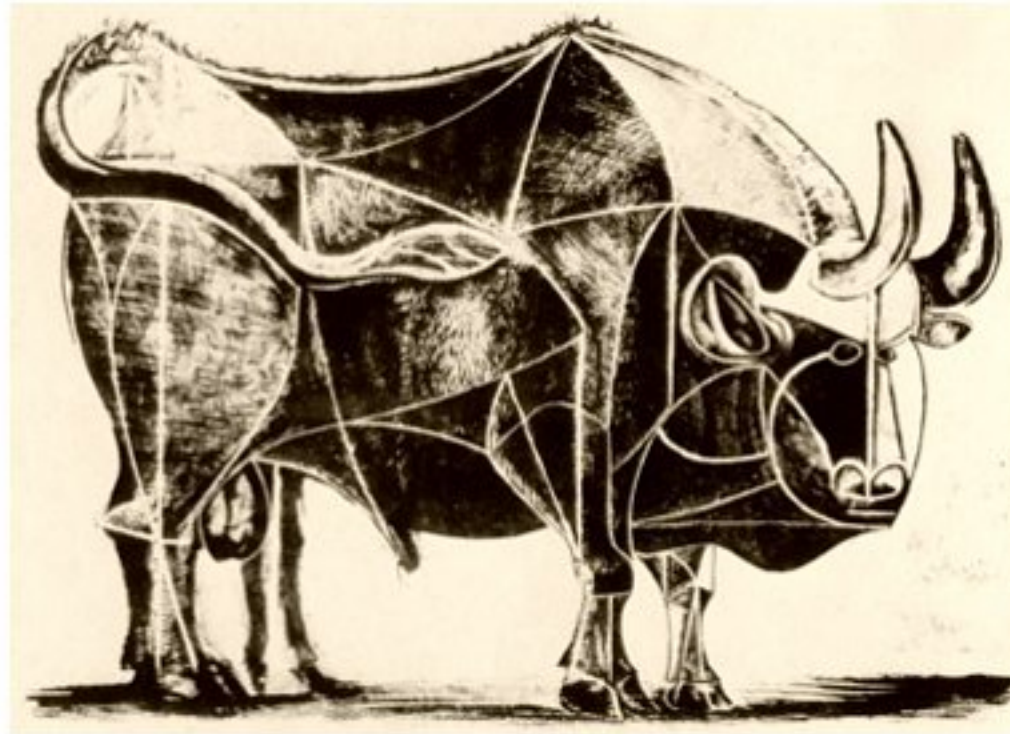
December 12, 1945

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm



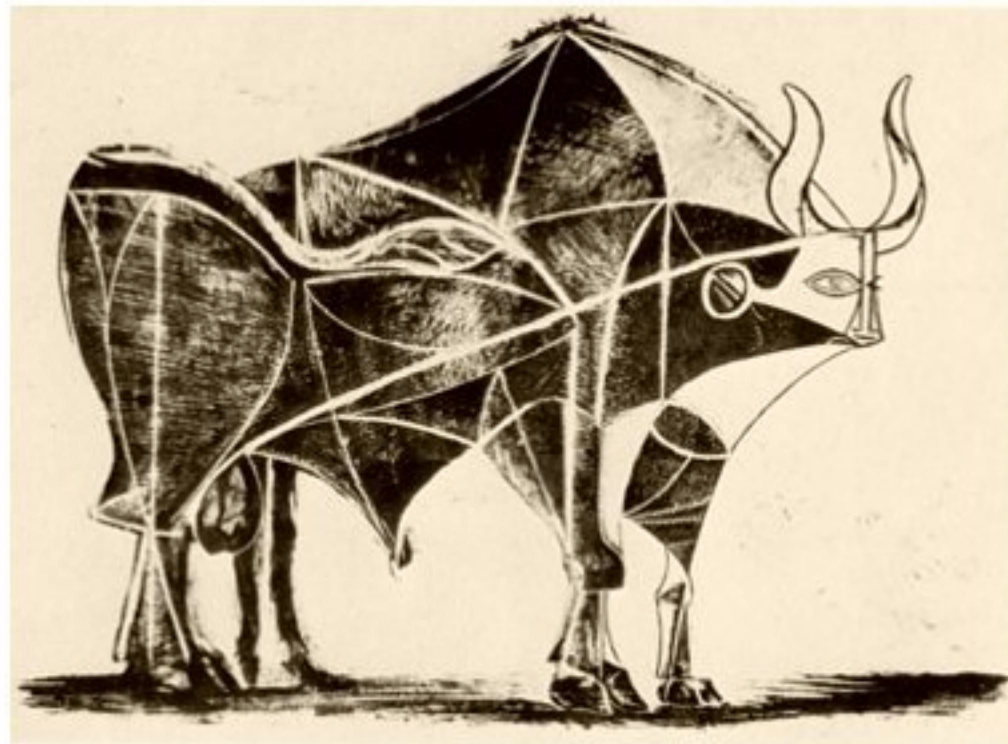
December 18, 1945

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm



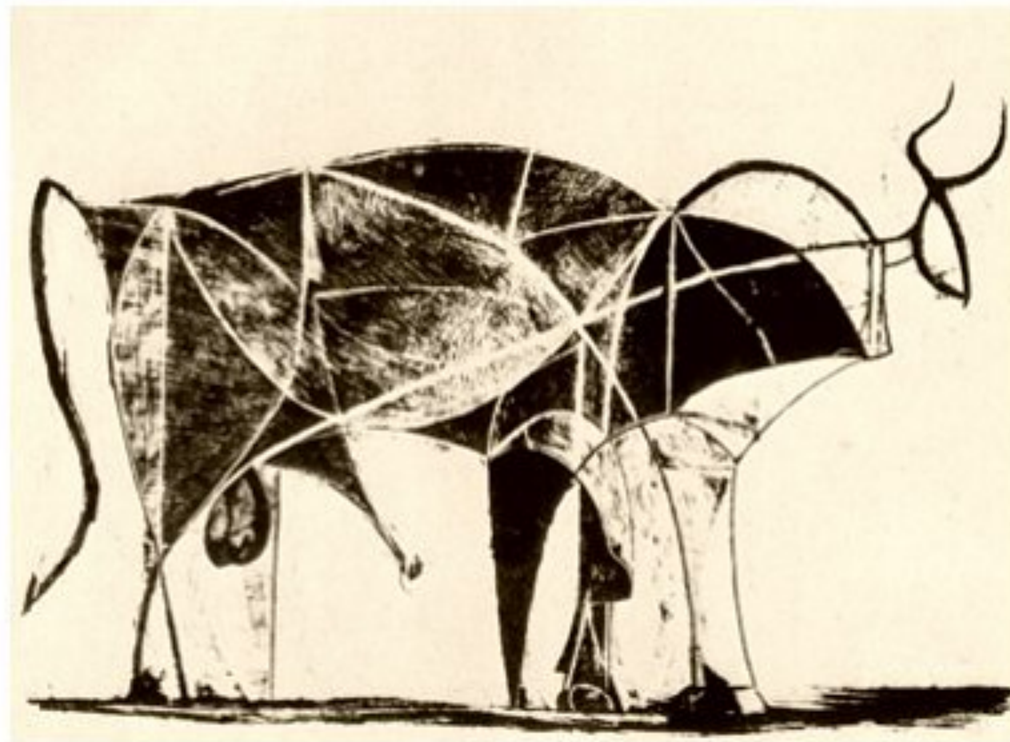
December 22, 1945

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm



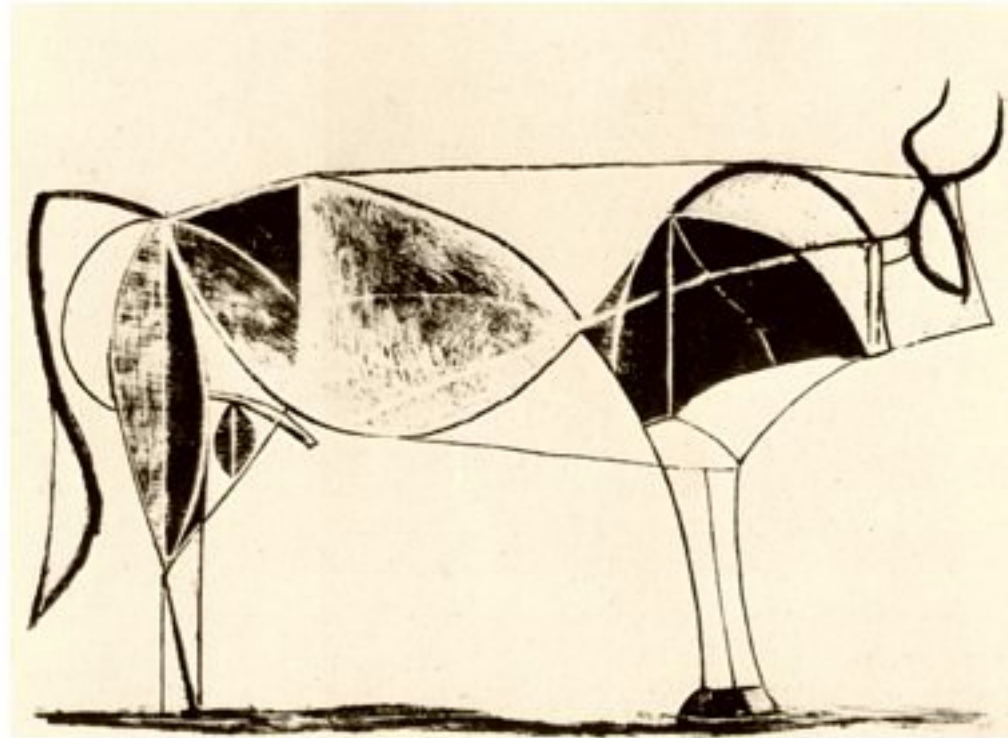
December 24, 1945

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm



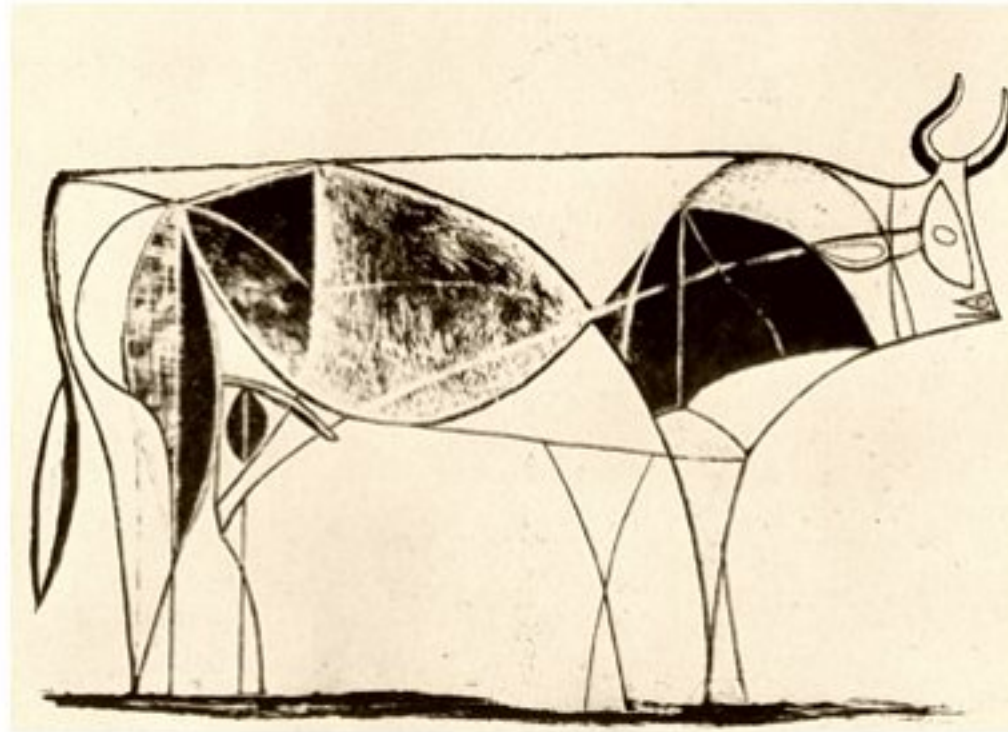
December 26, 1945

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm



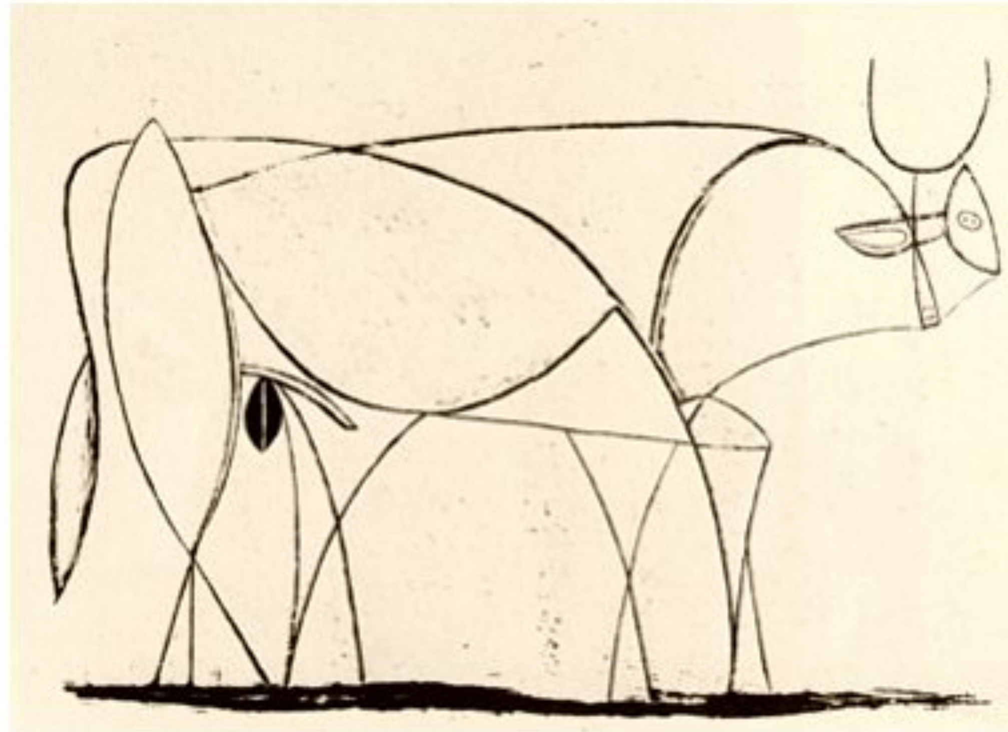
December 28, 1945

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm



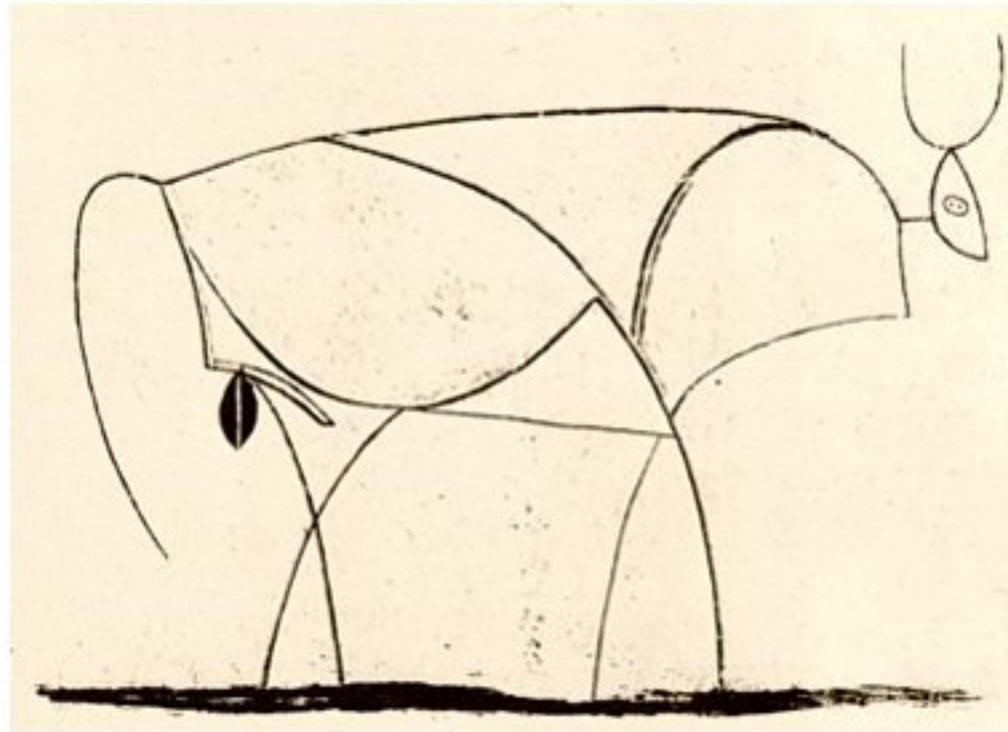
January 2, 1946

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm



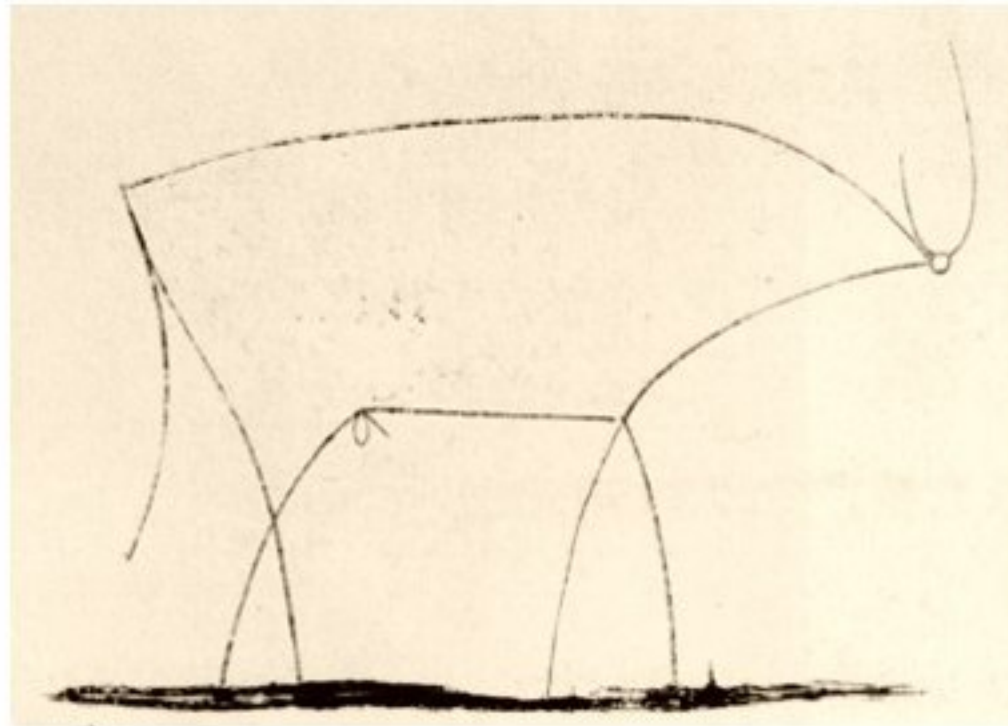
January 5, 1946

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm



January 10, 1946

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm



January 17, 1946

Drawing courtesy of http://www.artyfactory.com/art_appreciation/animals_in_art/pablo_picasso.htm

The Writing Process

Perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away.

(Antoine de Saint-Exupéry, L'Avion, Ch. III)

Recursive Structure

Paper title

- Abstract
- Section title
  - Abstract
  - Topic sentence + Paragraph body
  - Topic sentence + Paragraph body
  - …
  - Conclusion
- Section title
- …

Abstract

Deadlock immunity is a property by which programs, once afflicted by a given deadlock, develop resistance against future occurrences of that and similar deadlocks. We describe a technique that enables programs to automatically gain such immunity without assistance from programmers or users. We implemented the technique for both Java and POSIX threads and evaluated it with several real systems, including MySQL, JBoss, SQLite, Apache ActiveMQ, Limewire, and Java JDK. The results demonstrate effectiveness against real deadlock bugs, while incurring modest performance overhead and scaling to 1024 threads. We therefore conclude that deadlock immunity offers programmers and users an attractive tool for coping with elusive deadlocks.

1 Introduction

Writing concurrent software is one of the most challenging endeavors faced by software engineers, because it requires careful reasoning about complex interactions between concurrently running threads. Many programmers consider concurrency bugs to be some of the most insidious and, not surprisingly, a large number of bugs are related to concurrency [16].

The simplest mechanism used for synchronizing concurrent accesses to shared data is the mutex lock. When threads do not coordinate correctly in their use of locks, deadlock can ensue—a situation whereby a group of threads cannot make forward progress, because each one is waiting to acquire a lock held by another thread in that group. Deadlock immunity helps develop resistance against such deadlocks.

Avoiding the introduction of deadlock bugs during development is challenging. Large software systems are developed by multiple teams totaling hundreds to thousands of programmers, which makes it hard to maintain the coding discipline needed to avoid deadlock bugs. Testing, although helpful, is not a panacea, because exercising all possible execution paths and thread interleavings is still infeasible in practice for all but toy programs.

Even deadlock-free code is not guaranteed to execute free of deadlocks once deployed in the field. Dependencies on deadlock-prone third party libraries or runtimes can deadlock programs that are otherwise correct. Upgrading these libraries or runtimes can introduce new executions that were not covered by prior testing. Furthermore, modern systems accommodate extensions written by third parties, which can introduce new deadlocks into the target systems (e.g., Web browser plugins, enterprise Java beans).

Debugging deadlocks is hard—merely seeing a deadlock happen does not mean the bug is easy to fix. Deadlocks often require complex sequences of low-probability events to manifest (e.g., timing or workload dependencies, presence or absence of debug code, compiler optimization options), making them hard to reproduce and diagnose. Sometimes deadlocks are too costly to fix, as they entail drastic redesign. Patches are error-prone: many concurrency bug fixes either introduce new bugs or, instead of fixing the underlying bug, merely decrease the probability of occurrence [16].

We expect the deadlock challenge to persist and likely become worse over time: On the one hand, software systems continue getting larger and more complex. On the other hand, owing to the advent of multi-core architectures and other forms of parallel hardware, new applications are written using more threads, while existing applications achieve higher degrees of runtime concurrency. There exist proposals for making concurrent programming easier, such as transactional memory [8], but issues surrounding I/O and long-running operations make it difficult to provide atomicity transparently.

In this paper, we introduce the notion of deadlock immunity—a property by which programs, once afflicted by a given deadlock, develop resistance against future occurrences of similar deadlocks. We describe Dimmunix, a tool for developing deadlock immunity with no assistance from programmers or users. The first time a deadlock pattern manifests, Dimmunix automatically captures its signature and subsequently avoids entering the same pattern. Signatures can be proactively distributed to immunize users who have not yet encountered that deadlock. Dimmunix can be used by customers to defend against deadlocks while waiting for a vendor patch, and by software vendors as a safety net.

In the rest of the paper we survey related work (§2), give an overview of our system (§3-§4), present details of our technique (§5), describe three Dimmunix implementations (§6), evaluate them (§7), discuss how Dimmunix can be used in practice (§8), and conclude (§9).

Writing concurrent software is one of the most challenging endeavors faced by software engineers, because it requires careful reasoning about complex interactions between concurrently running threads.

The simplest mechanism used for synchronizing concurrent accesses to shared data is the mutex lock.

Avoiding the introduction of deadlock bugs during development is challenging.

Even deadlock-free code is not guaranteed to execute free of deadlocks once deployed in the field.

Debugging deadlocks is hard—merely seeing a deadlock happen does not mean the bug is easy to fix.

We expect the deadlock challenge to persist and likely become worse over time

In this paper, we introduce the notion of deadlock immunity—a property by which programs, once afflicted by a given deadlock, develop resistance against future occurrences of similar deadlocks.

locks into the target systems (e.g., Web browser plugins, enterprise Java beans).

Debugging deadlocks is hard—merely seeing a deadlock happen does not mean the bug is easy to fix. Deadlocks often require complex sequences of low-probability events to manifest (e.g., timing or workload dependencies, presence or absence of debug code, compiler optimization options), making them hard to reproduce and diagnose. Sometimes deadlocks are too costly to fix, as they entail drastic redesign. Patches are error-prone: many concurrency bug fixes either introduce new bugs or, instead of fixing the underlying bug, merely decrease the probability of occurrence [16].

We expect the deadlock challenge to persist and likely become worse over time: On the one hand, software systems continue getting larger and more complex. On

bugs or, instead of fixing the underlying bug, merely decrease the probability of occurrence [16].

We expect the deadlock challenge to persist and likely become worse over time: On the one hand, software systems continue getting larger and more complex. On the other hand, owing to the advent of multi-core architectures and other forms of parallel hardware, new applications are written using more threads, while existing applications achieve higher degrees of runtime concurrency. There exist proposals for making concurrent programming easier, such as transactional memory [8], but issues surrounding I/O and long-running operations make it difficult to provide atomicity transparently.

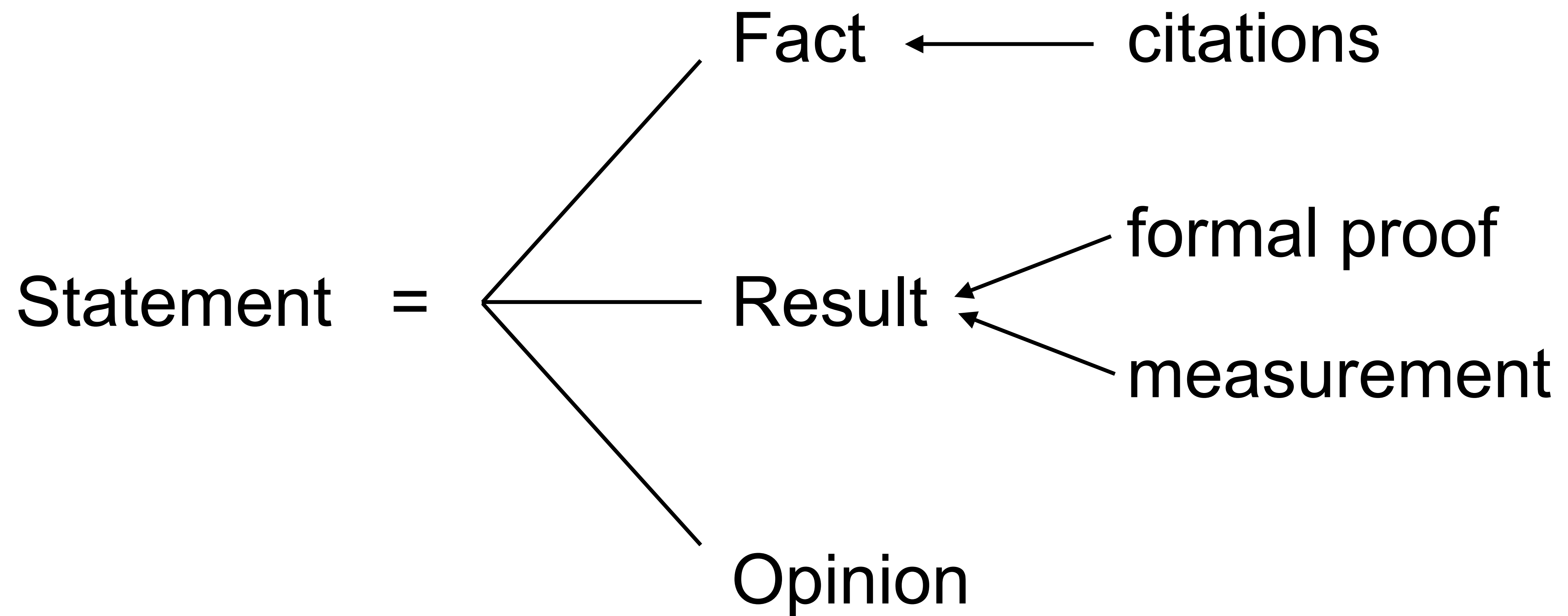
In this paper, we introduce the notion of deadlock immunity — a property by which programs, once afflicted by a given deadlock, develop resistance against future oc-

Recursive Structure

Paper title

- Abstract
- Section title
  - Abstract
  - Topic sentence + Paragraph body
  - Topic sentence + Paragraph body
  - …
  - Conclusion
- Section title
- …

Keep Opinions to Yourself



Avoid Vagueness

The performance of our cache becomes tremendously small when the data is accessed in a very adversarial manner.

The hit rate of the CPU cache drops by up to 95% if programs consistently write to the least-recently read memory address.

- Employ logic
- Quantify
- Avoid passive voice
- Avoid anthropomorphism
- Avoid hyperbolae
- Use consistent terminology

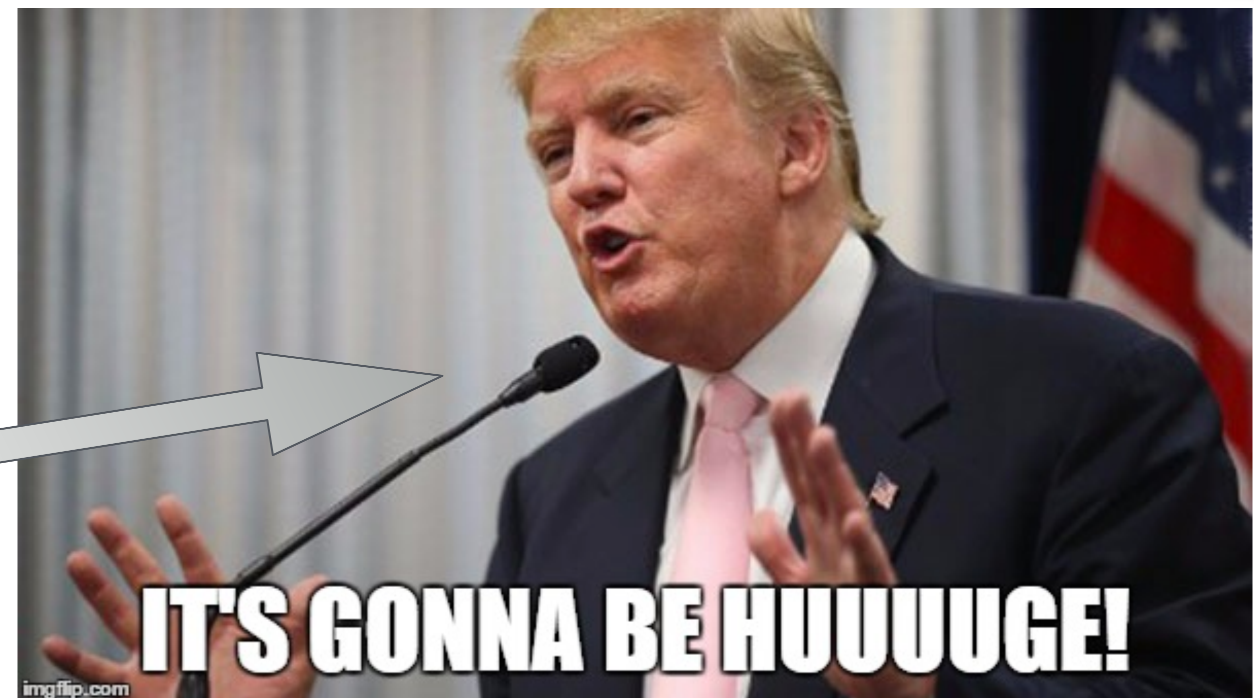


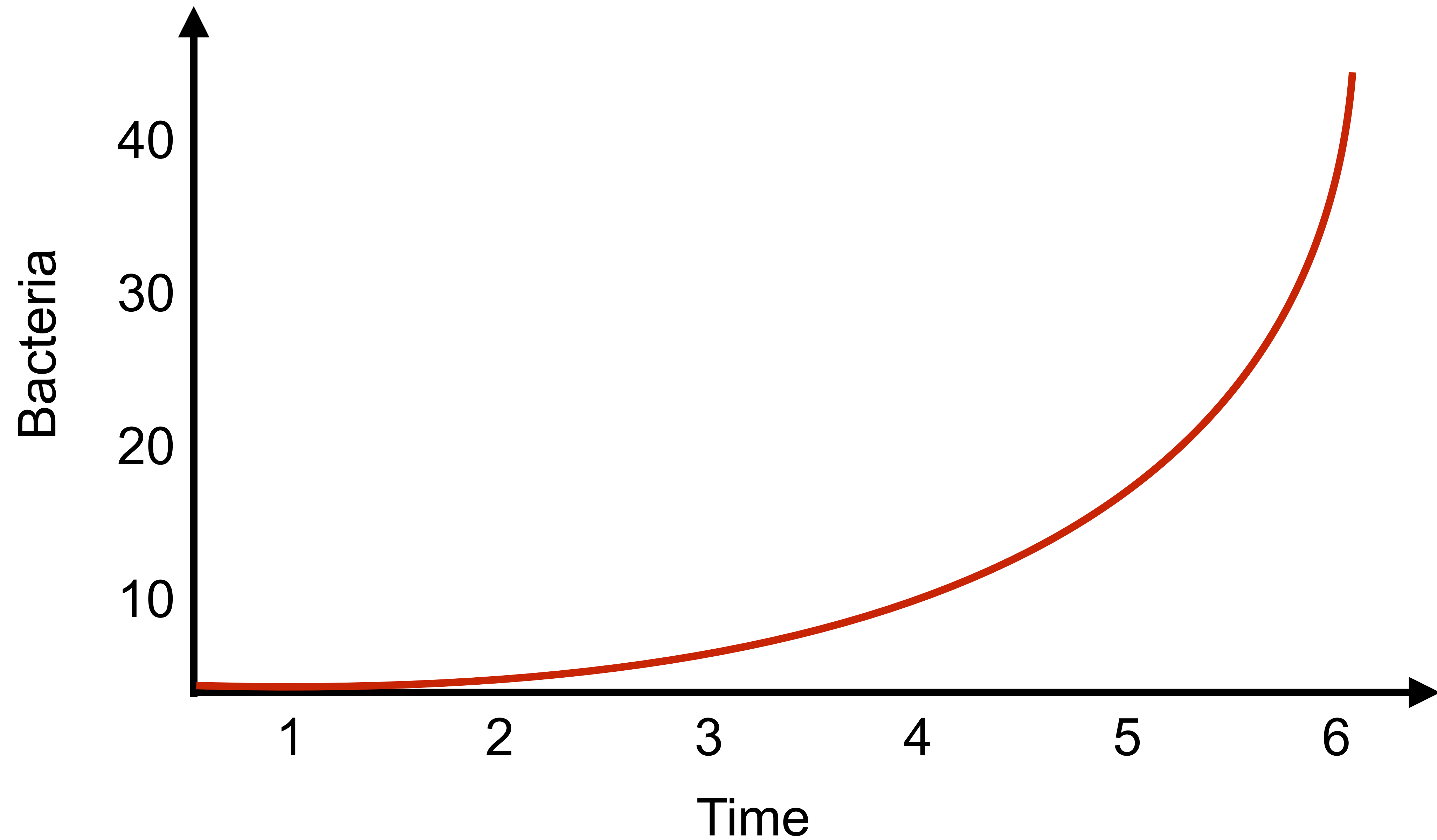
Image courtesy of https://imgflip.com/i/p8blw

Fewer Words, More Examples

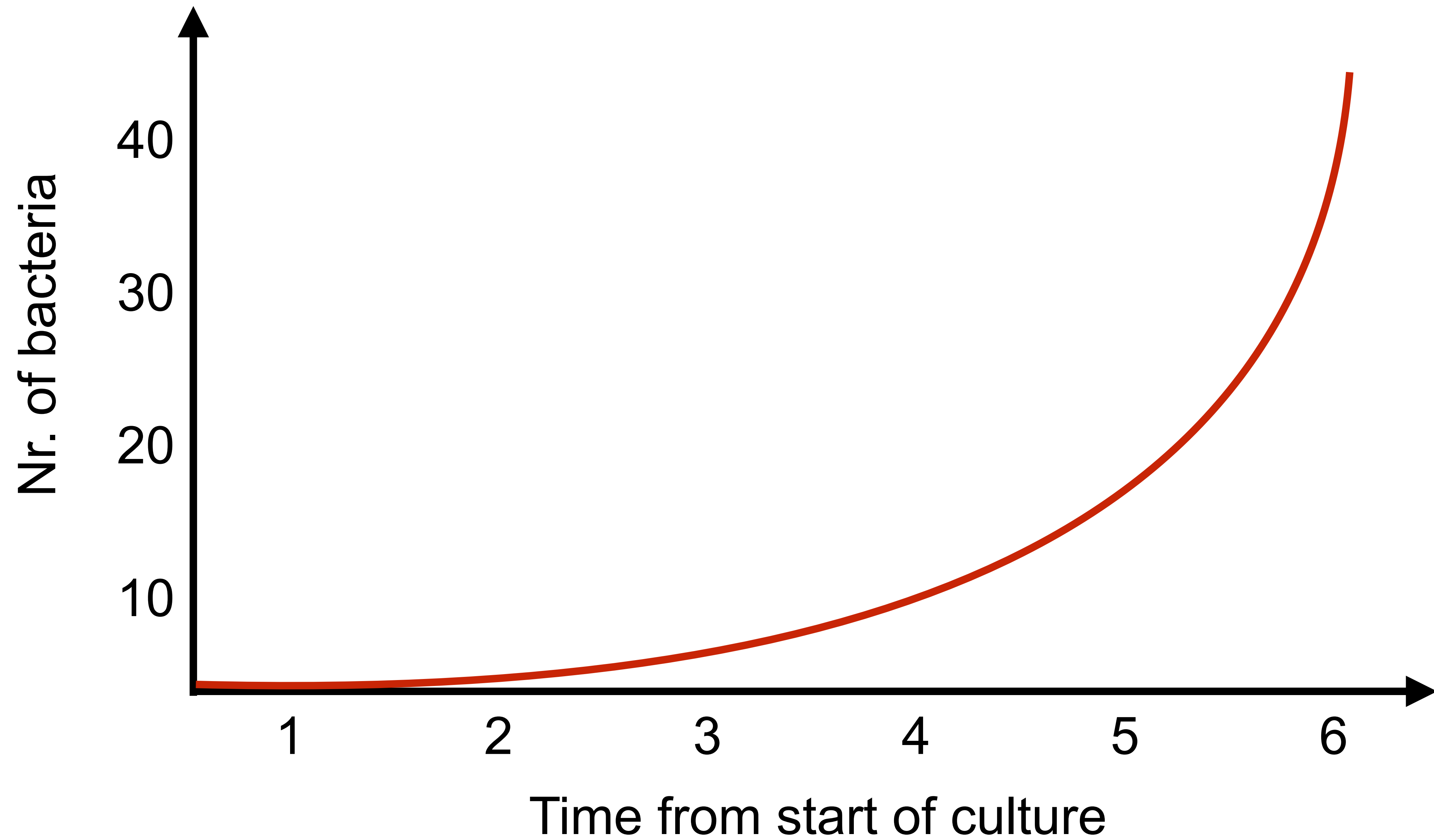
I would have written a shorter letter,
but I did not have the time.

(Blaise Pascal, Provincial Letters # XVI)

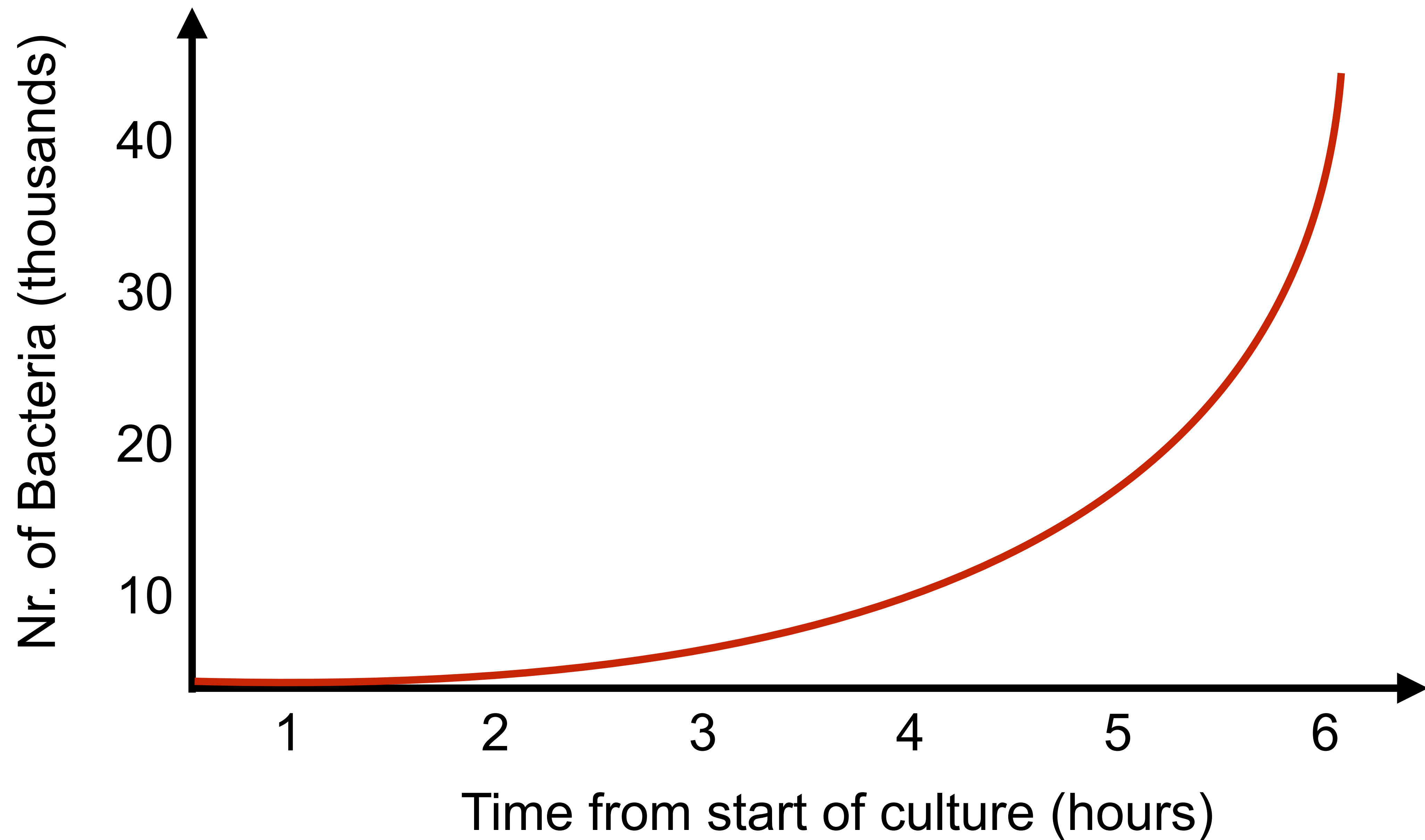
Clear Graphs



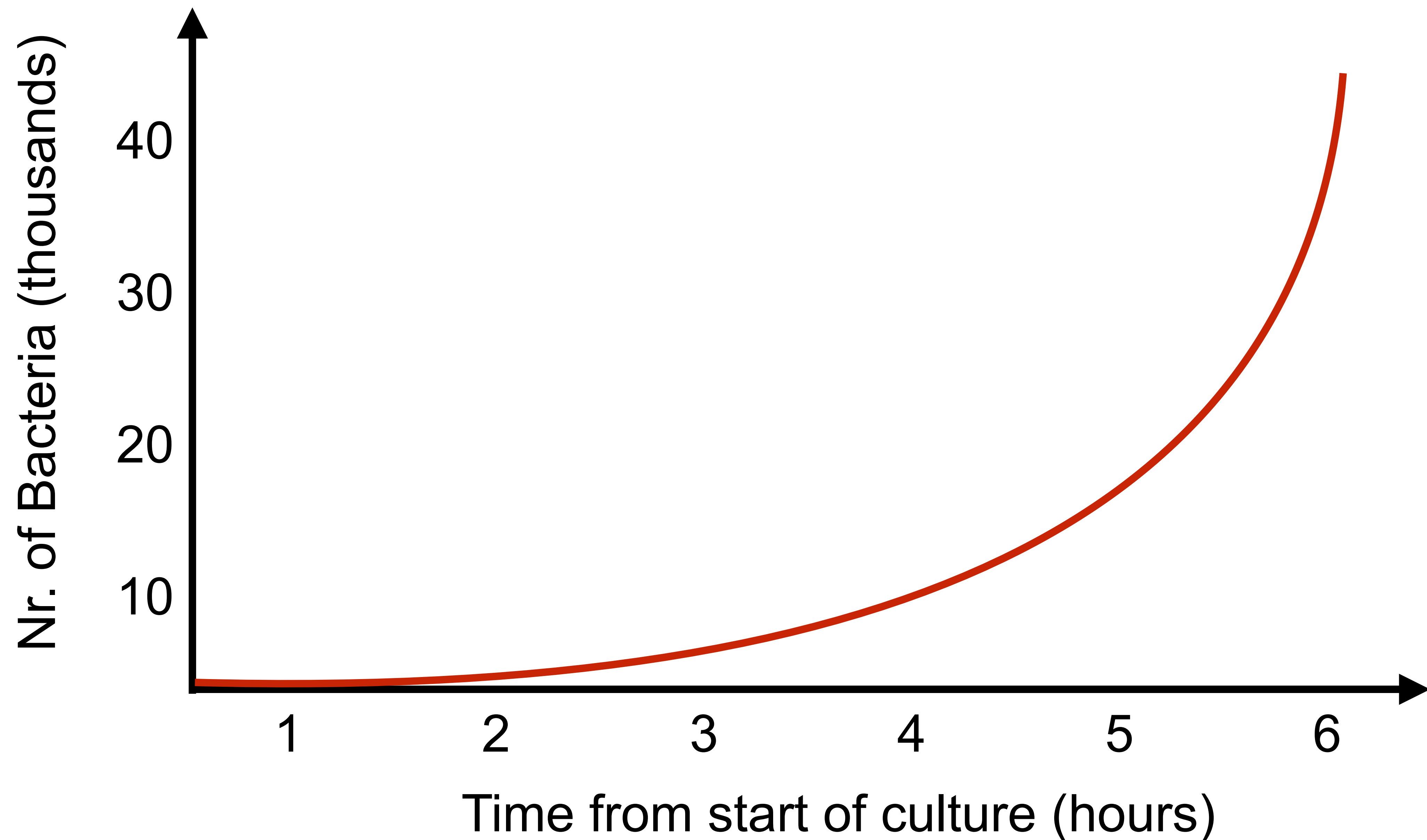
Clear Graphs



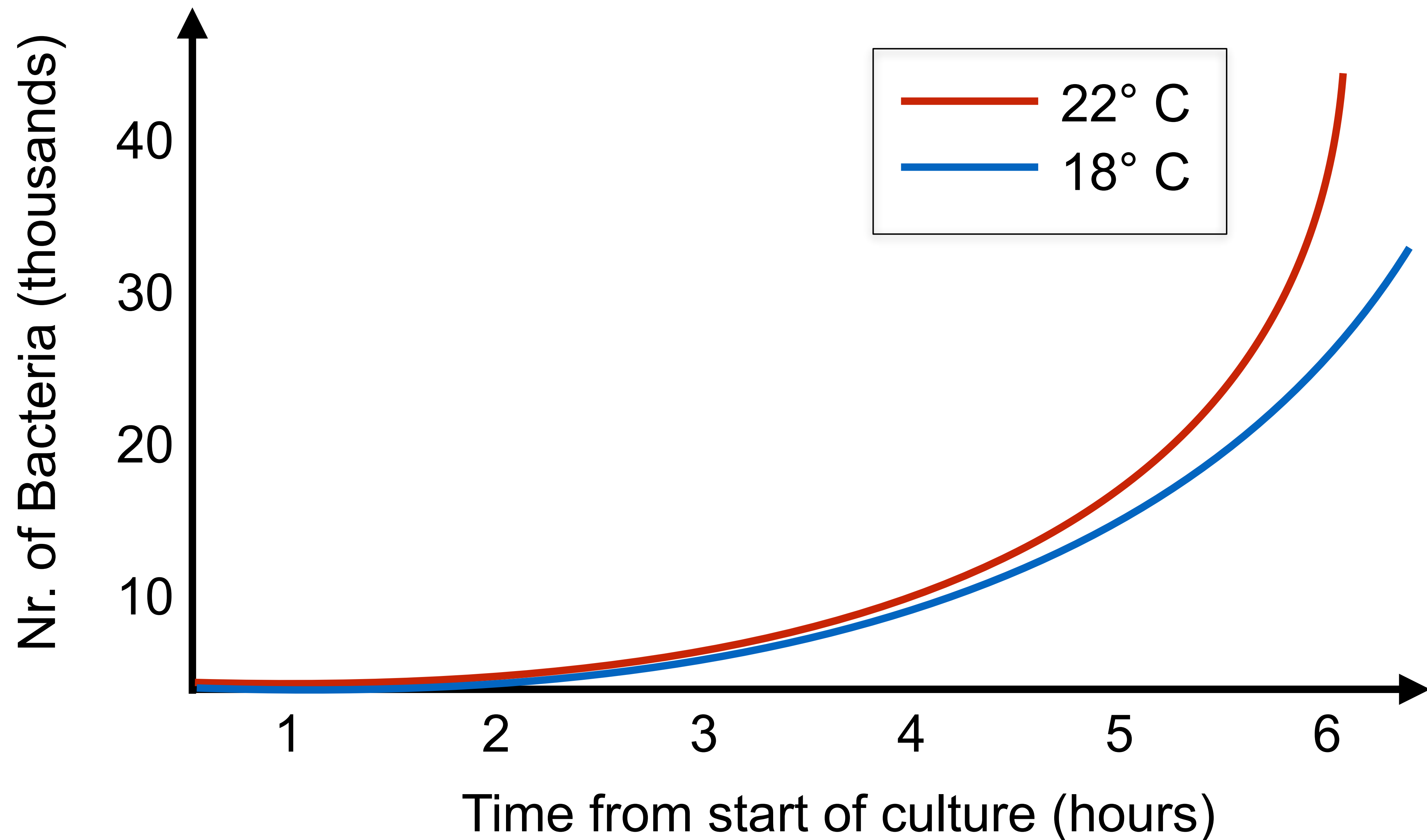
Clear Graphs



Clear Graphs



Clear Graphs



Explain Your Data

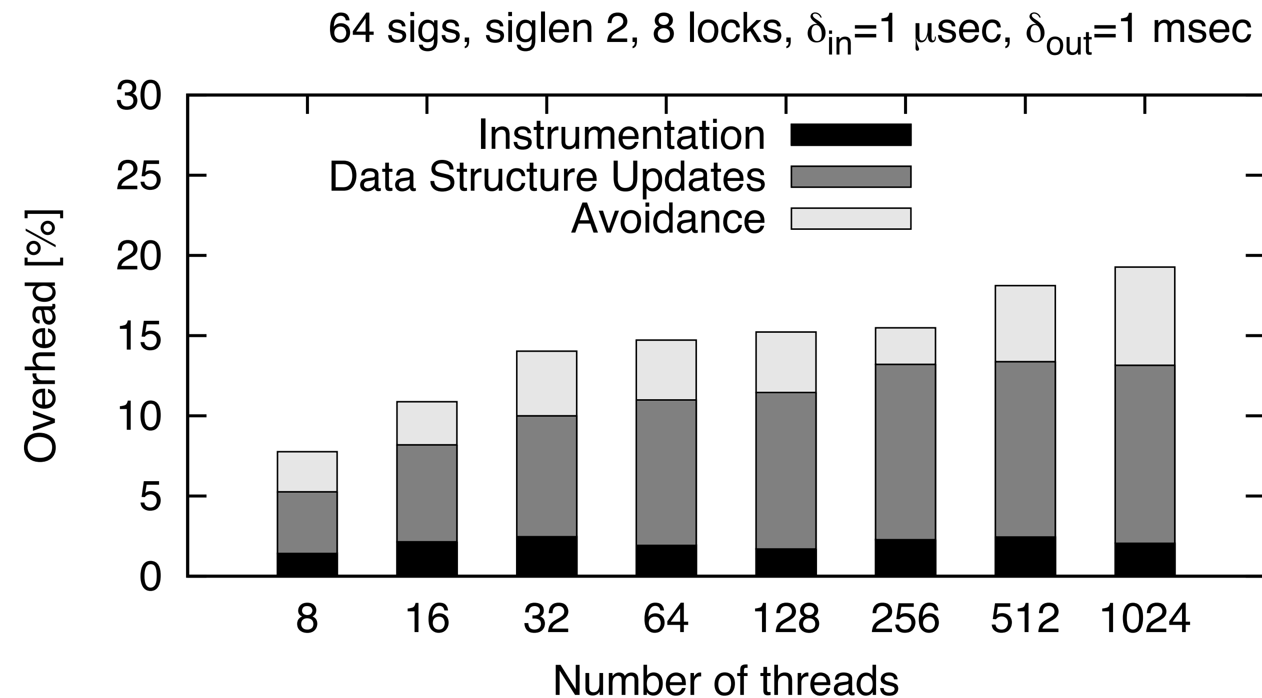


Figure 8: Breakdown of overhead for Java Dimmunix.

The results for Java are shown in Figure 8

Explain Your Data

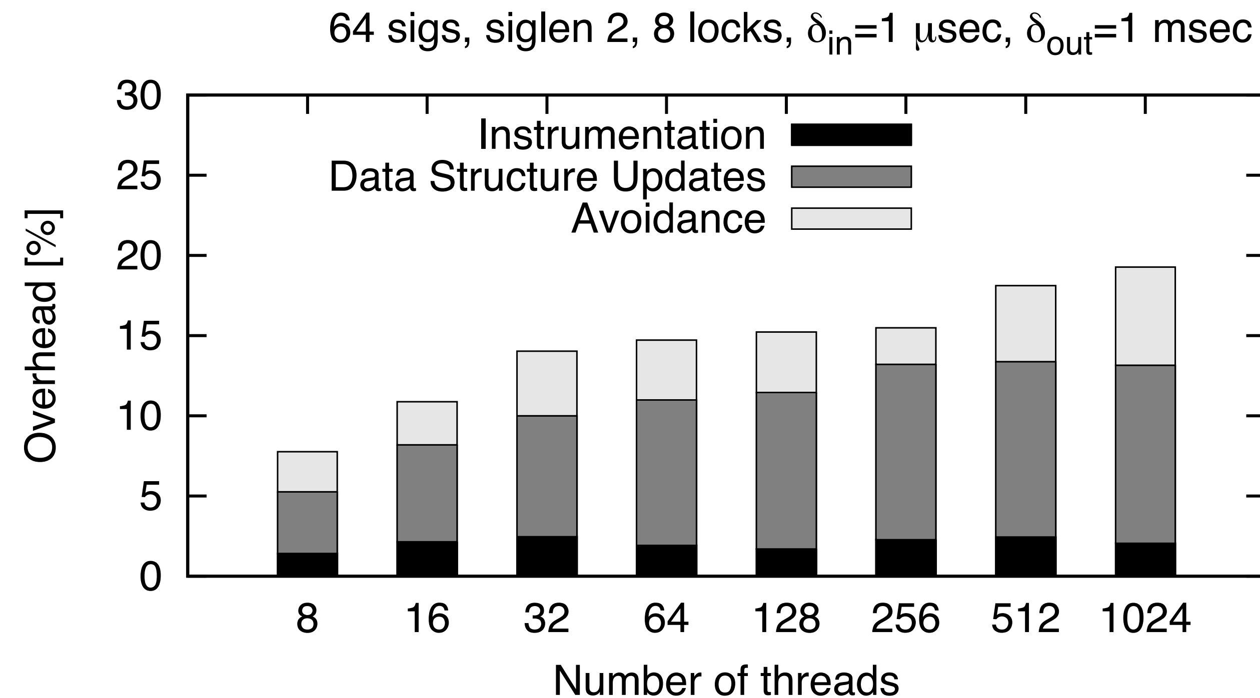


Figure 8: Breakdown of overhead for Java Dimmunix.

The results for Java are shown in Figure 8—the bulk of the overhead is introduced by the data structure lookups and updates. For pthreads, the trend is similar, except that the dominant fraction of overhead is introduced by the instrumentation code. The main reason is that the changes to the pthreads library interfere with the fastpath of the pthreads mutex: it first performs a compare-and-swap (CAS) and only if that is unsuccessful does it make a system call. Our current implementation causes that CAS to be unsuccessful with higher probability.

(Systems) Paper Structure

1. Introduction (a.k.a. Problem Stmt)
2. Design (a.k.a. Solution)
3. Prototype (a.k.a. Implementation)
4. Evaluation (a.k.a. Proof)
5. Related Work
6. Conclusion

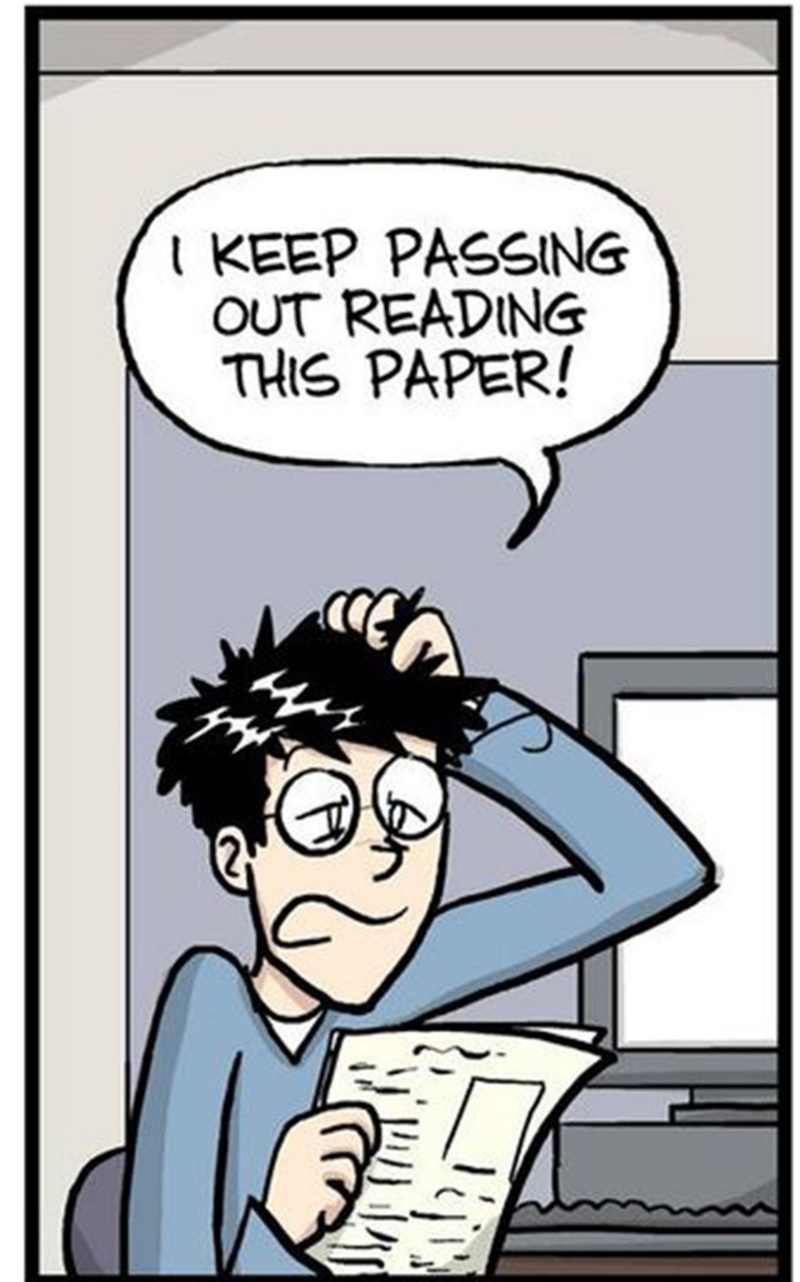


Image courtesy of http://phdcomics.com/

(Systems) Paper Structure

1. Introduction (a.k.a. Problem Stmt)
   - Problem Statement
   - Solution Overview (1 para)
   - Contributions (1 para)
   - Roadmap (1 para)
2. Design (a.k.a. Solution)
3. Prototype (a.k.a. Implementation)
4. Evaluation (a.k.a. Proof)
5. Related Work
6. Conclusion

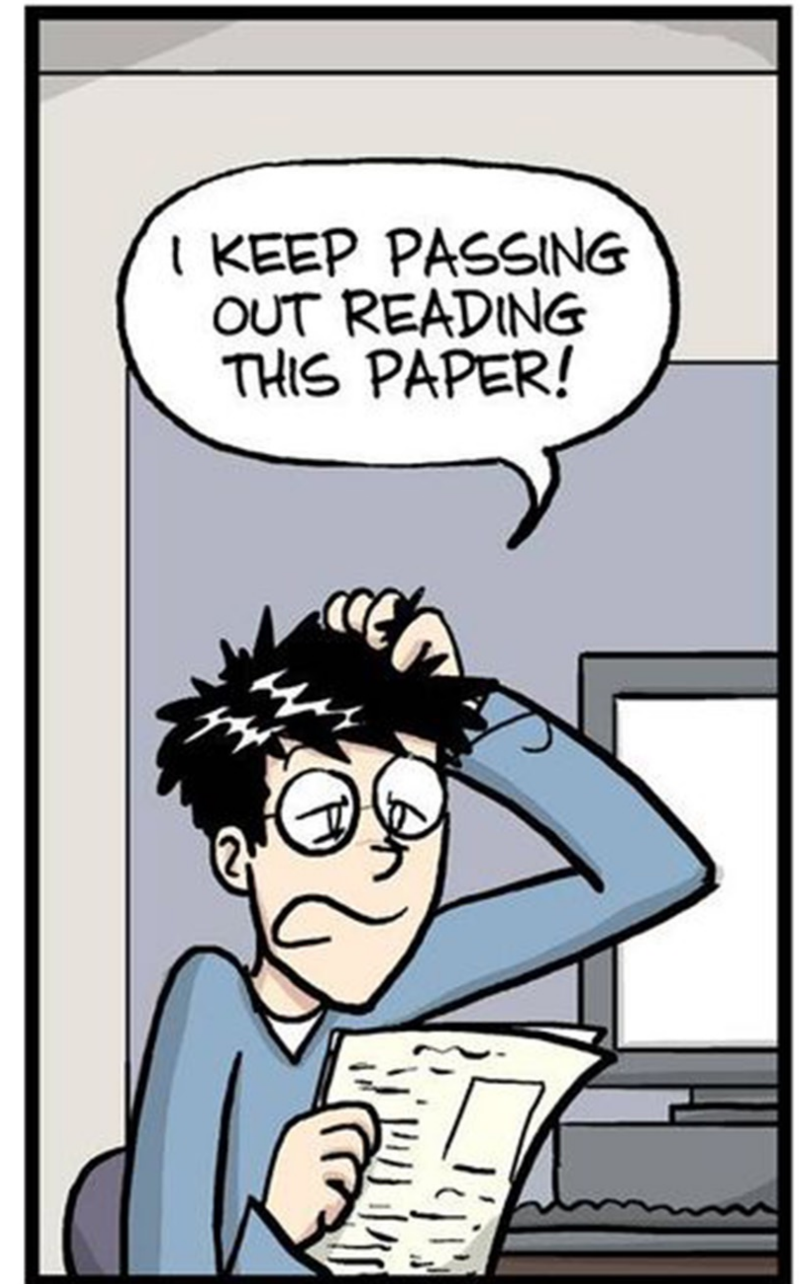


Image courtesy of http://phdcomics.com/

(Systems) Paper Structure

1. Introduction (a.k.a. Problem Stmt)
2. Design (a.k.a. Solution)
3. Prototype (a.k.a. Implementation)
4. Evaluation (a.k.a. Proof)
5. Related Work
6. Conclusion

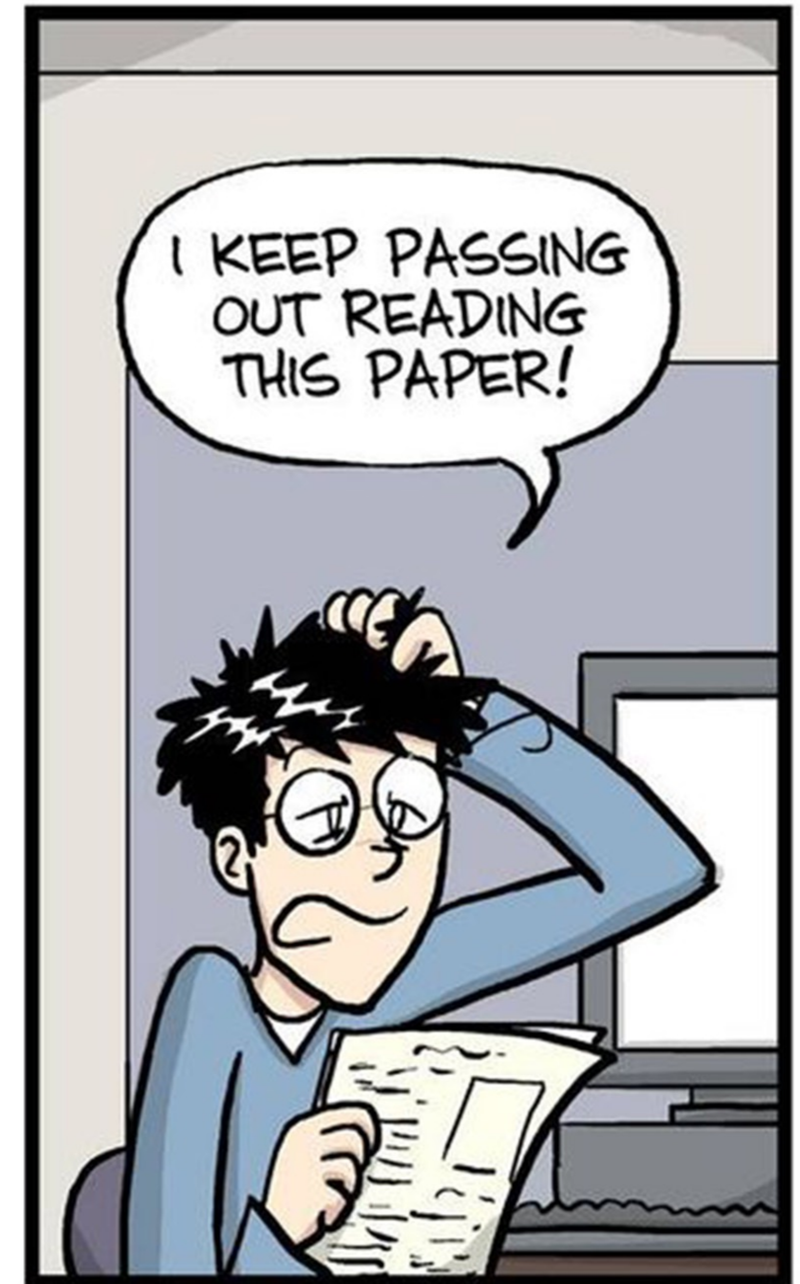


Image courtesy of http://phdcomics.com/

(Systems) Paper Structure

1. Introduction (a.k.a. Problem Stmt)
2. Design (a.k.a. Solution)
3. Prototype (a.k.a. Implementation)
4. Evaluation (a.k.a. Proof)
5. Related Work
6. Conclusion

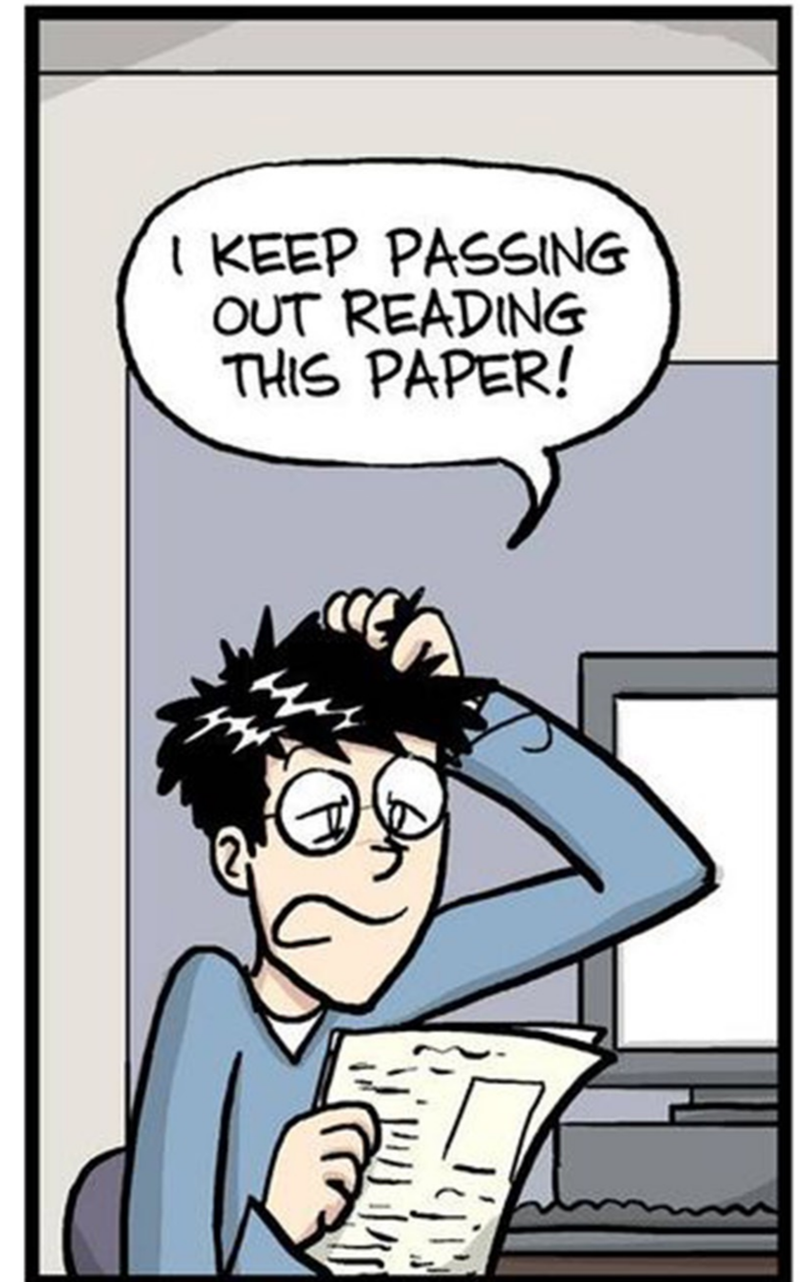


Image courtesy of http://phdcomics.com/

(Systems) Paper Structure

1. Introduction (a.k.a. Problem Stmt)
2. Design (a.k.a. Solution)
3. Prototype (a.k.a. Implementation)
4. Evaluation (a.k.a. Proof)
   - Questions to be answered
   - Experimental setup
   - Experiment 1, 2, …
   - Summary
5. Related Work
6. Conclusion

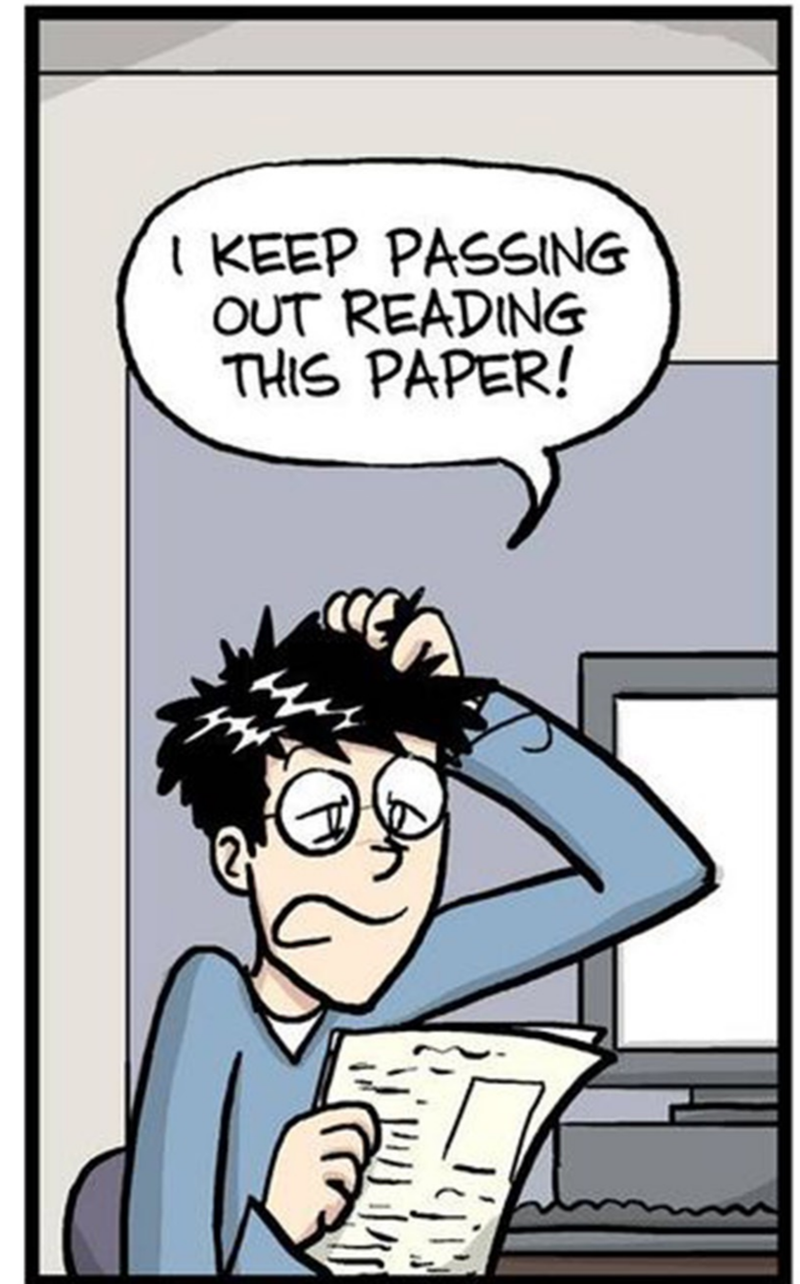


Image courtesy of http://phdcomics.com/

6. EVALUATION

S^2E's main goal is to enable rapid prototyping of useful, deep system analysis tools. In this vein, our evaluation of S^2E aims to answer three key questions: Is S^2E truly a general platform for building diverse analysis tools (Section 6.1)? Does S^2E perform these analyses with reasonable performance (Section 6.2)? What are the measured trade-offs involved in choosing different execution consistency models on both kernel-mode and user-mode binaries (Section 6.3)?

(Systems) Paper Structure

1. Introduction (a.k.a. Problem Stmt)
2. Design (a.k.a. Solution)
3. Prototype (a.k.a. Implementation)
4. Evaluation (a.k.a. Proof)
   - Questions to be answered
   - Experimental setup
   - Experiment 1, 2, …
   - Summary
5. Related Work
6. Conclusion

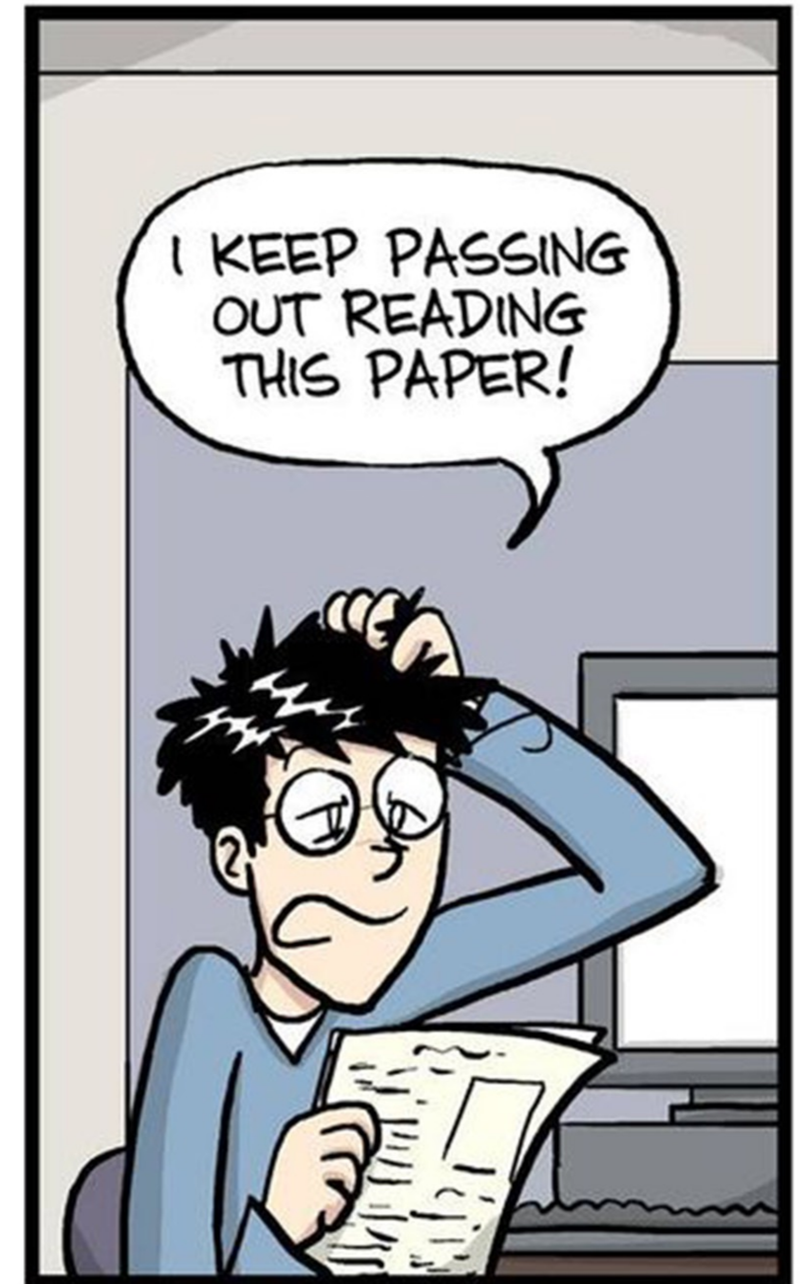


Image courtesy of http://phdcomics.com/

We evaluated RaceMob using a mix of server, desktop and scientific software: Apache httpd is a Web server that serves around 65% of the Web [17]—we used the `mpm-worker` module of Apache to operate it in multi-threaded server mode and detected races in this specific module. SQLite [42] is an embedded database used in Firefox, iOS, Chrome, and Android, and has 100% branch coverage with developer's tests. Memcached [12] is a distributed memory-object caching system, used by Internet services like Twitter, Flickr, and YouTube. Knot [43] is a web server. Pbzip2 [14] is a parallel implementation of the popular `bzip2` file compressor. Pfscan [11] is a parallel file scanning tool that provides the combined functionality of `find`, `xargs`, and `fgrep` in a parallel way. Aget is a parallel variant of `wget`. Fmm, Ocean, and Barnes are applications from the SPLASH suite [41]. Fmm and Barnes simulate interactions of bodies, and Ocean simulates ocean movements.

Our evaluation results are obtained primarily using a test environment simulating a crowdsourced setting, and we also have a small scale, real deployment of RaceMob on our laptops. For the experiments, we use a mix of workloads derived from actual program runs, test suites, and test cases devised by us and other researchers [48]. We configured the hive to assign a single dynamic validation task per user at a time. Altogether, we have execution information from $1,754$ simulated user sites. Our test bed consists of a 2.3 GHz 48-core AMD Opteron 6176 machine with 512 GB of RAM running Ubuntu Linux 11.04 and a 2 GHz 8-core Intel Xeon E5405 machine with 20 GB of RAM running Ubuntu Linux 11.10. The hive is deployed on the 8-core machine, and the simulated users on both machines. The real deployment uses ThinkPad laptops with Intel 2620M processors and 8 GB of RAM, running Ubuntu Linux 12.04.

We used C programs in our evaluation because RELAY operates on CIL, which does not support C++ code. Pbzip2 is a C++ program, but we converted it to C by replacing references to STL `vector` with an array-based implementation. We also replaced calls to `new/delete` with `malloc/free`.

(Systems) Paper Structure

1. Introduction (a.k.a. Problem Stmt)
2. Design (a.k.a. Solution)
3. Prototype (a.k.a. Implementation)
4. Evaluation (a.k.a. Proof)
    - Questions to be answered
    - Experimental setup
    - Experiment 1, 2, …
    - Summary
5. Related Work
6. Conclusion

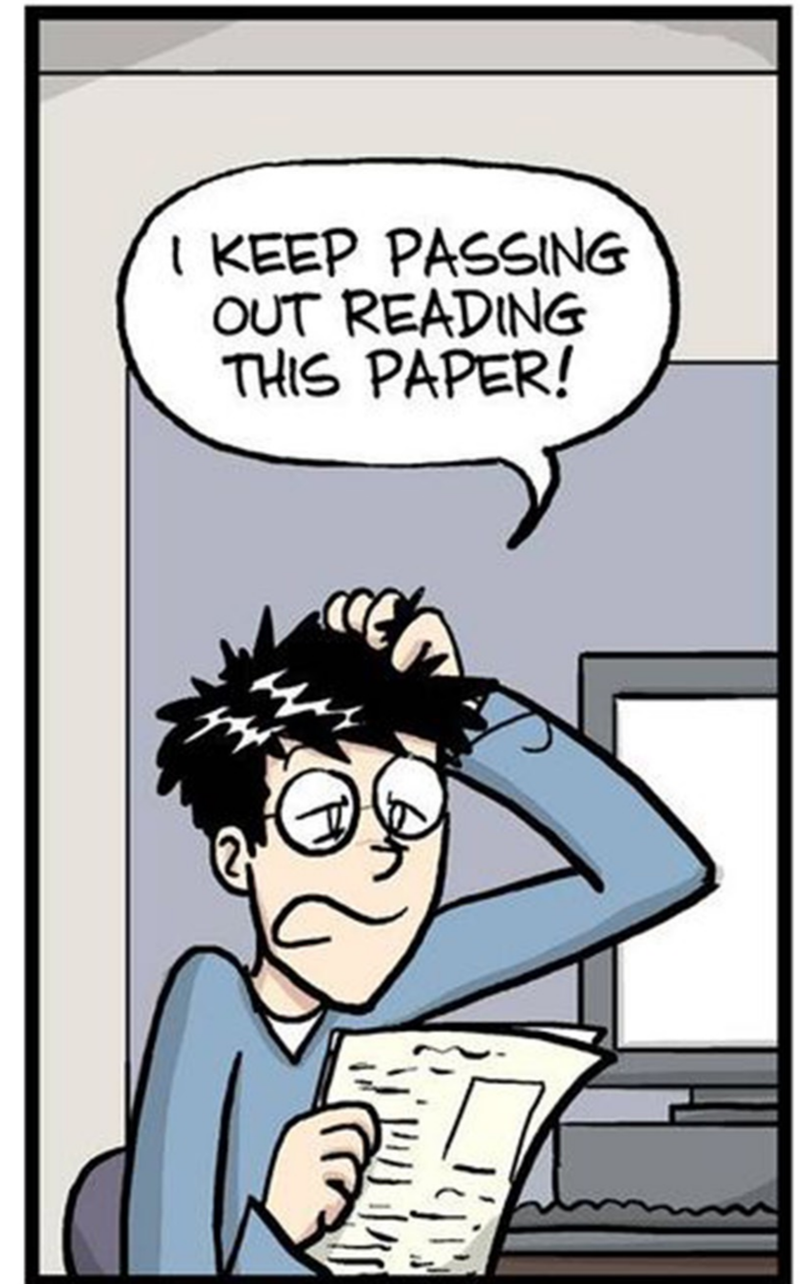


Image courtesy of http://phdcomics.com/

(Systems) Paper Structure

1. Introduction (a.k.a. Problem Stmt)
2. Design (a.k.a. Solution)
3. Prototype (a.k.a. Implementation)
4. Evaluation (a.k.a. Proof)
5. Related Work
6. Conclusion

Discussion (a.k.a. Limitations)

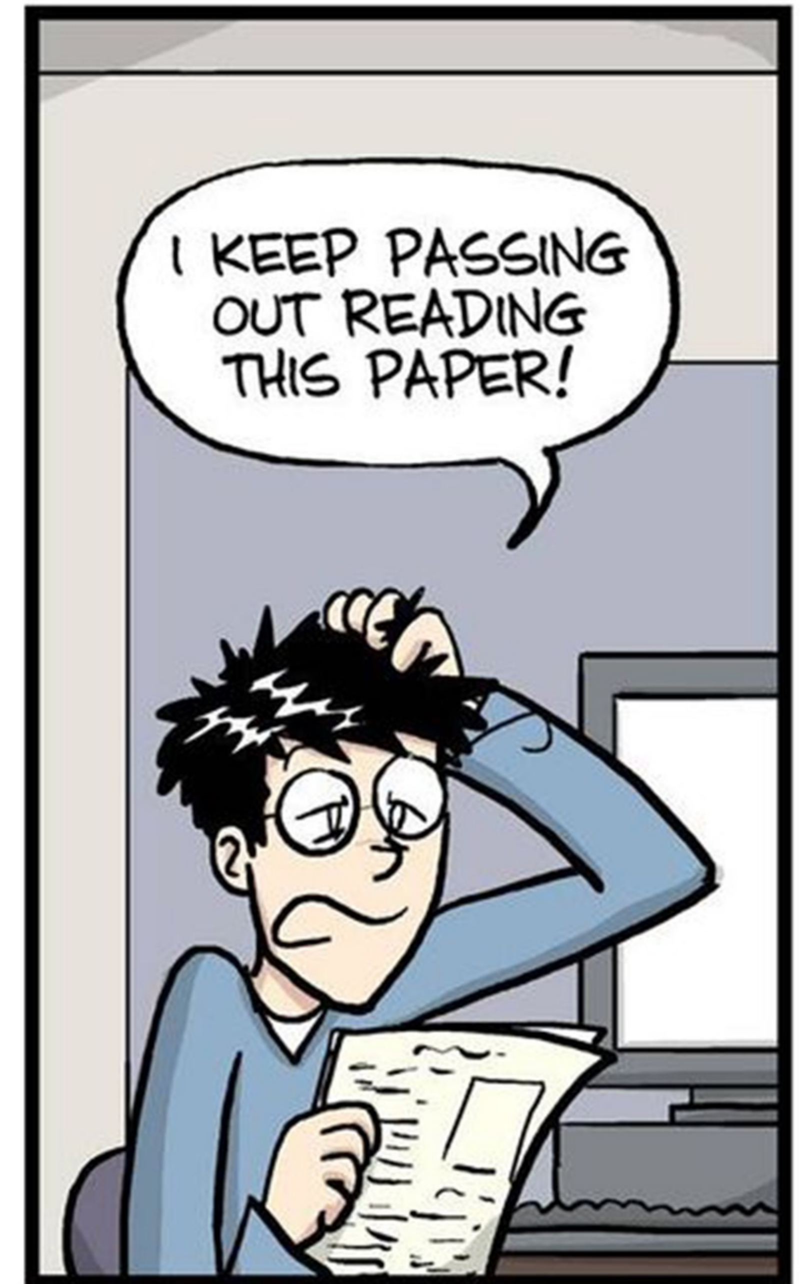


Image courtesy of http://phdcomics.com/

There is a spectrum of approaches for avoiding deadlocks, from purely static techniques to purely dynamic ones. Dimmunix targets general-purpose systems, not real-time or safety-critical ones, so we describe this spectrum of solutions keeping our target domain in mind.

Language-level approaches [3,15] use powerful type systems to simplify the writing of lock-based concurrent programs and thus avoid synchronization problems altogether. This avoids runtime performance overhead and prevents deadlocks outright, but requires programmers to be disciplined, adopt new languages and constructs, or annotate their code. While this is the ideal way to avoid deadlocks, programmers' human limits have motivated a number of complementary approaches.

Transactional memory (TM) [8] holds promise for simplifying the way program concurrency is expressed. TM converts the locking order problem into a thread scheduling problem, thus moving the burden from programmers to the runtime, which we consider a good tradeoff. There are still challenges with TM semantics, such as what happens when programmers use large atomic blocks, or when TM code calls into non-TM code or performs I/O. Performance is still an issue, and [14] shows that many modern TM implementations use lock-based techniques to improve performance and are subject to deadlock. Thus, we believe TM is powerful, but it cannot address all concurrency problems in real systems.

Time-triggered systems [13] and statically scheduled real-time systems [22] perform task synchronization before the program runs, by deciding schedules a priori based on task parameters like mutual-exclusion constraints and request processing time. When such parameters are known a priori, the approach guarantees safety and liveness; however, general-purpose systems rarely have such information ahead of time. Event-triggered real-time systems are more flexible and incorporate a priori constraints in the form of thread priorities; protocols like priority ceiling [20], used to prevent priority inversion, conveniently prevent deadlocks too. In general-purpose systems, though, even merely assigning priorities to the various threads is difficult, as the threads often serve a variety of purposes over their lifetime.

Static analysis tools look for deadlocks at compile time and help programmers remove them. ESC [7] uses a theorem prover and relies on annotations to provide knowledge to the analysis; Houdini [6] helps generate some of these annotations automatically. [5] and [21] use flow-sensitive analyses to find deadlocks. In Java JDK 1.4, the tool described in [21] reported 100,000 potential deadlocks and the authors used unsound filtering to trim this result set down to 70, which were then manually reduced to 7 actual deadlock bugs. Static analyses run fast, avoid runtime overheads, and can help prevent deadlocks, but when they generate false positives, it is ultimately the programmers who have to winnow the results. Developers under pressure to ship production code fast are often reticent to take on this burden.

Another approach to finding deadlocks is to use model checkers, which systematically explore all possible states of the program; in the case of concurrent programs, this includes all thread interleavings. Model checkers achieve high coverage and are sound, but suffer from poor scalability due to the ``state-space explosion'' problem. Java PathFinder, one of the most successful model checkers, is restricted to applications up to ~10 KLOC [10] and does not support native I/O libraries. Real-world applications are large (e.g., MySQL has >1 MLOC) and perform frequent I/O, which restricts the use of model checking in the development of general-purpose systems.

[..]

Deadlock immunity explores a new design point on this spectrum of deadlock avoidance solutions, combining static elements (e.g., control flow signatures) with dynamic approaches (e.g., runtime steering of thread schedules). This combination makes Dimmunix embody new tradeoffs, which we found to be advantageous when avoiding deadlocks in large, real, general-purpose systems.

There is a spectrum of approaches for avoiding deadlocks, from purely static techniques to purely dynamic ones. Dimmunix targets general-purpose systems, not real-time or safety-critical ones, so we describe this spectrum of solutions keeping our target domain in mind.

Language-level approaches [3,15] use powerful type systems to simplify the writing of lock-based concurrent programs and thus avoid synchronization problems altogether. This avoids runtime performance overhead and prevents deadlocks outright, but requires programmers to be disciplined, adopt new languages and constructs, or annotate their code. While this is the ideal way to avoid deadlocks, programmers' human limits have motivated a number of complementary approaches.

Transactional memory (TM) [8] holds promise for simplifying the way program concurrency is expressed. TM converts the locking order problem into a thread scheduling problem, thus moving the burden from programmers to the runtime, which we consider a good tradeoff. There are still challenges with TM semantics, such as what happens when programmers use large atomic blocks, or when TM code calls into non-TM code or performs I/O. Performance is still an issue, and [14] shows that many modern TM implementations use lock-based techniques to improve performance and are subject to deadlock. Thus, we believe TM is powerful, but it cannot address all concurrency problems in real systems.

Time-triggered systems [13] and statically scheduled real-time systems [22] perform task synchronization before the program runs, by deciding schedules a priori based on task parameters like mutual-exclusion constraints and request processing time. When such parameters are known a priori, the approach guarantees safety and liveness; however, general-purpose systems rarely have such information ahead of time. Event-triggered real-time systems are more flexible and incorporate a priori constraints in the form of thread priorities; protocols like priority ceiling [20], used to prevent priority inversion, conveniently prevent deadlocks too. In general-purpose systems, though, even merely assigning priorities to the various threads is difficult, as the threads often serve a variety of purposes over their lifetime.

Static analysis tools look for deadlocks at compile time and help programmers remove them. ESC [7] uses a theorem prover and relies on annotations to provide knowledge to the analysis; Houdini [6] helps generate some of these annotations automatically. [5] and [21] use flow-sensitive analyses to find deadlocks. In Java JDK 1.4, the tool described in [21] reported 100,000 potential deadlocks and the authors used unsound filtering to trim this result set down to 70, which were then manually reduced to 7 actual deadlock bugs. Static analyses run fast, avoid runtime overheads, and can help prevent deadlocks, but when they generate false positives, it is ultimately the programmers who have to winnow the results. Developers under pressure to ship production code fast are often reticent to take on this burden.

Another approach to finding deadlocks is to use model checkers, which systematically explore all possible states of the program; in the case of concurrent programs, this includes all thread interleavings. Model checkers achieve high coverage and are sound, but suffer from poor scalability due to the ``state-space explosion'' problem. Java PathFinder, one of the most successful model checkers, is restricted to applications up to ~10 KLOC [10] and does not support native I/O libraries. Real-world applications are large (e.g., MySQL has >1 MLOC) and perform frequent I/O, which restricts the use of model checking in the development of general-purpose systems.

[..]

Deadlock immunity explores a new design point on this spectrum of deadlock avoidance solutions, combining static elements (e.g., control flow signatures) with dynamic approaches (e.g., runtime steering of thread schedules). This combination makes Dimmunix embody new tradeoffs, which we found to be advantageous when avoiding deadlocks in large, real, general-purpose systems.

(Systems) Paper Structure

1. Introduction (a.k.a. Problem Stmt)
2. Design (a.k.a. Solution)
3. Prototype (a.k.a. Implementation)
4. Evaluation (a.k.a. Proof)
5. Related Work
6. Conclusion

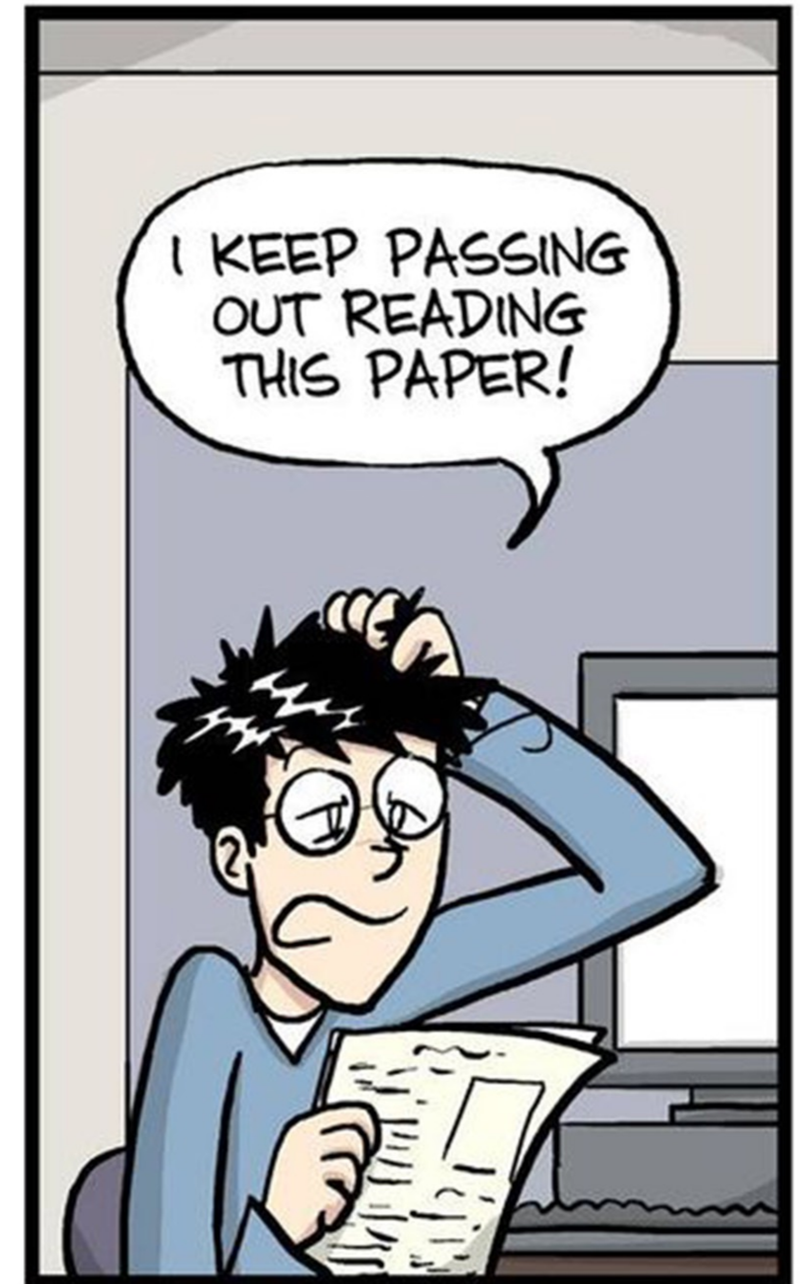


Image courtesy of http://phdcomics.com/

Conclusion

- Technical writing ≠ Lyrical writing
- Write iteratively (the way Picasso drew)
- Clean, recursive structure to ease reader's load
- Avoid opinions, vagueness
- Reduce # of words, increase # of examples
- Clear graphs with explained data
- Example structure for systems papers