Towards efficient and secure shared memory applications

Emmanuel Sifakis

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Emmanuel Sifakis

Thèse dirigée par **Saddek Bensalem**
et codirigée par **Laurent Mounier**

préparée au sein **Verimag**
et de **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Towards efficient and secure shared memory applications

Thèse soutenue publiquement le **6 Mai 2013**,
devant le jury composé de :

Mr. Roland Groz
Professeur à Grenoble INP, Président
Mr. Klaus Havelund
Senior Research Scientist at NASA JPL LARS, Rapporteur
Mr. Gilles Muller
Directeur de Recherche à INRIA REGAL, LIP6, Rapporteur
Mr. Ahmed Bouajjani
Professeur à Paris Diderot, Examinateur
Mr. Saddek Bensalem
Professeur UJF, Directeur de thèse
Mr. Laurent Mounier
Maître de conférences UJF, Co-Directeur de thèse

ABSTRACT

The invasion of multi-core and multi-processor platforms on all aspects of computing makes shared memory parallel programming mainstream. Yet, the fundamental problems of exploiting parallelism efficiently and correctly have not been fully addressed. Moreover, the execution model of these platforms (notably the relaxed memory models they implement) introduces new challenges to static and dynamic program analysis. In this work we address 1) the optimization of pessimistic implementations of critical sections and 2) the dynamic information flow analysis for parallel executions of multi-threaded programs.

Critical sections are excerpts of code that must appear as executed atomically. Their pessimistic implementation reposes on synchronization mechanisms, such as mutexes, and consists into obtaining and releasing them at the beginning and end of the critical section respectively. We present a general algorithm for the acquisition/release of synchronization mechanisms and define on top of it several policies aiming to reduce contention by minimizing the possession time of synchronization mechanisms. We demonstrate the correctness of these policies (i.e., they preserve atomicity and guarantee deadlock freedom) and evaluate them experimentally.

The second issue tackled is dynamic information flow analysis of parallel executions. Precisely tracking information flow of a parallel execution is due to non-deterministic accesses to shared memory. Most existing solutions that address this problem enforce a serial execution of the target application. This allows to obtain an explicit serialization of memory accesses but incurs both an execution-time overhead and eliminates the effects of relaxed memory models. In contrast, the technique we propose allows to predict the plausible serializations of a parallel execution with respect to the memory model. We applied this approach in the context of taint analysis, a dynamic information flow analysis widely used in vulnerability detection. To improve precision of taint analysis we further take into account the semantics of synchronization mechanisms such as mutexes, which restricts the predicted serializations accordingly.

The solutions proposed have been implemented in proof of concept tools which allowed their evaluation on some hand-crafted examples.

RÉSUMÉ

L'utilisation massive des plateformes multi-cœurs et multi-processeurs a pour effet de favoriser la programmation parallèle à mémoire partagée. Néanmoins, exploiter efficacement et de manière correcte le parallélisme sur ces plateformes reste un problème de recherche ouvert. De plus, leur modèle d'exécution sous-jacent, et notamment les modèles de mémoire "relâchés", posent de nouveaux défis pour les outils d'analyse statiques et dynamiques. Dans cette thèse nous abordons deux aspects importants dans le cadre de la programmation sur plateformes multi-cœurs et multi-processeurs: l'optimisation de sections critiques implémentées selon l'approche pessimiste, et l'analyse dynamique de flots d'informations.

Les sections critiques définissent un ensemble d'accès mémoire qui doivent être exécutées de façon atomique. Leur implémentation pessimiste repose sur l'acquisition et le relâchement de mécanismes de synchronisation, tels que les verrous, en début et en fin de sections critiques. Nous présentons un algorithme générique pour l'acquisition/relâchement des mécanismes de synchronisation, et nous définissons sur cet algorithme un ensemble de politiques particulières ayant pour objectif d'augmenter le parallélisme en réduisant le temps de possession des verrous par les différentes threads. Nous montrons alors la correction de ces politiques (respect de l'atomicité et absence de blocages), et nous validons expérimentalement leur intérêt.

Le deuxième point abordé est l'analyse dynamique de flot d'information pour des exécutions parallèles. Dans ce type d'analyse, l'enjeu est de définir précisément l'ordre dans lequel les accès à des mémoires partagées peuvent avoir lieu à l'exécution. La plupart des travaux existant sur ce thème se basent sur une exécution sérialisée du programme cible. Ceci permet d'obtenir une sérialisation explicite des accès mémoire mais entraîne un surcoût en temps d'exécution et ignore l'effet des modèles mémoire relâchées. A contrario, la technique que nous proposons permet de prédire l'ensemble des sérialisations possibles vis-a-vis de ce modèle mémoire à partir d'une seule exécution parallèle ("runtime prediction"). Nous avons développé cette approche dans le cadre de l'analyse de teinte, qui est largement utilisée en détection de vulnérabilités. Pour améliorer la précision de cette analyse nous prenons également en compte la sémantique des primitives de synchronisation qui réduisent le nombre de sérialisations valides.

Les travaux proposés ont été implémentés dans des outils prototypes qui ont permis leur évaluation sur des exemples représentatifs.

Acknowledgements

Foremost, I would like to express my deepest gratitude to my co-advisor Laurent Mounier. His immense enthusiasm, support, availability and attention to details made it a real pleasure to work with him. I also sincerely thank my advisor Saddek Bensalem for his integrity and support.

I would like to thank all members of the jury for their implication in the evaluation of my thesis. A special thanks to Klaus Havelund and Gilles Muller for thoroughly examining my thesis but also for their enriching comments. I also thank Ahmed Bouajjani for his participation in the jury and Roland Gros who accepted to president the dissertation.

I also thank all my colleagues and officemates at Verimag for their support and encouragement over all these past years. A special thanks to my friends Paris, Vasso and Tesnim with which we shared many good moments but also went through a lot.

Last but not least I am deeply grateful to my family. My parents George and Marie-Claude to whom I dedicate this thesis. I thank them for raising me with good values, and making my education always a priority. I also thank my sisters Catherine and Sandra for their encouragement. Lastly I thank Manolya for all her support especially during the hard part of redacting this document.

Contents

List of Figures

Chapter 1

Introduction

Computer systems are evolving with an astonishing rapid rate and are extensively used in various contexts, from scientific computations to critical systems, personal computers and consumer electronics. In all cases there is an insatiable demand for *performance*, since it allows to accomplish more complex tasks faster. Moreover, due to the importance of information manipulated by computer systems *correctness* and *security* are a major concern.

1.1 Increasing computing power

A computer is a machine based on the *Von Neuman* architecture. It consists of a *central processing unit* (CPU) connected to a *memory* and some *input/output* (I/O) devices to communicate with the environment. The CPU is capable of performing a finite set of arithmetic and logic operations on data with a constant speed defined by the CPU clock. The sequence of operations to be performed by the CPU are specified in programs. Both programs and data reside in memory.

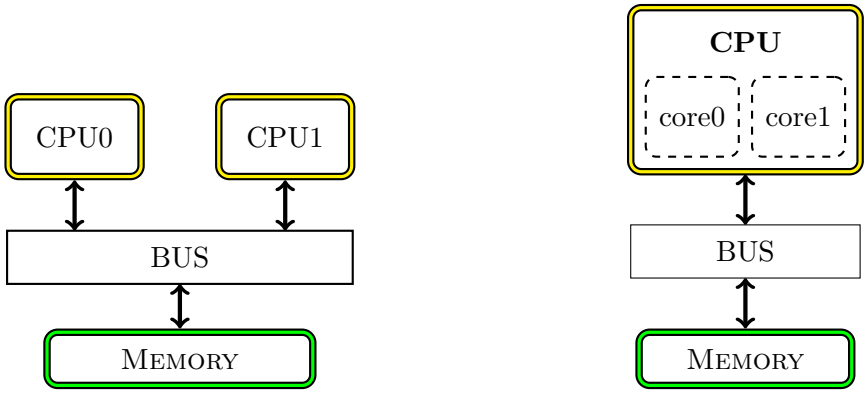
Increasing computing power implies performing more calculations in less time. For several decades the increase of performance in computer systems reposed mainly on the advance of hardware. According to *Moore's law* the cost effective density of transistors per chip should double approximately every two years. This implied that faster CPUs and larger memories could be fabricated all at reduced costs. Of course all this gain did not come for free. Faster CPUs need more electric power, and in combination with the reduced size heating of chips reaches critical levels.

Increasing the clock speed of a CPU was not sufficient itself for obtaining significant performance gains. The usage of CPUs had to be optimized (i.e., clock cycles should not be waisted). To that end several architectural changes have been applied such as: introduction of *cache memories* to hide latency for accessing main memory, or usage of dedicated buses for signals and data transfers. Another type of optimization was to add parallelism at the hardware level. It came in the form of *pipelining, instruction level parallelism (ILP) and simultaneous multi-threading*. This type of parallelism is implicit because it is transparent to the programmers.

Reaching physical limits and running out of coarse optimizations with significant benefits, hardware design was oriented towards explicit parallelism. That is, computers consisting of multiple processing units. This came in two flavors:

multiprocessor: multiple CPUs interconnected into some topology are placed inside a single computer system. Figure 1.1(a) illustrates a multiprocessor system consisting of two CPUs;

multicore: multiple processing units (called cores) are placed on a single CPU chip. Figure 1.1(b) presents a multicore CPU consisting of two cores.



(a) Multiprocessor (b) Multicore

Figure 1.1: Explicitly parallel computer systems.

Programmers must be aware of explicit parallelism in order to exploit it correctly and efficiently. This turns out to be an utterly challenging task. Parallelism adds nondeterminism which makes it more difficult to reason about a programs correctness. Moreover, analyses and debugging tools that were used on sequential programs must be adapted to deal with parallelism.

1.2 Exploiting computer power

The switch from single processor to parallel architectures (multicores and multiprocessors) has a great impact on the way software is conceived. Better performance of a program no longer comes for free. To improve performance, programs must be parallelized. There are two sources of parallelism, *data parallelism* and *task parallelism*. *Data* (or loop-level) parallelism takes advantage of independent data to partition them into data sets and execute the same computation function on each partitioning concurrently. Figure 1.2 on the facing page illustrates an example of loop parallelization. The original loop on the left can be split and be executed in parallel as is the case on the right. *Task parallelism* focuses on executing different computation functions concurrently, and potentially on the same data sets.

Parallel computation is not a new concept in computer systems. In the early 70s already *vector processing* was developed to increase performance of mathematical computations.

```
1 int data[1000];
2 for(i=0; i<1000; i++){
3   data[i]++;
4 }
```

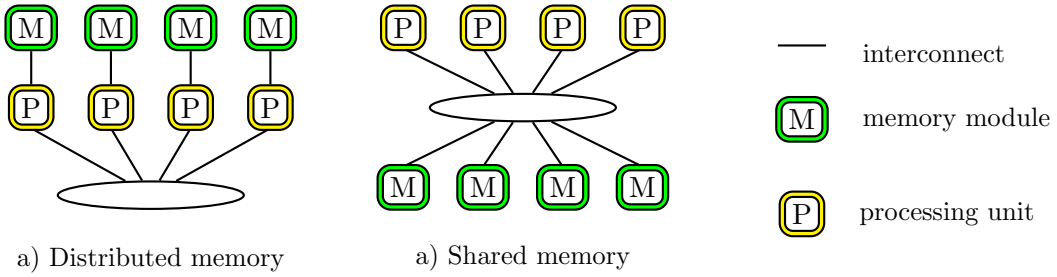
```
1 int data[1000]
2 % CPU0
3 for(i=0; i<500; i++){
4   data[i]++;
5 }
6 % CPU1
7 for(i=500; i<1000; i++){
8   data[i]++;
9 }
```

Figure 1.2: Data parallelism (loop parallelization).

Vector computers consist of a master processor which dispatches instructions to a number of processing units which execute them simultaneously on different memory locations. Such type of supercomputers were exclusively used for scientific and engineering purposes. A great break through to parallel computing came with the development of networks and the increased performance and reduced cost of personal computers. This allowed to build supercomputers by connecting custom of the shelf computers into *clusters* and *grids*.

A *cluster* is typically a set of homogeneous tightly connected computers, usually through a high speed local network (e.g., Myrinet). A *grid* is a collection of heterogeneous loosely connected computers, often through the *Internet*. Grids are mostly seen as a distributed system while clusters often appear as a single computational unit. Both clusters and grids are used in *high performance computing* (HPC) to solve complex problems. Two new trends in HPC are *cloud computing* and *general purpose graphics processing unit* (GPGPU) computing. Cloud computing offers access to remote computer resources through a service while GPGPU allows to exploit the computing power of graphics processing units, which consist of thousands of light cores specially designed for parallel performance.

Parallel computers independently of their scale (multi-core, multiprocessor, cluster, grid) can be abstracted using three building blocks: (i) processing units (ii) memory modules and (iii) interconnections. An interconnect links processing units between them or with memory modules. Figure 1.3 illustrates two common parallel architectures, the *distributed memory* on the left and the *shared memory* on the right. In the *distributed memory* architecture each processing unit has a private/local memory and can access the memory of other processing units through the network. In the *shared memory* case all processing units are directly connected to a globally visible shared memory.



a) Distributed memory a) Shared memory

Figure 1.3: Parallel systems abstraction

1.2.1 Parallel programming models

Processing units of a parallel system need to exchange data and synchronize in order to accomplish a common task. Communication is done either using *message passing* or through *shared memory*. In *message passing*, as its name indicates, messages are sent between processing units. This communication model is well adapted for distributed parallel systems where messages are sent through a network. An interface known as MPI (Message Passing Interface) has been standardized for message passing communications, and several implementations of it exist. In the *shared memory* model data are transfered by placing them into designated locations of the globally visible address space. The synchronization is also done through shared memory, using synchronization primitives such as locks.

Both the *message passing* and *shared memory* communication mechanisms do not need to correspond directly to the underlaying platform. That is, message passing can be implemented through shared memory and dually shared memory can be simulated through *distributed shared memories*. In the rest of the document we focus to parallel computer systems with *physically shared memory*. That is, we consider multiprocessor and multicore architectures. In these architectures the shared memory model fits naturally.

1.2.2 Shared memory architectures

Physically shared memory architectures can be devised into *uniform memory access* (UMA) and *non-uniform memory accesses* (NUMA) as illustrated in Figure 1.4. In UMA architectures accesses to the shared memory take the same time independently of which processor issued them. In NUMA architectures processors can still access any memory location, but the time required depends whether the memory location resides in its own local memory (immediate interconnection with memory, faster access) or to the local memory of other processors (non-immediate interconnection with memory, slower access). NUMA architectures have a structure similar to distributed memories.

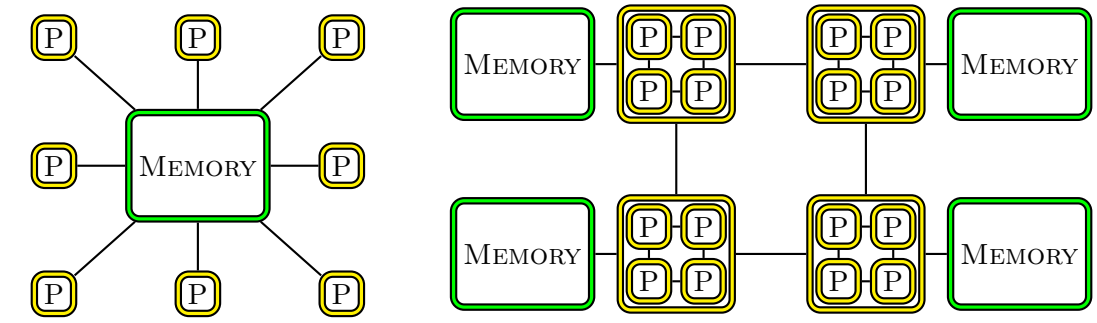


Figure 1.4: Physically shared memory architectures.

The architectures can also be classified by the types of processors they are composed of. In *symmetric multiprocessor* (SMP) systems all processors have the same characteristics. Dually, in *asymmetric multiprocessor* (ASMP) systems the processors used can have different characteristics (e.g., CPU speed).

1.2.3 Caveats of shared memory

In the shared memory model, independently of the underlaying architecture, concurrent modifications to common memory locations can cause *data races*. That is, the outcome of the program executed depends on the order in which memory updates took place. Data races can have disastrous results. Thus, precautions must be taken, when necessary, to eliminate them. In the shared memory model *synchronization mechanisms* that guarantee *mutual exclusion* are used to enforce an order among *critical sections* (portions of code containing that should be executed atomically). The mis-usage of synchronization mechanisms such as *locks* can introduce deadlocks i.e., disallow permanently the blocked parts to make any progress.

Mutual exclusion of critical sections implies they are serialized, which is necessary for the correctness of the programs. Serialization in parallel programs should be minimized in order to obtain significant performance gains. As formulated in *Amdahl's law*: the speedup of a program using multiple processors in parallel computing is limited by the time needed to execute the sequential fraction of the program.

The abstraction we used (processing units, memory, interconnection) to describe parallel computers provides an intuitive overview of the architectures but also hides many details such as *caches*. Caches are small and very fast memories which store copies of picked memory locations in order to hide the latency of main memory accesses. Maintaining caches coherent is a major issue because a processor may miss the update of a cached memory location residing on an other processor. To resolve those issues several *cache coherency* protocols exist.

Because shared memory parallel architectures can widely vary (UMA, NUMA, different levels of cache, cache coherence protocols used) the behavior of a program can also vary according to the platform it is executed on. It is hence necessary to define a *memory consistency model* which specifies the behavior of memory with respect to simultaneous accesses to the same location and issued by different processors. To gain in performance several *relaxed consistency models* have been proposed and applied to a vast majority of commercial architectures.

1.3 Security and correctness

1.3.1 Information security

The importance of information is invaluable and a number of security properties such as (i) confidentiality (ii) integrity and (iii) availability must be guaranteed. Nowadays information is stored, processed and transfered among a multitude of devices. Although encryption algorithms and communication protocols can ensure integrity and confidentiality, of data storage and transfer, they are not always used properly. Data are often stored un-encrypted and their security relies entirely on the operating system access control which is insufficient.

Benign software may unintentionally leak confidential data, or contain *vulnerabilities*

that could be exploited by malicious software (malware). Malwares come in many flavors, viruses, worms, trojan horses, spywares etc. and focus in disclosing confidential information, or causing denial of services (e.g., crash programs) or capture the host computer (zombie computer) in order to accomplish further malicious tasks. To protect against software vulnerabilities *information flow* analysis is mandatory.

Information flow analyses trace how data processed by a program transit inside memory at execution time. Two popular analyses focusing on preserving security are *non-interference* and *taint* analysis. *Non-interference* focuses on confidentiality. It ensures that *high* (confidential) data do not flow into *low* (public) data. A program is safe if the same outputs are observed for different values of *high* data. *Taint* analysis on the other hand tracks untrusted data such as user or network input and checks how they influence vulnerable statements (e.g., return address of functions). Taint analysis was originally implemented in the *Perl* language for identifying security risks on web sites such as *SQL injections* and *buffer overflow* attacks. The term taint analysis is now widely used as a synonym to *dynamic information flow tracing* (DIFT). Moreover, the type of information traced is not restricted to untrusted data and thus more general analyses can be implemented.

1.3.2 Program validation

Further to the security aspects, correctness of applications is also crucial. To state a program is correct a formal specification (mathematical description) must be provided and *formal verification* techniques should be employed to demonstrate it. Intuitively, an application is correct when it exhibits the expected behavior. Formal methods give verdicts about all executions of a program. Though, to overcome the complexity and non-determinism of the analyzed systems they use abstractions which usually over-approximate the systems behavior. This leads to false positives, i.e., finding errors that could not occur in a real execution of the program. A somehow dual technique called *testing* is widely used to validate program executions.

Testing has become an indispensable part of software development. It is applied to validate if a program meets its functional and extra-functional requirements for a specific (specially guided) execution. Testing is applied dynamically, that is by executing a program or fragment of a program and observe the execution for the tested property. We must note that testing is not exhaustive and thus it cannot be used for verification. In order to increase the confidence of programs *stress testing* is applied to increase coverage of tested executions. If any functional errors (bugs) are found during testing they should be resolved through the debugging process.

Debugging of programs consists in finding errors (bugs) and correcting them accordingly. In order to debug an error it should be *reproducible* so that it can be replayed as many times necessary to detect its source. Several tools called debuggers help developers in this process. They allow a step-by-step execution of the program and inspection of memory at any point.

The non-determinism of concurrent program executions makes information flow track-

ing, testing and debugging more challenging. Concurrent modifications to shared memory affect the propagation of taintness and observations of testing. Taint propagation or verdict of tests can be different for repeated executions using the same inputs. The non-deterministic execution is affected by many factors such as scheduling decisions taken by the operating system or even the serialization of events by the execution platform. Reproducing a non-deterministic execution precisely is nearly impossible. Thus, we can no longer specify precisely the execution that was tested, nor can we easily reproduce bugs in order to fix them. A partial solution to checking non-deterministic executions is *runtime prediction*. The intuition behind it is to infer/predict plausible serializations based on a concrete execution of the observed application. Thus, upon a single test execution we predict several neighboring executions which would be hard to enforce.

1.4 Our contribution

This thesis addresses two topics related to parallel executions of multi-threaded programs on shared memory architectures. The first problem addressed is the optimization of pessimistic implementations of critical sections. The second problem addressed is the taint analysis for parallel executions of multithreaded programs on multicore architectures.

1.4.1 Optimizing pessimistic critical sections

Critical sections are widely used to enforce mutual exclusion between conflicting accesses to shared resources, e.g., shared memory locations. They are traditionally implemented using synchronization mechanisms such as locks. The locks necessary to protect a critical section are in most cases obtained prior to entering the critical section and released only once the critical section has been executed.

In this work we identify the properties on locks that must be respected such that the semantic of critical sections is preserved. We then propose several policies intending to minimize possession of locks and thus increase parallelism at execution time. The contributions of this work are summarized as follows:

- The definition of a generic lock acquisition/release algorithm for critical sections.
- The instantiation of five policies on top of our algorithm. Correctness of these policies with respect to *deadlock freedom* and *absence of data races* is provided.
- The definition of a new exclusion mechanism called *read write intend*.
- The development of a library implementing several exclusion mechanisms, including *read write intend*. The library also provides functionality for serving atomically the acquisition and release of sets of synchronization mechanisms.

Publications

Sifakis Emmanuel and Mounier Laurent. Politiques de gestion de protections pour l'implémentation

de sections critiques. RENPAR, 2011

Sifakis Emmanuel and Mounier Laurent. Politiques de gestion de protections pour l'implémentation de sections critiques. TSI special RENPAR, 2012

1.4.2 Predictive information flow analysis

Information flow analysis is often applied at runtime since it is more precise. Performing information flow analysis on shared memory programs executing in parallel is utterly difficult. To overcome the difficulties introduced by parallel execution the majority of existing works impose the serialized execution of the analyzed applications. Note that in this case information flows inferred are restricted to the observed serialization.

In this work we propose a predictive information flow analysis for parallel programs. Our analysis allows the parallel execution of applications which is the input to our prediction algorithm. The prediction is applied to bounded portions of code that were executed "simultaneously". Predictions are inferred by the iterative algorithm we propose. The prediction focuses into capturing information flows produced by plausible serializations of the parallel execution observed. The predictions should take into account the characteristics of the execution platform. We use *taint* analysis as a representative information flow analysis and consider sequentially consistent platforms. A considerable effort has been done to reduce false predictions. The contributions of this work are summarized as follows:

- Proposition of a runtime prediction technique for parallel executions.
- Definition of an algorithm for precise predictive taint analysis. The precision of predictions comes from:
  - (i) taking into account the underlaying memory model;
  - (ii) safely un-tainting memory locations;
  - (iii) respecting semantics of synchronization mechanisms (locks).
- The algorithm we propose avoids the enumeration of all serializations.
- Implementation of a proof of concept tool.

Publications

Sifakis Emmanuel and Mounier Laurent. Dynamic information-flow analysis for multi-threaded applications. *ISOLA*, 2012

1.5 Organization of the thesis

The thesis is organized in six chapters. Chapter 2 provides background information on threads and formalizes their execution. In chapter 3 we present the contribution on *optimizing critical sections*, while in chapter 4 the contribution on *offline predictive taint*

analysis. Chapter 5 presents the proof of concept experimentations we conducted as well as some tools that were developed. Finally, chapter 6 provides some perspectives and concludes the thesis. Below, are some more detailed descriptions on the content of each chapter.

Chapter 2 provides background information on *thread programming model*. We start with a detailed description of different types of threads (kernel, user level). Next, we abort the challenges introduced by thread programming, *races* and *synchronization*. The chapter ends with a short presentation of weak memory models and the formalization of threads and their execution.

Chapter 3 initially details critical sections and gives an overview on the two main implementation approaches *optimistic* and *pessimistic*. Next, it presents some literature on improving pessimistic implementations of critical sections. Then we present our algorithm for acquiring and releasing protections and how it is used to compute protections to be held by each instruction in the critical section according to our policies. In total, we define five polices and prove they respect the desired properties.

Chapter 4 first presents taint analysis (both implicit and explicit). Then it provides background information on dynamic taint analysis. The framework for offline explicit taint prediction is detailed and applied to a completely relaxed memory model, as well as to a sequentially consistent one. For the sequentially consistent memory model we also provide details on the refinement of taint predictions by taking into account untainting of variables and the semantic of synchronization mechanisms (locks).

Chapter 5 contains the experimentations conducted and the tools developed. The experiments on critical sections although hand-crafted tend to simulate computations commonly found in image processing, with high contention. The experiments on taint propagation are also on hand-crafted examples and the implemented tool is mostly a proof of concept.

Chapter 6 concludes with a summarization of the thesis and presents some perspectives for the works on *optimizing critical sections* and *predictive information flow*.

Chapter 2

Thread programming model

With SMP and multicore architectures becoming ubiquitous shared memory programming goes mainstream. The *thread programming model* which was widely used even on mono-processor architectures is suitable for exploiting the parallelism offered by the multicore and multiprocessor architectures. In this chapter we provide an overview of thread programming and the challenges programmers have to face.

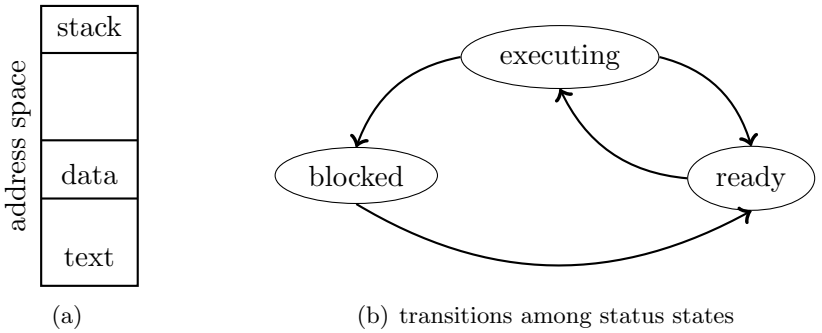
2.1 What are threads?

A *thread* or *light weight process* defines a separate stream of execution within a *process*. To make the definition more concrete we present hereafter some background information on *operating systems* (OS). We detail processes and threads and point out their differences. The information we provide is kept at a high level and is based on the *Unix* OS. More detailed information on operating systems is provided in [Tan01].

2.1.1 Description of a process

A fundamental task of operating systems is to instrument the execution of several programs concurrently. This is called *multitasking*. Because the resources (most importantly the CPU) are limited the OS has to rapidly switch the execution of programs so that they appear as executing in parallel. The process abstraction:

- provides isolation between running programs by assigning to each a virtual portion of main memory (RAM) called the *address space* in which the program can read and write. The address space is logically split into three parts as illustrated in Figure 2.1(a). The *text* segment contains the programs code while the *data* segment the global variables. The stack segment stores information on routines executed by the program. The free space between data and stack is used for dynamic memory allocation.
- stores at the OS level (e.g., in the process table) a multitude of attributes related to a programs execution such as:



(a) (b) transitions among status states

Figure 2.1: Address space and status of a process

- *general registers*
- *program counter*
- *stack pointer*
- *status*, which can be either of:
  - *executing* when the process uses a CPU;
  - *blocked* when a process gets blocked by an event e.g., reading from hard disk;
  - *ready* when it is waiting to be scheduled on a CPU.

  Figure 2.1(b) illustrates in an automaton the transitions among the status of a process.
- *file descriptors*
- *priority*
- *signals*

2.1.2 Description of threads

As mentioned earlier a thread defines a separate execution stream within a process. An execution stream consists of the following information which is necessary for resuming the execution on a CPU:

- program counter
- stack pointer
- register values
- status

Defining a thread implies storing the minimum information listed above for the thread. The rest of the information stored for a process is shared among threads. A process can hence be conceived as a mean of grouping resources (address space (memory), files, devices etc.) which are accessible by its threads. By default a process consists of a single thread.

Creation and scheduling of threads

There are two ways of creating a thread: manually (*user-level* threads) or through the OS (*kernel* threads). Each has its advantages and weak points. *User-level threads* were most popular when OS did not support threads (still some do not). In this case information on threads is entirely stored in the address space of the process. Moreover, the programmer is responsible of storing and resuming correctly threads. *Kernel threads* are created and managed by the OS so information on resuming a threads execution is stored at the OS level. Dually, the stack associated to the thread resides in the address space of the process. The number of threads the OS can manage may be limited. In any case, the creation of threads is much faster than that of processes because no resources need to be acquired (e.g., assign a new address space).

For *user-level* threads the OS assumes managing a regular process, thus in case of a multiprocessor it will always provide only one CPU to execute on. Moreover, scheduling of *user-level* threads is at the complete responsibility of the programmer. This can be advantageous because any scheduling policy can be applied according to the needs of the program. *Kernel* threads on the other hand are scheduled by the OS (multi-threading). Dually to *user-level threads* the OS can assign multiple CPUs to the process and thus execute threads in parallel. In both cases switching between threads is much faster than switching between processes. Finally, the two types of threads can be combined inside a process.

Figure 2.2 presents an instance of a computer system stacked into three layers. At the bottom is the *hardware* in the middle the OS, and at the top the *processes* being executed. The OS has a set of available scheduling units which are linked to a process. A schedulable unit is assigned to a process for each *kernel thread* it contains. A process always has at least one *kernel thread* (as in proc1) which executes it. Process three (proc3) for instance has three *kernel threads* one of which is the main thread and the other two were spawned, as well as two *user-level threads*. Finally, the lines connecting a schedulable unit to a CPU denote the scheduling at a given time.

2.2 Common usage of threads

In section 2.1 we gave an insight of threads implementation and how they are related to processes. Processes can be seen as a way of grouping resources while threads as entities to be scheduled for execution. Hereafter we present how the benefits of threads are most commonly exploited.

The main motivation behind threads is to allow a process to progress while waiting on a blocking instruction such as reading a file. A great example is a text editor which stores a backup of the current work while the user edits the text. Instead of blocking the text editor, a thread is spawned and performs the blocking I/O operation while an other thread keeps processing the text. Similarly, all programs with user interaction use threads to remain reactive to user input.

Another common case combines two properties of threads. Access to the same set of

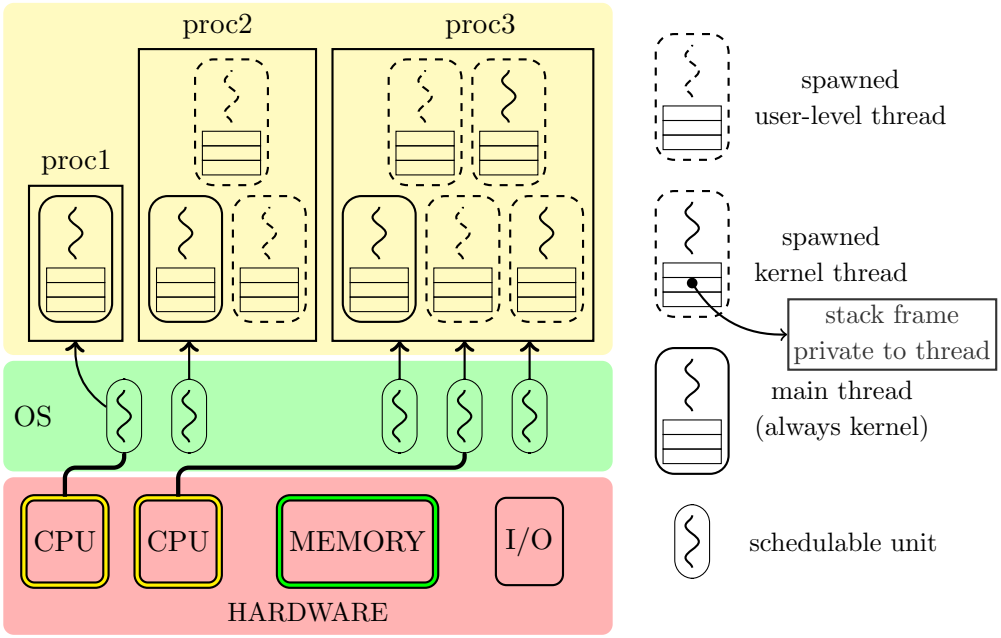


Figure 2.2: Overview of processes and threads.

resources and cheap creation and destruction cost. A server which has to reply to a large number of requests can take advantage of these two characteristics of threads. First, it can rapidly create a thread per request, thus it can concurrently serve many clients. Secondly, by sharing the resources it can be more efficient. For example, a web server could read an *html page* once and forward it to a number of clients that requested it.

Finally, multicore and multiprocessor architectures allow the parallel execution of threads, at least for the number of kernel threads. This can be used to increase performance of applications by parallelizing loops or doing parallel sorting etc. Parallel algorithms are privileged because thread synchronization is much easier to achieve (compared to using inter process communication) and data are immediately shared.

2.3 Challenges of threads

The benefits of thread programming do not come for free. Concurrent modifications to the shared memory of a process can cause several bugs such as inconsistent data, memory leaks, program crashing, but can also have disastrous consequences [LT93, Pou04]. Concurrent implies both interleaving instructions of threads and executing them in parallel (i.e., simultaneously each on a CPU or core). In any case non-determinism creates races which can be categorized into *race conditions* and *data races*. The two types of races overlap and thus are often used interchangeably. We provide the definition of each and give an example to clear any ambiguity.

Race condition: occurs when the ordering of events affects the programs correctness. The ordering may vary due to several reasons such as non-deterministic scheduling, interrupts, memory operations on multiprocessors etc.

Data race: is a special case of race condition where two (or more) threads attempt to access a memory address simultaneously, at least an operation is a write and no ordering of the events is enforced. We must note that a data race can be benign.

The overlapping of the definitions is due to the fact that data races are susceptible to turn into race conditions. Resolving data races by no means implies elimination of race conditions. Race conditions are more related to the semantic of what is executed and thus are harder to reason about, identify and resolve.

To disambiguate the definitions we use the example of a bank account withdraw (inspired from [Reg11]). It is intuitive that each withdrawal should be atomic. That is, concurrently withdrawing from an account should reduce the balance of the account by an amount equivalent to the sum of all concurrent withdraws. Listing 2.1 contains a naive implementation of function `withdraw` which removes the specified `amount` from the given account `acc`.

```
1 withdraw(Account acc,int amount){
2   bal = acc.balance;
3   if( bal >= amount ){
4     acc.balcance = bal-amount;
5   }
6 }
```

Listing 2.1: Naive withdraw example.

If two threads $t1, t2$ issued concurrently a withdraw from an account X then the effect of the withdraw issued by a thread could be suppressed. Such a case occurs if both threads read the same initial balance from account X and then race for writing the new balance of the account. The outcome depends on which thread will write last, since thats the value that will persist. In this naive implementation we have both race conditions because the outcome depends on the scheduling of the threads but also data races since no ordering on memory accesses is enforced.

To correct the implementation of `withdraw` we must eliminate both data races and race conditions. We use the mutex synchronization mechanism which, as its name denotes, provides mutual exclusion. We assume each account is attributed a mutex (accessible through `acc.mutex`). More details on synchronization mechanisms are provided in section 2.5.

The implementation of `withdraw` presented in Listing 2.2 is data race free because all accesses to attributes of the account are synchronized. This however does not prevent race conditions to occur.

```
1 withdraw(Account acc,int amount){
2   lock(acc.mutex);
3     bal = acc.balance;
4   unlock(acc.mutex);
5   if( bal >= amount ){
6     lock(acc.mutex);
7       acc.balcance = bal-amount;
8     unlock(acc.mutex);
9   }
10 }
```

Listing 2.2: Withdraw without data races.

Eliminating race conditions necessitates to execute a set of instructions in mutual exclusion with all conflicting instructions of other threads. These portions of code are called *critical* or *atomic* sections. In our example we want to ensure the balance is updated correctly. Thus our critical section should ensure the balance has not been modified between the check and its update. The implementation of Listing 2.3 is both free of race conditions and data races, at least as long as it concerns concurrent withdraws.

```
1 withdraw(Account acc,int amount){
2   lock(acc.mutex);
3     bal = acc.balance;
4     if( bal >= amount ){
5       acc.balcance = bal-amount;
6     }
7   unlock(acc.mutex);
8 }
```

Listing 2.3: Withdraw without data races and race conditions.

We mentioned earlier that data races can be benign. We extend the `withdraw` example such that it contains a data race which is benign. For that we assume the accounts have a flag which denotes the account has been accessed. This information can be used for random checking of accounts for instance. Thus for performance it may be profitable not to protect accesses to it. The implementation of Listing 2.4 has no race conditions although it contains a data race.

```
1 withdraw(Account acc,int amount){
2   lock(acc.mutex);
3     bal = acc.balance;
4     if( bal >= amount ){
5       acc.balcance = bal-amount;
6     }
7   unlock(acc.mutex);
8   acc.flag = true;
9 }
```

Listing 2.4: Withdraw with data race but without race conditions.

2.4 Data race detection

Data races exist in almost any multi-threaded application for a multitude of reasons: (i) programmers were unable to anticipate interleavings or (ii) left them voluntarily for performance reasons (iii) miss-use of synchronization mechanisms (iv) priority inversion or (v) erroneous assumptions on atomicity of instructions. Because data races can be the source of various bugs and race conditions, several works have focused on their detection. Static and dynamic techniques have been extensively used to that end. Even some special hardware has been designed [MSQT09, ZTZ07, MC91].

Several static analyses and type systems have been defined for race detection, targeting most commonly *C* and *Java* languages. Pratikakis et al. [PFH11] propose *LOCKSMITH* a data flow analysis tool for detecting data races in *C* programs using *POSIX* threads and *mutexes*. The principle of their tool is to deduce a correlation between locks and the memory locations they protect. A program is race-free if all accesses to a location are consistently protected by the same lock. RacerX [EA03] is another race detection tool which also reasons about deadlock freedom. Moreover, it provides a ranking of identified races according to (i) the likelihood of being a false positive and (ii) the difficulty of inspection.

Flanagan et al. [FF00, FLL+02, FFLQ08] have conducted a great amount of work on static race detection of *Java* programs. In [FF00] they propose a type system capable of capturing common synchronization patterns such as classes with internal synchronization and thread-local classes. The type system requires programmers to annotate fields with a locking expression that protects them. In [FLL+02] they present *Extended Static Checker* for Java (ESC/Java). Apart from detecting races it also finds common programming errors such as null dereferences. Most importantly though it uses an automatic theorem-prover to reason about the semantics of the program and which gives it the capability of detecting errors observable at runtime only. Finally, in [FFLQ08] they propose a type system for inferring atomicity of *Java* methods.

A drawback of static techniques is the excessive number of false positives they produce. The reason is most of them make pessimistic assumptions on feasible interleavings. Dynamic analyses on the other hand are more precise because they reason on feasible executions and all aliasing issues are eliminated. A plethora of such tools exist. Some of these tools such as [PK96, YRC05, JBPT09] are based on the *happens before* relation defined by Lamport in [Lam78]. In this approach accesses to a shared location by different threads should be ordered based on a synchronization. Another technique is the so called lockset analysis employed by Eraser [SBN+97]. It consists in monitoring every shared memory access and verify consistent locking behavior is observed. That is, all accesses to a memory location are protected by the same lock. Hybrid tools combining happens before and lockset technique have also been implemented [OC03, SI09]. Because dynamic methods incur great overheads at execution time [MMN09] proposes a sampling method for monitoring running applications. Data race detection analysis is only performed during selected portions of a programs execution.

Finally, although dynamic data race detection tools are precise a great number of races

detected are benign. Narayanasamy et al. [NWT+07] split their data races into potentially *benign* and *harmful*. To be able to triage they record the execution into logs and then perform all analyses offline during replay. To classify a data race two different schedules are replayed, if the outcome is the same for both schedules then it is considered as benign. Of course there is no guarantee that all schedules are also benign, but at least the tool cannot demonstrate a harmful execution. A more recent race detection classification tool is Portend [KZC12]. It is also based on replaying a concrete execution but does a finer classification into four categories and has improved accuracy.

2.5 Synchronization mechanisms

Synchronization mechanisms allow to impose an order on the execution of threads and thus control access to shared resources. There exist several mechanisms to synchronize threads. We detail hereafter the most commonly used ones. The implementation and behavior of these mechanisms varies but always relies on special hardware instructions which guarantee atomic update of the information related to the synchronization mechanism.

lock: is a binary variable with two states *locked* and *unlocked*. Only one thread can obtain the lock at a time, all other threads are prevented from obtaining it until it is unlocked. Locks provide *mutual exclusion* and are thus used for protecting critical sections (sequences of instructions susceptible to create race conditions). There are typically two variations of locks:

- **spin lock** a thread that was not able to obtain the lock will repeatedly try to obtain it, resulting into consuming CPU cycles without making any progress. This tactic is advantageous when the time spinning to acquire the lock is smaller than a context switch to another thread.
- **mutex** threads that are not able to obtain the lock get blocked allowing other threads to execute. When the lock is released they are signaled and can try again to obtain the lock.

semaphores: are mostly used to control access to countable shared resources. A semaphore can be conceived as a counter with atomic increment (V) and decrement (P) interface. To obtain a resource a process has to decrement the semaphore. If the semaphores value becomes negative the thread is blocked and put in a waiting list. To release a resource a thread increments the semaphore. If the previous value was negative it unblocks the first thread in the waiting queue.

condition variable: allows to block a thread until a condition is true. A condition variable is always used in conjunction with a lock. The lock is needed for atomically checking the condition. If the condition is false then the lock is released (so other threads can update the condition) and the thread sleeps in a queue related to the condition variable. When another thread updates data of the condition it should signal a single or all threads waiting on the condition so they can test it again.

barriers: are used to synchronize a set of threads. All threads must reach to the state designated by the barrier prior to continuing. For example in a parallel sorting of a table all threads must finish sorting their sub-table prior to merging the sorted parts.

2.5.1 Synchronization issues

Synchronization mechanisms affect the execution of a process by blocking threads when necessary. This brings to surface some well known problems that of *deadlock*, *livelock* or *starvation* and *priority inversion*.

Deadlock

A deadlock (or deadly embrace) occurs when two threads mutually wait on a resource held by the other thread to be released. For a deadlock to occur several conditions known as Coffman conditions [CES71] must hold.

i) *mutual exclusion:* a resource is either held by a thread else it is available.

ii) *hold and wait:* threads can hold some resources while waiting for others

iii) *no preemption:* resources obtained by a thread cannot be forcibly released. The thread must release them voluntarily.

iv) *circular wait:* there exists a circular chain between two or more threads each waiting on a resource obtained by the next in the chain. Figure 2.3 illustrates such a case, where circles represent the resources (locks for instance) and squares the threads. Edges exiting threads represent resources obtained while incoming edges represent the requested resources.

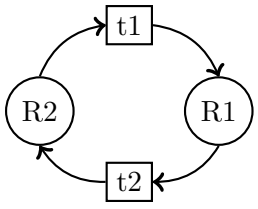


Figure 2.3: Deadlock.

Deadlock handling

If deadlocks are ignored, then most probably at some point the entire process will get blocked. To avoid this undesired event several solutions can be used. They are categorized into: (i) detection and recovery (ii) avoidance (iii) prevention.

detection and recovery: In this approach action is taken after a deadlock is detected. To detect deadlocks a graph of resources must be kept up to date. Then, an algorithm for detecting cycles can be used to detect deadlocks. Recovery occurs by forcibly removing a resource from a thread, that is braking Coffman condition iii.

avoidance: Dynamically decides whether allowing a thread to obtain a resource can lead to a deadlock. For this approach to work knowledge of all resources a thread will need is required. The *bankers algorithm* proposed by Dijkstra solves this problem.

prevention: To prevent deadlocks it suffices to eliminate one of the Coffman conditions.

i mutual exclusion can be eliminated by using *lock free* algorithms

ii to lift the hold and wait condition, all threads must obtain the resources they require prior to starting their execution. Because resources are usually obtained one by one backing of is necessary i.e., if a resource is not immediately available all resources previously acquired must be released and try again.

iii removing the no preemption condition correctly necessitates a roll-back of all actions executed by a thread before releasing the resource. This can be too expensive to implement or even impossible (in the case of I/O for instance).

iv to eliminate circular waits Dijkstra originally proposed defining a partial order over the resources and then enforce they are acquired respecting that ordering.

Livelock or starvation

This problem occurs when a thread which is not blocked does not manage to progress. This can happen due to unfair scheduling for instance. Another case is when higher priority threads monopolize a resource thus excluding lower priority threads from accessing it.

Priority inversion

Priority inversion occurs when a low priority thread surpasses a higher priority thread. Here is a typical case of priority inversion as presented in [MSD10]. To demonstrate it three threads are required: *t1* with low priority, *t2* with medium and *t3* with high priority. Initially *t2,t3* are blocked and thus *t1* manages to acquire a resource to be shared with *t3*. While *t1* has not yet released the resource *t2* preempts it because it has a higher priority. Eventually *t2* will run to completion surpassing *t3* which will resume only after *t2* finishes and *t1* releases the shared resource. To avoid such circumstances priority inheritance is often used.

2.6 Executing threads in parallel

Threads are executed *concurrently*, which incurs their execution can be arbitrarily *interleaved* or occur in *parallel* (i.e., at the same time). Figure 2.4 illustrates the scheduling of a multi-threaded application consisting of two threads on a mono-processor and on a

multi-processor system. When executing on the mono-processor, threads are interleaved according to the schedule and memory accesses are serialized respecting program order of executed threads. When executing on a multi-processor (or multicore) system real parallelism can be obtained, and memory accesses may not appear in the same order to all threads. In the example of Figure 2.4 we assume all variables are initialized to zero. For the mono-processor execution the values of `z` and `w` will always be `1` and `8` respectively, which is the expected result. For the parallel execution though the values corresponding to `z` and `w` can be `1` and `0` respectively, which is a rather unexpected result since the assignment to `y` precedes that to `x` and thus should have taken effect too.

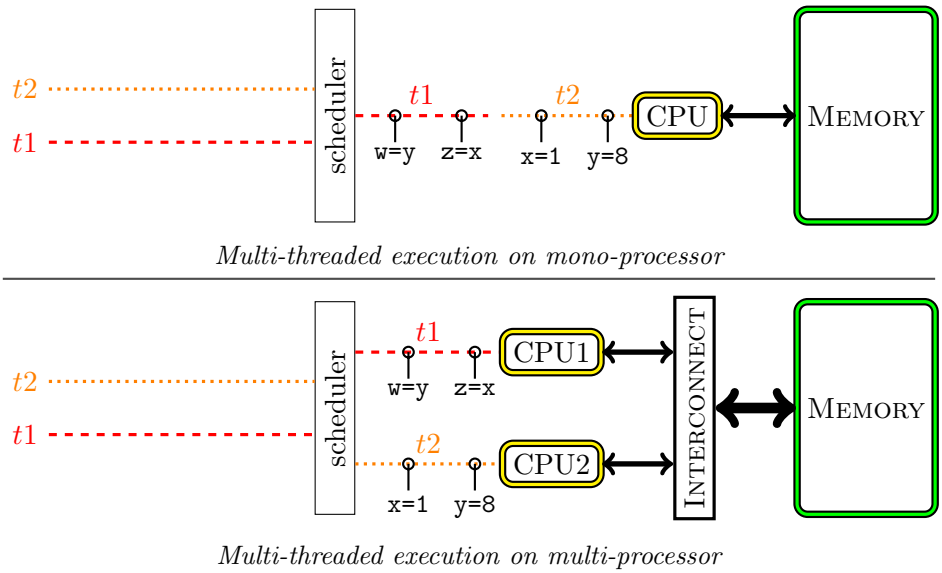


Figure 2.4: Execution of multithreaded application.

The memory inconsistencies that can be observed by a multi-processor system are specified in its *weak/consistency memory model*. The memory model specifies how the memory system behaves. That is, it correlates the values read by load operations with the value written by store operations to the same memory location in a parallel execution of a program. The inconsistencies are caused by several optimizations introduced by compilers and the executing platform. Compiler optimizations are correct for single-threaded applications while hardware optimizations for mono-processor systems. Adve et al [AG96] provide more details on shared memory consistency models and the effect of various optimizations.

2.6.1 Sequential consistency

Prior to introducing relaxations of weak memory models, we present *sequential consistency* (SC) which is the memory model programmers are used to reason about. *Sequential consistency* is defined by Lamport in [Lam79] and focuses on:

program order: a processor must issue memory operations in the same order as they appear in the program, and prior to issuing a memory operation it must ensure that

its previous memory operation is complete.

write atomicity: writes to the same memory location are made visible in the same order to all processors.

2.6.2 Relaxing sequential consistency

For performance reasons most architectures break sequential consistency by using out-of-order execution, non strict cache coherency protocols, and complex memory interconnections (e.g., switch-based). Various memory models can be defined by relaxing the *program order* and *write atomicity* which are necessary for SC. The relaxations are bounded within a time interval δ. We enumerate hereafter some relaxations and present in Table 2.1 how they are incorporated in a number of relaxed memory models.

program order: in this case accesses are made to different locations

- write to read (W → R): a read operation completes before a preceding write
- write to write (W → W): the order of writes is inversed
- read to read or write (R → RW): a read or write operation completes before a preceding read

write atomicity: in this case accesses are to the same location

- read own write early: a processor reads the new values it wrote prior they are made visible to other processors.
- read others write early: a processor can read new values prior they are made visible to all processors.

Relaxation	W → R	W → W	R → RW	Read Own Write early	Read Others Write early
SC				✓	
IMB370	✓				
PC	✓			✓	✓
TSO	✓			✓	
PSO	✓	✓		✓	
RMO	✓	✓	✓	✓	
WO	✓	✓	✓	✓	
PowerPC	✓	✓	✓	✓	✓

Table 2.1: Relaxations accepted by most common memory models

As we can note in Table 2.1 one of the most common relaxations is the *write to read* (W→R). Because writes are more costly operations they are often placed in a write buffer while waiting their completion. Subsequent reads can be served as long as there is no pending write to the same address in the buffer. We use the example presented in [AG96] which demonstrates how this relaxation breaks *Dekkers* algorithm for mutual exclusion.

Figure 2.5 illustrates the pseudo-code to be executed by each thread, and a possible execution on an architecture which relaxes W→R. Under *sequential consistency* Dekkers algorithm guarantees mutual exclusion between two threads. Each thread notifies (by updating its flag) the other it attempts to access the critical section and then checks (reads the other flag) if the competing thread has accessed its critical section first. In the illustrated execution, the circled numbers define the order in which the accesses are served by the memory. As we can note, both threads update their flag, which goes in the respective write buffer and subsequently test the competing threads flag which in both cases returns 0. Thus, both threads assume they can proceed with the execution of their critical section which breaks mutual exclusion.

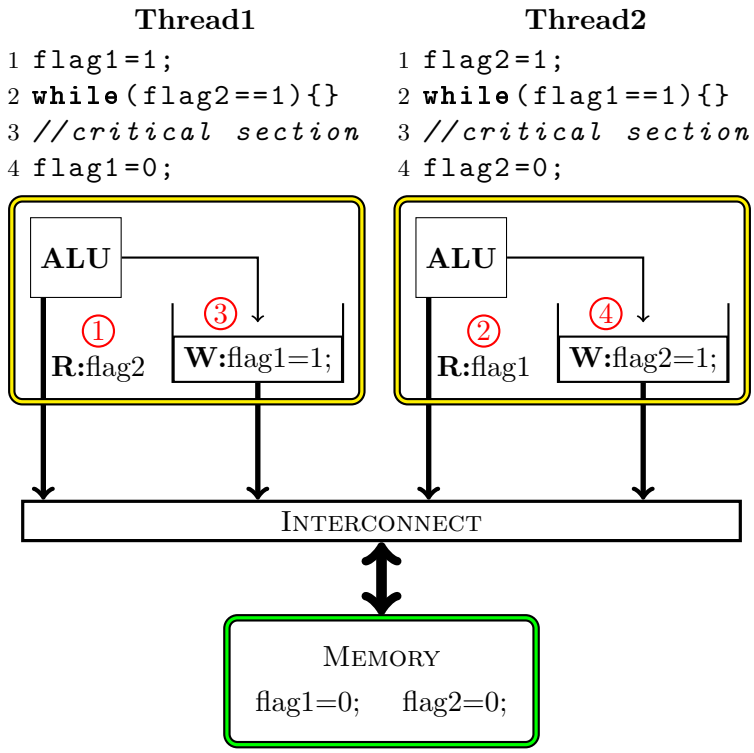


Figure 2.5: *Dekkers* algorithm broke under Write → Read relaxation

Programmers developing high performance libraries (e.g., concurrent data structures) often reside to such algorithms for mutual exclusion and synchronization. To enforce an ordering of memory accesses low level primitives called *memory barriers* or *fences* are used to prohibit relaxations and hence serialize memory accesses.

2.7 Formalization of a multithreaded program execution

As described previously, a multithreaded program consists of a *main thread* with a life span same as that of the hosting process and a set of threads dynamically growing and shrinking as threads are created and terminated at execution time. Each thread defines a *sequence of events* which are scheduled and subsequently executed by the underlaying platform.

The sequence of events produced by a thread is usually described in a high level imperative programming language such as *C/C++* or *Java*. At this level, events are statements of the language such as assignments, conditionals, function calls etc. Programmers are often mistaken and assume high level statements to be atomic. Prior to executing an event described in a high level language, it is faithfully translated into assembly instructions/events for the targeted execution platform. At the assembly level the events corresponding to a thread are much finner. They specify load, store accesses to thread-private or shared memory locations and control the execution flow. Some shared memory locations may be associated to synchronization mechanisms (e.g., mutexes). Choosing the events to consider depends on the abstraction level needed in order to capture significant information about the threads execution. Usually the set of events consists of synchronizations and data accesses. We provide hereafter an example of some types of events:

- `read(x)` read the memory location `x`
- `write(x)` write the memory location `x`
- `lock(m)` acquire the mutex synchronization mechanism `m`
- `unlock(m)` release the mutex synchronization mechanism `m`
- `spawn(`t`,`t'`)` thread t spawns thread t'
- `thread_start(`t`)` thread t is initialized, it denotes the start of production of events
- `join(`t`,`t'`)` thread t waits for thread t' to end
- `thread_end(`t`)` thread t ends its execution, it denotes the end of production of events

A thread T^a (where a is the threads identifier) defines a *totally ordered* sequence of events e_i^a where i is a unique identifier of the event (capturing the order relation) and a is the thread that produced it. The sequence of events is delimited by an initial event e_1^a `:= thread_start(`a`)` and a final event e_n^a `:= thread_end(`a`)`. To specify the ordering of events, we introduce the reflexive operator $\blacktriangleleft_a$ which denotes that an event e precedes e' ($e \blacktriangleleft_a e'$), where e, e' belong to the same thread T^a. We formalize a thread as a tuple consisting of a set of events, and the thread specific precedence operator:

Definition 2.7.1 (Thread)

$$T^a = (\{e_i^a\}, \blacktriangleleft_a)$$

A multithreaded program $\mathfrak{P}$ statically or dynamically creates a number of threads $\mathcal{T}$, each uniquely identifiable. Thus, we formalize a multithreaded program as the union of all these threads:

Definition 2.7.2 (Multithreaded program)

$$\mathfrak{P} = \bigcup_{a \in \mathcal{T}} T^a \text{ , where } \mathcal{T} \text{ is the set of all threads executed in } \mathfrak{P}$$

A multithreaded program can be scheduled in many different ways. A schedule Σ is a total function mapping events to discrete timestamps in $\mathbb{N}$ for which the usual $<$ and $\leq$ order relations can be established. The timestamps assigned to events belonging to a same thread T^a are always distinct and respect the ordering $\blacktriangleleft_a$ associated to it.

Definition 2.7.3 (Schedule of multithreaded program $\mathfrak{P}$)

$$\Sigma(e) : e \in \mathfrak{P} \rightarrow t \in \mathbb{N} \textit{ such that } \forall e_k \blacktriangleleft e_m \Rightarrow \Sigma(e_k) < \Sigma(e_m) \textit{ where } \blacktriangleleft = \bigcup_{a \in \mathcal{T}} \blacktriangleleft_a$$

The scheduling of a multithreaded program can be of two types, *sequential* or *parallel* which we detail hereafter. The execution of a schedule Σ is a serialization $\sigma^o_{\Sigma,\pi}$ of all events as observed by an observer o. The order of events one can observe, depends on the execution platform $\pi = (\delta, \mu)$ where μ is the platforms memory model and δ is the max time span during which the memory model can affect ordering of events. We presented earlier in section 2.6.2 some relaxed memory models.

2.7.1 Sequential schedule and serialization

In the sequential case, the scheduler maps a distinct timestamp to each event $e \in \mathfrak{P}$, thus a total ordering of events can be inferred. A sequential schedule Σ_s respects the following properties:

Definition 2.7.4 (Sequential schedule Σ_s of $\mathfrak{P}$)

$$\begin{array}{ll} i) & e_k \neq e_m \Rightarrow \Sigma_s(e_k) \neq \Sigma_s(e_m) \\ ii) & \forall e_k \blacktriangleleft e_m \Rightarrow \Sigma_s(e_k) < \Sigma_s(e_m) \end{array}$$

The serializations $(\sigma^o_{s,\pi})$ that can be observed, by any observer o, for the execution of a sequential schedule Σ_s are all equivalent and in accordance with the sequence of events defined by the schedule Σ_s. A sequential schedule can be conceived as executing the multithreaded program on a mono-processor. The threads are interleaved, that is a context switch occurs between two events e^a_k, e^b_{k+1} which guarantees e^b_{k+1} will definitively observe e^a_k as a preceding event. Concerning how a thread observes itself we remind that the effect of weak memory models are transparent to the thread itself when executed in isolation with other threads as is the case here.

2.7.2 Parallel schedule and serialization

In the parallel case, the scheduler maps events to partially ordered timestamps while respecting the intra-thread precedence relation of each thread. That is, events executed by different threads might be assigned the same timestamp. The definition of the mapping function for a parallel scheduler $\Sigma_{\parallel}$ is the same as that of multithreaded program (Definition 2.7.3). We just recall the property to be respected:

Definition 2.7.5 (Parallel schedule $\Sigma_{\parallel}$ of $\mathfrak{P}$)

$$i) \quad \forall e_k \blacktriangleleft e_m \Rightarrow \Sigma_{\parallel}(e_k) < \Sigma_{\parallel}(e_m)$$

Contrarily to sequential schedules, different serializations ($\sigma_{\parallel,\pi}$) can be observed for the execution of a parallel schedule $\Sigma_{\parallel}$. There are two sources for observing different serializations from a parallel schedule. First, events executed in parallel (i.e., with the same timestamp) can be serialized in any order. Second, effects of weak memory model (μ) allows threads to observe serializations under which events produced by other threads appear in an order other than that specified by their precedence relation i.e., we observe intra-thread interleaving of events. As presented in section 2.6.2 there are several weak memory models each allowing different relaxations. We introduce the predicate $\mathcal{M}(\pi, e_k, e_m)$ which for a given platform $\pi = (\delta, \mu)$ asserts whether events e_k and e_m can be swapped with respect to the memory model μ and time interval δ in which it applies. We formalize the serialization observable by a thread T^a for a parallel schedule $\Sigma_{\parallel}$ executed on platform π as follows:

Definition 2.7.6 (Serialization obtained by the execution of a $\Sigma_{\parallel}$)

$$\sigma^a_{\parallel,\pi} = \{ (e_1, \ldots, e_n) \mid \forall e^a_k \blacktriangleleft e^a_m \Rightarrow k < m \bigvee |\Sigma_{\parallel}(e_k) - \Sigma_{\parallel}(e_m)| < \delta \wedge \forall j \textit{ such that } k \leq j \leq m \,.\, \mathcal{M}(\pi, e_k, e_m) \}$$

Figure 2.6 allows us to summarize the definitions of different scheduling types for a multithreaded program $\mathfrak{P}$. At the top of the figure we provide an instance of $\mathfrak{P}$ consisting of three threads. For each thread we can see a snippet of its sequence of events, and their effect (reads and writes). We note that at this level there is no notion of time.

A sequential schedule Σ_s of $\mathfrak{P}$ is then presented which adds unique timestamps to each event in $\mathfrak{P}$. We also provide the serialization $\sigma^o_{s,\pi}$ obtained, which gives exactly the same sequence of events as the schedule Σ_s.

Just below we juxtapose a parallel schedule $\Sigma_{\parallel}$ of $\mathfrak{P}$. Compared to Σ_s above only the timestamping of events has changed. We aligned events that were executed in parallel i.e., those with the same timestamp, one above the other. As detailed earlier, a parallel schedule can produce several serializations, and threads executed in parallel within a time interval δ may each observe a different serialization of events in δ.

We illustrate an execution of $\Sigma_{\parallel}$ for a weak memory model $\mathcal{M}$ which allows the W $\rightarrow$ W relaxation. We focus on the serialization of events as observed by threads during the designated time interval δ. We assume that for the events preceding the interval δ the following serialization has been established: e^c_6, e^b_3, e^a_3.

To decipher the arrows connecting events and how they affect the serialization observed we need to take into account what each event represents. This information is in the initial description of $\mathfrak{P}$. The arrows connect read events to the corresponding write events. That is the arrow connecting e^a_4 to e^b_5 implies that e^a_4 reads the value written by e^b_5.

A serialization of events in δ can re-order them in any way as long as the memory model $\mathcal{M}$ is respected. The memory model we assume allows the W $\rightarrow$ W relaxation, thus T^a may

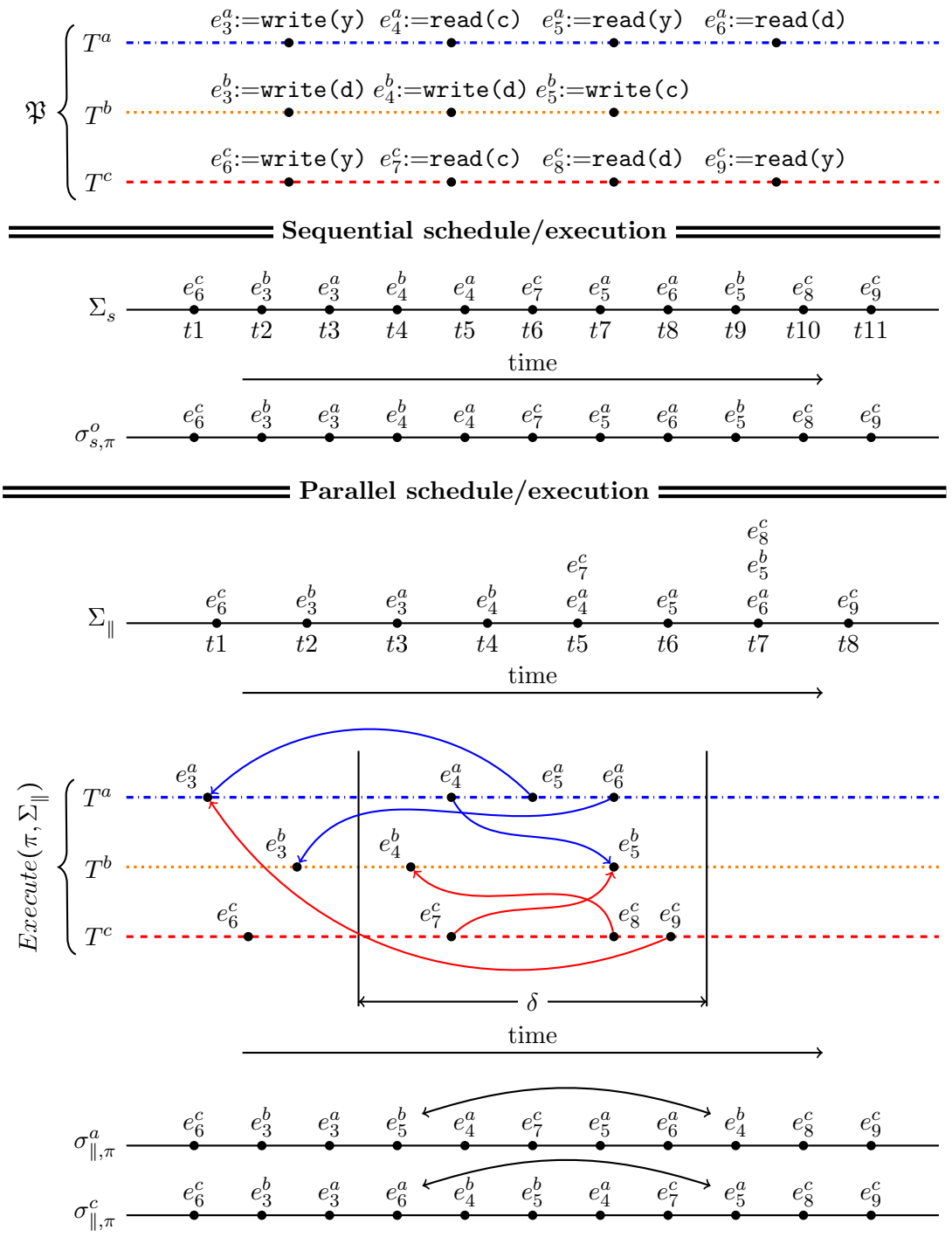


Figure 2.6: Obtaining serializations for a multithreaded program $\mathfrak{P}$

observe the write to variable `c` prior to that of `d` executed by events e_5^b and e_4^b respectively. Thus a possible serialization of events is that corresponding to $\sigma^a_{\delta,\mathcal{M}}$, which respects the sequence of events observed both by T^a and T^c. An other possible serialization could be $\sigma^c_{\delta,\mathcal{M}}$ which again respects the observations of T^a and T^c.

A last point to note is the consensus on the ordering of events prior to δ. Although two writes to variable `y` occurred possibly in a same δ' both T^a and T^c consent that the write of e_3^a persisted.

2.7.3 Constraining interleavings of a multithreaded execution

For both sequential and parallel schedules of multithreaded programs we allowed the unconstrained interleaving of events produced by threads. This is not strict enough since some events imply *happens-before* and *mutual exclusion* relations which should be respected by all valid schedules. Such events can be for instance thread creation and lock acquisition.

The thread creations introduce a transitive happens before relation between the thread that spawns and the newly created thread. Figure 2.7 illustrates a multithreaded program $\mathfrak{P}$ consisting of three threads where T^a spawns T^b and subsequently T^b spawns T^c. The happens before relation established (thick dashed arrows) during a thread creation specifies that all events preceding the spawn event (e^b_m) have occurred strictly prior to the events succeeding the matching thread start event (e^c_1). In the example of Figure 2.7 the following happens before relations are established due to thread creations: $e^a_m \xrightarrow{hb} e^b_1$ and $e^b_m \xrightarrow{hb} e^c_1$ (where $e \xrightarrow{hb} e'$ denotes event e must precede e'). Because events of a thread are totally ordered, by transitivity we can also infer that $e^a_m \xrightarrow{hb} e^c_1$. Finally, we illustrate with crossed out edges infeasible interleavings due to precedence. Similarly to thread creation, precedence can be established between events preceding a threads end and events succeeding a matching join event.

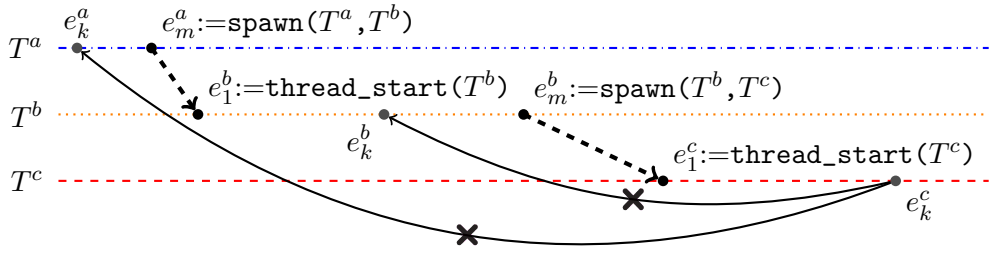


Figure 2.7: Transitive happens before on thread create

As mentioned in section 2.5 locks are used for *mutual exclusion*. That is, to guarantee sequences of events produced by different threads cannot interleave. Such sequences of events are surrounded by two special events *lock* and *unlock* which designate a critical section. We note a critical section as a tuple (e^a_l,e^a_u) where e^a_l is the event associated to the lock operation and e^a_u is the matching unlock. The lock and unlock operations are applied to a shared synchronization variable. The mutual exclusion relation between two critical sections is denoted as $(e^a_l, e^a_u) \overset{\mathtt{m}}{\leftrightarrow} (e^b_l, e^b_u)$ where $\mathtt{m}$ is the synchronization variable on which the lock and unlock operations apply.

The semantics of locks ensure that at any time only one thread executes its critical section. That is, even if two threads simultaneously demand to lock a synchronization variable only one will succeed.

We introduce the happens-before and mutual exclusion restrictions to the scheduling of multithreaded programs as follows.

Definition 2.7.7 (Schedule of multithreaded programs respecting constraints)

$$\Sigma(e) : e \in \mathfrak{P} \to t \in \mathbb{N} \textit{ such that } \quad \begin{array}{l} \forall e_k \blacktriangleleft e_m \Rightarrow \Sigma(e_k) < \Sigma(e_m) \textit{ where } \blacktriangleleft = \bigcup_{a \in \mathcal{T}} \blacktriangleleft_a \ \bigwedge \\ e_k^a \xrightarrow{hb} e_k^b \Rightarrow \Sigma(e_k^a) < \Sigma(e_k^b) \ \bigwedge \\ (e_k^a, e_m^a) \overset{\mathtt{m}}{\leftrightarrow} (e_k^b, e_m^b) \Rightarrow \Sigma(e_m^a) < \Sigma(e_{k+1}^b) \vee \\ \qquad\qquad\qquad\qquad \Sigma(e_m^b) < \Sigma(e_{k+1}^a) \end{array}$$

2.8 Summary

The thread programming model is well suited for exploiting the new massively parallel architectures. It can be used as a structuring mechanism of applications but also to exploit parallelism. Threads are cheaper to create and destroy than processes and their cooperation is easier and faster since it occurs through the shared memory.

The major challenge in thread programming is non-deterministic concurrent accesses to shared memory. Synchronization mechanisms resolve non-determinism but are cumbersome to use and even neglected for performance reason. Finally, the type of execution, serial vs parallel, has a great impact on the correctness of a multithreaded program.

Chapter 3

Optimizing critical sections

In this chapter we approach the problem of maintaining efficiently a coherent shared state in multi threaded applications. As detailed in section 2.3 on page 14 *critical* or *atomic* sections are blocks of code that appear to be executed atomically (free of race conditions) and thus capable of preserving the shared state coherent. Critical sections are most commonly implemented manually by surrounding the instructions of the critical section with the appropriate synchronizations (e.g., Listing 2.3 on page 16). Implementing critical sections manually can be error-prone, hence their automatic implementation has captured the interest of several researchers. The problem of implementing critical sections is orthogonal to that of detecting races.

Ultimately, with automatic implementation of *critical sections* programmers should not be concerned about critical sections; compilers should be capable of (i) identifying harmful data races and race conditions in a program and (ii) take necessary and optimal measures for eliminating them. In a first step towards completely automatic solutions programmers are asked to identify critical sections using primitives provided by the language (e.g., `synchronized` in *Java* or `#pragma omp critical` in *OpenMP*). These critical sections are then automatically protected without further implying the programmer. Listing 3.1 and 3.2 present examples of manual and automatic critical section respectively, using the classic example of transferring an `amount` of money between a source (`accFrom`) and destination (`accTo`) bank accounts.

Hereafter we take a closer look to the two implementations of `transfer`. In Listing 3.1 the programmer manually defines the synchronization for the critical section. In this case we assume `ACCOUNTS` is a mutex lock protecting all accounts. Although the choice of such a coarse grain lock simplifies the reasoning (only need to protect accesses to accounts using `ACCOUNTS`) it drastically reduces parallelism by serializing all accesses to accounts. Moreover, the programmer can easily make errors e.g. forgetting to release lock on `ACCOUNTS` at a return point as in line 6 (we intentionally commented it).

In Listing 3.2 on the other hand, the primitive `critical` is used to designate the critical section which eliminates potential synchronization errors. The programmer though has no control over the implementation of the critical section and hence cannot optimize it if needed. Most implementations of primitives like `critical` provide the guarantee that

```
1 bool transfer(amount,
2                accFrom,accTo){
3
4   lock(ACCOUNTS)
5   if( accFrom.balance < amount ){
6     // unlock(ACCOUNTS)
7     return false;
8   }
9   accTo.balance   += amount;
10  accFrom.balance -= amount;
11  unlock(ACCOUNTS);
12  return true
13 }
```

Listing 3.1: Manual critical section

```
1 bool transfer(amount,
2                accFrom,accTo){
3
4   critical{
5     if( accFrom.balance < amount ){
6       return false;
7     }
8     accTo.balance   += amount;
9     accFrom.balance -= amount;
10    return true
11  }
12 }
```

Listing 3.2: Automatic critical section

the execution of the critical section will appear atomic to all other critical sections. This guarantee is called *weak atomicity* and is what we assume for critical sections in the remaining of the document. Another type of guarantee also exists called *strong atomicity* which guarantees that the critical section will appear atomic to any statement in the program. The ATOMOS [CMC+06] programming language guarantees *strong atomicty* for its critical sections.

In the remaining of the chapter we provide some information on the implementation of *critical sections* respecting weak atomicity. We present the two main approaches, one called *optimistic* and the other *pessimistic*. Further, we give an insight on related work and position our work prior to presenting it in details. Experimentations conducted on the optimization of *critical sections* are provided in chapter 5.

3.1 Relaxing atomicity of critical sections

Critical or *atomic* sections are supposed to execute atomically. Atomic execution incurs that the effect of an instruction or set of instructions appears to the rest of the system instantaneously. This definition of atomicity corresponds to the *strong atomicity* we mentioned earlier. Most processors have in their instruction sets a number of atomic instructions such as *atomic increment* for instance which in one step reads and increments the value of a variable. To execute a large number of instructions atomically is hard and penalizes performance by exclusively using the processor.

As stated earlier the most commonly accepted semantic of critical sections is to guarantee *weak atomicity* which relaxes the notion of *atomicity* by limiting it to other critical sections and not any instruction in the program. To achieve this type of atomicity it suffices to execute all *critical sections* in mutual exclusion. Achieving it is straight forward, we just have to synchronize the execution of all critical sections using a single *lock*. This level of atomicity is still too conservative and drastically reduces performance since only one *critical section* can execute at a time.

The semantic of *critical sections* respecting *weak atomicity* is equivalent to *serializable*

transactions in a database system. This allows to relax the execution of critical sections i.e., interleave their instructions as long as their serializability is guaranteed. Two transactions or *critical sections* are serializable when the outcome is equivalent to that of a serial execution. Serial means without overlap, i.e., the transactions/critical sections were executed serially the one after the other in any order.

Figure 3.1 illustrates the impact of relaxing atomicity of critical sections on parallelism. The example consists of three threads ($t1$, $t2$, $t3$) where $t1$ and $t2$ must each execute a critical section (possibly accessing same shared variables) while $t3$ has no critical section to execute. Under *strong atomicity* while executing a critical section nothing else can execute, not even instructions of $t3$ because the critical section must also appear atomic to them. In *weak atomicity* the mutual exclusion of critical sections with non-critical sections is relaxed. Thus the concurrent execution of critical section in thread $t1$ and $t2$ with thread $t3$ is allowed as illustrated. Though, concurrent execution of critical sections is still not permissible. Finally, allowing *serializable weak atomic* executions of critical sections may also allow overlapping of critical sections provided the outcome is equivalent to their serial execution.

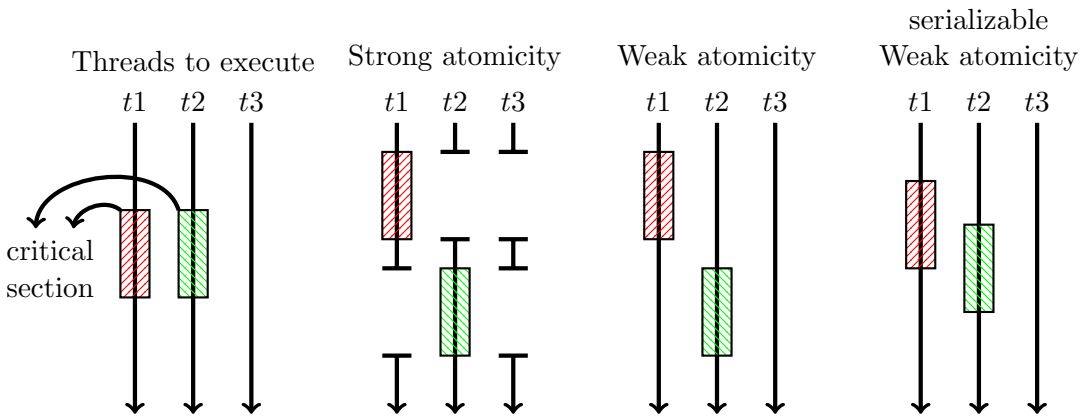


Figure 3.1: Relaxing atomicity of critical sections

3.2 Implementing critical sections

The automatic implementation of *critical sections* is divided into two approaches: *optimistic* and *pessimistic*. The optimistic approach reposes on the usage of *transactional memories* while the pessimistic uses *synchronization primitives* (such as locks) to enforce correct executions.

3.2.1 Optimistic implementation of critical sections

In this approach a *critical section* maps to a *transaction*. Transactions are executed under the supervision of transactional memories ([ST95, HLMS03, HMJH08]) which guarantee their serializability. Transactional memories assume no conflict will occur during the execution of a transaction. Thus, each thread executes its critical section while the

transactional system keeps a log of all shared memory accesses. If conflicting accesses occurred concurrently by different threads, then their modifications to the shared memory are undone (roll-back) and the transactions are re-executed from scratch. Transactional memories can be implemented with special hardware [Kni90, HM93, RG02] or through software [ST95, DSS06].

3.2.2 Pessimistic implementation of critical sections

The pessimistic approach makes no assumption on the occurrence of conflicts at execution time and thus uses synchronization mechanism (mostly locks) for enforcing atomicity of critical sections. The challenges to overcome are: (i) infer sufficient synchronizations such as atomicity of critical sections is not violated and (ii) avoid deadlocks which can be caused by the usage of synchronization mechanisms, while not reducing drastically the parallelism. We cite hereafter some distinctive works on the pessimistic implementation of critical sections.

McCloskey et al. [MZGB06] have developed the tool *Autolocker* which performs a *source-to-source* transformation of *C* code. Programmers must annotate the program with information relating locks to shared data. The type-system they define guarantees the correctness of the transformation. Experimental results they provide exhibit substantial gain over optimistic approaches. A major drawback of this work is it relies on user annotations.

In their works Emmi et al. [EFJM07] and Hicks et al. [HFP06] do not require any annotations. They first infer a set of locks necessary to guarantee absence of conflicts and then perform optimizations to reduce the number of locks. In [HFP06] for instance coalesced locks are compacted. Both works draw the conclusion that defining a minimal set of mutexes which guarantee absence of conflicts is NP-hard.

Cherem et al. [CCG08] propose a backward tracking of expressions within the scope of the critical section. This allows to identify more precisely shared data accessed and thus infer very fine grained locks. When fine locks cannot be inferred they repose on coarser locks. An interesting aspect of this approach is to consider a hierarchy of partially ordered locks. The multi-granularity locking scheme uses the deadlock avoidance protocol of Gray et al. [GLP75] and is implemented in a runtime library they provide. Finally, Upadhyaya et al. [UMP10] present a new strategy for inferring locks. They use data structure knowledge to make more precise alias analysis.

3.2.3 Optimistic versus pessimistic concurrency

Pessimistic solutions perform better when a high contention is expected. The reason is that each critical section is executed once, while in optimistic solutions a critical section might have to execute several times until it succeeds to complete. Some transactional memory implementations like TL2 [DSS06] may use the lock-based approach in case of multiple failures to complete a critical section. This technique improves their performance compared to other STM. Moreover, optimistic solutions incur a significant overhead due to log keeping and transaction committing.

Optimistic solutions can be profitable when contention is low and thus a small number or roll-backs is required. A main advantage of optimistic solutions is composability. Arbitrary critical sections can be composed, which is equivalent to executing them in the same transaction. This is infeasible with pessimistic implementations because a deadlock may occur.

3.3 Improving pessimistic implementations of critical sections

The works we presented earlier on pessimistic implementation of critical sections propose cunning analyses and algorithms to infer the finest locks necessary to protect *critical sections*. Inferring fine grained locks is not sufficient to obtain performance gains. Locks must also be used appropriately and their implementation must be optimized. We present hereafter some optimizations proposed in order to improve pessimistic implementation of critical sections.

Kagi et al. [KBG97] focus on providing more efficient mutual exclusion through better locks. They state that to maximize performance of fine-grained parallel applications, the delay associated to the transfer of exclusively accessed resources must be minimized. They define the notion of synchronization period which we reproduce in Figure 3.2. It illustrates the life cycle of a critical section protected by a single lock X and accessed by two processes P_A and P_B. The synchronization period consists of three phases: (i) *transfer*, the lock is transfered from P_A to P_B (ii) *load/compute*, P_B loads exclusive data and updates them and (iii) *release*, P_B releases the lock on X. The *transfer* and *release* phases are considered overhead related to the lock management which can be improved.

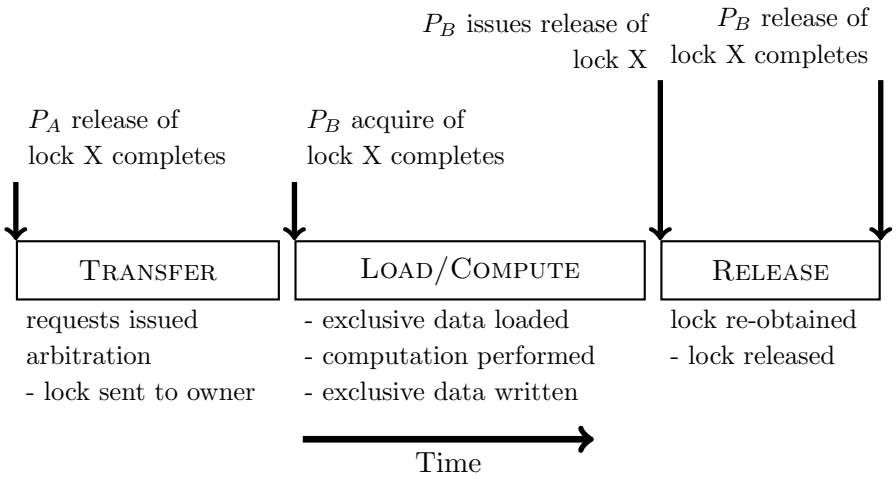


Figure 3.2: Synchronization period for high contention lock. (from [KBG97])

After defining the synchronization period they present a set of optimizations for synchronization mechanisms (local spinning, queue-based locking, collocation, synchronous prefetch) and how they are incorporated into six synchronization primitives (TS, TTS, MCS locks [MCS91], LH and M locks [MLH94], reactive synchronization [Lim95] and

QOLB [GVW89]). The six synchronization primitives are compared on well established benchmark suites such as SPLASH [SWG92]. Their main conclusion is that QOLB, which uses all four optimizations, provides consistent and large performance gains.

Suleman et al. [SMQP09] propose *accelerated critical sections* which leverages the high performance cores of *Asymmetric Chip Multiprocessors* (ACMP). Selected critical sections are executed on high performance cores and thus their execution latency is reduced. Their solution necessitates some modifications of the instruction set for relocating the execution of critical sections to the high performance core and notify regular cores of completion. Relocating all critical sections to the same core causes what they call *false serialization*. To reduce it they use simultaneous multi-threading (SMT) on the high performance core and *selective serialization* (SEL) which dynamically chooses where the critical section will be executed. SEL is susceptible of producing deadlocks when used with nested critical sections.

A most recent work by Lozi [LDT+12] proposes a new locking algorithm entirely implemented in software called *remote core locking* (RCL). In RCL lock acquisitions are replaced by remote procedure calls, executed on a dedicated server core. Executing all critical sections on a server core would introduce false serialization. Thus, the authors provide a profiler which guides the programmer towards which locks should be transformed into RCL locks. Moreover, there can be more than one servers which are executed on dedicated threads i.e., application threads are not penalized by being the server interchangeably as in [HIST10].

3.3.1 Positioning of our work

The aim of our work is to improve the performance of pessimistic implementations of critical sections, by minimizing the possession of locks. Contrarily to the works presented in section 3.2.2 which focus in identifying the finest possible locks necessary to protect a critical section, we are interested in the usage of the identified locks (i.e., how they are acquired and released) and their type (i.e., what kind of exclusion is provided). We exhibit that the most commonly used policy which consists in obtaining all protections prior to entering a critical section and releasing them once exited (we call it *global*) is not optimal. Hence, we propose a number of more flexible policies which: (i) respect the semantics of critical section, (ii) guarantee *deadlock freedom* and (iii) perform better than the *global policy*.

The problems of identifying synchronizations needed to protect a section and which policy to use are complementary. Thus, the finest synchronizations are identified, the greater will be the impact of the applied policy; allowing more concurrency.

3.4 Mutual exclusion mechanisms

We present hereafter three mutual exclusion mechanisms for controlling access to shared variables. We call these mechanisms *protections*, each allowing or restricting access to be exclusive or shared among threads. The granularity of a protection may vary. That is, it

can protect a single variable or a set of shared variables.

For the remaining of the chapter we adopt the following notations:

- $\mathcal{V}$ is a set of *shared variables* used in critical sections
- $CS = [i_1,\ldots,i_k,\ldots,i_n]$ is a *critical section* to be executed by a *thread*. A critical section is a *sequence* of instructions. Each instruction i_k can be represented by a sequence of *reading* (read(x)) and *writing* (write(x)) of shared variables. The sequence can be empty if shared variables are not used in the instruction. The example below illustrates on the left side an excerpt of a critical section and on the right the sequence of read/write statements we consider for each instruction. Variables `lb,lc` are neglected because they are thread local variables and hence they do not require any protection.

```
critical{
  lc = x + lb;   ⟶ i1 : [ read(x) ]
  x = y + z;     ⟶ i2 : [ read(y), read(z), write(x) ]
  lb = lc + 5;   ⟶ i3 : [ ∅ ]
}
```

- p_x is a *protection* either exclusive or not on a shared variable x ($x \in \mathcal{V}$)
- $P_k = \{p_x | x \in \mathcal{V}\}$ is the minimum set of protections needed for the safe access to shared variables used by instruction i_k. More precisely:
  - if i_k reads a shared variable x then at least a non-exclusive protection on x should be held;
  - if i_k writes a shared variable x then a non-exclusive protection must be held.
- $\mathcal{P} = \bigcup_{k=1}^{n} P_k$ is the set of all protections needed in the critical section.

We present hereafter three types of protections: **mutexes**, **read/write** and **read-/write intend**. The first two are classic in the literature while the third one is a variation of read/write we propose. For each protection we provide its intended use inside a critical section (i.e., how to compute P_k).

Mutex : this type of protection gives exclusive access (read/write) to the thread that obtained it. This protection should be held whenever the shared variable associated to it is accessed.

$$P_k = \{\, m_x \,|\, read(x) \in i_k \vee write(x) \in i_k \,\}$$

where m_x is a *mutex* protection related to variable x.

Read/Write : This type of protection distinguishes read from write access to a variable. Read access is non-exclusive and thus multiple threads can read the protected variable in parallel. Dually, write access is exclusive and only the thread holding the protection can write (or read) the variable.

The correct utilization of this protection is not as straightforward as for mutexes. As we will see later in this chapter protections may be obtained incrementally for a critical section. In this case when an instruction i_k reads a shared variable x which is later on written in the critical section by an instruction i_m, then it is necessary to anticipate the exclusive (write) protection even for reading x at instruction i_k.

$$\begin{aligned} P_k \quad = \quad & \{\, r_x \,|\, read(x) \in i_k \,\} \bigcup \\ & \{\, r_x, w_x \,|\, read(x) \in i_k \wedge write(x) \in i_m, m > k \,\} \bigcup \\ & \{\, r_x, w_x \,|\, write(x) \in i_k \,\}, \end{aligned}$$

where r_x is a read protection on variable x and w_x is a write protection on variable x.

Figure 3.3 bellow illustrates how a deadlock can occur when (i) protections necessary for a critical section are obtained incrementally and (ii) there is no anticipation on obtaining the write protection. Assuming critical sections CS_1 and CS_2 are executed concurrently by two threads. Then, both threads could obtain the read protection on x (r_x) which is not exclusive. Later on when each tries to obtain the write protection on x (w_x) gets blocked due to incompatibility with the r_x protection previously obtained by the competing thread.

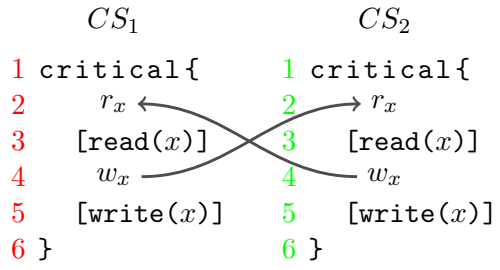


Figure 3.3: Deadlock by obtaining incrementally read/write protections.

Write Intend : This type of protection is a refinement to the Read/Write protection presented above. It increases parallelism by making the distinction between obtaining the write protection and using it (i.e., actually writing the protected variable). Concretely, instead of obtaining the write protection we just reserve it for later use. This is done by obtaining the *write intend* protection until the first write is encountered, prior to which we must obtain the write protection. The switch from write intend to write is guaranteed because write intend protection is exclusive to write and write intend protections but non-exclusive to reads.

$$\begin{aligned} P_k \quad = \quad & \{r_x \,|\, read(x) \in i_k\} \bigcup \\ & \{r_x, w'_x \,|\, read(x) \in i_k \wedge write(x) \in i_m \ , \ m > k\} \bigcup \\ & \{r_x, w'_x, w_x \,|\, write(x) \in i_k\} \end{aligned}$$

where r_x is a read protection on variable x, w'_x is a relaxed protection (write

intend) on the writing of variable x and w_x is the write protection on variable x.

This type of protection can improve cases such as that illustrated in Figure 3.4 where CS_3 only wants to access variable x for reading. If no write intend protection is used (as in CS_1) then CS_3 cannot be executed in parallel with a critical section that modifies x. When write intend protection is used (as in CS_2) then CS_3 can execute in parallel with the critical section intending to write x. A gain is obtained if the reading thread releases its read protection before the writing thread requires the switching from write intend to write protection. If not, then the writing thread will be blocked until reader finishes. Ideally in this example CS_3 will get executed in parallel with instructions 3, 4 of CS_2.

CS_1	CS_2	CS_3
1 `critical{`	1 `critical{`	1 `critical{`
2 r_x , w_x	2 r_x , w'_x	2 r_x
3 `[read(`x`)]`	3 `[read(`x`)];`	3 `[read(`x`)];`
4	4 w_x	4
5 `[write(`x`)]`	5 `[write(`x`)]`	5 `[read(`x`)]`
6 `}`	6 `}`	6 `}`

Figure 3.4: Benefit of write intend protection.

Table 3.1 below summarizes the incompatibilities between protection types. On the horizontal and vertical axes we have concurrent threads competing for the protections. The cross symbol (✖) is used to denote that threads cannot obtain simultaneously that type of protection.

Protection	Mutex Lock
Mutex Lock	✖

Mutex protection

Protection	Read	Write
Read		✖
Write	✖	✖

Read/Write protection

Protection	Read	WriteIntend	Write
Read			✖
WriteIntend		✖	✖
Write	✖	✖	✖

Write intend protection

Table 3.1: Incompatibilities for each protection type

As discussed earlier in section 2.5.1 the usage of synchronization mechanisms such as the protections we presented above are susceptible to producing deadlocks, if not used properly. A deadlock is produced when two threads are mutually waiting for a resource previously obtained by the concurring thread. A classic solution for avoiding deadlocks is to define a total order over the shared resources [Hav68, CES71] and then make sure all threads obtain protections respecting this ordering. Deadlocks are avoided because the first thread to obtain a common protection will be able to obtain all locks necessary to complete its critical section.

We introduce hereafter the transitive binary operator $<_{\mathcal{V}}$ which is used to define a total order on a set of shared variables $\mathcal{V}$. Assuming the following set of shared variables $\mathcal{V} = \{x, z, y\}$ and the order relation $x <_{\mathcal{V}} y <_{\mathcal{V}} z$. If in a critical section $CS_1 = [i_1, \ldots, i_k, \ldots, i_m, \ldots, i_n]$ instruction i_m accesses variable x and i_k accesses y then; despite the order in which they appear in the critical section, protections on variable x must be obtained prior to those on y.

Figure 3.5 illustrates an example of how ordering is applied on protections. Assuming the following set of shared variables $\mathcal{V} = \{x, q, z, y\}$ to which we assign accordingly the protections (p_x,p_q,p_z, p_y). We define the following ordering $q <_{\mathcal{V}} x <_{\mathcal{V}} y <_{\mathcal{V}} z$. For the critical section in Figure 3.5 a protection on variable y is needed at instruction 3 while protections on x and z are required at instruction 6. According to the ordering defined for these variables, protections on x should always be obtained prior to those on y and z. Thus, protection p_x is moved to instruction 2 prior p_y.

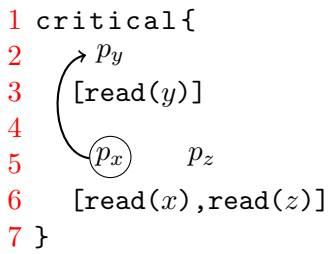


Figure 3.5: Acquiring protections respecting order

To facilitate the identification of protections that need to be obtained in advance for avoiding deadlocks, we define the predicate $Prefix(P)$. For a given set of protections P it returns a set containing all protections that should be obtained before protections in P according to a predefined order relation $<_{\mathcal{V}}$ over a set of shared variables $\mathcal{V}$. We explicit the definition of $Prefix(P)$ for each protection type:

Mutex:

$$Prefix(P) = \{\, m_x \mid \exists y \,.\, x <_{\mathcal{V}} y \;\wedge\; m_y \in P \,\}$$

Read/Write

$$Prefix(P) = \{\, r_x, w_x \mid \exists y \,.\, x <_{\mathcal{V}} y \;\wedge\; (r_y \in P \vee w_y \in P) \,\}$$

Write intend

$$Prefix(P) = \{\, r_x, w_x, w'_x \mid \exists y \,.\, x <_{\mathcal{V}} y \;\wedge\; (r_y \in P \vee w'_y \in P \vee w_y \in P) \,\}$$

For example: given $\mathcal{V} = \{x, z, y\}$ and the ordering $x <_{\mathcal{V}} y <_{\mathcal{V}} z$ then $Prefix(\{m_z\}) = \{m_x, m_y\}$ and $Prefix(\{r_z\}) = \{r_x, w_x, r_y, w_y\}$ in the case of *read/write* protections and $Prefix(\{r_z\}) = \{r_x, w_x, w'_x, r_y, w_y, w'_y\}$ for *write intend* protections.

3.5 Policies for acquisition/release of protections

The problem of implementing critical sections using pessimistic synchronizations can be decomposed into the following:

- ensure the set of protections P_k necessary for the execution of an instruction i_k are obtained when it is executed;
- guarantee serializability of critical sections (using two-phase locking);
- avoid deadlocks;
- finally, allow the maximum parallelism between threads (avoidance of using a single global lock for all critical sections).

First, we provide a general algorithm for managing acquisition and release of protections in a critical section. Then we formalize the properties that should be respected in order to have a correct implementation of critical sections using protections. Finally, we instantiate the algorithm with several policies and prove their correctness.

3.5.1 General algorithm for managing protections

The principle of this algorithm is that, in order to execute any instruction i_k of the critical section CS, an extended set of protections $H_k \supseteq P_k$ which guarantees serializability and deadlock freedom of the section must be held by the thread executing it. The algorithm consists in:

- obtaining the missing protections $H_k \setminus H_{k-1}$ prior to executing i_k
- releasing the unnecessary protections $H_k \setminus H_{k+1}$ after executing i_k

Table 3.2 presents the principle of the algorithm. In the middle column we list the instructions i_k of a critical section and, separated with a colon, the set of protections P_k needed to execute them. Above them we have H_k which is the extended set of protections the executing thread should hold at that point. The computation of H_k sets is defined by policies. On the left column we have the protections that need to be obtained prior to executing i_k. These are the protections needed by i_k that have not yet been obtained at instruction i_{k-1}. We must note the special case of instruction i_1 where we obtain exactly H_1 since prior to entering a critical section no protections are held. Dually, on the right column we depose the protections to be released after executing i_k. These are instructions no longer needed. Again we note that at the last instruction i_n we release all protections held since we are about to leave the critical section.

Protections to obtain	Protections held	Protections to release
$\mathbf{H_1}$		
	H_1	
	$i_1 : P_1$	
		$H_1 \setminus H_2$
	$\vdots$	
$\mathbf{H_k \setminus H_{k-1}}$		
	$\mathbf{H_k}$	
	$i_k : \mathbf{P_k}$	
		$\mathbf{H_k \setminus H_{k+1}}$
$H_{k+1} \setminus H_k$		
	H_{k+1}	
	$i_{k+1} : P_{k+1}$	
		$H_{k+1} \setminus H_{k+2}$
	$\vdots$	
$H_n \setminus H_{n-1}$		
	H_n	
	$i_n : P_n$	
		$\mathbf{H_n}$

Table 3.2: Principle of general algorithm for managing protections

We formalize hereafter the desired properties of the algorithm presented above.

(P1) Absence of data races. All protections p_x necessary for accessing variables during instruction i_k must be obtained. That is, weak atomicity is respected.

$$\textbf{No data race} \qquad \forall x \in \mathcal{V}\ ,\ \forall k \in [1,n] : \quad p_x \in P_k \Rightarrow p_x \in H_k$$

(P2) Deadlock freedom. Protections are obtained with respect to the total order ($<_{\mathcal{V}}$) defined on the set of shared variables $\mathcal{V}$:

$$\textbf{Deadlock freedom} \qquad \left.\begin{array}{r} \forall x, y \in \mathcal{V}\ ,\ \forall k, m \in [1,n] : \\ x <_{\mathcal{V}} y \ \wedge\ k < m \\ p_y \in P_k \ \wedge\ p_x \in P_m \end{array}\right\} \Rightarrow p_x \in H_k$$

(P3) Serializability (two-phase locking). The first phase consists in obtaining (growing phase) all protections needed to execute the critical section. The second phase consists in releasing (shrinking phase) the protections. This implies that at some point in the critical section we hold all protections used $\mathcal{P}$ and that a protection can only be obtained once per critical section :

$$\textbf{Two-phase locking} \qquad \textbf{(i)}\ \exists\, i_0 \in [1,n]\,.\, H_0 = \mathcal{P}$$
$$\textbf{(ii)}\ \forall k, m \in [1,n]\ : \begin{cases} k \leq m \leq i_0 \ \Rightarrow\ H_k \subseteq H_m \\ i_0 \leq k \leq m \ \Rightarrow\ H_k \supseteq H_m \end{cases}$$

3.5.2 Policies for acquisition/release of protections

In this section we present five instantiations of the generic algorithm presented in section 3.5.1 above. Each instantiation corresponds to an acquisition/release policy of protections. Policies are first presented intuitively and then we provide their formalization i.e., the computation of the set of protections that must be held (H_k) prior to executing an instruction i_k. All policies presented respect the properties **P1,P2,P3** cited above. The proofs are given after the computation of H_k sets.

For each policy we provide a figure illustrating a critical section and the duration protections are held according to the policy. The code of critical sections is abstracted using hatched zones representing the range between the first and last usage of a shared variable. The duration a protection assigned to a shared variable is held is represented by a vertical line traversing the hatched zone of the variable. To simplify the examples we assume each shared variable (x) is protected by a mutex protection (m_x).

Figure 3.6 explains in more details the mapping between a critical section and its abstract representation as well as how duration of holding a protection comes into the picture. On the left side of the figure there is the *critical section* code (what a programmer actually writes) and on the right side the equivalent code (automatically produced) which will guarantee the correct execution of the critical section following a given policy. In the middle we have the figure itself summarizing those fragments of code. The critical section is abstracted by simply illustrating the region between first and last access to a shared variable. In this example x and y are the shared variables. The ordering for avoiding deadlocks is given explicitly above the abstracted code. Finally, the duration of holding a protection is mapped to the acquisition (`lock(`m_x`)`) and release (`unlock(`m_x`)`) of the protection (m_x) in the actual implementation of the critical section.

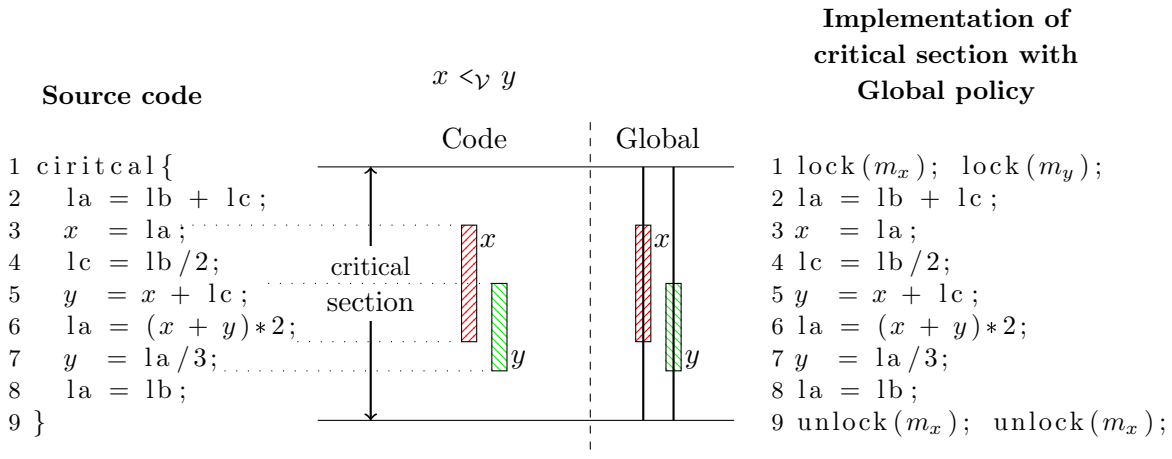


Figure 3.6: Policies for acquiring/releasing protections

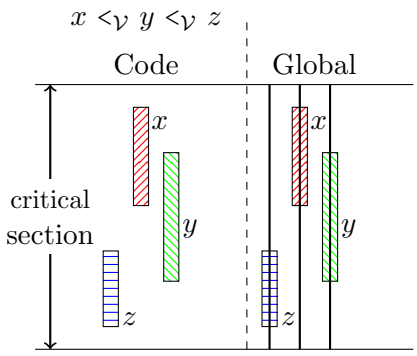


Figure 3.7: Global policy

Global policy

This policy is the most widely used in pessimistic implementations of critical sections. It consists in acquiring all protections used in the critical section prior to starting its execution. The protections are then released once the critical section has been completely executed. As illustrated in Figure 3.7 the vertical lines traversing each variable cover the entire critical section. We must note that still the acquisition of protections at the beginning of the critical section respects the ordering $<_{\mathcal{V}}$ to avoid deadlocks.

The computation of protections to hold at each point is defined as follows:

$$\forall k \in [1,n] : H_k = \bigcup_{j=1}^{n} P_j \quad = \mathcal{P}$$

At each instruction i_k all protections needed in the critical section must be held

Proof of properties P1-P3

From definition of H_k we have:

- P1 :

No data race	$\forall x \in \mathcal{V}\ ,\ \forall k \in [1,n] :\quad p_x \in P_k \Rightarrow p_x \in H_k$

$$\forall k \in [1,n]\ P_k \in H_k \Rightarrow p_x \in H_k$$

- P2 :

Deadlock freedom	$\forall x,y \in \mathcal{V}\ ,\ \forall k,m \in [1,n] :$ $\left.\begin{array}{r} x <_{\mathcal{V}} y \ \wedge\ k < m \\ p_y \in P_k \ \wedge\ p_x \in P_m \end{array}\right\} \Rightarrow p_x \in H_k$

$$\text{from } \mathbf{P1} \quad \left.\begin{array}{l} p_x \in P_m \Rightarrow p_x \in H_m \\ \forall k, m \in [1,n]\ H_k = H_m \end{array}\right\} p_x \in H_k$$

- P3 :

Two-phase locking

$$\textbf{(i)}\ \exists\, i_0 \in [1,n]\,.\, H_0 = \mathcal{P}$$

$$\textbf{(ii)}\ \forall k, m \in [1,n]\ :\ \begin{cases} k \le m \le i_0 \ \Rightarrow\ H_k \subseteq H_m \\ i_0 \le k \le m \ \Rightarrow\ H_k \supseteq H_m \end{cases}$$

(i) Any instruction $k \in [1,n]$ can be chosen as i_0 (because $H_k = \mathcal{P}$ for all k)

(ii) from definition of H_k $\forall k, m \in [1,n]\ H_k = H_m$ thus for both cases $k \le m \le i_0$ and $i_0 \le k \le m$ the property holds.

Eager policy

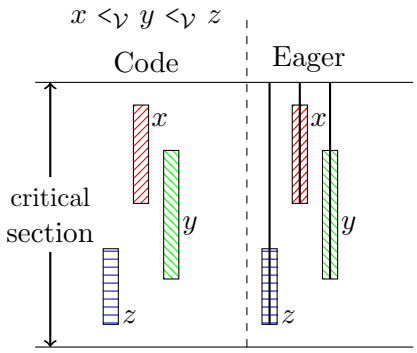


Figure 3.8: Eager policy

The principle of this policy is to release protections as soon as they are no longer needed in the critical section. All protections are acquired prior to executing the critical region. The release of a protection occurs after executing the last instruction using it in the critical section. In Figure 3.8 we can see the horizontal lines expanding from the beginning of the critical section down to the end of the hatched zone of each variable.

The computation of protections to hold at each point is defined as follows:

$$\forall k \in [1,n] : H_k = \bigcup_{j=k}^{n} P_j$$

At instruction i_k we must hold all protections used in subsequent instructions of the critical section.

Eager policy could be profitable when some shared variables are only used at the beginning of a critical section. Listing 3.3 exhibits such a case, where shared variable x is only used in the first instruction of the critical section (line 2). The function call `ackermann`

at line 4 is a computation intensive function. In this example an early release of x could allow other critical sections blocked on x to progress.

```
1 critical{
2   y = x ;
3   y = y +(a/2);
4   z = ackermann(a,y);
5 }
```

Listing 3.3: Example for Eager policy.

Proof of properties P1-P3

- P1 :

$$\textbf{No data race} \qquad \forall x \in \mathcal{V}\ ,\ \forall k \in [1,n] : \quad p_x \in P_k \Rightarrow p_x \in H_k$$

From definition of H_k we have:

$$\forall k \in [1,n] P_k \subseteq H_k \Rightarrow p_x \in H_k$$

- P2 :

$$\textbf{Deadlock freedom} \qquad \left.\begin{array}{r} \forall x, y \in \mathcal{V}\ ,\ \forall k, m \in [1,n] : \\ x <_{\mathcal{V}} y \ \wedge\ k < m \\ p_y \in P_k \ \wedge\ p_x \in P_m \end{array}\right\} \Rightarrow p_x \in H_k$$

$\forall k, m \in [1,n]$ such that $k < m$ we have $H_k \supseteq H_m$ thus $p_x \in H_k$

- P3 :

$$\textbf{Two-phase locking} \qquad \textbf{(i)}\ \exists\, i_0 \in [1,n]\,.\, H_0 = \mathcal{P}$$
$$\textbf{(ii)}\ \forall k, m \in [1,n]\ : \begin{cases} k \leq m \leq i_0 \ \Rightarrow\ H_k \subseteq H_m \\ i_0 \leq k \leq m \ \Rightarrow\ H_k \supseteq H_m \end{cases}$$

(i) $H_1 = \mathcal{P}$ thus we choose $i_0 = i_1$

(ii) from definition, H_k is decreasing that is $\forall k, m \in [1,n]$ such that $k < m$ we have $H_k \supseteq H_m$

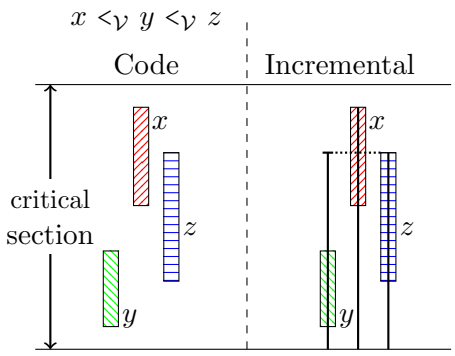


Figure 3.9: Incremental policy

Incremental policy

The principle of this policy is to acquire protections as late as possible. That is, just before the associated variable is used. Protections are released all together at the end of the critical section. We must note that for this policy (and the remaining ones) anticipation (i.e., earlier acquisition of protections) may be needed in order to avoid deadlocks. The anticipation is imposed by the statically defined total ordering on variables and by the order they appear in the critical section. In Figure 3.9 we can note protection on y is anticipated up to before obtaining protection on z. Finally, we observe protections are released at the exit of the critical section.

The computation of protections to hold at each point is defined incrementally. First, we compute H_k^d which extends the set of used variables P_k by adding necessary protections for avoiding deadlocks. Here is the computation of H_k :

$\forall k \in [1, n]$:

- $H_k^d = (\bigcup_{j=k}^{n} P_j) \cap Prefix(P_k)$
- $H_k = \bigcup_{j \leq k} (H_j^d \cup P_j)$

At instruction i_k we must hold the protections used by all instructions preceding it plus the protections of instructions used latter that could cause a deadlock.

Incremental policy could be beneficial when a shared variable is only used in the end of a critical section. Listing 3.4 exhibits such a case, where shared variable x is only used in the last instruction of the critical section (line 4). In this example a late acquisition of x could allow the progress of the critical section even if the protection on x was not yet available.

```
1 atomic{
2   y = ackermann(z,a);
3   y = y+a/2;
4   x = y;
5 }
```

Listing 3.4: Example for Incremental policy.

Proof of properties P1-P3

- P1 :

$$\textbf{No data race} \qquad \forall x \in \mathcal{V}\ ,\ \forall k \in [1,n] : \quad p_x \in P_k \Rightarrow p_x \in H_k$$

From definition of H_k we have:

$$\forall k \in [1,n] P_k \subseteq H_k \Rightarrow p_x \in H_k$$

- P2 :

$$\textbf{Deadlock freedom} \qquad \left.\begin{array}{r} \forall x, y \in \mathcal{V}\ ,\ \forall k, m \in [1,n] : \\ x <_{\mathcal{V}} y \ \wedge\ k < m \\ p_y \in P_k \ \wedge\ p_x \in P_m \end{array}\right\} \Rightarrow p_x \in H_k$$

$$\left.\begin{array}{ll} x <_{\mathcal{V}} y & \Rightarrow p_x \in Prefix(k) \\ k < m & \Rightarrow p_x \in \bigcup_{j=k}^{n} P_j \end{array}\right\} \Rightarrow p_x \in H_k^d \Rightarrow p_x \in H_k$$

- P3 :

$$\textbf{Two-phase locking} \qquad \textbf{(i)}\ \exists\, i_0 \in [1,n]\,.\, H_0 = \mathcal{P}$$
$$\textbf{(ii)}\ \forall k, m \in [1,n]\ : \left\{\begin{array}{l} k \leq m \leq i_0 \ \Rightarrow\ H_k \subseteq H_m \\ i_0 \leq k \leq m \ \Rightarrow\ H_k \supseteq H_m \end{array}\right.$$

(i) $H_n = \mathcal{P}$ thus we choose $i_0 = i_n$

(ii) from definition, H_k is increasing that is $\forall k, m \in [1,n]$ such that $k < m$ we have $H_k \subseteq H_m$

Incremental/Eager policy

As its name implies it is a combination of the two policies described above. Protections are acquired following the *incremental* policy and released using the *eager* policy. We must note though that to respect *two phase locking* protections can only be released once

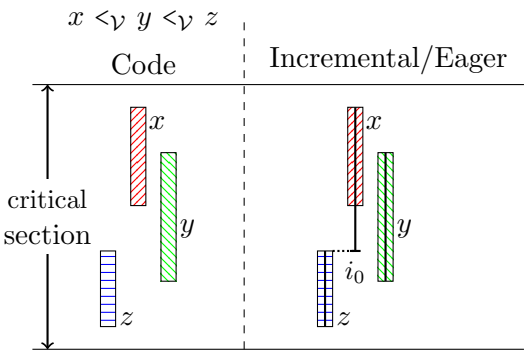


Figure 3.10: Incremental/Eager policy

all protections necessary to the critical section are obtained. In Figure 3.10 we can note that the possession of protection on variable x is extended until point i_0 where the last protection (on z) is acquired. Point indicated as i_0 is the turning point of two phase locking.

The computation of protections to hold depends on their position relative to i_0. For instructions prior to i_0 the incremental policy is applied, thus special care must be taken to avoid deadlocks, while for instructions subsequent to i_0 eager policy is applied. Instruction i_0 is identified as the instruction at which we have discovered all protections needed for the critical section ($\mathcal{P}$).

- $i_0 = Min(\{k \in [1,n] \mid (\mathcal{P} \setminus \bigcup_{j \le k} P_j) = \varnothing\})$
- $\forall k \in [1,n] : H_k^d = (\bigcup_{j=k}^{n} P_j) \cap Prefix(P_k)$
- $$H_k = \begin{cases} \bigcup_{j \le k} (H_j^d \cup P_j) & \text{if } k \le i_0 \\ \bigcup_{j \ge k} P_j & \text{if } k > i_0 \end{cases}$$

Compute H_k as in *Incremental* policy until last protection needed is reached, then switch to *Eager* policy.

Proof of properties P1-P3

This policy is the combination of *incremental* and *eager*. Each policy is applied respectively to the sub-critical sections $[1, i_0]$ and $[i_0, n]$. The switching of policy is correct because it occurs when all protections are obtained.

- P1 :

No data race $\quad \forall x \in \mathcal{V}\ ,\ \forall k \in [1,n] : \quad p_x \in P_k \Rightarrow p_x \in H_k$

From definition of H_k we have:

$$\forall k \in [1,n] P_k \subseteq H_k \Rightarrow p_x \in H_k$$

- P2 :

$$\textbf{Deadlock freedom} \quad \left.\begin{array}{r} \forall x, y \in \mathcal{V}\ ,\ \forall k, m \in [1, n] : \\ x <_{\mathcal{V}} y \ \wedge\ k < m \\ p_y \in P_k \ \wedge\ p_x \in P_m \end{array}\right\} \Rightarrow p_x \in H_k$$

- for interval $[1, i_0]$ same as P2 for *incremental policy*
- for interval $(i_0, n]$ same as P2 for *eager policy*

- P3 :

$$\textbf{Two-phase locking} \quad \textbf{(i)}\ \exists\, i_0 \in [1, n]\,.\, H_0 = \mathcal{P}$$
$$\textbf{(ii)}\ \forall k, m \in [1, n]\ : \begin{cases} k \leq m \leq i_0 \ \Rightarrow\ H_k \subseteq H_m \\ i_0 \leq k \leq m \ \Rightarrow\ H_k \supseteq H_m \end{cases}$$

(i) The instruction i_0 is explicitly defined such as it respects the property. In the worst case, i_0 will be i_n and the policy is equivalent to *incremental*

(ii) from definition of H_k: in the interval [1,i_0] H_k is increasing and decreasing in (i_0,n]). Hence, in both cases the property is respected.

Incremental/Priority release policy

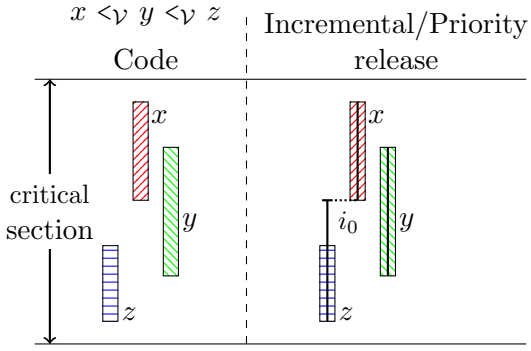


Figure 3.11: Incremental/Priority release policy

This policy is very similar to *incremental/eager* since again protections are acquired incrementally up to an instruction i_0 and released eagerly after that point. Instruction i_0 is defined differently this time so that early release of protections is prioritized. That is, we set i_0 to be the first instruction after which we no longer use a protection. This time respecting *two-phase locking* enforces the anticipation of protections. In Figure 3.11 we note that i_0 is now located at the last usage of x. After that point p_x can be released since it is no further used in the critical section. To allow the release we must first obtain protection on z. As we can see this results in obtaining protection on z earlier than we would if a purely incremental policy was applied.

- $i_0 = Min\left(\{k \in [1,n] \mid \exists x.\, x \in P_k \wedge x \notin \bigcup_{j>k} P_j\}\right)$
- $\forall k \in [1,n] : H_k^d = \left(\bigcup_{j=k}^{n} P_j\right) \cap Prefix(P_k)$
- $$H_k = \begin{cases} \bigcup_{j \le k}(H_j^d \cup P_j) & \text{if } k < i_0 \\ \mathcal{P} & \text{if } k = i_0 \\ \bigcup_{j \ge k} P_j & \text{if } k > i_0 \end{cases}$$

Compute H_k as in *Incremental* policy until last occurrence of a protection, then acquire all protections missing from $\mathcal{P}$ and switch to *Eager* policy.

Proof of properties P1-P3

This policy is the combination of *incremental* and *eager*. Each policy is applied respectively to the sub-critical sections $[1, i_0]$ and $[i_0, n]$. Instruction i_0 is explicitly defined to prioritize release of protections. The switching of policy at i_0 is correct because it occurs when all protections are obtained.

- P1 :

No data race $\quad \forall x \in \mathcal{V}\ ,\ \forall k \in [1,n] : \quad p_x \in P_k \Rightarrow p_x \in H_k$

From definition of H_k we have:

$$\forall k \in [1,n] P_k \subseteq H_k \Rightarrow p_x \in H_k$$

- P2 :

Deadlock freedom
$$\left.\begin{array}{r} \forall x, y \in \mathcal{V}\ ,\ \forall k, m \in [1,n] : \\ x <_{\mathcal{V}}\ y\ \wedge\ k < m \\ p_y \in P_k\ \wedge\ p_x \in P_m \end{array}\right\} \Rightarrow p_x \in H_k$$

- for interval $[1, i_0]$ same as P2 for *incremental policy*
- for interval $(i_0, n]$ same as P2 for *eager policy*

- P3 :

Two-phase locking
$$\textbf{(i)}\ \exists\, i_0 \in [1,n]\,.\, H_0 = \mathcal{P}$$
$$\textbf{(ii)}\ \forall k, m \in [1,n]\ :\ \begin{cases} k \le m \le i_0 \ \Rightarrow\ H_k \subseteq H_m \\ i_0 \le k \le m \ \Rightarrow\ H_k \supseteq H_m \end{cases}$$

(i) The instruction i_0 is explicitly defined such as it respects the property. In the best case, i_0 will be i_1 and the policy is equivalent to *eager*.

(ii) from definition of H_k: in the interval [1,i_0] H_k is increasing and decreasing in (i_0,n]). Hence, in both cases the property is respected.

3.6 Observations on policies

Figure 3.12 puts all policies side by side and shows how each behaves for a common critical section. We can note that all the policies we propose reduce the time protections are held. Policies *Eager* and *Incremental* have a dual behavior. In this example, no anticipation is needed for incremental policy. This is an ideal case since it gives the minimal time protections can be held for this policy (each protection acquired exactly prior to its usage). Policies *Incremental/Eager* and *Incremental priority release* seem to behave best with respect to the other policies since they limit the overall time protections are held. But each policy can prioritize/penalize the protection on a variable. Policy *Incremental/Eager* prioritizes variable z and penalizes x while *Incremental priority release* has the inverse effect on these variables. As we detail next the definition of i_0 (the point where all protections $\mathcal{P}$ are held and we switch from incremental to eager policy) is very important in optimizing a critical section.

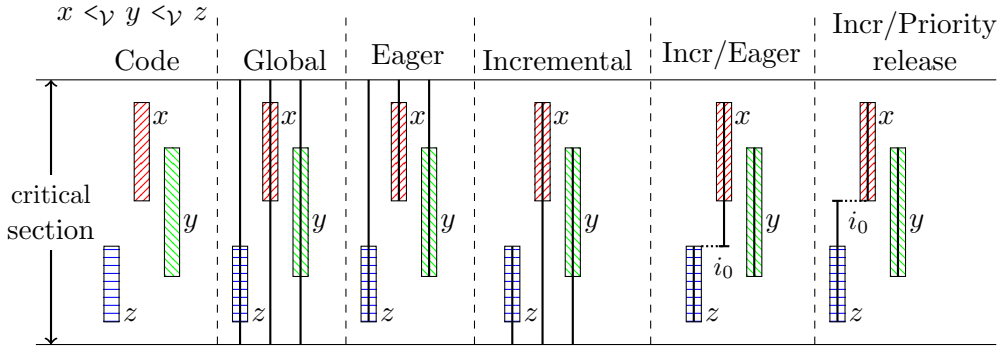


Figure 3.12: Policies for acquiring/releasing protections

3.6.1 Equivalence of *Incremental/Eager* and *Incremental priority release*

We present here a special case where i_0 is implicitly defined in the critical section. As illustrated in Figure 3.13 the regions of access to shared variables overlap. All instructions in the region where the totality of shared variables in the critical region overlap consist explicit definitions of i_0. We note in the figure i_0 *incr/eager* (respectively *pr. release*) the point matching to instruction i_0 for *Incremental/Eager* policy (respectively *Incremental priority release*). Any of them can be chosen as i_0 and both *Incremental/Eager* and *Incremental priority release* policies produce the same acquisitions and release of protections. In fact policy *Incremental priority release* has no meaning since there is no release to prioritize over the acquisitions of protections. As we can observe in the middle of the figure, protection on variable y needs to be anticipated. If the following ordering was chosen $x <_\mathcal{V} z <_\mathcal{V} y$ then the policies would give the optimal solution since the protection on each variable would be held exactly while it is accessed. This optimal solution is illustrated at the right side of the figure. This observation raises the issue of defining an optimal total ordering of the variables.

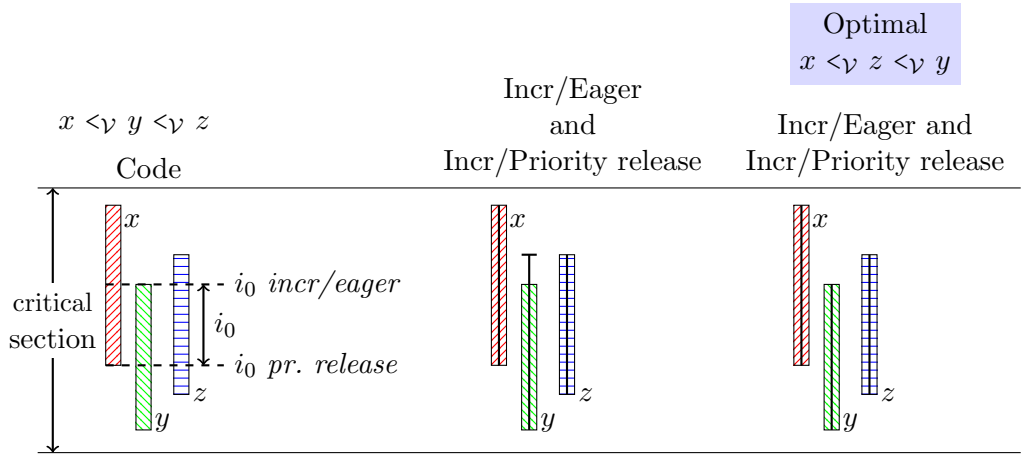


Figure 3.13: Equivalence of *Incremental/Eager* and *Incremental priority release*

3.6.2 Optimizing critical sections implemented with *Incremental* policies

As observed in the example of Figure 3.13 the total order chosen on variables for deadlock avoidance has a great impact on the efficiency of incremental policies since they heavily reside on it. For policies *Incremental/Eager* and *Incremental priority release* the choice of instruction i_0 also strongly affects performance. First, we focus on the effect of choosing an instruction i_0 between i_0 *pr. release* and i_0 *incre/eager* when i_0 is not implicitly defined. Then we provide heuristics for defining orderings of variables.

In a critical section with irregular accesses to shared variables as for instance in Figure 3.14 it is impossible to have an optimal solution as that in the right side of Figure 3.13. For the example illustrated in Figure 3.14 we assume a strong contention on variable w is expected. In this case, defining either point marked i_0 *opt* or i_0 *opt* $'$ as i_0 would be the best choice since they would guarantee that protection on w will be released right after its last access.

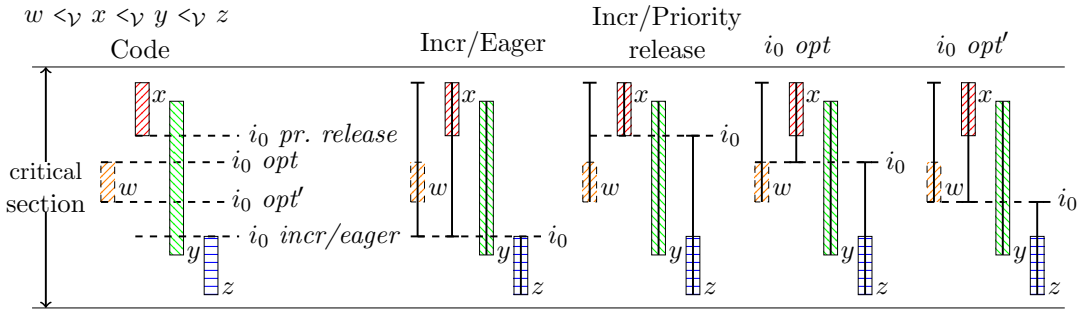


Figure 3.14: Choosing arbitrary i_0

For the order chosen $w<_\mathcal{V}$ $x<_\mathcal{V}$ $y<_\mathcal{V}$ $z<_\mathcal{V}$ and each i_0 delimited we provide the region in the critical section each protection will be held. We focus on the effect of i_0 *opt* and i_0 *opt*$'$. For both of them we can observe that they give the same hold regions which is

also identical to that obtained by policy *Incremental priority release*. In the case of i_0 *opt* variable x is less penalized than z while the contrary is observed in the case of i_0 *opt*$'$.

To optimize access to variable w a better ordering must be defined. Figure 3.15 illustrates how protections are held with the new ordering. We can observe how i_0 *opt* and i_0 *opt*$'$ optimize access to w. Moreover, with this ordering policy *Incremental priority release* penalizes access to w since it forces the acquisition of its protection earlier. In the case of policy *Incremental eager* we can note that variable w could be released immediately after its last access. This does not happen since the algorithm will not release anything until it reaches point i_0.

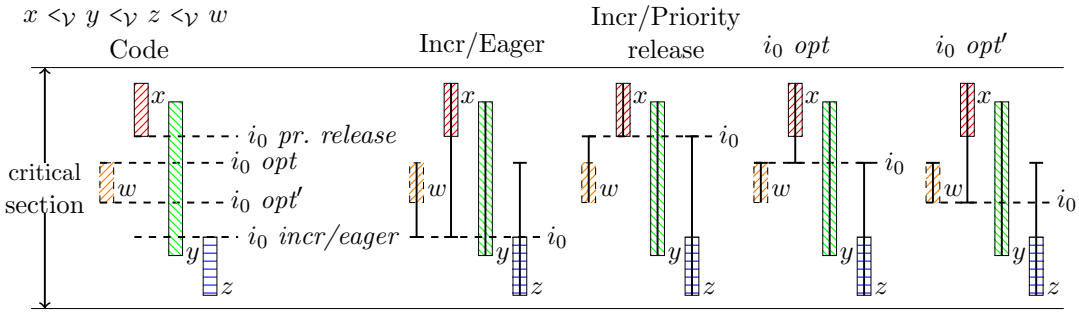


Figure 3.15: Optimizing access to a variable

3.6.3 Inferring optimal total order of variables

As mentioned earlier to avoid deadlocks a total order must be specified on all shared variables. This ordering affects all *incremental* policies. We provide here two heuristics for specifying orderings that will optimize performance of incremental policies but also in general of the application since even for *global* or *eager* policy the protections must still be obtained respecting this ordering. The principle behind defining an optimized ordering is that variables that appear higher in the ordering have less dependencies.

Optimize access to a single variable implies that it should never be anticipated. A variable is anticipated when in a critical section variable x is accessed prior to variable y and the variables are ordered in the inverse way (i.e., $y <_\mathcal{V} x$). To ensure a protection on a variable z is never anticipated it should be assigned the higher order:

$$\forall\, x \in \mathcal{V} \;\Rightarrow\; x <_\mathcal{V} z$$

Optimize overall critical sections implies finding an ordering such that the acquisition is optimized for the greatest number of variables. Figure 3.16 illustrates five different critical sections of a program. For each critical section we form a word by concatenating the shared variables in the order they appear when crossing the critical section. The word formed appears below each critical section. The order that minimizes overall anticipations is that identified as the longest substring of these

words. In this example substring xy is the longest substring appearing the most often. Thus, the ordering chosen should always respect that $x <_{\mathcal{V}} y$.

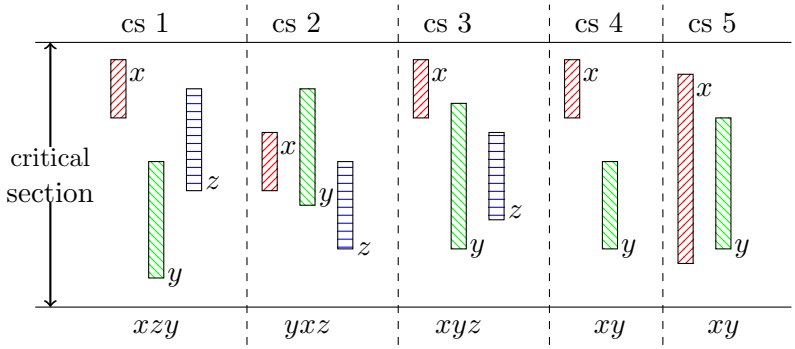
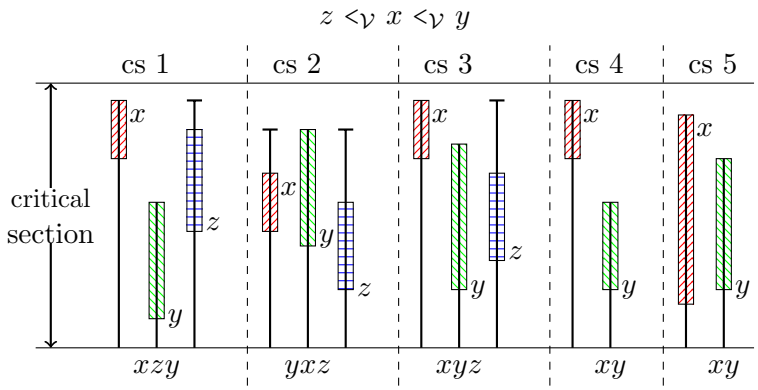
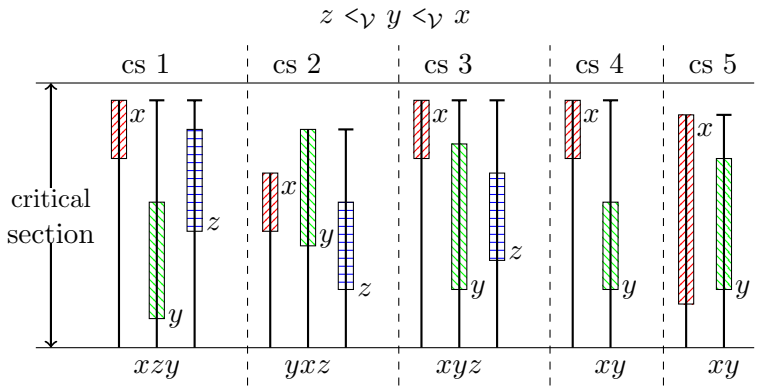


Figure 3.16: Optimizing order for critical sections

Figure 3.17 illustrates how protections would be held for all five critical sections using *Incremental policy* and two different orderings. Figure 3.17(a) is using the ordering respecting the heuristic on longest common prefix while Figure 3.17(b) violates it. As we can note, when the *good* ordering is chosen much less anticipation is needed (see variable y).



(a) Good ordering of variables



(b) Bad ordering of variables

Figure 3.17: Good vs bad ordering

3.7 Extending critical sections

The critical sections we considered were sequences of assignments. This assumption simplifies reasoning on protections but it could be too restrictive in practice. In this section we discuss the integration of (i) loops (ii) conditionals and (iii) function calls into critical sections. These constructs modify dynamically the execution flow of the program. Thus, the algorithm should be modified in order to deal with sets of sequential execution paths.

3.7.1 Loops and conditionals

Reasoning on loops and conditionals is typically done using a *control flow graph* (CFG). A control flow graph is a simple representation of a program (or an excerpt of code) consisting of *instructions* connected with *directed edges*. A CFG usually has a single *entry* and *exit* point and in-between branching and merging instructions. Instructions that form sequences can be grouped together into so-called *basic blocks*.

Figure 3.18 presents the notations we use to illustrate CFGs of *critical sections* for the remaining of the chapter. On the left side resides the code of a critical section and on the right its corresponding CFG. As we can note the entry (critical_in) and exit (critical_out) points are denoted by distinctive nodes at the top and bottom of the figure respectively. Individual instructions and basic blocks are put into boxes; branching conditionals (**`if`**) in diamond shapes; and loop conditionals (**`while`**) in trapezium shapes. Loops are further recognized by the cyclic edges in the graph. In this example after executing the code of while, we return to its condition to check again if it is still satisfied or not. For instructions of a CFG we define the following:

$succ(k)$ function returning all *successors* of instruction k. In Figure 3.18 the successors of instruction marked as i_3 are all instructions on the dashed path starting at i_3 and ending at exit point critical_out.

$pred(k)$ function returning all *predecessors* of instruction k. In Figure 3.18 the predecessors of instruction marked as i_3 are all instructions on the dash-dotted path starting at i_3 and going upwards to the entry point critical_in.

$loop(k, m)$ a predicate deciding whether or not instructions k and m belong to the same loop or into nested loops.

$allPaths(k)$ a predicate which decides whether instruction k belongs to *all paths* or not. The predicate makes the distinction between instructions and conditionals. In Figure 3.18 the instructions for which the predicate is true are i_1 and i_8. We must note that i_2 is excluded despite it is executed in all paths because it is a predicate.

Conditionals and loops diverge the execution flow at runtime forming different *paths* (sequences of instructions) connecting the entry and exit point of the critical section. The algorithm we presented in section 3.5.1 is based on computing precisely the protections that must be held prior to executing an instruction. Computing this set precisely is no

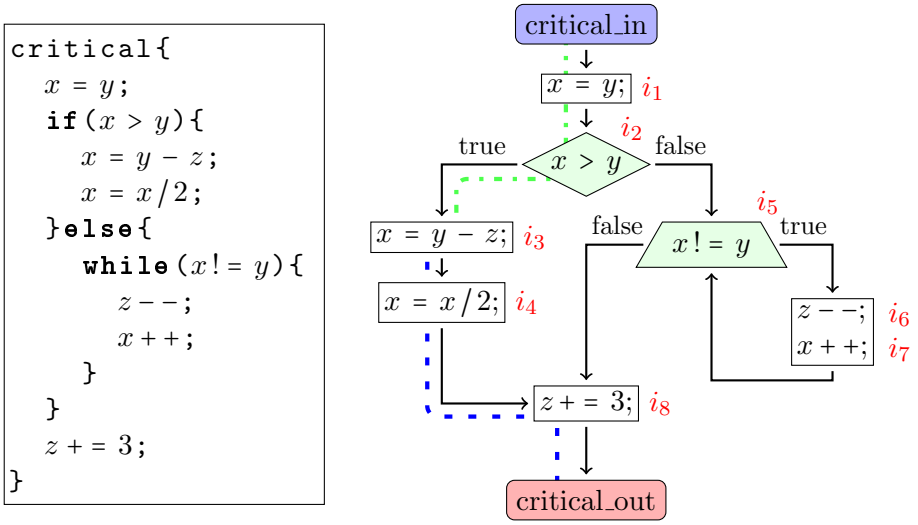


Figure 3.18: Control Flow Graph (CFG) example.

longer feasible statically due to non determinism. We discuss hereafter how this influences the policies we proposed.

Global policy

The *global* policy is not affected by *loops* nor by *conditionals*. This is due to the fact that protections are acquired and released similarly for all execution paths of the critical section. That is, protections are obtained independently of belonging to the path that will actually get executed or not. Parallelism is reduced when protections that are not necessary are obtained.

With *control flow graph* notations the definition of variables to be held by each instruction would be the following:

$$\forall k \in [1,n] : H_k = \bigcup_{j \in succ(1)} P_j \quad = \mathcal{P}$$

Eager policy

According to this policy protections are released after the last access to the protected variable. When the last access occurs inside a conditional then special care should be taken on where to position the release of the protection. We distinct the following cases:

loop conditional: in this case if the condition is true then the loop body is executed and unless a **`break`** instruction is encountered the condition is re-evaluated (at least once more). Hence, releasing the protection inside the loop body could cause: (i) re-accessing the shared variable without holding the adequate protections and (ii) releasing a protection without holding it (since it was released on previous iteration). Releasing a protection without holding it can have several side-effects depending on

the implementation of protections. For instance, in the implementation of *Pthreads - fast mutexes* there is no control if the releasing thread is the same as the one that obtained it and thus we could release the mutex obtained by some other thread.

Finally, for loop conditionals releasing the protection of a variable that appears last in the conditional should be executed at the exit of the loop. Figure 3.19 illustrates a critical section consisting of a while loop. On the transitions between nodes we added, where necessary, the protections to be *acquired* (Acq) and *released* (Rel). We note the release of p_w being executed after exiting the loop.

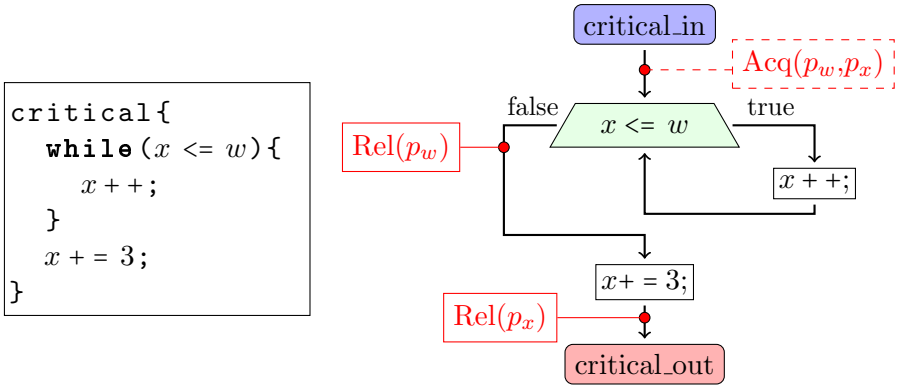


Figure 3.19: While loop conditional release protection.

branching conditional: in this case two execution paths are feasible. Since the variable is no longer accessed it should be released on both paths. Figure 3.20 illustrates the releasing of protections. As we can note, independently of the branch executed the protection will be released immediately after its last access (inside the conditional).

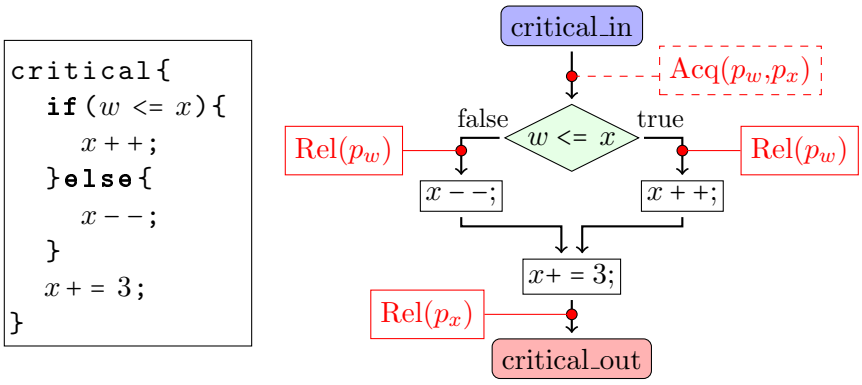


Figure 3.20: Branch loop conditional release protection.

When the last occurrence of a variable is inside the body of a loop then apart the two problems we aborted previously, on releasing repeatedly inside the loop, we are faced with a dual problem of *not releasing a protection* due to un-executed paths. Often loops contain instructions such as **`break`** and **`continue`** which cause a direct *jump* outside the loop and to the conditional check respectively. In both cases instructions of the loop subsequent to these instructions are not executed and thus the releasing of protections could be skipped. This would result into exiting the critical section while possessing protections.

To solve this problem, we postpone again all releases until exiting the loop. In case of nested loops releases must be postponed until the end of the outermost loop. Figure 3.21 presents on the left a critical section containing two nested **while** loops and on the right the corresponding CFG illustrating the solution. On the transitions between instructions we hook the sets of protections that should be acquired (Acq) and released (Rel) as computed by the algorithm presented in section 3.5.1 on page 41. The release of p_y at the designated position is incorrect as premature. On the other hand, releasing of p_z will never occur due to the **break** instruction preceding it. Dashed lines illustrate the postponing of releases outside of the outermost loop.

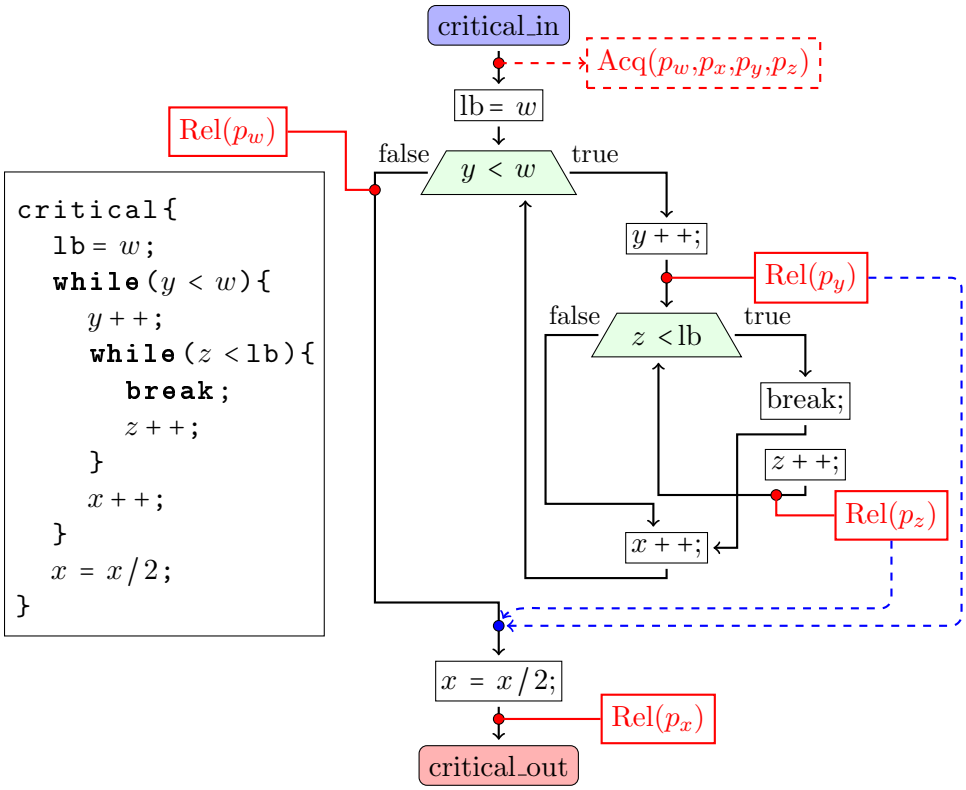


Figure 3.21: Nested loops in critical section.

Finally, we provide the formalization of computing hold sets for an instruction i_k in a loop. The set H_k no longer holds only the protections necessary for the subsequent instructions, but also the protections of preceding instructions residing inside the same loop scope. This modification is necessary to enforce releasing protections after exiting the loop. Although protections are released immediately after their last access the overhead of holding the protections is equal to that of executing remaining instructions on last iteration.

$$\forall k \in [1,n] : H_k = \bigcup_{j \in succ(j)} P_j \cup \bigcup_{j \in pred(j)\,.\,loop(j,k)} P_j$$

For branching conditionals (**if** statements) no problem occurs as long as the code executed in them consists of sequences of assignments and nested branchings. In this case,

the computation of protections to be held at an instruction i_k is defined as the necessary protections of all successors. Protections on not executed branch can be released immediately.

$$\forall k \in [1,n] : H_k = \bigcup_{j \in succ(j)} P_j \quad = \mathcal{P}$$

Incremental policy

According to this policy, protections are obtained as late as possible. We remind that anticipation is often needed to avoid deadlocks. The problems to be faced due to multiple executions are the following: (i) how to compute protections to be anticipated and (ii) what to release at the end of the critical section.

To avoid deadlocks the set H_k^d must be correctly computed for each instruction. Many execution paths with different sets of variables used on each may exist from instruction i_k to the end of the critical section. To be sure no protection is obtained out of order H_k^d is defined as the union of protections used on all paths originating from i_k. This implies that some protections may be anticipated even though their path may not get executed. Figure 3.22 illustrates an example where protection p_y is over-approximated since it must be obtained prior to executing the conditional. More formally the re-definition of H_k^d:

$$\forall k \in [1,n] : H_k^d = (\bigcup_{j \in succ(j)} P_j) \bigcap Prefix(P_k)$$

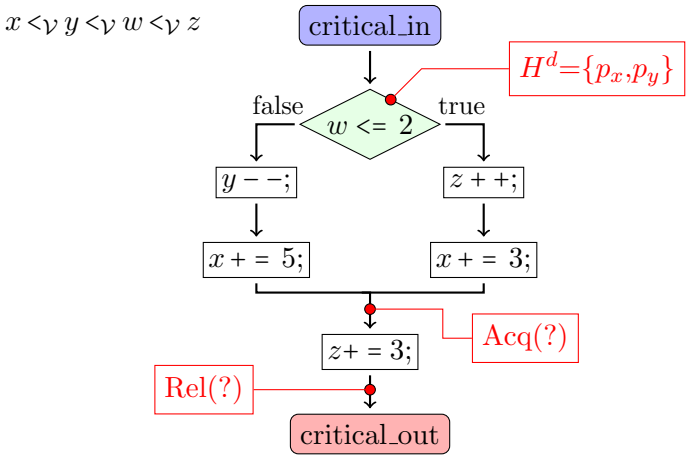


Figure 3.22: Pessimistic H_k^d computation

Another problem with incremental acquisition is that after merging two execution paths, we can no longer specify the set of protections held at the merging point. In Figure 3.22 at the merging of the conditional protection on variable z is already held if the *true* branch was executed and dually not held if *false* branch was executed. A safe solution consists in assuming protections that do not belong to the intersection of all paths not to be acquired. Thus in the example, protection p_z should be acquired. Because the protection may already be held we must be sure that the implementation of protections

used is *re-entrant* (i.e., a thread can obtain a protection it already holds). This problem of re-acquiring protections also occurs when iterating loops.

The final problem to be solved is related to not knowing exactly the set of protections that were obtained during the execution of the critical section and thus what should be released prior leaving the critical section. As mentioned earlier releasing a protection not owned can be dangerous. A solution we propose to this problem is having a primitive that allows the release of all protections held by a thread.

3.7.2 Function calls

Function calls introduce several challenges in the implementation of critical sections caused by: (i) calling library functions of which the source code is not available for analysis (ii) recursive functions (iii) nested critical sections, when the function called contains itself a critical section.

We consider the simplest case where the called function contains no recursion and does not call any library functions. In this case the code of the function should be inlined such that is is analyzed along with instructions preceding and following its call. Moreover, if some called function contains critical sections they should be removed while in-lining.

3.8 Recapitulation

In this chapter we addressed the problem of optimizing critical sections with the pessimistic approach i.e., using synchronization mechanisms. Initially, we formalized the usage of two classic synchronization mechanisms *mutexes* and *read/write* locks. We also proposed *write intend* locks, a variation of *read/write* lock which can be beneficial when there is a big number of reading threads and a few that update a value. Next, we focused on the optimization of critical sections by reducing the overall time protections are held. We presented a generic scheme for the acquisition/release of protections and used it to define five policies. We proved that all policies respect a set of properties that guarantee the correct implementation of a critical section with the pessimistic approach. Finally, we presented how to extend our work for critical sections consisting of instructions other than simple assignments.

Comparison to existing work

Existing works on optimizing pessimistic implementation of critical sections can be divided into three categories: (i) lock free implementations (ii) identification of the finest possible locks to protect the critical section and (iii) optimize implementation of synchronization mechanisms.

Our work is complementary to that of identifying the finest possible locks. Definitively, finding the finest locks reduces contention on synchronization mechanisms and thus allows more parallelism. Our algorithm assumes this tedious work has been applied to critical

sections before applying it. With the policies we proposed, except the *global* which is the most commonly used in the literature, we minimize possession of synchronization mechanisms to smallest possible portion of the critical section without violating its semantics. Finally, we exhibit through a series of experimentation (presented in chapter 5) the effect of each policy.

Chapter 4

Information flow analysis for multithreaded programs

Information flow analyses infer the data and control dependencies that can occur in a program. In software security such information is useful for detecting and preventing the exploit of software vulnerabilities and confidentiality breaches. In debugging and testing it can be useful for understanding how errors occur and what are their sources. Moreover, parallelizing compilers can also benefit of it since accesses to independent data can be safely parallelized. Tracing information flow in sequential programs is difficult due to dynamic memory allocations, control flow branchings etc. Adapting information flow analyses to multithreaded programs is even more challenging due to the non-deterministic execution, caused by the scheduling of threads and the relaxations of the execution platform.

Both *static* and *dynamic* techniques have been proposed to address the information flow tracing problem. *Static* approaches usually reason on source code level. A vast majority reposes on *type systems* to define languages that guarantee by construction secure information flows, i.e. executions that do not leak any confidential information. Volpano [VS97] and Sabelfeld [SM06] have proposed such sequential languages while Barthe [BRRS10] and Smith [SV98] include in their languages some basic primitives for multithreaded development. A drawback of these approaches is that they can reject programs that occasionally flow sensitive data. For instance a benign program may leak sensitive information only when sending a crash report to its developers. *Dynamic* approaches e.g., TaintEraser [ZJS+11] are better adapted since they monitor at runtime the flow of sensitive data and can interfere in order to prevent the leaking. Dynamic information flow tracing (DIFT) or taint analysis is widely used for detecting software vulnerabilities and avoid their exploit. As it applies dynamically it is much more precise than static analyses. We detail taint analysis and how it propagates in section 4.1.

Hereafter we motivate taint analysis and give an overview of the techniques employed to address the problem. Further, we introduce runtime prediction: a method to generalize executions of multi-threaded programs. We present our algorithm for predictive taint analysis. The implementation of our algorithm along with a proof of concept experimentation are presented in chapter 5.

4.1 Taint analysis

Taint analysis is a dynamic information flow tracing technique which consists of tainting (marking) sensitive or untrusted data and tracing their flow through a program. To perform taint analysis we need to specify: (i) the taint sources and (ii) a propagation policy of taintness. Often, untrusted data such as user input and network traffic are used as taint sources. The propagation of taintness may occur *explicitly* through copy of value (e.g., assignment) or *implicitly* through covert channels (e.g., control flow).

To limit propagation of taintness, a dual process of untainting is used to mark data as safe. This occurs by assigning an untainted value to data or by sanitizing it i.e., check they respect some rules and if necessary modify them such that they conform to these rules. We introduce the following notation for abstracting taint sources and sanitization:

T stands for $Taint$ and is used to abstract all possible taint sources. For instance, user input obtained through `scanf` function will be replaced by an assignment of T in the variable written as in the example:

$$\texttt{scanf("\%d",val);} \implies \texttt{val} = T;$$

U stands for $Untaint$ and is used to abstract sanitization functions. Sanitization is often used on untrusted data in order to ensure they are harmless and thus they can safely be untainted after its completion. Assuming function `clean_search` sanitizes a search string for SQL injections then we can make the following replacement:

$$\texttt{clean\_search(input);} \implies \texttt{input} = U;$$

4.1.1 Explicit information flow

Listing 4.1 presents an excerpt of code where initially variable `a` gets tainted at instruction 2 (by reading user input into it). Also variable `e` gets eventually tainted through the dependency path $\texttt{e} \Rightarrow \texttt{d} \Rightarrow \texttt{a} \Rightarrow T$. The propagation of taintness in variable `d` is straight forward since we have an explicit copy of the tainted value. In the case of variable `b` the effect of the assignment is subtle to the taint propagation policy chosen. For instance, it may be assumed that merging tainted data with untainted absorbs the tainting effect. Most often though it suffices one operand to be tainted in order to propagate taint. Thus, in most existing taint analyses `b` would be considered tainted too. Finally, we note the sanitation of `a`.

4.1.2 Implicit information flow

Listing 4.2 presents an implicit information flow. Again variable `a` gets initially tainted. The tainted data control program execution and thus information about it can leak. In this example, an external observer can infer information about the value of `a` by looking at the printed value of `b`. Thus information about `a` is implicitly propagated to all variables

set inside the scope of control (this includes `c`). Variable `d` is not tainted since it is not affected by the value of `a`. Implicit flows are very hard to detect since covert channels can be implemented in many ways. Some typical examples are timing and storage channels.

```
1 int a,b,c,d,e;
2 a = T; // scanf("%d",a);
3 c = 21;
4 d = a;
5 b = c + a;
6 a = U; // sanitize(a);
7 e = d;
```

Listing 4.1: Explicit flow

```
1 int a,b,c,d;
2 a = T; //scanf("%d",a);
3 if( a >10 ){
4   b = 1;
5 }else{
6   b = 0;
7   c = 2;
8 }
9 d = 10;
10 printf("%d",b);
```

Listing 4.2: Implicit flow

Because implicit flows are based on covert channels they are tedious to track both statically and dynamically and are often neglected. Implicit information flows are mostly critical for the non-interference property where confidential data can leak unconsciously. In the context of taint analysis and vulnerability detection implicit flows affect subtly the exploitation of a vulnerability. Moreover, implicit propagation of taint introduces too many false positives which degrades the efficiency of the analysis.

4.1.3 Application of taint analysis

As mentioned earlier taint analysis targets mostly vulnerability detection and prevention. Therefore, most taint analyses are performed dynamically at runtime. Tainted data are tracked down and the appropriate checks are performed when necessary in order to respect the security property.

```
1 int read_input(){
2   char buf[32];
3   gets(buf);
4   return 0;
5 }
```

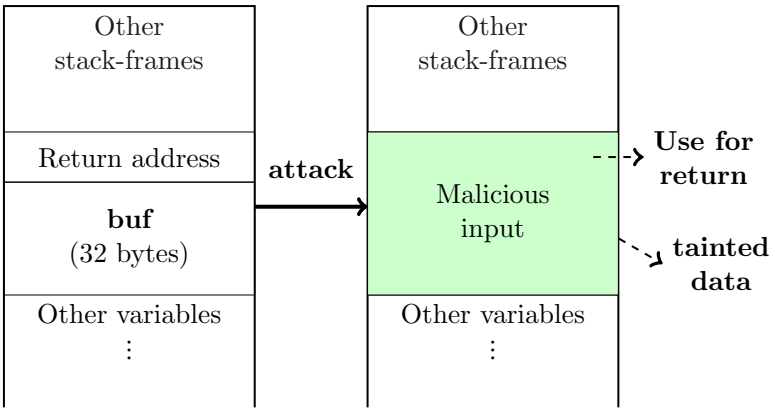


Figure 4.1: Stack smashing by buffer overflow.

A typical example demonstrating how taint analysis is used for vulnerability prevention is stack smashing caused by a buffer overflow. Figure 4.1 describes such an attack. On the

left of the figure we present the code of the function which reads user input into a buffer of size 32 bytes. The programmer assumes the size associated to the buffer is enough to hold the input provided. If a user enters a longer input then the data will overflow the buffer resulting into writing a new value to the return address associated to this stack frame (illustrate on the right stack frame). This vulnerability can be exploited by ensuring that the return address is not overwritten with random data but with an address to a malicious set of instructions. To prevent that from happening a taint analysis should check if the return address is tainted prior to jumping to it.

4.2 Tracing taintness

Dynamic analysis techniques are widely used in the context of multithreaded applications for runtime error detection like deadlocks ([LELS05, CFC12]) and data races ([SBN+97, SI09]. Although detecting data races could be useful for information-flow analysis, it is not sufficient as such. Hence, more focused analyses are developed to deal with malware detection ([BKK10, ESKK08]) and enforcement of security policies ([ZJS+11, CM09]).

Building dynamic analysis tools necessitates integrating some monitoring facilities to the analyzed application. Monitoring features are added either at source code level or binary level, either statically or dynamically. Waddington et al. [WRS] present a survey on these techniques. Working at the binary level allows to analyze programs for which source code is not available such as malwares or libraries. A major drawback is that high level information is lost making it harder to reason about the program.

Instrumentation code is often added statically in applications as explicit logging instructions. It necessitates access to the source code and can be added accordingly by the developers (which is a tedious and error-prone procedure) or automatically. To automate this process source-to-source transformations can be applied, for instance using aspect-oriented programming. Apart from the source level, static instrumentation can also be applied directly at the binary level, e.g., using binary rewriting functionality of frameworks like Dyninst [BH00]. Hereafter we take a closer look to dynamic binary instrumentation (DBI) techniques since they are the most widely used.

4.2.1 Dynamic binary instrumentation

In general, DBI frameworks ([NS07, BH00, LCM+05]) consist of a front-end and a back-end. The front-end is an API allowing to specify instrumentation code and the points at which it should be introduced at runtime. The back-end introduces instrumentation at the specified positions and provides all necessary information to the front-end.

There are two main approaches for controlling the monitored application: *emulation* and *just-in-time* (JIT) instrumentation. The emulation approach consists in executing the application on a *virtual machine* while the JIT approach consists in linking the instrumentation framework dynamically with the monitored application and inject instrumentation code at runtime.

Valgrind [NS07] is a representative framework applying the emulation approach. The analysed program is first translated into an intermediate representation (IR). This IR is architecture independent, which makes it more comfortable to write generic tools. The modified IR is then translated into binary code for the execution platform. Translating code to and from the IR is time consuming. The penalty in execution time is approximately four to five times (with respect to an un-instrumented execution).

Pin [LCM+05] is a widely used framework which gains momentum in analysing multi-threaded programs running on multi-core platforms. Pin and the analysed application are loaded together. Pin is responsible of intercepting the applications instructions and analysing or modifying them as described by the instrumentation code written in so-called pintools. Integration of Pin is almost transparent to the executed application.

The pintools use the frameworks front-end to control the application. Instrumentation can be easily added at various granularity levels from function call level down to processor instructions. An interface exists for accessing abstract instructions common to all architectures. If needed more architecture specific analyses can be implemented using specific APIs. In this case the analysis written is limited to executables of that specific architecture.

Adapting a DBI framework to parallel architectures is not straight forward. Hazelwood et al. [HLC09] point out the difficulties in implementing a framework that scales well in a parallel environment and present how they overcame them in the implementation of Pin. As mentioned in their article, extra care is taken to allow frequently accessed code or data to be updated by one thread without blocking the others. Despite all this effort in some cases the instrumenter will inevitably serialise the threads execution or preempt them.

4.2.2 Sequential taint analysis

All taint analyzes are decomposed into three distinct phases:(i) tainting (ii) tracing and (iii) asserting. The first two were presented earlier in section 4.1 and consist in defining what data are tainted and how taintness propagates. The third phase consists into checking how tainted data are used. The property to be asserted affects the first two phases too.

Often taint analyzes implemented with DBI [NS05, ZCYH05, CZYH06, QWL+06, ZJS+11, GLG12] focus on the same properties such as buffer overflows, format string attacks, stack smashing etc. and compare with each other in terms of precision and performance. The major overheads in these analyzes are caused by *instrumentation* and *updating shadow memory*.

Shadow memories are used to store information about taintness. A mapping exists between the registers and address space of the application to the shadow memory, as illustrated in Figure 4.2. For performance and memory usage optimization shadow memories are usually implemented as bitvectors. Each bit indicates whether the mapped memory is tainted or not. The granularity of the mapping may vary, but most often a bit corresponds to a byte of address space or register. Because the lookup and updating of shadow memory occurs practically for each instruction executed by a processor Nagarajan and Gupta [NG09] propose architectural support for their implementation. Their proposal

focuses on multiprocessors and thus incurs modifications both at the instruction set and at the cache coherency protocols.

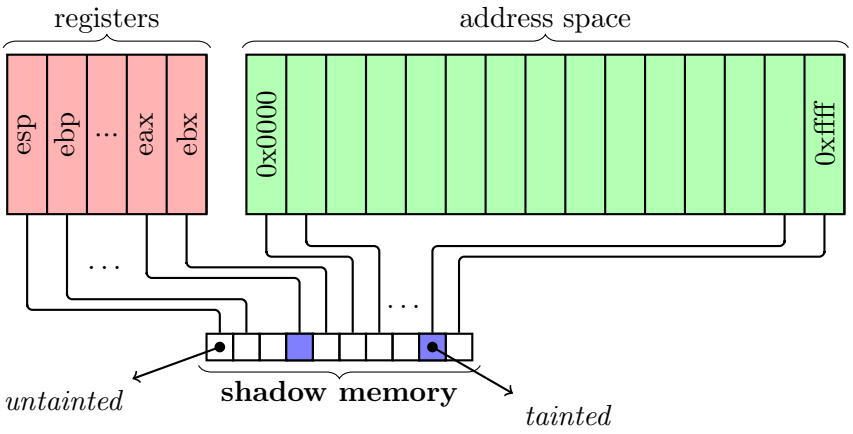


Figure 4.2: Shadow memory mapping.

Newsome and Dong proposed TaintCheck [NS05] for detecting exploits on commodity software and produce signatures for their early detection and avoidance. They used Valgrind [NS07] for instrumentation which penalizes the monitored execution due to its emulation approach. Their information flow tracing is limited to *move* (e.g., load, store, push, pop) and *arithmetic* (e.g., add, sub, xor) instructions. During arithmetic operations some special registers are updated called EFLAGS which are not taken into account. For shadow memory they map each byte of memory (register or address space) to a four-byte pointer, linked to either a tainted data structure [1] or to NULL depending on its taint status. Shadow memory is kept in a page-table like structure in order to reduce its size. The assert phase checks if tainted data are used in jump instructions or passed as arguments to system calls.

Further to taint propagation TaintCheck keeps a log allowing to trace the flow from the source that tainted it down to the exploit position. This is necessary for generating the signature of the attack. Moreover, once an exploit is detected the programs execution may continue under a constrained environment which allows to learn what is the goal of the attack. This information is useful for undoing, if possible, the damage done by a malware.

Zhao et al. present DOG [ZCYH05] a program monitoring framework built on top of Dynamorio [Bru04] for detecting exploits but also preventing confidentiality leaks. DOG provides a graphical interface from which one can define the taint sources, and associate to each source a propagation policy and a set of assertions and actions to perform if an exploit is detected. The propagation of taint supported is similar to TaintCheck [NS05] apart that they take into account the EFLAGS and allow for implicit propagation through control. Because implicit propagation can introduce many false positives DOG allows the user to specify regions in the program where it can be applied. To optimize taint tracing a bit-vector is used. Each byte corresponds to a bit with value 1 marking it as tainted and 0 as untainted. Moreover, they do not use a page-table based strategy as in TaintCheck but instead they devise a mapping where it suffices to add a *shadow_base* to the address to

[1] this is similar to the taint object T we defined, its a fixed point for taintness

locate its mapping. The taint checks provided by DOG are somehow typical, i.e., format string attacks, stack smashing, etc.

Dytan [CLO07] is yet another framework for taint analysis. It is implemented using Pin [LCM+05] and as DOG it supports both implicit and explicit flow propagation. In addition to a simple XML configuration the framework also provides an easily extendable interface allowing the rapid development of more elaborated taint analyzes. The taint information is also stored in bit-vectors and the granularity of memory mapped is one byte.

A most recent work TaintEraser [ZJS+11] focuses on confidentiality, it blocks unintended data exposure to the network or local file system by applications. TaintEraser makes several optimizations in taint analysis without losing in precision. First, it uses *function summaries* which resume the effects of a functions execution and thus there is no need to instrument it at runtime. Moreover, they perform on-demand instrumentation, i.e., they do not instrument the entire program execution. Finally, to enforce confidentiality of sensitive data it allows to log the leak, block the action or replace the sensitive data with random ones. The output channels protected are network connections and files.

All frameworks presented above use DBI to add monitoring. Figure 4.3 illustrates an overview of their mode of operation. The DBI framework observes the instructions executed by the processing unit and updates accordingly the shadow memory and takes action if needed. As presented in the figure, the execution of multithreaded programs is serialized. This is convenient for monitoring since the DBI frameworks observe a sequential schedule Σ_s (see Definition 2.7.4) which allows the shadow memory to be updated precisely.

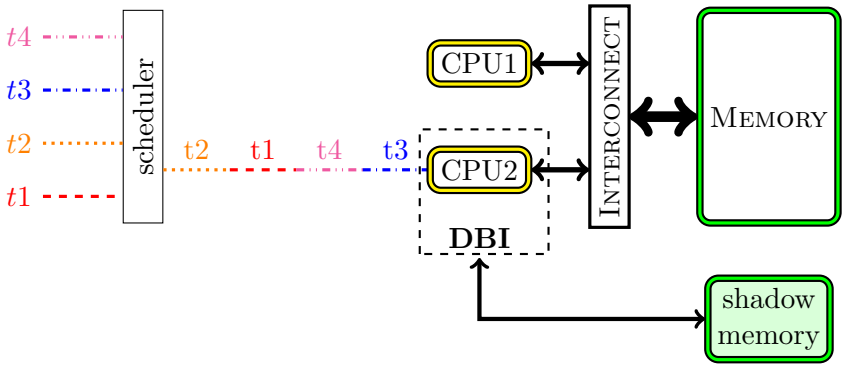


Figure 4.3: DIFT analysis using Dynamic Binary Instrumentation frameworks

Adding instrumentation at runtime incurs two drawbacks. First, it penalizes execution having to produce the instrumentation dynamically, at least for the first [1] time they get executed. Second, it is hard to apply any optimizations since high level program structure has been lost. Saxena et al. [SSP08] try to weaken these problems by proposing a binary rewriting technique for adding instrumentation. Prior to adding the monitoring code they extract as much high level information as possible from x86 executables, so that optimizations can be applied.

[1] instrumented code is stored into code caches for later use

4.2.3 Optimizing DIFT

A great challenge DBI frameworks are facing is dealing efficiently and correctly with multithreaded programs and their parallel execution achieved on the multicore platforms. As illustrated in Figure 4.3 existing DIFT analyses based DBI frameworks serialize their execution. Though necessary for tracing information flow precisely it incurs a great penalization of the execution time. To improve DIFT analyzes several solutions requiring the support of specialized hardware have been proposed.

Nagarajan et al. [NKWG08] takes advantage of multicore processors to perform DIFT transparently and efficiently. Their solution reposes on spawning a new thread dedicated to the analysis and running on a dedicated core in parallel with the main thread (the monitored application). The monitoring core tracks taintness and sends an interrupt to the main thread when the use of a tainted value violates the specified security policy. Intense communication between the cores executing main and monitor thread respectively is required. Initially shared memory was used for their communication but it added too much overhead to the execution. Thus, they proposed the usage of a dedicated hardware FIFO [1] buffer. Although this buffer does not exist in current multicore processors, it has been proposed by several other works [RVS+06, SKSP06]. For their experimentations they used the *Simics* full system simulator on which they implemented the hardware FIFO queue. The results they obtained showed a 48% overhead which is much better than aforementioned frameworks which introduced an overhead of about 300% and more.

The work of Ruwase et al. [RGM+08] reposes on the *log-based architecture* (LBA [CKS+08]) to implement a parallel dynamic information flow analysis. LBA introduces several hardware components in the CPU design that allow the extraction of a log trace for a monitored application. The log can be read by the monitoring thread. The analysis of the log happens in parallel. It is broke into segments each processed by a worker thread running on a dedicated core. The worker threads create summaries of segments and send them to the monitoring/master thread which updates meta-data and makes the appropriate checks. For the parallelized DIFT they proposed a big number of worker threads is necessary, but it does not always guarantee a speedup compared to sequential frameworks.

A most recent work by Ozsoy et al. proposes SIFT [OPAGS11] which takes advantage of symmetric multithreading (SMT). The implementation of their work though necessitates modifications which increase the size of the processor core by core by 4.5%. Although it is relatively small compared to the solution proposed in Raksha [DKK07] which increases chip size by 20%. Hardware-based DIFT solutions seem very appealing but necessitate non-trivial hardware modifications which make the design of processing units more complex. Thus chip manufacturers are not willing into adapting them.

[1] First In First Out or queue like storage and processing policy

4.3 Extending monitored traces

Dynamic information flow analysis performed either at software level, using a DBI framework, or with support of sophisticated hardware, allow the meticulous analysis of a single execution trace. That is, the verdict concerns only the specific execution which is often serialized. Such a solution is very intrusive and hides sources of concurrency bugs such as races caused by non-deterministic scheduling and effects of weak memory model relaxations.

Although monitoring of applications is useful itself, as long as the performance losses are acceptable, in some contexts such as debugging or testing, limiting the verdict to a single execution is too restrictive. To overcome this problem *runtime prediction* is used to expand the analysis by inferring executions. The inferred executions capture different interleavings for the executed application.

Depending on the accuracy of the interleaving computation the prediction may under-approximate (miss errors) or over-approximate (produce false positives). In the former case, the initial execution trace is usually captured as a totally ordered sequence of events which is relaxed pessimistically i.e., allowing only a subset of feasible interleavings. In the latter case, execution traces are conceived as unordered sets of events and interleavings are computed by enumerating all possible interleavings and then eliminating some unfeasible paths (e.g., based on happens before relations).

4.3.1 Runtime prediction for concurrency bugs

Runtime prediction has been widely used in the identification of concurrency bugs such as race conditions and deadlocks. To perform such analysis the frameworks proposed in the literature [JNPS09, SFM10, WG12] abstract executions by logging information necessary to discover interleavings susceptible to cause concurrency bugs. The logs consist of shared memory accesses and various synchronization primitives such as lock acquisitions and releases, thread creation and join etc. The frameworks are differentiated by the logged information, the algorithms detecting interleavings and either they over or under approximate. The algorithms used can be split into *enumerative* and *symbolic*. In the former case all interleavings are enumerated and then bogus ones are filtered, while in the latter case constraints on interleavings are encoded into logic formulas fed to SMT [1] solvers [DM06].

Several works use enumerative algorithms. Some of them [WS06b, WS06a] over-approximate since they solely rely on the algorithms inferring the interleavings. To reduce false positives CalFuzzer [JNPS09] and PENELOPE [SFM10] try to infer a schedule capable to exhibit the concurrency bug. The inferred schedule is executed and if the bug occurs then it is reported by the framework, else it is dropped.

In the symbolic category Wang et al. [WG12] provide a detailed survey. Moreover they briefly present their contribution in the domain. First, they mention a theoretical optimal solution they proposed, the CTP [WCGY09] (Concurrent Trace Program), which captures

[1] Satisfiability Modulo Theories

all interleavings that can possibly be inferred from a single trace, without introducing any bogus interleavings. Subsequently they present two abstractions UCG [KW10] (Universal Causality Graph) which over-approximates and a dual work, TSA [SMWG11] which under-approximates the set of traces computed in TCP.

4.3.2 Runtime prediction applied to information flow

In the context of information flow *runtime prediction* has not yet been widely used. We present hereafter two recent analyses: DTAM [GLG12] and Butterfly [GVC+10].

In their work Ganai et al. [GLG12] propose DTAM analysis which identifies a subset of tainted input sources and shared objects that can affect the execution of a multithreaded program. That is, the tainted data have an impact on the control-flow of the program or its shared state. The tainted data get classified according to six relevancy types that describe how they tainted data can affect the program execution. To infer the information flow dependencies DTAM proposes a serial variation DTAM_{serial} and two parallel ones $\text{DTAM}_{parallel}$ and DTAM_{hybrid}.

DTAM_{serial} monitors the serialized execution of the multithreaded program and keeps track of taintness as in usual DIFT analyses. In the parallel variations each thread performs thread-local taint propagation. The information flow between threads is taken into account during the offline phase. For the offline phase relevant information needs to be logged. Each thread logs shared memory accesses along with the runtime taint value they have computed. Some basic synchronization primitives are also kept into the log such as fork/joins and wait/notifies allowing to infer happens before relations. For their relevancy analysis even conditionals are logged.

The difference between the $\text{DTAM}_{parallel}$ and DTAM_{hybrid} is that $\text{DTAM}_{parallel}$ does not take into account happens before relations and thus the results are less accurate. Else, the inter-thread propagation in both cases is rather coarse. Once a shared memory location is tainted in a thread, it remains indefinitely and propagates to all other threads. Such an approach drastically over-taints and it can result into considering everything as tainted.

The main objective of Goodstein et al. [GVC+10] is to provide a *lifeguard* mechanism for (multi-threaded) applications running on multi-core architectures. It is a runtime enforcement technique, which consists in monitoring a running application to raise an alarm (or interrupt the execution) when an error occurs (e.g., writing to an unallocated memory). The main difficulty is to make the lifeguard reasoning about the *set* of parallel executions. To solve this issue, the authors considered (monitored) executions produced on specific machine architectures [CKS+08] on which *heartbeats* can be sent regularly as *synchronization barriers*, to each core. This execution model can be captured by a notion of *uncertainty epochs*, corresponding to code fragments such that a *strict happens-before* execution relation holds between non-adjacent epochs. These assumptions allow to define a conservative data-flow analysis, based on sliding window principle, taking into account a superset of the interleaving that could occur in three consecutive epochs. The result of this analysis is then used to feed the lifeguard monitor. This approach can be used to check various properties like use-after-free errors or unexpected tainted variable propagation.

4.3.3 Positioning of our work

Our work is inspired from Butterfly analysis [GVC+10] though the objectives are not the same. Our intention is to provide some *verdict* to be used in a property oriented test-based validation technique for multi-core architectures. As such, our solution does not need to be necessarily conservative: false negatives are not a critical issue. A consequence is that we do not require any specific architecture (nor heartbeat mechanism) at execution time. Another main distinction is that we may proceed in a *post-mortem* approach: we first produce log files which record information produced at runtime, then this information is analyzed to provide various test verdicts (depending on the property under test).This makes the analysis more flexible by decoupling the execution part and the property checking part. From a more technical point of view, we also introduced some differences in the data-flow analysis itself. In particular we considered a sliding window of two epochs (instead of three). From our point of view, this makes the algorithms simpler, without sacrificing efficiency. Finally, a further contribution is that we take into account lock-set information to reduce the number of false positives.

4.4 Predictive explicit taint analysis

Hereafter we present our approach to *predictive explicit taint analysis* of multithreaded programs. The motivation is to use it for test validation, that is extend the results of a tested parallel execution to the set of plausible serializations that could have occurred. Since we are in a testing context our predictions do not need to be sound (taint value can be over or under approximated). For a test to be representative of a concrete execution the monitoring should be as transparent as possible. As presented in section 4.2.2 most works force the serialization of multithreaded applications, which is very intrusive. We do not impose such restrictions to the scheduling, i.e., we allow the parallel execution of the application, and reason a posteriori about taint propagations.

4.4.1 Overview of our approach

In our approach, an abstract view of which is presented in Figure 4.4, we propose an *offline sliding window-based analysis*. First, the multithreaded application is executed and a parallel schedule $\Sigma_{\parallel}$ (see Definition 2.7.5) is captured in the form of log files. A log file is recorded per executed thread containing the timestamped sequence of events produced by the thread mapped to it (upper part of Figure 4.4). Next, the log files are sliced into so called *epochs* and the sliding window-based taint analysis is applied (lower part of Figure 4.4).

Due to the information flow property we are interested in (taint propagation), the logging of events is exhaustive. Typically all memory accesses, affecting both *shared* and *thread-local* variables, in the form of *use/def* relations and some synchronization events. We remind the T and U notations introduced in section 4.1 where T is a fixed tainted variable and dually U an untainted one. For an event e we introduce the following functions:

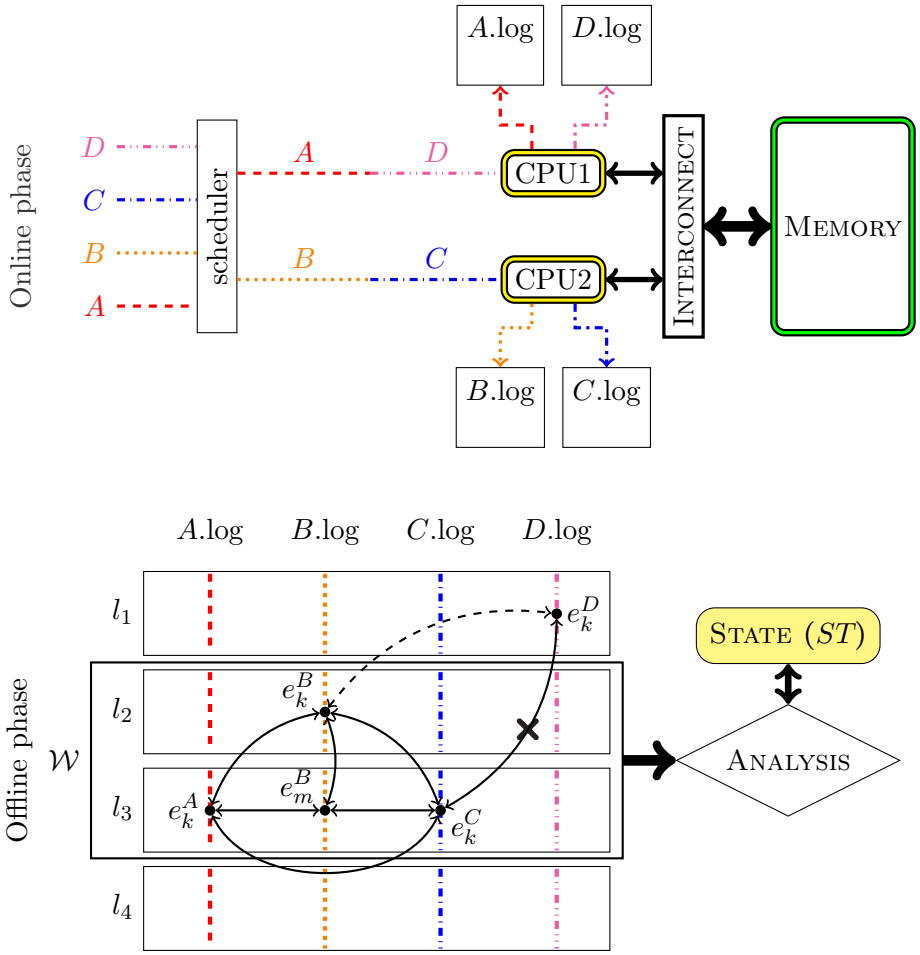


Figure 4.4: Overview of our approach

$Def(e)$	returns the singleton set of variables *defined* i.e., written by event e
$Used(e)$	returns the set of variables *used* i.e., read by event e

Computing all interleavings (i.e., all serializations) of logged events is impractical. To avoid the interleaving explosion problem we propose (i) the slicing of logs into epochs (l) which limits the number of events to interleave and (ii) a processing algorithm which infers taint propagation without enumerating all serializations. Because the slicing can occur between arbitrary events there is no guarantee that a happens before relation is established for events belonging to consecutive epochs. Thus, we extend the bounding of interleavings to events belonging in a window consisting of two adjacent epochs. Section 4.4.2 provides more details on slicing.

The lower part of Figure 4.4 illustrates a window consisting of two consecutive epochs ($\mathcal{W}$=$\{l_2$,$l_3\}$) and the considered interleavings of events e_k^A, e_k^B, e_m^B, e_k^C in it. We note that also events belonging to the same thread can be interleaved. This allows to reason on taint propagation under relaxed memory models. Moreover the figure presents how slicing bounds the prediction to events belonging to adjacent epochs only. For instance, the interleaving between events e_k^B and e_k^D denoted with a dashed line is considered in the preceding window consisting of epochs l_1, l_2. On the contrary the interleaving between

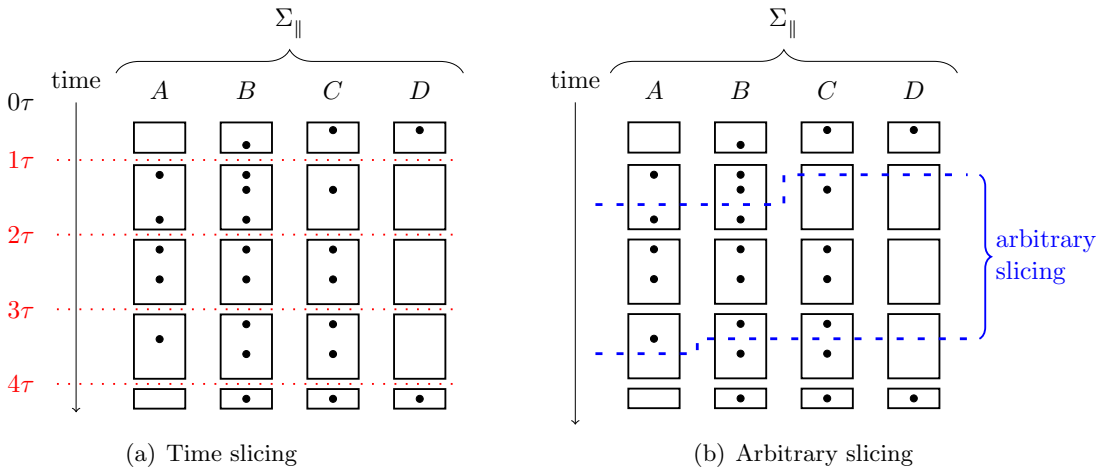
e_k^C and e_k^D is crossed out because it will not be considered by the analysis since the events do not belong to adjacent epochs. Similarly, the interleaving of events e_k^A and e_m^B with e_k^D are not taken into account.

We apply our analysis using a sliding window consisting of two adjacent epochs. The window slides over epochs thus all interleavings of an event with events in its preceding and succeeding epochs are explored. The analysis identifies taint propagations inside the currently analyzed window $\mathcal{W}$ and summarizes their effect in state ST, which acts as a *shadow memory*.

4.4.2 Slicing the parallel schedule $\Sigma_{\parallel}$ (log files)

The slicing of log files into epochs affects the prediction since it defines which events can be interleaved. The slicing technique we use is *time-based*, that is we define a time period τ which slices the logs as illustrated in Figure 4.5(a). The time slicing is well adapted for our purpose because it allows the analysis to consider interleavings of events that were executed simultaneously, or at least in parallel with respect to the chosen period τ.

Choosing the value of τ is delicate. In principle it should be large enough to capture (i) the delta between the execution of an event and the assignment of the timestamp and (ii) the effects of the platform on the ordering of the executed instructions (weak memory models). Taking into account criterion (ii) is meaningful only when the logging of events is at the assembly level and the timestamping utterly precise. We note that, by setting a large value for τ the analysis may infer taint propagations caused by different schedules. Dually, choosing a small value will under-approximate taint propagation, and thus the analysis will not infer taintness for all feasible serializations under the observed schedule. Finally, if the entire execution log is split into just two epochs (i.e., one window) then the analysis reasons about all possible executions of the program.



(a) Time slicing (b) Arbitrary slicing

Figure 4.5: Slicing $\Sigma_{\parallel}$ into epochs

In general, the slicing can be performed arbitrarily. Figure 4.5(b) illustrates such an arbitrary slicing delimited by thick loosely dashed lines. Some heuristics that can produce interesting slices are to use context switches or synchronization barriers as the slicing

points. In the case of synchronization barriers for instance, slicing at these points is not sufficient. A dummy epoch should be introduced such that it forces the analysis not to reason about the interleavings. The dummy epoch should define empty blocks for the threads concerned by the synchronization.

We introduce hereafter some key notations that we use in the sequel. As mentioned previously the slicing of log files defines epochs as illustrated in Figure 4.6. The events of a thread belonging to an epoch form a *block*. Each block is uniquely identified by a tuple (l, t) where l is the epoch it resides in and t is the thread identifier. Events within a block are uniquely identified by a triplet (l, t, i) where l, t specify the block it belongs to, and i is the identifier of the event. As illustrated in Figure 4.6 event $e_i^B = (l, B, i)$.

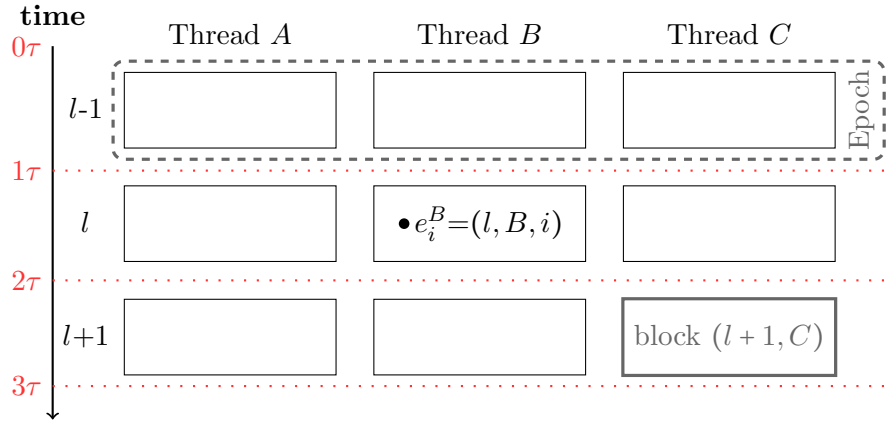


Figure 4.6: Basic notations

We further define the functions $Thr(e)$ and $Epoch(e)$ which return respectively the thread that executed event e and the epoch it belongs to. Finally, we introduce a binary reflexive operator $\nleftrightarrow$ which denotes two events can be interleaved based on the window interleaving assumption. More formally:

$$e_k \nleftrightarrow e_m \Rightarrow |Epoch(e_k) - Epoch(e_m)| \leq 1$$

4.5 Sliding window-based explicit taint prediction

As mentioned earlier what we propose is an *offline sliding window-based analysis* for predicting explicit taint propagation. There are two aspects in our analysis:

(i) prediction of explicit taint propagation within a window, and summarization of its effects;

(ii) reasoning correctly about the sliding windows which causes them to overlap.

We introduce hereafter the notations used in our sliding window analysis. Figure 4.7 illustrates two consecutive windows $\mathcal{W}' = \{l_h, l_b\}$ and $\mathcal{W} = \{l_b, l_t\}$ where $\mathcal{W}$ is the *currently*

analyzed window consisting of epochs labeled l_b, l_t while $\mathcal{W}'$ is the preceding window consisting of epochs labeled l_h, l_b. The labeling of epochs is relative to the currently analyzed window and is adopted from [GVC+10]. The upper epoch of the currently analyzed window ($\mathcal{W}$ in Figure 4.7) is called *body* (l_b) while the lower one *tail* (l_t), finally the epoch preceding body is called *head* (l_h). We remind that $ST_{\mathcal{W}'}$ summarizes the taint predictions down to the indexed window ($\mathcal{W}'$).

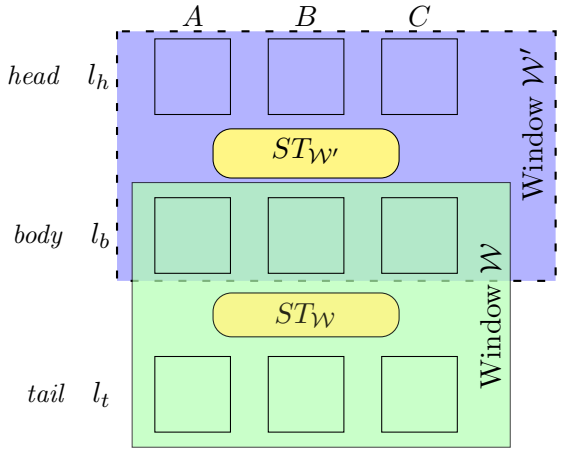


Figure 4.7: Sliding window based analysis

Prior to abording explicit taint prediction of a window we define explicit tainting and un-tainting of a variable. Next, we provide a formal definition of taintness on a serialized execution of events. Finally, we adapt this definition to fit a serialization of events in our window-based taint prediction.

We use the oracle $isTainted(x)$ which asserts if a variable x is tainted or not. It allows us to define explicit tainting/generation ($gen(e)$) and respectively un-tainting/killing ($kill(e)$) of the variable defined by an event e:

$$gen(e) = \begin{cases} \{Def(e)\} & \text{if } \exists x \in Used(e) \quad \text{s.t. } isTainted(x) \\ \{\varnothing\} & \text{if } \nexists x \in Used(e) \quad \text{s.t. } isTainted(x) \end{cases}$$

$$kill(e) = \begin{cases} \{Def(e)\} & \text{iff } \nexists x \in Used(e) \quad \text{s.t. } isTainted(x) \\ \{\varnothing\} & \text{if } \exists x \in Used(e) \quad \text{s.t. } isTainted(x) \end{cases}$$

Taint propagation occurs through a series of taintings over a serialization. That is, there is a tainting source that causes variables to be tainted. Moreover, a tainted variable remains so until some event un-taints it. We provide hereafter the definition of taintness for a variable x at an event e of a serialization σ.

Definition 4.5.1 (Taintness on a serialized execution: $taint(\sigma, x, e_k)$)
Let σ be a valid serialization of the analyzed application, x a variable, and e_k an event in σ. We (recursively) define predicate $taint(\sigma, e_k, x)$, meaning that variable x is tainted at event e_k on σ. Note that event indexes correspond to the position/order of events on the serialization.

$$taint(\sigma, e_k, x) \equiv \begin{cases} x = T \quad \bigvee \\ \exists m \leq k \text{ such that: } Def(e_m) = x \quad \wedge \\ \qquad\qquad\qquad\qquad \exists y \in Used(e_m) \,.\, taint(\sigma, e_m, y) \quad \wedge \\ \qquad\qquad\qquad\qquad \forall n \,.\, m < n < k \Rightarrow Def(e_n) \neq x \end{cases}$$

Intuitively, a variable `x` is tainted at event e_k of σ if it was assigned with a tainted variable at a preceding event e_m (or at the event e_k itself), and never re-assigned in between. Figure 4.8 illustrates the definition of $taint(\sigma, e_k, \texttt{x})$. The serialization is not complete, since the occurrence of events post e_k do not affect the taint value of `x` at event e_k. Applying the taint predicate recursively, tracks back to the initial taint source which is always variable T. We remind that T is a constantly tainted variable which abstracts all taint sources.

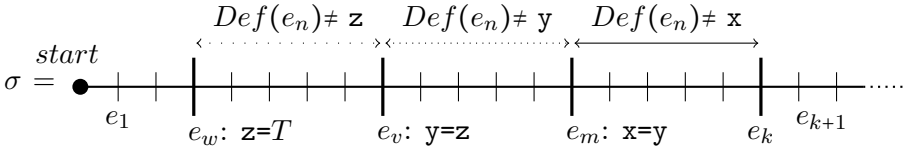


Figure 4.8: Taint definition for a concrete serialization

We adapt the taint definition above to our window-based prediction analysis. Note that, the definition is applied on a valid serialization $\sigma^i_{\mathcal{W}}$ of events in $\mathcal{W}$. Since the serialization is bounded to events belonging to the window $\mathcal{W}$, recursively applying the taint definition may not be able to reach the constantly tainted variable T. Thus, the taint definition must rely on summarizations of taint predictions, that is it suffices to reach a variable in $ST_{\mathcal{W}'}$. We provide the definition of *window-based taintness* ($taint\mathcal{W}(\sigma^i_{\mathcal{W}}, e_k, x)$) on a serialization $\sigma^i_{\mathcal{W}}$ of events in window $\mathcal{W}$.

Definition 4.5.2 (Taintness on a serialization of a window $\mathcal{W}$)
Let $\sigma^i_{\mathcal{W}}$ be a valid serialization of the currently analyzed window $\mathcal{W}$, x a variable, and e_k an event in $\sigma^i_{\mathcal{W}}$. We (recursively) define predicate $taint\mathcal{W}(\sigma^i_{\mathcal{W}}, e_k, x)$, meaning that variable x is tainted at event e_k on $\sigma^i_{\mathcal{W}}$.

$$taint\mathcal{W}(\sigma^i_{\mathcal{W}}, e_k, x) \equiv \begin{cases} x = T \quad \bigvee \\ x \in ST_{\mathcal{W}'} \wedge \nexists j < k \text{ such that: } Def(e_j) = x \quad \bigvee \\ \exists j \leq k \text{ such that: } Def(e_j) = x \quad \wedge \\ \qquad\qquad\qquad\qquad \exists y \in Used(e_j) \,.\, taint\mathcal{W}(\sigma^i_{\mathcal{W}}, e_j, y) \quad \wedge \\ \qquad\qquad\qquad\qquad \forall m \,.\, j < m < k \Rightarrow Def(e_m) \neq x \end{cases}$$

Figure 4.9 illustrates the application of definition $taint\mathcal{W}(\sigma^i_{\mathcal{W}}, e_k, x)$ on an example. The events preceding $\mathcal{W}$ are abstracted in the dotted path and the predictions of their plausible serializations, with respect to a given slicing and the application of window-based taint prediction to it down to $\mathcal{W}'$, are summarized in $ST_{\mathcal{W}'}$. Applying the definition for

variable x at event e_k we obtain the recursive calls of events pointed by the arrows initiated from e_k. The recursion ends in the summary of the preceding window $ST_{\mathcal{W}'}$.

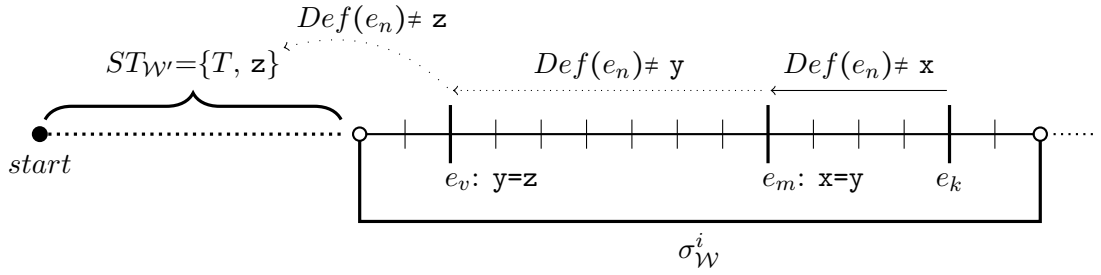


Figure 4.9: Taint definition for a plausible serialization of events in a window $\mathcal{W}$

4.6 Iterative explicit taint prediction in a window

To introduce explicit taint prediction of a window $\mathcal{W}$ we consider the simple case where:

(i) events in $\mathcal{W}$ can arbitrarily interleave, even those produced by the same thread. That is, any serialization $\sigma^i_{\mathcal{W}}$ of events is considered valid (no memory model restrictions). In section 4.7 we illustrate how to enforce sequential consistency.

(ii) events that kill/un-taint variables are ignored. This assumption simplifies propagation of taintness and is often used e.g., [GLG12]. This assumption will be raised in the case of sequentially consistent serializations in section 4.7.2.

We provide hereafter the definition of *relaxed taintness* which introduces the ignoring of killing/un-tainting variables:

Definition 4.6.1 (Relaxed taintness on a serialization of a window $\mathcal{W}$)
Let $\sigma^i_{\mathcal{W}}$ be an arbitrary serialization of the currently analyzed window $\mathcal{W}$ ($\sigma^i_{\mathcal{W}} = \{(e_1, \ldots, e_n) \mid \forall k < m \Rightarrow e_k \nrightarrow e_m\}$), x a variable, and e_k an event in $\sigma^i_{\mathcal{W}}$. We (recursively) define predicate $taint\mathcal{R}(\sigma^i_{\mathcal{W}}, e_k, x)$, meaning that variable x is tainted at event e_k on $\sigma^i_{\mathcal{W}}$.

$$taint\mathcal{R}(\sigma^i_{\mathcal{W}}, e_k, x) \equiv \begin{cases} x = T \quad \bigvee \quad x \in ST_{\mathcal{W}'} \quad \bigvee \\ \exists j \leq k \text{ such that: } Def(e_j) = x \quad \wedge \\ \qquad\qquad\qquad \exists y \in Used(e_j) \,.\, taint\mathcal{R}(\sigma^i_{\mathcal{W}}, e_j, y) \end{cases}$$

Figure 4.10 presents the definition of relaxed taintness propagation in a window. As illustrated, the events killing variables are ignored i.e., the condition of not redefining a variable between its tainting and its usage to taint some other variable has been removed. For example, the effect of event e_w (which is crossed out) is ignored, thus it does not prevent the taint propagation at event e_m.

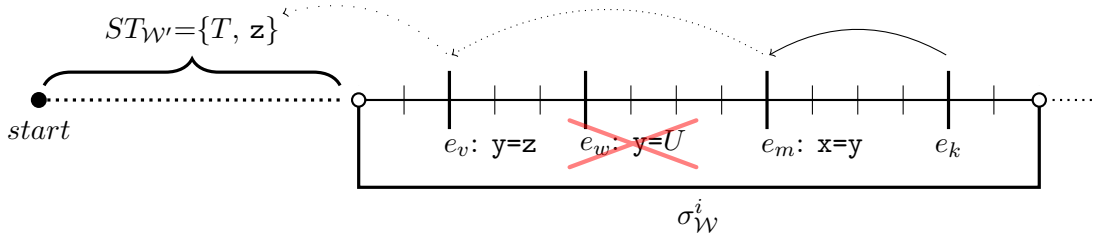


Figure 4.10: Relaxed taint definition for a plausible serialization of events in a window $\mathcal{W}$

4.6.1 Enumerative approach

A natural way of predicting taint propagation in a window $\mathcal{W}$ is to compute all serializations $\sigma^i_{\mathcal{W}}$ by enumerating all permutations of events in it. Given the cardinality (number of events) of $\mathcal{W}$ denoted as $|\mathcal{W}|$, there will be $|\mathcal{W}|!$ such serializations, since all interleavings are feasible. A sequential taint analysis should then be applied on each serialization $\sigma^i_{\mathcal{W}}$, with $i \in [1, |\mathcal{W}|]$, using as initial state $ST_{\mathcal{W}'}$ and producing a local state $ST^i_{\mathcal{W}}$ which predicts *relaxed taint propagation* for the analyzed serialization. That is, for each variable x in $ST^i_{\mathcal{W}}$ the predicate $taint\mathcal{R}(\sigma^i_{\mathcal{W}}, last(\sigma^i_{\mathcal{W}}), x)$ holds, and likewise. By $last(\sigma^i_{\mathcal{W}})$ we denote the last event of serialization $\sigma^i_{\mathcal{W}}$.

$$x \in ST^i_{\mathcal{W}} \Leftrightarrow taint\mathcal{R}(\sigma^i_{\mathcal{W}}, last(\sigma^i_{\mathcal{W}}), x)$$

We present in Algorithm 1 how each plausible serialization of window $\mathcal{W}$ is analyzed. First, a copy of taint predictions down to the preceding window ($ST_{\mathcal{W}'}$) is made. The copy is updated locally such that when a variable gets tainted it is added to the state. Dually, when a variable is untainted no action is taken since these events are ignored.

Algorithm 1 Relaxed taint analysis of a serialization (kills are ignored)

In: $\sigma^i_{\mathcal{W}}$, $ST_{\mathcal{W}'}$
1: $ST^i_{\mathcal{W}} \leftarrow ST_{\mathcal{W}'}$
2: **for all** $e \in \sigma^i_{\mathcal{W}}$ **do**
3: **if** $Used(e) \cap ST^i_{\mathcal{W}} \neq \varnothing$ **then**
4: $ST^i_{\mathcal{W}} \leftarrow ST^i_{\mathcal{W}} \cup Def(e)$
5: **end if**
6: **end for**
Out: $ST^i_{\mathcal{W}}$

The analysis of each serialization $\sigma^i_{\mathcal{W}}$ with Algorithm 1 computes its *relaxed taintness* into $ST^i_{\mathcal{W}}$ which contains the set of variables that can be tainted by $\sigma^i_{\mathcal{W}}$, without taking un-taintings into account. To *summarize* the predictions of all serializations in $\mathcal{W}$, i.e., to compute $ST_{\mathcal{W}}$, it suffices to take the union of all local predictions. Figure 4.11 illustrates the enumerative approach.

$$ST_{\mathcal{W}} = \bigcup_{i \in [1, |\mathcal{W}|!]} ST^i_{\mathcal{W}}$$

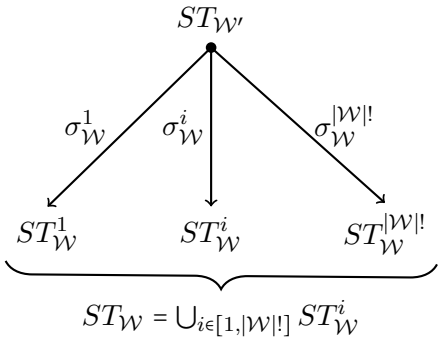


Figure 4.11: Enumerative prediction of taint propagation

4.6.2 Iterative approach

The enumerative approach presented above has an exponential complexity ($|\mathcal{W}|!$ serializations to process) which makes it impractical. We present here an iterative algorithm with linear complexity for predicting relaxed taintness propagation. Our algorithm iterates over an arbitrary serialization of events in a window at most $|\mathcal{W}|$ times and infers $ST_{\mathcal{W}}$. We justify the correctness of our solution by showing taint prediction is equivalent to solving a *boolean equation system* (BES). Note that the transformations we present hereafter are only used to illustrate the correctness of the solution and never occur in the analysis. In appendix A on page 131 we provide basic notations and definitions for boolean equation systems.

Equivalence to boolean equation systems

Taintness is a binary value that characterizes a variable as either tainted (*true*, $\top$) or untainted (*false*, $\bot$). Thus, taint propagation can be expressed in terms of *boolean equations*. As mentioned earlier, taintness is explicitly propagated to a variable if there exists a tainted variable among those used to define it. By associating to each variable x in the logs a boolean shadow variable denoted as $\widehat{x}$, we can transform an event e_k into an equivalent boolean equation as follows:

$$e_k \equiv \widehat{Def}(e_k) = \bigvee_{x \in Used(e_k)} \widehat{x}$$

Figure 4.12 illustrates how events in a block can be transformed into an equivalent boolean equation system. The first step transforms each event into an equivalent boolean equation as detailed above. After this first transformation we might have several boolean equations defining the same variable. To obtain a boolean equation system for the block we must eliminate all duplicate definitions. We achieve this by taking the disjunction of all equations defining the same variable. This merging of boolean equations is valid with respect to $taint\mathcal{R}$. Recall that according to $taint\mathcal{R}$ a variable x is tainted if there exists an event that assigns it a tainted value.

The boolean equation system $\mathcal{E}$ we obtain by the above transformation of events consists

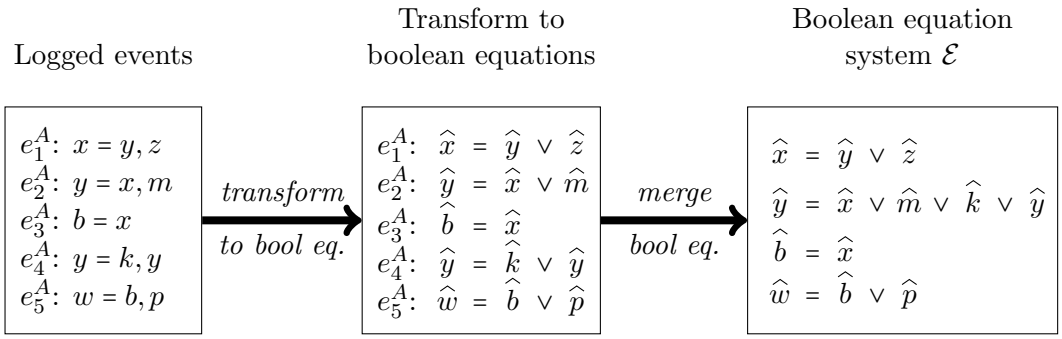
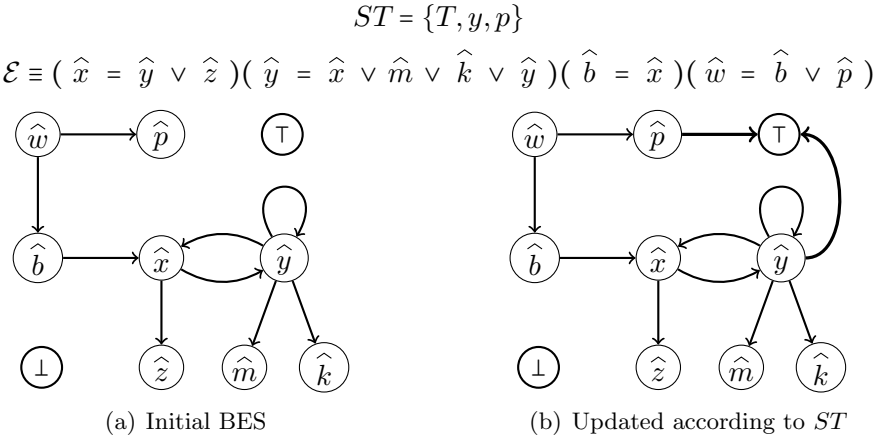


Figure 4.12: Obtaining boolean equation system.

of *disjunctive* boolean equations. Such boolean equation systems are often represented as directed graphs $G_{\mathcal{E}} = (V, E)$, where $V = \{ \widehat{x} \mid \widehat{x} \in \mathcal{E}\} \cup \{\top, \bot\}$ is the set of vertices and E is the set of directed edges, representing dependency between variables. Figure 4.13(a) illustrates the dependency graph for the BES $\mathcal{E}$ obtained from the block in Figure 4.12.



(a) Initial BES (b) Updated according to ST

Figure 4.13: Variable dependency graph of disjunctive boolean equation system.

For each variable $\widehat{x}$ defined in the boolean equation system $\mathcal{E}$ we show that if there exists a solution that makes it *true* then there also exists a serialization σ of logged events such that $taint\mathcal{R}(\sigma, last(\sigma), x)$ holds and conversely.

Finding a solution that assigns a variable $\widehat{x}$ of $\mathcal{E}$ with *true* is equivalent to identifying a path in the dependency graph of the BES that leads from vertex mapped to $\widehat{x}$ to the *true* vertex. Before searching for such a solution we must update the dependency graph such that there exists an edge connecting each variable in the ST with the *true* vertex. This update introduces the information about tainted/*true* variables in the BES. Figure 4.13(b) illustrates the updated dependency graph with respect to $ST = \{T, y, p\}$. We can now easily identify which variables can reach the *true* vertex. For instance $\widehat{b}$ is assigned a true value through the path ($\widehat{x}$, $\widehat{y}$,$\top$).

We argue now why the existence of a path that propagates *true* value to a variable $\widehat{x}$ implies the existence of a serialization that propagates taintness. The path in the dependency graph defines in which order the equations should be applied such that *true* value reaches the desired equation defining $\widehat{x}$. Under the current assumptions (completely

relaxed memory model and not taking untaintings into account) events can be arbitrarily re-ordered. Thus, we can produce a serialization where the ordering of events matches the order imposed on boolean equations. That is, we group all events defining a variable and subsequently order these groups such that they match the ordering of boolean equations. The events defining variables for which the path does not precise an ordering can be placed arbitrarily.

Figure 4.14 illustrates a plausible serialization that propagates taintness to b with respect to $taint\mathcal{R}$. On the left side of the figure we have the set of boolean equations that correspond to the block on Figure 4.12. In the middle we re-order the boolean equations such that *true* value can reach variable $\widehat{b}$. Finally, on the right side we exhibit a serialization that taints b under $taint\mathcal{R}$.

$$ST = \{T, y, p\}$$

$$\mathcal{E}_{\mathcal{W}} \equiv (\ \widehat{x} = \widehat{y} \vee \widehat{z}\ )(\ \widehat{y} = \widehat{x} \vee \widehat{m} \vee \widehat{k} \vee \widehat{y}\ )(\ \widehat{b} = \widehat{x}\ )(\ \widehat{w} = \widehat{b} \vee \widehat{p}\ )$$

Boolean equation system	Re-order to propagate $\top$ to $\widehat{b}$	Serialization propagating taintess to b
$\widehat{x} = \widehat{y} \vee \widehat{z}$ $\widehat{y} = \widehat{x} \vee \widehat{m} \vee \widehat{k} \vee \widehat{y}$ $\widehat{b} = \widehat{x}$ $\widehat{w} = \widehat{b} \vee \widehat{p}$	$\widehat{y} = \widehat{x} \vee \widehat{m} \vee \widehat{k} \vee \widehat{y}$ $\widehat{x} = \widehat{y} \vee \widehat{z}$ $\widehat{b} = \widehat{x}$ $\widehat{w} = \widehat{b} \vee \widehat{p}$	e_4^A: $y = k, y$ e_2^A: $y = x, m$ e_1^A: $x = y, z$ e_3^A: $b = x$ e_5^A: $w = b, p$

Figure 4.14: Equivalence between path in dependency graph and tainting serialization

Although there are many efficient algorithms for solving disjunctive BES our taint analysis is based on the iterative one. Note that, as explained in the following section, we do not iterate over the BES itself but directly on the logged events.

Iterative algorithm

As argued above, we can iterate over a block to obtain the predictions of *relaxed taintness*. This can be generalized to a window where we iterate on on an arbitrary serialization $\sigma^{it}_{\mathcal{W}}$. We choose this serialization to be defined as the concatenation of blocks in the window respecting program order. We concatenate first blocks in l_b followed by those in l_t. Figure 4.15 illustrates the serialization of events that is iterated. The iteration is divided into two phases: (i) *horizontal* and (ii) *vertical*. The *horizontal* phase makes a pass over events in an epoch by crossing blocks left to right. The *vertical* phase iteratively initiates horizontal passes over the *body* and *tail* epochs successively.

Algorithm 2 presents the *vertical* phase, which iterates over the serialization $\sigma^{it}_{\mathcal{W}}$ of events in $\mathcal{W}$. The phases of the algorithm are also illustrated on the left side of Figure 4.15. The algorithm for horizontal processing of an epoch is provided in Algorithm 3. The horizontal algorithm applies a *transfer* function on each block. Here the transfer function is equivalent to Algorithm 1 where the serialization processed is the blocks events in program order (i.e.,

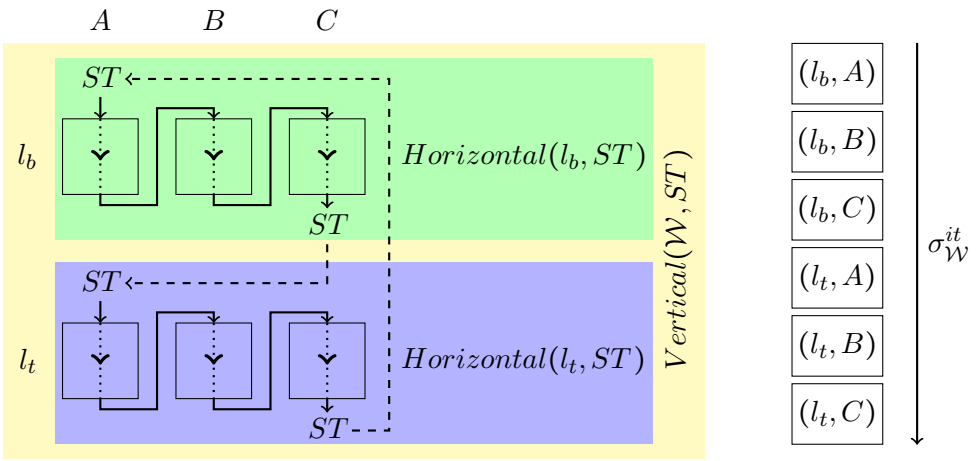


Figure 4.15: Iterating over the window

as they appear in the log). We must note that the *transfer* function should be monotonic on ST, that is it either adds or removes elements from it. This is required required to terminate the iteration of *vertical* algorithm.

Algorithm 2 Vertical processing $Vertical(\mathcal{W}, ST_{\mathcal{W}'})$

In: $\mathcal{W} = \{l_b, l_t\}, ST_{\mathcal{W}'}$
1: $ST \leftarrow ST_{\mathcal{W}'}$
2: **repeat**
3: $\quad ST \leftarrow Horizontal(l_b, ST)$
4: $\quad ST \leftarrow Horizontal(l_t, ST)$
5: **until** (ST **unmodified**)
6: $ST_{\mathcal{W}} \leftarrow ST$
Out: $ST_{\mathcal{W}}$

Algorithm 3 Horizontal processing of epoch $Horizontal(l, ST)$

In: l, ST
1: **for all** block $bl \in l$ **do**
2: $\quad ST \leftarrow Transfer(bl, ST)$
3: **end for**
Out: ST

We covered so far the aspect of predicting a relaxed form of explicit taint analysis within a window. We provided an intuitive enumerative method and an equivalent iterative. In subsection 4.6.3 that follows we abord the sliding window phase of the analysis.

4.6.3 Sliding windows - overlapping

As mentioned earlier the slicing of log-files into epochs limits the interleaving of events to be taken into account. Due to the arbitrary slicing we extend the interleaving of events to adjacent epochs. The explicit taint prediction analysis is applied on a sliding window consisting of two epochs. As illustrated in Figure 4.7 the sliding window allows each event

to be interleaved with events in its preceding and succeeding epochs. In window $\mathcal{W}'$ events in epoch l_b are interleaved with events in l_h, while in window $\mathcal{W}$ with events in l_t.

The disjoint processing of interleavings for events in an epoch can affect the approximation of the predictions. That is, they can either over or under approximate explicit taint propagations. These issues are not observable for the *relaxed taintness* where no killing/untainting of variables occurs. We will focus on the effect of sliding window for the case of taint prediction under sequential consistency in section 4.7.3.

4.7 Iterative explicit taint propagation under sequential consistency

The iterative prediction is a comfortable and efficient way of predicting taint propagation when any serialization of events is valid and kills are not taken into account. In this section we present how to adapt the iterative algorithm such as *explicit taint* propagation under *sequential consistency* is predicted within a window. Briefly, we must filter out taint propagation that is caused by non sequentially consistent serializations of events. Initially, we maintain the assumption that killing variables is not affecting taint propagation (i.e., they are ignored). We recall from section 2.6.1 on page 21 that *sequential consistency* enforces (i) program order and (ii) write atomicity. Write atomicity is meaningful only for parallel executions, thus it does not affect reasoning on serializations as is the case.

4.7.1 Respecting program order without kills

The taint property we are interested in is *relaxed taintness* propagation (see Definition 4.6.1 on page 79) applied to *sequentially consistent serializations*. We remind the precedence binary operator $\blacktriangleleft$ which defines ordering of events for a single thread (order in which events of a thread were logged). Furthermore, we introduce a more general binary operator $\lhd$ which denotes precedence between events produced by any thread and respecting the window interleaving assumption. We remind the more detailed notation of events for a given slicing $e_i^t \equiv (l,t,i)$, where $Epoch(e_i^t) = l$. The operators are formalized as follows:

$$(l,t,i) \lhd (l',t',j) \equiv \left(t = t' \wedge \left(l < l' \vee (l = l' \wedge i < j) \right) \right) \vee \left(t \neq t' \wedge l' \geq l - 1 \right)$$

$$(l,t,i) \blacktriangleleft (l',t',j) \equiv t = t' \text{ and } (l,t,i) \lhd (l',t',j)$$

We provide here the definition of a sequentially consistent serialization consisting of events belonging to a window $\mathcal{W}$. Since events are restricted to a window, the events belonging to different threads can appear in any order. Though, for an event e_m in the serialization we must ensure that all events e_k in the same thread that precede it ($e_k \blacktriangleleft e_m$) also precede it in the serialization.

$$\sigma^i_{\mathcal{W}} = \{(e_1,...,e_n) \mid \forall\, k,m \in [1,n] \text{ s.t. } k < m \Rightarrow e_k \lhd e_m \\ \forall\, m \in [1,n]\,,\, e_k \in \mathcal{W} \text{ s.t. } e_k \blacktriangleleft e_m \Rightarrow e_k \in \sigma^i_{\mathcal{W}} \wedge k < m\}$$

We provide hereafter an example on which we apply the iterative taint prediction as presented earlier and sketch the proposed adaptation. Figure 4.16 illustrates a window consisting of two blocks (l_b, A) and (l_b, B) (the tail epoch is empty), which are iterated in order A, B. On the right side are the summaries obtained by each iteration. We focus on the second iteration where variables `y,w,d` are marked as tainted. While the tainting of variables `y,w` respects program order, that of variable `d` does not. The serialization $\sigma^i_{\mathcal{W}}$ for which $taint\mathcal{R}(\sigma^i_{\mathcal{W}}, \texttt{d}, e^B_1)$ holds is the following (e^B_2, e^A_1, e^A_2, e^A_3, e^B_1). Executing e^B_2 before e^B_1 does not conform with *program order* and thus tainting of `d` should not be included in the taint predictions.

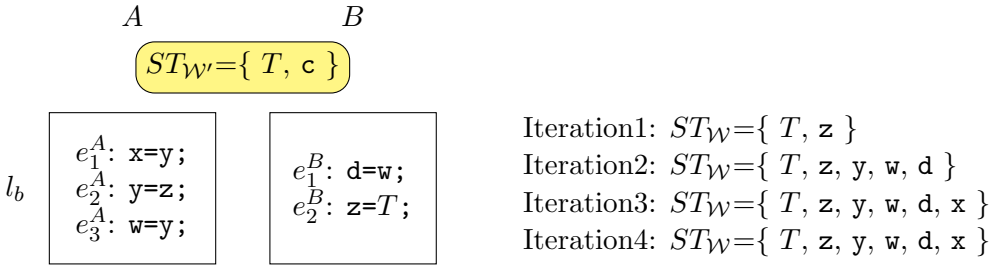


Figure 4.16: Example, sequential consistency and iteration

To filter out taint predictions that do not respect program order, we should verify that for each predicted taint propagation there exists a sequentially consistent serialization of events that justifies it (note that, still killings/untaintings of variables are ignored). To do so, we keep track of events that taint variables as well as the tainting source (i.e., through which variables taintness is propagated). This information about generated/-tainted variables is stored in what we call the *gen history* of the window ($GH_{\mathcal{W}}$). $GH_{\mathcal{W}}$ maps each tainted variable x to a set of *markings*. A *marking* $\mathtt{m}$ is a pair (e, V) where e is the event that taints x ($Def(e) = x$) and V is the set of variables that cause it to be tainted ($V \subseteq Used(e)$). The *gen history* stores information for the currently analyzed window only. We define the function $GH_{\mathcal{W}}(x)$ which returns the markings associated to a variable x inside a window $\mathcal{W}$.

$$GH_{\mathcal{W}}(x) = \{\, (e,V) \mid Def(e) = x \;\bigwedge\; \forall y \in V :\; y \in Used(e) \;\wedge\; \exists\, \sigma^i_{\mathcal{W}} \text{ such that } taint\mathcal{R}(\sigma^i_{\mathcal{W}}, e, y) \,\}$$

Table 4.1 illustrates the usage of $GH_{\mathcal{W}}$ for the example of Figure 4.16. During first iteration we update $GH_{\mathcal{W}}(\texttt{z})$ with the marking $(e^B_2, \{T\})$. The marking encodes the information that variable `z` was tainted at event e^B_2, and that the variable that caused it to be tainted is T which belongs to $ST_{\mathcal{W}'}$. Similarly, each successive iteration adds the markings accordingly. $GH_{\mathcal{W}}$ captures all the necessary information to track the source of tainting inside the window, and infer whether it respects sequential consistency or not. We point out with a colored background the invalid markings, i.e., false taint propagations. The filtering of false predictions can occur either online, the variable is neither added to the ST nor a marking is added to $GH_{\mathcal{W}}$, or offline. Finally, we introduce the function $Events(GH_{\mathcal{W}}(x))$ which returns a set containing all events that taint variable x.

Vars	$ST_{\mathcal{W}}$	Iteration1	Iteration2	Iteration3
T	✓			
c	✓			
z		$(e_2^B,\{T\})$		
y			$(e_2^A,\{\texttt{z}\})$	
w			$(e_3^A,\{\texttt{y}\})$	
d			$(e_1^B,\{\texttt{w}\})$	
x				$(e_1^A,\{\texttt{y}\})$

Table 4.1: Gen history example

Definition 4.7.1 ($Events(GH_{\mathcal{W}}(x))$)

Returns the set containing all events that taint variable x according to $GH_{\mathcal{W}}$.

$$Events(GH_{\mathcal{W}}(x)) = \{e_k \mid \exists\, \mathtt{m} = (e_k, V) \in GH_{\mathcal{W}}(x)\}$$

We provide in Algorithm 4 the *transfer function* to be applied on blocks such that $GH_{\mathcal{W}}$ is updated properly and used to filter out online the non sequentially consistent taint propagations. Of utmost importance is function $TaintingVars$ which returns the set of variables that can taint the variable defined by the current event e. If the set of tainting variables tv is not empty, then we update the $GH_{\mathcal{W}}$ accordingly by adding the mapping $\big(Def(e), (e, tv)\big)$ (line 4) and also update the set of taint predictions (at line 5).

Algorithm 4 Transfer function for taint analysis

In: B , ST
1: **for all** $e \in B$ **do**
2: $tv = TaintingVars(e)$;
3: **if** $tv \neq \emptyset$ **then**
4: $GH_{\mathcal{W}} \leftarrow GH_{\mathcal{W}} \cup \{\big(Def(e)\,,\,(e, tv)\big)\}$;
5: $ST \leftarrow ST \cup Def(e)$;
6: **else**
7: // ignore kills, do nothing
8: **end if**
9: **end for**
Out: ST

To define function $TaintingVars$ we need to introduce first the notion of *taint dependency path*. We start with a more generic definition, that of a backward dependency path $\mathcal{P}^b$, which is a sequence of events that form a chain of defined and used variables.

Definition 4.7.2 (Backward dependency path $\mathcal{P}^b$)

$$\mathcal{P}^b = \{(e_1, \ldots, e_n) \mid \forall k \in [1, n-1] \,.\, Used(e_k) \cap Def(e_{k+1}) \neq \varnothing\}$$

A *taint dependency path* $\mathcal{P}$ for a variable x is a backward dependency path limited to a window $\mathcal{W}$. The path is retrieved in $GH_{\mathcal{W}}$ and ends in the initial set of tainted variables $ST_{\mathcal{W}'}$ where we recall $\mathcal{W}'$ is the window preceding $\mathcal{W}$.

Definition 4.7.3 (Taint dependency path $\mathcal{P}$)

$$\begin{aligned}\mathcal{P} = \{(e_1, \ldots, e_n) \mid \quad & \forall k \in [1,n] : e_k \in \mathcal{W} \ \wedge \ Used(e_n) \cap ST_{\mathcal{W}'} \neq \varnothing \ \wedge \\ & \forall k \in [1, n-1] : Used(e_k) \cap Def(e_{k+1}) \neq \varnothing \wedge \\ & \qquad \exists\, y \in Used(e_k) \cap Def(e_{k+1}), (e_m, V) \text{ such that} \\ & \qquad\quad (e_m, V) \in GH_{\mathcal{W}}(y) \ \wedge \ e_m = e_{k+1} \ \}\end{aligned}$$

A taint dependency path is the sequence of events that correspond to the recursive invocations of $taint\mathcal{R}$. Recalling Figure 4.10 on page 80 the taint dependency path for variable x is $\mathcal{P} = (e_m, e_v)$. Note that, inversing a taint dependency path $\mathcal{P}$ produces a partial serialization, enforcing the ordering of some key events, such that $Def(e_1)$ gets tainted, where e_1 is the first event in $\mathcal{P}$. We use the notation $\sigma(\mathcal{P})$ to represent the partial serialization of events corresponding to a taint dependency path $\mathcal{P}$. That is, given $\mathcal{P} = (e_1, e_2, e_3)$ then $\sigma(\mathcal{P}) = (e_3, e_2, e_1)$. Moreover, we call $TDP(GH_{\mathcal{W}}, x)$ the set of taint dependency paths for variable x with respect to $GH_{\mathcal{W}}$. The set of paths can be obtained with a recursive exploration of markings for variable x.

Back to the definition of $TaintingVars(e)$ in Algorithm 4. The function computes the set tv of variables that taint $Def(e)$ (the variable defined by e). For every variable $y \in tv$ there must exist a taint dependency path $\mathcal{P}$ such that: if e is added to it as the first event, the resulting path is *valid*. The definition of a *valid path* can be modified accordingly to capture any restrictions that must apply to serializations of events (i.e., capture different weak memory models). For now a path is valid if the events respect sequential consistency. For sequential consistency, the call to $isValid(\mathcal{P})$ is equivalent to calling $isConsistent(\mathcal{P})$ which we define here:

Definition 4.7.4 (Predicate $isConsistent(\mathcal{P})$)

A taint dependency path $\mathcal{P}$ is sequentially consistent if the serialization of events it defines, which is the inverse order of events, is sequentially consistent. Thus predicate $isConsistent(\mathcal{P})$ is defined as:

$$\forall e_k, e_m \in \mathcal{P} \ \ then \ \ (k < m \Rightarrow e_m \lhd e_k)$$

Algorithm 5 presents the computation of tv. For each variable in $Used(e)$ it obtains its taint dependency paths and extends them by adding event e as the first event (line 12). We use the following notation $e.\mathcal{P}$ to denote the concatenation of paths or of an event e with a path $\mathcal{P}$. The resulting path is checked for validity. If it is valid then variable

$y \in Used(e)$ can successfully taint $Def(e)$ and is added to tv. There is a special case where variable y belongs to $ST_{\mathcal{W}'}$. In this case we produce an immediate path consisting uniquely of event e and after checking it for validity we add y to tv.

Algorithm 5 $TaintingVars(e)$, set of variables that produce valid taint propagation

In: e

```
1:  tv ← ∅
2:  for all y ∈ Used(e) do
3:    if y ∈ ST_W' then
4:      P ← e // P is a path consisting of just event e
5:      if isValid(P) then
6:        tv ← y ∪ tv
7:        continue;
8:      end if
9:    end if
10:   TP_y ← TDP(GH_W, y)
11:   for all P ∈ TP_y do
12:     P ← e.P // add event e as first event of P
13:     if isValid(P) then
14:       tv ← y ∪ tv
15:       break;
16:     end if
17:   end for
18: end for
```

Out: tv

To clarify the definition of $TaintingVars(e)$ we apply it to the example of Figure 4.16 and the corresponding illustration of its *gen history* in table 4.1. First, we apply $TaintingVars(e_2^B)$ during first iteration. Since $Used(e) = T$ which belongs to $ST_{\mathcal{W}'}$ the path to check is $\mathcal{P} = (e_2^B)$ which respects sequential consistency. Thus, variable T is added to tv. We apply now $TaintingVars(e_1^B)$ during the second iteration. There is only one taint dependency path for variable `w` which is $\mathcal{P} = (e_3^A, e_2^A, e_2^B)$. We add e_1^B as the first element, $\mathcal{P} = e_1^B.\mathcal{P} = (e_1^B, e_3^A, e_2^A, e_2^B)$. The resulting path $\mathcal{P}$ is not sequentially consistent because e_2^B does not precede e_1^B ($e_2^B \not\lhd e_1^B$).

Note that in this section we make the assumption that kills are ignored. Thus, if there exists a sequentially consistent path $\mathcal{P}$ that generates a variable `x`, then there also exists a serialization $\sigma_{\mathcal{W}}^i$ that generates it. To produce $\sigma_{\mathcal{W}}^i$ it suffices to extend the partial serialization $\sigma(\mathcal{P})$ obtained from $\mathcal{P}$ such that all remaining events in $\mathcal{W}$ are positioned on $\sigma_{\mathcal{W}}^i$ respecting program order for all threads. Based on the assumption the effect of these events is ignored.

We illustrate in Figure 4.17 the composition of a serialization. On the left side we illustrate the currently analyzed window (for clarity we assume its tail epoch is empty) consisting of two blocks. The arrows depict the taint dependency path $\mathcal{P}$ that taints `x`. On the right side, we explicit the path $\mathcal{P}$ and the derived partial serialization $\sigma(\mathcal{P})$. Below it we appose the remaining events denoted as $\mathcal{W} \setminus \mathcal{P}$. An arrow initiated from each remaining event indicates its plausible positioning such that a sequentially consistent $\sigma_{\mathcal{W}}^i$ is obtained.

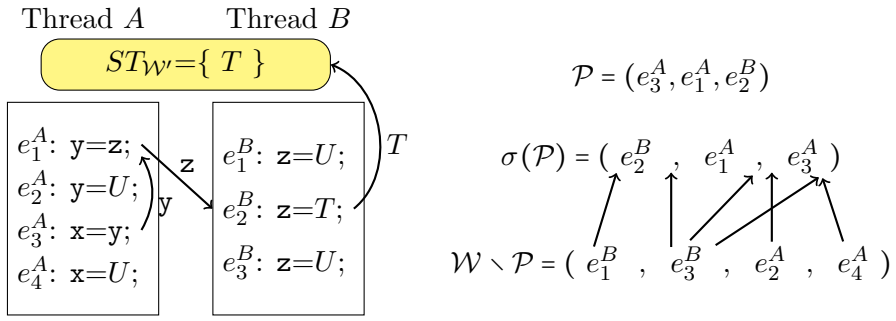


Figure 4.17: Composing a sequentially consistent serialization based on a TDP

4.7.2 Taking kills into account

In this section we introduce the killing/un-tainting of variables. Taking kills into account makes taint predictions more accurate and thus reduces *false positives*. Though, extra care must be taken since we do not want our analysis to miss any valid taint propagations. The killing of a variable affects taint prediction in two ways:

i) it prevents taint propagation between variables;

ii) it excludes variables from the summarization of the window.

Prior to detailing the two cases we give their intuition using the example of Figure 4.17. In the first case, kill/untainting of a variable occurs between the tainting point of a variable and its usage to propagate taintness to another variable. In the example event e_2^A must be executed between e_1^A and e_3^A. With kills taken into account the given path $\mathcal{P}$ cannot produce a serialization such that x is tainted. For the second case we shall focus on variable z which is explicitly tainted at event e_2^B. Note that the succeeding event e_3^B untaints z. Thus on all sequentially consistent serializations of the window variable z will eventually be untainted and thus should not be included in the taint predictions of the window (i.e., $ST_{\mathcal{W}}$).

We approach the killing/untainting of variables by separating the two cases identified. The killing of variables that break tainting paths is treated *online* during the iterative algorithm. Dually, the killing of variables that excludes them from the summarization of the window is treated *offline* i.e., after the iterative algorithm (Algorithm 2)has completed.

Killing a taint dependency path (TDP)

For the killing of tainting paths $\mathcal{P}$ we need to verify that there exists a sequentially consistent partial serialization $\sigma^i_{\mathcal{P}}$ consisting of all events preceding the events in $\mathcal{P}$, such that $taint\mathcal{W}(\sigma^i_{\mathcal{P}}, e_1, Def(e_1))$ (see Definition 4.5.2 on page 78) holds, where e_1 is the first event in $\mathcal{P}$. Figure 4.18 illustrates with a light background the events that must be included in $\sigma^i_{\mathcal{P}}$. In this abstract example the path $\mathcal{P}$ is designated by the arrows. The check for the existence of a $\sigma^i_{\mathcal{P}}$ is performed online during the computation of $TaintingVars$ (see Algorithm 5 on the preceding page) as part of the $isValid(\mathcal{P})$ predicate. More precisely,

predicate $isValid(\mathcal{P})$ is the conjunction of predicates $isConsistent(\mathcal{P})$ (see Definition 4.7.4 on page 88) and $noKill(\mathcal{P})$ which we define after the presentation of some key examples.

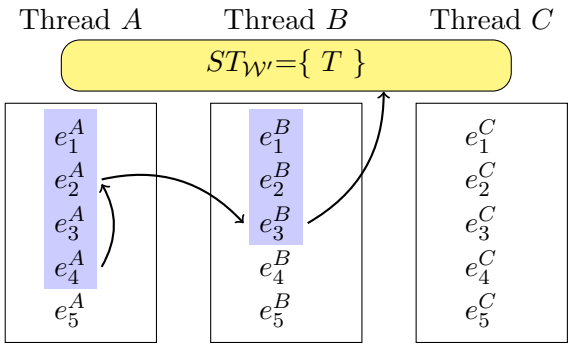
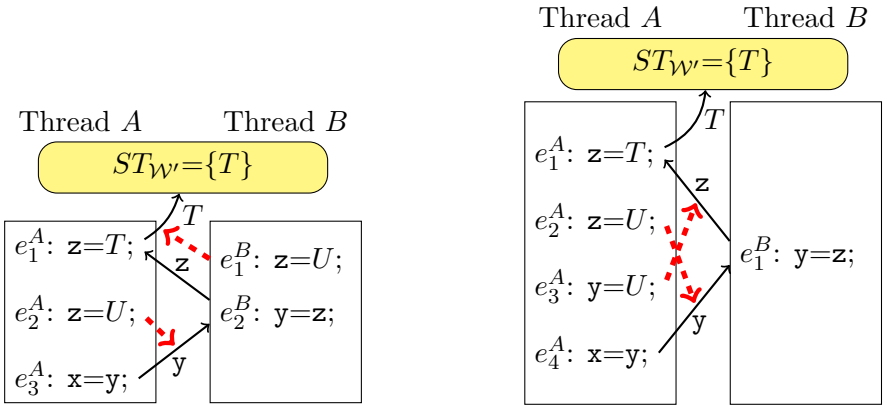


Figure 4.18: Events contained in $\sigma^i_{\mathcal{P}}$

Figure 4.19 illustrates two examples. In both examples the solid arrows, labeled by the variable that propagates taintness, represent the tainting path $\mathcal{P}$ that causes variable `x` to be tainted at events e_3^A and e_4^A accordingly. The dashed thick arrows designate where the killing events should be placed such that they do not break $\mathcal{P}$. In Figure 4.19(a) event e_2^A must be positioned after e_2^B such that it kills variable `z` only after it has propagated its taintness to variable `y`. Similarly e_1^B must be placed before e_1^A such that it kills `z` before it gets tainted. In this example the killing events can be serialized such that $\mathcal{P}$ is capable of tainting `x`. Dually, in Figure 4.19(b) the path $\mathcal{P}$ does not hold. As illustrated e_2^A must be place after e_1^B while inversely e_3^A before e_1^B. Due to the program order imposed one of the two events will inevitably break $\mathcal{P}$.



(a) valid serialization exists (b) no valid serialization exists

Figure 4.19: Inferring a valid serialization for a path $\mathcal{P}$

To verify if there exists a serialization $\sigma^i_{\mathcal{P}}$ such that the killing events do not break $\mathcal{P}$ we perform some sanity checks on the composition of $\mathcal{P}$ with the preceding events. Here are the observations that allow us to make these checks in a incremental way.

- all events that are not part of $\mathcal{P}$ are considered as kills. Figure 4.20 illustrates an example where two tainting paths are present, one defined by solid and the other by

dashed arrows. The solid path is considered as invalid because event e_2^A is considered a kill of variable `z` witch breaks the solid path. Variable `x` though gets tainted by the dashed path.

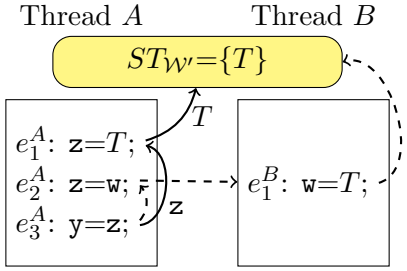


Figure 4.20: Events not belonging to $\mathcal{P}$ are considered kills

- events belonging to different threads can be serialized independently

Before giving the definition of predicate $noKill(\mathcal{P})$ we introduce the following notations:

$\sigma^a \oplus \sigma^b$ is a sequentially consistent merge operator. Given two serializations of events σ^a and σ^b, themselves respecting sequential consistency, it produces all plausible sequentially consistent serializations containing events of σ^a and σ^b.

$\sigma(e_k^A, e_m^A)$ defines a sub-sequence of events produced by a thread. The events contained are scoped by e_k^A and e_m^A, where $e_k^A \blacktriangleleft e_m^A$. For example, $\sigma(e_1^A, e_6^A) = (e_2^A, e_3^A, e_4^A, e_5^A)$

$taintSource(\mathcal{P})$ returns the tainting source, i.e., the variable that is reached by $\mathcal{P}$ in $ST_{\mathcal{W}'}$. In the example of Figure 4.20 $taintSource(\mathcal{P}) = T$ (both for the solid and dashed paths).

Definition 4.7.5 (Predicate $noKill(\mathcal{P})$)

Under sequential consistency, a taint dependency path $\mathcal{P}$ is not broken by a killed variable if there exists a sequentially consistent serialization of events belonging in $\mathcal{P}$ and the events preceding them such that, between two successive events e_i, e_{i+1} of $\mathcal{P}$ there is no event e_j such that $Def(e_j) = Def(e_i)$.

$$\forall e_k \in \mathcal{P} = (e_1, ..., e_k, ..., e_m, ..., e_n) \text{ we distinct the following cases:}$$

- $\exists\, m > k$ *such that* $e_m \blacktriangleleft e_k \wedge \nexists e_q$ *such that* $m > q > k \wedge e_m \blacktriangleleft e_q$
  - *if* $m = k + 1$ *then* (e.g., Figure 4.20 e_3^A,e_1^A)

$$\forall e_j \text{ s.t. } e_m \blacktriangleleft e_j \blacktriangleleft e_k \Rightarrow Def(e_j) \neq Def(e_m)$$

  - *else* (e.g., Figure 4.19(b) e_4^A,...,e_1^A)

$$\exists \sigma = (\ldots, e_i, \ldots, e_j, \ldots, e_{i+1}, \ldots) \in \sigma(\mathcal{P}) \oplus \sigma(e_m, e_k) \text{ s.t.}$$
$$\forall e_i, e_{i+1} \in \sigma(\mathcal{P}), e_j \in \sigma(e_m, e_k) \Rightarrow Def(e_j) \neq Def(e_i)$$

- $\nexists\, m > k$ *such that* $e_m \blacktriangleleft e_k$

  - *if* $k = n$ *then* (e.g., Figure 4.19(b) e_1^A)

    $$\forall e_j \text{ such that } e_j \blacktriangleleft e_k \Rightarrow Def(e_j) \neq taintSource(\mathcal{P})$$

  - *else (we introduce a dummy event)* (e.g., Figure 4.19(b) e_2^B)

    $$\mathcal{P}' = \mathcal{P}.e_{k'} \text{ where } Def(e_{k'}) = taintSource(\mathcal{P}) \wedge \forall e_j \in \mathcal{W} \text{ such that } Thr(e_j) = Thr(e_k) \Rightarrow e_{k'} \blacktriangleleft e_j$$

    Now we can apply the first case where:

    $$\exists\, m > k \text{ such that } e_m \blacktriangleleft e_k \wedge \nexists e_q \text{ such that } e_m \blacktriangleleft e_q$$

    (note that $m > k+1$)

To simplify the definition of predicate $noKill(\mathcal{P})$ in the case were $m > k+1$ we make use of the merge operator $\oplus$ implying that all sequentially consistent serializations consisting of events in $\sigma(\mathcal{P})$ and $\sigma(e_m, e_k)$ are computed. This may be misleading since we stated earlier that we verify the existence of $\sigma^i_{\mathcal{P}}$ incrementally. We provide here the checks that verify that:

$$\exists\, \sigma = (\dots, e_i, \dots, e_j, \dots, e_{i+1}, \dots) \in \sigma(\mathcal{P}) \oplus \sigma(e_m, e_k) \text{ s.t.}$$
$$\forall e_i, e_{i+1} \in \sigma(\mathcal{P}), e_j \in \sigma(e_m, e_k) \Rightarrow Def(e_j) \neq Def(e_i)$$

Definition 4.7.6 (Positioning kill events)

Given a taint dependency path $\mathcal{P} = (e_1, \dots, e_k, \dots, e_m, \dots, e_n)$ *where* $e_m \blacktriangleleft e_k$ *and* $m > k+1$. *We want to check that events in* $\sigma(e_m, e_k)$ *can be ordered such that they respect sequential consistency and do not kill a variable between the point it is defined and used to propagate taintness. That is between events* e_i, e_{i+1} *in* $\sigma(\mathcal{P})$ *where* $k \leq i < m$. *To respect program order, for an event* e_j *(*$e_m \blacktriangleleft e_j \blacktriangleleft e_k$*) that kills a variable of* $\sigma(\mathcal{P})$ *all events preceding should be able to be positioned before it and dually all succeeding events after it.*

Given $\mathcal{P} = (e_1, \dots, e_k, \dots, e_m, \dots, e_n)$ *where* $m > k+1$ *and* $e_m \blacktriangleleft e_k$*:*

- $\forall e_j$ *s.t.* $e_m \blacktriangleleft e_j \blacktriangleleft e_k \Rightarrow Def(e_j) \cap \left(\bigcup_{i \in [k+1,m]} Def(e_i)\right) \neq \varnothing$

$$\bigvee$$

- $\forall e_j,\, i \in [k+1, m]$ *s.t.* $e_m \blacktriangleleft e_j \blacktriangleleft e_k \wedge Def(e_j) = Def(e_i)$

$$\textit{then:} \begin{cases} \forall e_{j'} \text{ s.t. } e_m \blacktriangleleft e_{j'} \blacktriangleleft e_j \Rightarrow \exists i' \in [i+1, m] \text{ s.t. } Def(e_{i'}) \neq Def(e_{j'}) \\ \bigvee \\ \forall e_{j'} \text{ s.t. } e_j \blacktriangleleft e_{j'} \blacktriangleleft e_k \Rightarrow \exists i' \in [k+1, i-1] \text{ s.t. } Def(e_{i'}) \neq Def(e_{j'}) \end{cases}$$

To conclude taking into account kills in taint propagation we remind that the predicate $noKill(\mathcal{P})$ is used in conjunction with $isConsistent(\mathcal{P})$ in the call of $isValid(\mathcal{P})$ at Algorithm 5 on page 89. Thus, the gen history of the current window ($GH_{\mathcal{W}}$) is precisely updated.

Excluding variables from window summarization

As mentioned earlier the killing of variables such that they are excluded from the summarization of a window are treated offline. After the iterations of Algorithm 2 have completed we hold $ST_{\mathcal{W}}$ which over-approximates the set of tainted variables down to $\mathcal{W}$ and $GH_{\mathcal{W}}$ which contains the markings for generated variables. For each marking $\mathfrak{m}$ in $GH_{\mathcal{W}}$ the following is true:

$$\mathfrak{m} = (e_k, V) \in GH_{\mathcal{W}} \Leftrightarrow \exists\, \sigma^i_{\mathcal{P}} \text{ s.t. } taintW(\sigma^i_{\mathcal{P}}, e_k, Def(e_k))$$

We remind that, a variable x is included in the summarization of the window if there exists a serialization $\sigma^i_{\mathcal{W}}$ of all events in $\mathcal{W}$ such that $taintW(\sigma^i_{\mathcal{W}}, last(\sigma^i_{\mathcal{W}}), x)$ holds. Dually, to exclude a variable x from the summarization then on all serializations $\sigma^i_{\mathcal{W}}$ the last assignment to x should be with an untainted value. Since we do not construct all serializations, but instead use the iterative algorithm which guarantees us to identify all propagations, we cannot precisely (at least not cost-effectively) identify variables killed in $\mathcal{W}$. Thus, we under-approximate the killing of variables by identify the following cases for which we are certain variables are killed in $\mathcal{W}$:

- x is defined and never generated

$$\exists e_k \in \mathcal{W} \text{ s.t. } Def(e_k) = x \wedge GH_{\mathcal{W}}(x) = \varnothing$$

- all threads that generate x successively kill it

$$\forall e_k \in Events(GH_{\mathcal{W}}(x)) \Rightarrow \exists e_m \in \mathcal{W} \text{ s.t. } e_k \blacktriangleleft e_m \wedge Def(e_m) = x \wedge e_m \notin Events(GH_{\mathcal{W}}(x))$$

The first case is straight forward. Variable x is defined in $\mathcal{W}$ but no marking exists for it in $GH_{\mathcal{W}}$. Thus, all events in $\mathcal{W}$ that define it assign an untainted value. We conclude that on all serializations $\sigma^i_{\mathcal{W}}$ x is killed and thus can be removed from $ST_{\mathcal{W}}$. Figure 4.21 illustrates an abstract example where only the code of events that define `x` is given explicitly. We assume that there does not exist any valid tainting path such that variable `d` is tainted at e^C_1. Thus, on all serializations $\sigma^i_{\mathcal{W}}$ variable `x` is lastly assigned an un-tainted value. As illustrated `x` is removed from $ST_{\mathcal{W}}$.

In the second case variable x is generated. Thus, there exists at least a partial serialization $\sigma^i_{\mathcal{P}}$ for which x is tainted at event e_k. To ensure that finally x is assigned an untainted value, on all plausible serializations, it must be killed by an event e_m of the same thread that succeeds e_k i.e., $e_k \blacktriangleleft e_m$. Figure 4.22(a) illustrates an abstract example

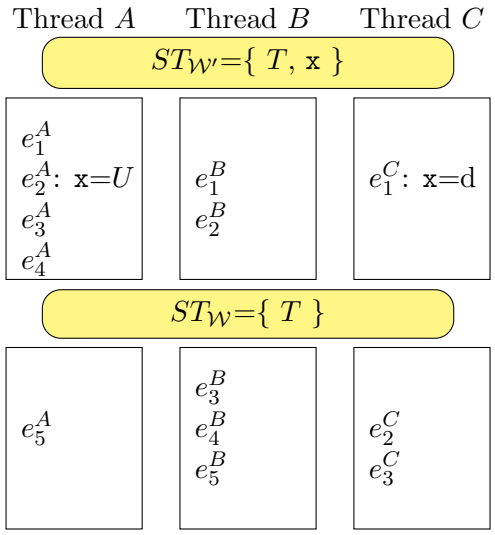
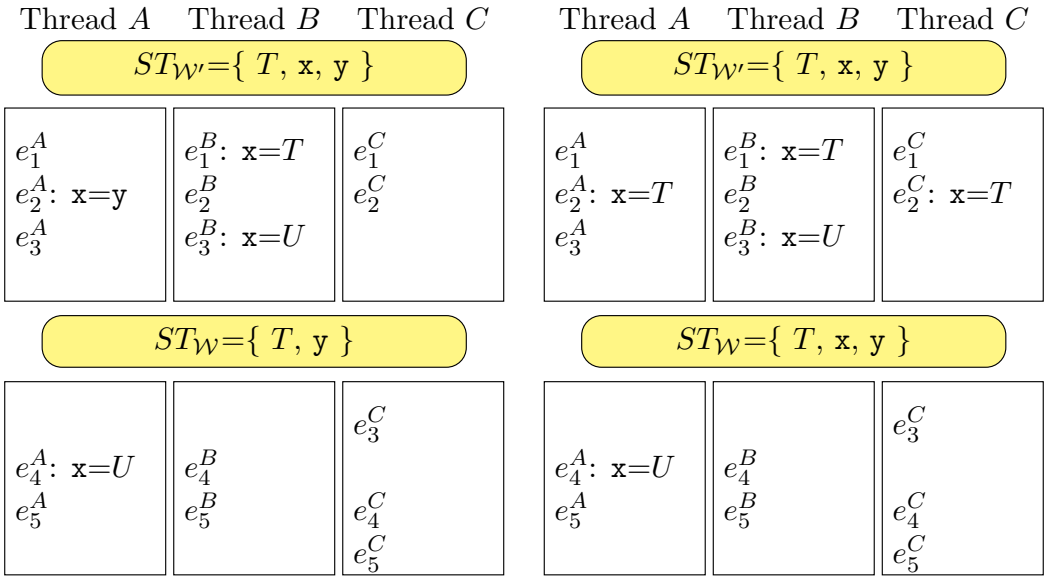


Figure 4.21: Kill in $\mathcal{W}$ when not generated

where all threads that taint x successively kill it. We can note that program order guarantees that eventually the last assignement to x is always an untainted value. Hence, x is safely excluded from $ST_{\mathcal{W}}$. Dually in Figure 4.22(b) thread C generates x at event e_1^C and there does not exist any succeeding event in C that kills x. Obviously for the serialization $\sigma^i_{\mathcal{W}} = (\ldots, e_2^C, e_3^C, e_4^C, e_5^C)$ x ens up tainted.



(a) Kill after generating

(b) Generating without killing

Figure 4.22: All threads generating x must eventually kill it

To conclude removing killed variables from the summarization we provide Algorithm 6 which updates Algorithm 2 by adding the offline processing that refines the summarization of the currently analyzed window. We also give in an algorithmic-like form the removing of killed variables in Algorithm 7.

Algorithm 6 Vertical processing $Vertical(\mathcal{W}, ST_{\mathcal{W}'})$

In: $\mathcal{W} = \{l_b, l_t\}, ST_{\mathcal{W}'}$
1: $ST \leftarrow ST_{\mathcal{W}'}$
2: **repeat**
3: $ST \leftarrow Horizontal(l_b, ST)$
4: $ST \leftarrow Horizontal(l_t, ST)$
5: **until** (ST **unmodified**)
6: $ST_{\mathcal{W}} \leftarrow WindowKills(ST, GH_{\mathcal{W}})$
Out: $ST_{\mathcal{W}}$

Algorithm 7 $WindowKills(ST, GH_{\mathcal{W}})$

In: $ST, GH_{\mathcal{W}}$
1: $ST_{\mathcal{W}} \leftarrow ST$
2: **for all** $x \in ST$ **do**
3: **if** $GH_{\mathcal{W}}(x) = \varnothing \ \wedge \ \exists e_k \in \mathcal{W}$ s.t. $Def(e_k) = x$ **then**
4: $ST_{\mathcal{W}} \leftarrow ST_{\mathcal{W}} \setminus \{x\}$
5: **else**
6: no_kill_exists $\leftarrow false$
7: **for all** $e_k \in Events(GH_{\mathcal{W}}(x))$ **do**
8: **if** $\nexists e_m \in \mathcal{W}$ s.t. $Def(e_m) = x \ \wedge \ e_k \blacktriangleleft e_m \ \wedge \ e_m \notin Events(GH_{\mathcal{W}}(x))$ **then**
9: no_kill_exists $\leftarrow true$
10: break;
11: **end if**
12: **end for**
13: **if** no_kill_exists $= false$ **then**
14: $ST_{\mathcal{W}} \leftarrow ST_{\mathcal{W}} \setminus \{x\}$
15: **end if**
16: **end if**
17: **end for**
Out: $ST_{\mathcal{W}}$

4.7.3 Effects of sliding window

We mentioned earlier in section 4.5 that there are two aspects in our sliding window-based analysis: (i) predicting explicit taint propagations within a window and (ii) reasoning correctly about the overlapping of interleavings caused by sliding windows.We develop here how the sliding of windows affects our predictions, and what precautions can be taken such that our predictions remain an over-approximation of taintness, but also increase the confidence on the soundness of the predictions.

Killing variables in tail causes under-approximates taint propagation

We focus on the killing of variables within a window. The definition we gave before is correct when restricted to a window. When sliding windows, it may cause *under-approximation* of taint predictions. Figure 4.23 illustrates such an example. We note that, in window $\mathcal{W}'$ variable `x` is killed since on all serializations $\sigma^i_{\mathcal{W}'}$ it is finally assigned a non-tainted value. Though, the killing is premature because it eliminates the propagation

of taintness to variable z. The propagation is feasible under our assumptions since e_1^B can be executed prior the untainting of x. The arrow shows the interleaving under which taintness is propagated in $\mathcal{W}$.

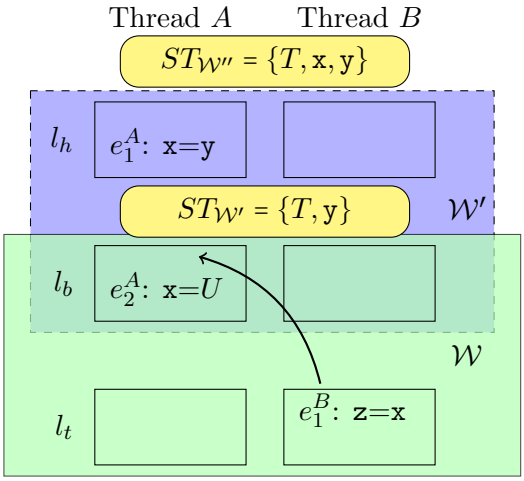


Figure 4.23: Delay killing in tail

The example of Figure 4.23 shows that kills that occur in the tail epoch of a window may hide valid taint propagations on the consecutive window. To overcome these issues, we shall only exclude a variable from the summarization if it is killed in the body of a window.

Incompatible TDPs over-approximate taint propagation

Windows consisting of two epochs allow us to reason only on valid interleavings, based on our initial assumptions, but has the disadvantage of predicting possibly incompatible propagations. For two consecutive windows $\mathcal{W}'$, $\mathcal{W}$ the interleavings of events in the common epoch l_b with $l_h$in $\mathcal{W}'$ and l_t in $\mathcal{W}$ are computed independently. Thus, it is possible that the serializations that propagate taintness in $\mathcal{W}'$ and $\mathcal{W}$ are conflicting.

Figure 4.24 illustrates an example of incompatible TDPs. In the first window $\mathcal{W}' = \{l_h, l_b\}$ variable x is tainted through $\mathcal{P}_1 = (e_1^B, e_1^C, e_3^A, e_1^A)$ traced with a solid line. The summary $ST_{\mathcal{W}'}$ correctly contains x as there exists a serialization which taints x. Sliding to the next window $\mathcal{W} = \{l_b, l_t\}$ we identify with a dashed line $\mathcal{P}_2 = (e_4^A, e_2^A, e_3^B)$ which taints variable y using x. Taint dependency path $\mathcal{P}_2$, although valid in $\mathcal{W}$, is *not compatible* with the one that generated x (which is the tainting source for variable y). Namely, the two TDPs cannot be merged and hence they do not provide a concrete serialization demonstrating how y gets tainted.

Merging TDPs computed in different windows is not always feasible. However, TDPs indicating why a variable is tainted within a window are not unique (although finding a single path is sufficient in our algorithm). For instance, a closer look at Figure 4.24 shows a second $\mathcal{P}_3 = (e_2^B, e_1^A)$ (with dash-dotted path) for variable x which is compatible with $\mathcal{P}_2$.

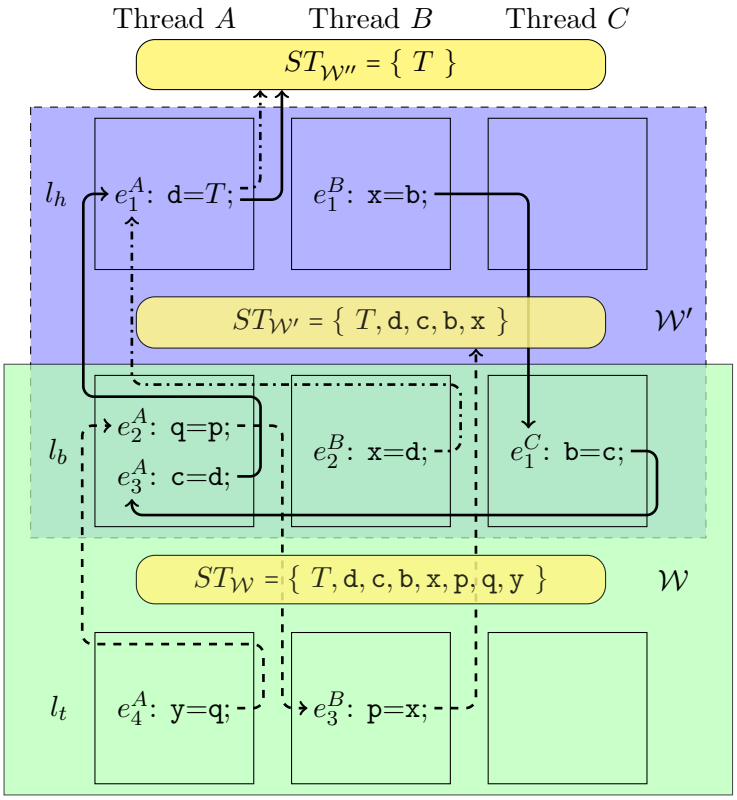


Figure 4.24: Propagation through incompatible TDPs

The incompatibility of TDPs over-approximates our predictions. To reduce the number of false positives we can classify the tainted variables into two categories *strong* and *weak*. *Strongly* tainted variables are those for which taintness propagation occurred through mergeable tainting paths. Dually, *weakly* tainted variables are those for which non-mergeable tainting paths may exist. For the strongly tainted variables a witness execution can be constructed.

We provide hereafter two heuristics that can be used to identify *strongly* tainted variables:

1. If a tainting path $\mathcal{P}$ contains uniquely events from the window's body and its tainting source variable is strongly tainted, then it defines a strongly tainted variable as well. Since $\mathcal{P}$ contains only instructions in body epoch it has no conflicts with paths in the succeeding window.

2. If a variable x is tainted in two consecutive epochs and (i) there is no kill of this variable in the common epoch and (ii) the variable that made it tainted in the first epoch is *strongly* tainted, then x is strongly tainted.

4.8 Respecting synchronization primitives

As presented in section 2.5 on page 18 several synchronization mechanisms exist to impose an ordering on the execution of threads. We focus on the usage of mutexes for the synchronization of critical sections and take advantage of their semantics to infer more accurate *taint dependency paths* in our analysis. That is, we add extra restrictions to the $isValid(\mathcal{P})$ function.

We remind a mutex is a binary variable with states *locked* and *unlocked*. Critical sections are portions of code that should be executed atomically. To synchronize access to critical sections all threads need to acquire the necessary mutexes prior to entering their critical section, and release them once its execution is completed. A thread acquires/locks a mutex m by calling a blocking function $lock(m)$ and releases/unlocks it by calling $unlock(m)$. A successful call to $lock(m)$ allows the thread to enter the critical section and prevents other threads from entering their critical section protected by the same mutex m until it is released ($unlock(m)$) by the thread that initially obtained it. A mutex acquisition always has a matching mutex release. As mentioned earlier in section 4.4.1 on page 73 synchronization events are also logged. We define hereafter a *protection* [1] which is a triplet (m, e_l, e_u) where m is the mutex used to synchronize threads, e_l is the logged event corresponding to the locking of the mutex ($lock(m)$) while e_u is the event corresponding to the *matching* release of the mutex ($unlock(m)$). The locking event always precedes the unlocking event, thus $e_l \blacktriangleleft e_u$. Finally, we will use a dot notation in the sequel to refer to the elements of a protection. Thus, given a protection $p = (q, e_k, e_m)$ then $p.m = q$, $p.e_l = e_k$ and $p.e_u = e_m$.

A critical section may be synchronized using several mutexes. Moreover, the set of mutexes protecting events of a critical section may not be the same for all events. We call *context* the set of protections surrounding an event and define the function $cont(e)$ which returns the context for an arbitrary event e. More precisely:

$$cont(e) = \{\, p \mid p.e_l \blacktriangleleft e \ \wedge\ e \blacktriangleleft p.e_u \}$$

We provide hereafter a small example to clarify the notion of *protection* and *context* we just introduced. Listing 4.3 presents an excerpt of a log file with events produced by thread A. On the right side, we identify the two protections present in Listing 4.3 namely p_a, p_b matched to mutexes m_a and m_b respectively. We also provide the context for events e_3^A and e_5^A.

[1] There is no connection with *protections* in chapter 3

e_1^A: $lock(m_a)$
e_2^A: $lock(m_b)$
e_3^A: `x = y;`
e_4^A: $unlock(m_b)$
e_5^A: `w = z;`
e_6^A: $unlock(m_a)$

$p_a = (m_a, e_1^A, e_6^A)$
$p_b = (m_b, e_2^A, e_4^A)$
$cont(e_3^A) = \{\ p_a,\ p_b\ \}$
$cont(e_5^A) = \{\ p_a\ \}$

Listing 4.3: Critical section

Moreover, we introduce two operators for contexts: $\sqcap$ which computes the set of mutexes shared between two contexts, and $\boxplus$ which computes the set of mutexes *used in same critical sections* and shared between two contexts. Finally, we define the function $Mutex(c)$ which gives the set of mutexes for context c.

$$c_1 \sqcap c_2 \ = \ \{m' \mid \exists (p \in c_1\ ,\ q \in c_2) \ \text{ such that } \ p.m = q.m = m'\}$$

$$c_1 \boxplus c_2 \ = \ \{m' \mid \exists (p \in c_1\ ,\ q \in c_2) \ \text{ such that } \ p = q \ \wedge \ m' = p.m\}$$

We note that the equality for two protections is defined as a complete match of all elements of the *protection* triplet:

$$p = q \ \Rightarrow \ p.m = q.m \ \wedge \ p.e_l = q.e_l \ \wedge \ p.e_u = q.e_u$$

$$Mutex(c_1) \ = \ \bigcup_{p \in c_1} p.m$$

4.8.1 Inferring order from mutexes

By definition of mutual exclusion critical sections protected by the same mutexes never execute concurrently. Thus, in the produced log files timestamps of synchronization events should be in accordance with the order they were executed. While the timestamps allow us to infer the exact ordering, we try to predict different serializations of entire critical sections if possible.

Figure 4.25 illustrates how the execution of two critical sections can be interleaved. In this instance we can clearly identify that the critical sections were executed in the order A, B. This ordering implies that only variable `y` should end up tainted. With the current slicing our analysis assumes all events are interleavable. Although the analysis should not interleave events belonging to a critical section (i.e., e_2^A, e_3^A, e_2^B, e_3^B), it can interleave the lock/unlock events resulting into considering a different synchronization where the ordering of critical sections is B, A. As illustrated in the summary $ST_{\mathcal{W}}$ our analysis predicts both serializations of the critical sections and thus both `x` and `y` are considered tainted.

In the example of Figure 4.25 the execution of critical sections could be interleaved. This is not always the case. For critical sections to be interleavable by the analysis they should appear within two consecutive epochs, i.e. be bounded in a window. If this is not the case then a fixed ordering between events in the critical section is imposed and it

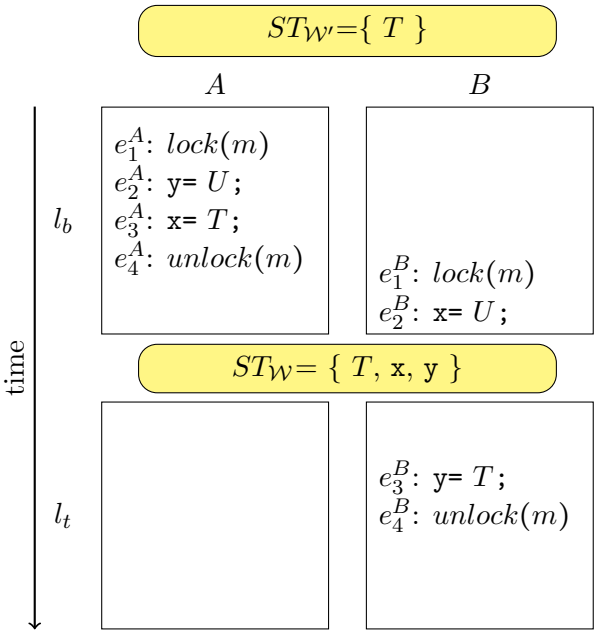


Figure 4.25: Interleaving critical sections

should be respected by the inferred *taint dependency paths*. For that reason we introduce the binary operator $p_a \lessdot p_b$ which checks if events protected by protection p_a precede those protected by p_b. The operator is defined as follows:

$$p_a \lessdot p_b \Rightarrow (p_a.m = p_b.m) \wedge \Big(\begin{array}{ll} Epoch(p_a.e_l) < Epoch(p_b.e_l) - 1 & \vee \\ Epoch(p_a.e_l) < Epoch(p_b.e_u) - 1 & \Big) \end{array}$$

Figure 4.26 illustrates the conditions for defining precedence based on the protections. The darkened areas are events in critical sections protected by the same mutex (let it be m) and the events surrounding it are the lock and unlock events. Each critical section defines a protection: $p_a = (m, e_k^A, e_m^A)$, $p_b = (m, e_k^B, e_m^B)$, $p_c = (m, e_k^C, e_m^C)$. We note that the lock release event (e_m^C) for critical section in thread C does not appear in the figure, but it definitively exists in a subsequent epoch. The relation $p_a \lessdot p_c$ obviously holds because the acquisitions of the mutex are not interleavable. Contrarily though, the acquisitions of the mutex are interleavable between critical sections of thread A and B. Despite that, the relation $p_a \lessdot p_b$ also holds because the acquisition event of thread A (e_k^A) cannot interleave with the release of mutex by thread B (e_m^B) which is necessary for swapping the execution order of the critical sections.

The precedence operator is also used to compare contexts $c_1 \lessdot c_2$. For a context to precede another we need to identify precedence between any two protections belonging to the contexts, that is:

$$c_1 \lessdot c_2 \Rightarrow \exists\, (p_a \in c_1\,,\, p_b \in c_2\,) \text{ such that } p_a \lessdot p_b$$

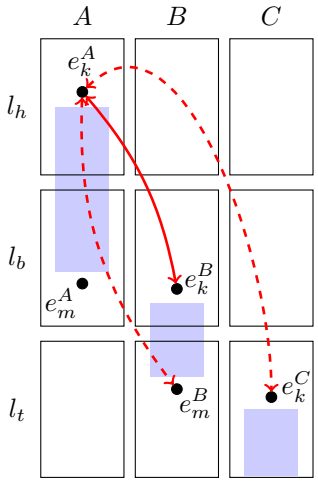


Figure 4.26: Ordering critical sections using mutexes

4.8.2 Enforcing explicit mutex ordering in taint dependency paths

When the critical sections cannot be interleaved, then their execution order must be respected by all inferred taint dependency paths. That for, we enforce the *consistency check* function ($isConsistent()$) by adding an extra restriction that enforces the precedence of events that constitute a taint dependency path based on their contexts:

Definition 4.8.1 (Predicate $isConsistent(\mathcal{P})$)

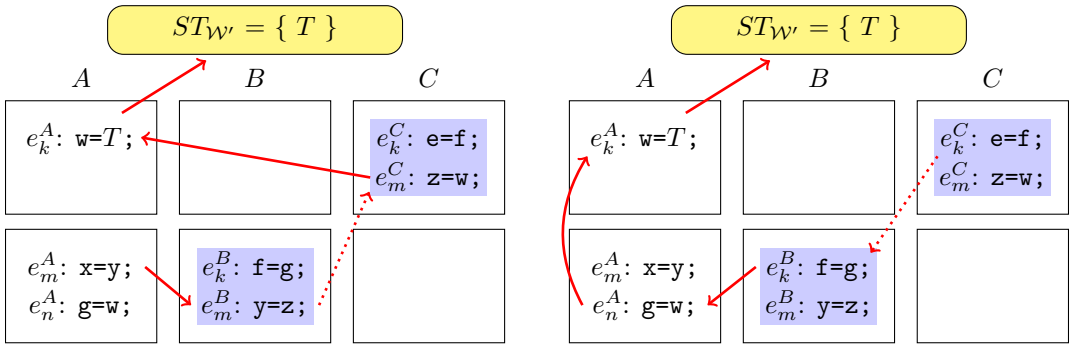
Ensures interleaving assumptions and explicit ordering imposed by critical sections are respected:

$$\forall e_k, e_m \in \mathcal{P} \text{ then } \begin{cases} k < m \Rightarrow e_m \lhd e_k \\ \qquad \bigwedge \\ cont(e_m) \neq \varnothing \wedge cont(e_k) \neq \varnothing \Rightarrow cont(e_m) \lessdot cont(e_k) \end{cases}$$

4.8.3 Enforcing implicit mutex ordering in taint dependency paths

As mentioned earlier critical sections can be re-ordered by the analysis when they reside within the same window. The order in which they are executed is defined by the taint paths containing events belonging to those critical sections. Once an order is set it should be respected throughout the path.

Figure 4.27 illustrates two instances of the same window each depicting a different taint dependency path imposing a different ordering of the critical sections. On Figure 4.27(a) the tainting path $\mathcal{P}_1 = (e_m^A, e_m^B, e_m^C, e_k^A)$ implies the execution order C, B of the critical sections since e_m^C precedes e_m^B. Dually, the tainting path $\mathcal{P}_2 = (e_k^C, e_k^B, e_n^A, e_k^A)$ in Figure 4.27(b) implies the execution order B, C of critical sections.



(a) Path defining C, B order (b) Path defining B, C order

Figure 4.27: Taint paths define implicitly order of critical sections

The examples provided above illustrate how a taint dependency path imposes the ordering of critical sections by visiting events inside them. Once the execution order is set between two (or more) critical sections we need to ensure that the path (i) is not bouncing between two critical sections, protected by the same mutexes and (ii) the serialization of events respects the order imposed by the critical sections.

Figure 4.28(a) illustrates a path bouncing between two critical sections. That is, there exist two events in the tainting path that belong to the same critical section (e_m^B, e_k^B) and inbetween there exists an event that belongs to a mutually excluded critical section (e_m^C). In this example, initially events e_m^B, e_m^C connected with a dotted edge define the order C, B between the critical sections. Subsequently, events e_m^C, e_k^B linked with a dashed edge define the inverse ordering of critical sections (B, C). Thus the tainting path is no longer valid. To prevent such paths from propagating taintness we introduce a new path restriction $noRe-entry(\mathcal{P})$:

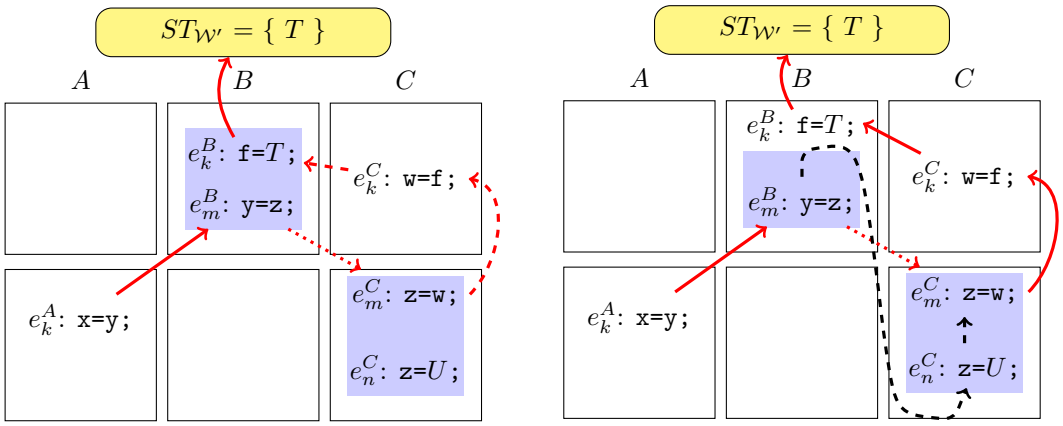
Definition 4.8.2 (Predicate $noRe-entry(\mathcal{P})$)

Ensures implicit ordering of critical sections is respected throughout the path:

$$\forall e_k, e_n \in \mathcal{P} \text{ such that } k < n \text{ then}$$

$$(\mathcal{M} = cont(e_k) \boxplus cont(e_n) \neq \varnothing) \Rightarrow \nexists e_m \text{ such that } Thr(e_m) \neq Thr(e_k) \wedge k < m < n \wedge Mutex(cont(e_m)) \cap \mathcal{M} \neq \varnothing$$

The second thing we need to take care of when considering the implicit ordering of critical sections, is to respect entirely the serialization of all events. Figure 4.28(b) has slightly modified the example of Figure 4.28(a) such that the bouncing between critical sections is avoided, but illustrates the problem of killing the linking variable (`z` in our example) on the serialization of the critical sections. Again the dotted edge connecting events e_m^B, e_m^C specifies the ordering of the critical sections to be C, B. Because the events



(a) Invalid path, redifing order of critical sections (b) Serializing events respecting order of critical sections

Figure 4.28: Implicit precedence of critical sections

belonging to different critical sections are explicitly connected (only one linking variable is used) all events on dashed path must be taken into account. More precisely we have to check there is no event that kills the linking variable.

To filter out these incorrect paths we add one more check called $atomicKill(\mathcal{P})$. It checks if there does not exist any kill (i.e., redefinition) of linking variable between two subsequent events of the path e_k, e_{k+1} that are executed by different threads and protected by a common mutex. The definition of $atomicKill(\mathcal{P})$ we provide takes into account the case of events protected by sets of mutexes (contexts). The events that should be checked are those preceding event e_k and succeed e_{k+1} and for which the set of protections is common for e_k and e_{k+1}.

Definition 4.8.3 (Predicate $atomicKill(\mathcal{P})$)

Ensures the linking variable between events belonging to separate threads and protected by a common set of mutexes is not killed by an event belonging in their serialization.

$$\forall e_k, e_{k+1} \in \mathcal{P} \text{ such that } (\mathcal{M} = cont(e_k) \sqcap cont(e_{k+1}) \neq \varnothing) \text{ then:}$$

$$\nexists e_m \text{ such that } e_m \blacktriangleleft e_k \wedge (cont(e_m) \boxplus cont(e_k)) \cap \mathcal{M} \neq \varnothing \wedge Def(e_m) = Def(e_{k+1})$$

$$\nexists e_m \text{ such that } e_{k+1} \blacktriangleleft e_m \wedge (cont(e_m) \boxplus cont(e_{k+1})) \cap \mathcal{M} \neq \varnothing \wedge Def(e_m) = Def(e_{k+1})$$

4.9 Recapitulation

In this chapter we addressed the problem of taint analysis for multithreaded programs, a representative information flow analysis widely used in vulnerability detection. We proposed an offline sliding window-based taint analysis which allows the prediction of taint propagations that could have occurred under valid serializations of the executed multithreaded program. We give hereafter an overview of our analysis and the refinements we did on taint prediction.

Online phase:

- *Unrestricted multithreaded program execution:* the program is executed without imposing any scheduling restrictions (i.e., it is not serialized) and memory accesses to both shared and thread local variables are logged.

Offline phase:

- *Slicing of log files into epochs:* such that events that were executed within a bounded time interval belong to the same or adjacent epochs.
- *Sliding window-based analysis:*
  - use a window of two consecutive epochs;
  - apply *taint prediction* on the window and create a summary that contains plausible taint propagations down to the analyzed window.
- *Taint prediction:* we use an *iterative algorithm* which allows predicting taintness propagation without enumerating all serializations of events in the analyzed window. The iterative algorithm is based on the equivalence between computing taint propagation and solving disjunctive boolean equation systems. The serializations inferred by the predictive algorithm can account for the memory model. We applied it to *sequential consistency* and made the following refinements:
  - safely untainting variables;
  - taking lock synchronizations into account.

Comparison to existing work

Existing works on dynamic information flow tracking of multithreaded programs force them to execute sequentially. This allows to apply typical dynamic information flow analyses as in the case of sequential programs. This somehow naive approach penalizes execution time of analyzed application but also eliminates the effects of weak memory models and simultaneous memory accesses. Moreover the analysis results are restricted to the serialized execution.

To the best of our knowledge the only works addressing the problem of dynamic information flow for parallel executions of multithreaded programs are those of Ganai et

al. [GLG12] and Goodstein et al. [GVC+10]. We introduced these works in section 4.3.2. Having fully presented our work allows a closer comparison.

The work of [GLG12] is closer to ours since (i) it does not need specialized hardware and (ii) has an offline prediction phase for taint propagation between threads. The goal of their offline prediction phase is slightly different from ours since they try to predict the effect of different operating system schedules while we target mostly into predicting the effect for plausible serializations of the executed schedule.

A first point of comparison is on the runtime execution and information logged. In [GLG12] they perform thread local dynamic information flow at execution time and only log information on accesses to shared variables. When logging a write to a shared variable they also include the locally computed taintness such that it can be used for offline propagation. This choice leads to more concise logs but less precise predictions since not all propagations can be re-calculated offline.

The second point of comparison is the prediction algorithm. As mentioned above their logs are less precise thus taint propagations cannot be computed offline. Instead, they simply propagate taintness if there exists a write of a shared variable with a tainted value and a read of this variable by different threads. The order in which they were logged does not matter. Finally, they do not take untainting into account since they assume all interleaving of events to be plausible, at least in the implementation of $\text{DTAM}_{parallel}$. In the implementation of DTAM_{hybrid} they refine the propagation by taking into account happens before relations.

Regarding the work of Goodstein et al. [GVC+10] it is based on a specialized architecture and performs online prediction for usage in the context of lifeguards. The specialized architecture produces synchronization barriers among the cores. This architectural support is required to switch between program execution and program analysis. In addition it forms bounded blocks of code executed in parallel. This corresponds to the notion of epochs we obtain by slicing offline the logs obtained by a parallel execution.

Their prediction mechanism is similar to ours [1] in that it uses a sliding window (consisting of three epochs instead of two in our case) and constructs summaries to capture the effect of inferred serializations. Their prediction methodology focuses on implementing classic dataflow analyses (e.g., available expressions) and thus consists into analyzing blocks independently and propagating necessary information between blocks. Concerning taint analysis, they state it is not straight forward to implement in their framework and give some elements on how to proceed. They provide, as we do, a taint prediction for a completely relaxed memory model and for sequential consistency. They also account for untainting to reduce false positives. Comparing the accuracy in taint prediction we have the benefit of taking synchronization mechanisms into account which they do not.

[1] our work is inspired from [GVC+10]

Chapter 5

Implementation and experimentation

In this chapter we present the tools implemented and experimentations conducted to validate the theory presented in chapters 3 and 4. We start with the optimization of critical sections where we compare the *policies* detailed in section 3.5, as well as the types of *protections* which we developed in a library. Next, we present a tool-chain for performing *predictive taint analysis* under sequential consistency as described in section 4.7.

5.1 Optimizing critical sections

In this section we present several experimentations which allowed us to compare the performance of policies described in section 3.5 and of different types of protections presented in section 3.4. Prior to presenting the experimentation we introduce a library for managing protections we implemented.

5.1.1 Library for managing protections

We have developed a library for managing protections in *C++*. Our library is capable of handling all three types of protections described in section 3.4. Its main characteristics are the following:

- re-entrant acquisition of protections. This implies that if a thread requests a protection it already possesses it wont get blocked.
- atomic acquisition of protections. This implies that either all protections in the requested set will be obtained (if they are available) or none (if any of the protections is not available) resulting into blocking the thread.

The atomic acquisition is extremely useful when a total order on variables ($<_{\mathcal{V}}$) cannot be established (e.g., in case of dynamic memory allocations). The policies that can be

used in such a case are *Global* and *Eager* which do not require the definition of predicate $Prefix()$ which relies on $<_{\mathcal{V}}$.

The main components of the library are the classes implementing each protection type (`Mutex`, `ReadWrite`, `WriteIntend`) and the `ProtectionManager` (`PM`) which manages the protections at runtime. The library uses common data structures provided by the *Standard Template Library* (*STL*) and synchronization mechanisms provided by the *Pthreads* library.

The implementation of each protection resides on using sets which hold *thread identifiers* and ensures the exclusion restrictions presented in table 3.1 on page 39 are respected. For instance the implementation of *read/write* class contains two sets one for *readers* and one for *writers*. Listing 5.1 below shows a snippet of the read/write protection implementation focusing on how to test if the read protection is available and what are the updates in case of acquisition and release.

```
1 class ReadWrite : public Protection{
2   private:
3     set<int> readers;
4     set<int> writers;
5
6   pubic:
7     bool testAcquireRead( int thread_id ){
8       if( writers.empty() ||
9           writers.count( thread_id ) == 1 )
10        return true;
11      return false;
12    }
13    bool AcquireRead(int thread_id ){
14      readers.insert( thread_id );
15      return true;
16    }
17    bool ReleaseRead(int thread_id ){
18      readers.erase( thread_id );
19      return true;
20    }
21    ⋮
22 };
```

Listing 5.1: Implementation of Read/Write protection

The `ProtectionManager` (`PM`) is the most important component since it is responsible of managing the protections i.e., **update** the shared data structure containing the state of protections and **blocking** the threads when needed. Protections of supported types can be added and removed at runtime from the `PM`. The `PM` is actually a *global variable* accessible to all threads. Each thread can initiate an acquisition/release of protections through a call to `serve(REQ)`. `REQ` is a set of requests the `PM` serves atomically. Each

request specifies (i) the protection to be accessed (ii) if it is an acquire or release request and (iii) the type of exclusion (depending on the type of the protection). Listing 5.2

```
1 class ProtectionManager{
2   private:
3     pthread_mutex_t MAN_MUTEX;
4     pthread_cond_t  COND;
5     list<Protection*> PROT;
6   public:
7     bool serve(set<Request> REQ);
8     ⋮
9 };
```

Listing 5.2: ProtectionManager snippet.

The `ProtectionManager` keeps all protections in a *linked list* (`PROT`) which is protected by a *Pthread mutex* (`MAN_MUTEX`). Moreover a condition variable (`COND`) is used to notify blocked threads of modifications in the `PM`. Figure 5.1 illustrates an activity diagram for function `serve(REQ)`. The execution cycle of the function is as follows. First the lock on `PROT` must be acquired. Then a *test* is performed to see if all requests in `REQ` can be served. If so (i) the update of appropriate protections is performed (ii) the lock is released and (iii) threads waiting on `COND` are signaled the state of `PM` has been modified. If the test was not successful then the thread releases the lock and gets blocked.

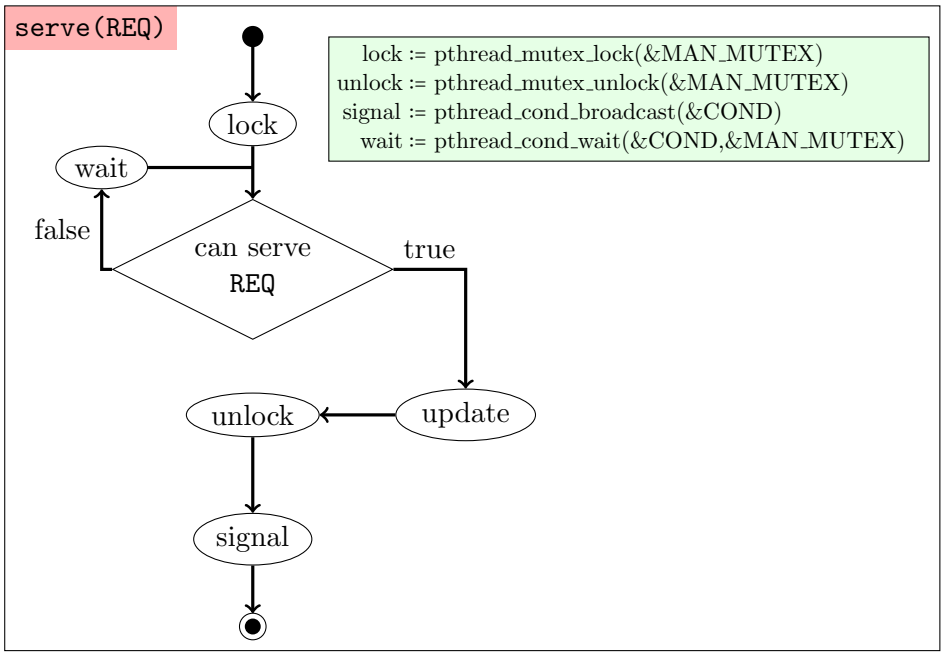


Figure 5.1: Activity diagram for serve function.

5.1.2 Experimentations

The experimentations we present hereafter are based on handcrafted applications which exhibit the benefits of policies (see section 3.5) and protections (see section 3.4). Moreover, we point up the gains in using atomic acquisition of sets of variables when using our library presented in section 5.1.1.

Description of applications

We describe hereafter the applications we use in our experimentations. Applications 1 and 2 compute the average value of nodes structured in different topologies while application 3 simulates communication in a tree-like topology.

Application 1 and 2

These applications compute in parallel the average value of a set of nodes organized in two types of data structures: a circular list and a two dimensional torus (2D-torus). In our case each node is an integer variable. We assume that a protection is assigned to each node and also a thread is spawned per node.

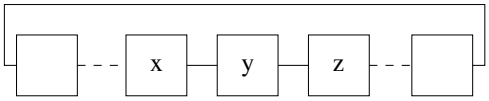
The parallel computation of the average is performed as follows. The nodes are split into small intersecting clusters (clustering depends on the nodes structuring). Each cluster has a *master node* which atomically computes the average value of the cluster and then updates all nodes with the computed value. The intersecting nodes propagate the clusters average to neighboring clusters. The computation ends when all clusters have stabilized to the same value which is the average of all nodes.

For the needs of our experimentations we slightly modify the algorithm described above. The modification consist in: (i) making different executions more comparable and (ii) amplifying the effect of synchronization mechanism.For the first point (i) despite keeping the same clusterings and initial values of nodes, how fast will the average stabilize depends on the communication order as well as rounding happening on divisions etc. To solve this problem we decided to fix the number of computations of each cluster (i.e., each cluster will compute the average value for a fixed number of times independently if algorithm stabilized or not). For the second point (ii) the problem is that computations are very simple and the gains of each synchronization mechanism or policy are hard to observe. To overcome this problem we added some overhead in memory accesses and computation in the form of `ackermann` [Sun71] computations. We chose to add `ackermann` since it is computation intensive that is it consumes *CPU* cycles and hence the thread cannot be preempted as would be the case of a `sleep` call for instance.

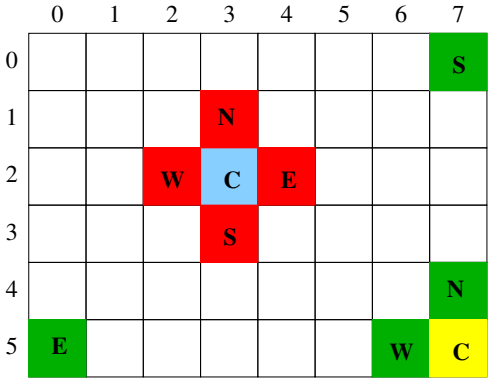
Application 1 illustrated in Figure 5.2(a) is a circular list. The cluster is defined as three consecutive nodes. Each node of the structure is the *master node* of the cluster consisting of itself and its left and right neighbor. The pseudo-code for critical section executed by threads associated to master nodes is illustrated in Listing 5.3. The pseudo-code is in accordance with Figure 5.2(a) which focuses on a cluster consisting of nodes `x`, `y`, `z`.

```
1 critical{
2   // a,b,c,ave : local variables
3   // x,y,z     : shared variables
4
5   a = x;   ackermann();
6   b = y;   ackermann();
7   c = z;   ackermann();
8   ave = (a+b+c)/3;
9   x = ave; ackermann();
10  y = ave; ackermann();
11  z = ave; ackermann();
12 }
```

Listing 5.3: Critical section of circular list master node



(a) Average of list



(b) Average of 2D-torus

Figure 5.2: Average computation, Applications 1 and 2

Application 2 illustrated in Figure 5.2(b) is a two dimensional torus. The cluster is defined as five neighboring nodes. Each node of the structure is the *master node* (C) of the cluster consisting of itself and its north (N), east (E), south (S) and west (W) neighbors. The critical section executed by each thread is identical to that of application 1 presented in Listing 5.3 except that now more nodes participate. Figure 5.2(b) illustrates two clusters one in the center of the torus and one on the bottom right corner where we can see how its south and east neighbors fold in the torus.

Network communication, Application 3

Application 3 is inspired from communication algorithms used in wireless sensor networks. We consider a set of nodes deployed in a tree-like structure where *leaf nodes* try to propagate information towards the *root*. Figure 5.3 illustrates such a topology. The arrows show the direction of messages sent. As we can note a child node transmits potentially

to several parent nodes. A thread and a shared variable are associated to each node. A transmission is represented by a write of the shared variable of the receiving node. To simulate errors in the network we introduce a success probability. If transmission failed then we assume its re-transmission is always successful (i.e., a message is transmitted maximum two times towards a destination). Moreover, the transmission of messages to multiple parents is done atomically. Finally, transmission time is simulated with a *sleep*. The transmission time is used again to amplify the gains of each policy .

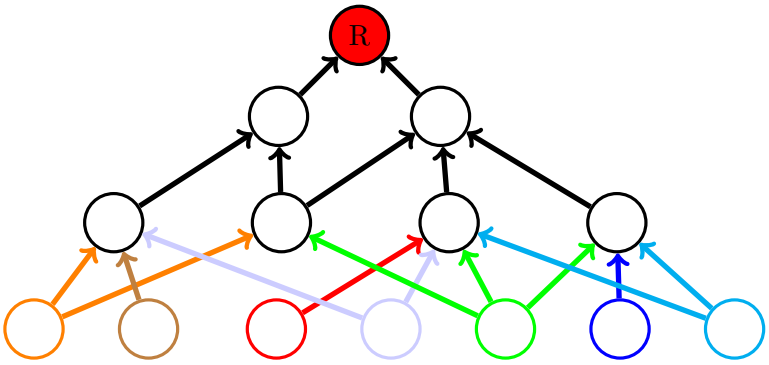


Figure 5.3: Communication in a network Application 3

The nodes behavior can be split into *leafs*, *intermediate* and *root*. The *root* node is unique and it behaves as a sink,it only receives messages from its children. Dually, the leafs are source nodes, they only transmit messages to their parents. Intermediate nodes combine both behaviors, they receive messages from their children (leaf nodes or other intermediate nodes) and forward them to their parent nodes (also intermediate nodes or the root). Listing 5.4 contains the pseudo-code for an intermediate node. It checks whether a "fresh" (i.e., not re-transmitted) message has been received, and if so it is relayed to its parents.

```
1 critical{
2   // msg is message shared variable of the node
3   if( msg.isFresh() ){
4       // transmit message to each parent
5       for( p ∈ Parents){
6         sleep( TRANSMISSION_TIME );
7         // randomly decide success of transmission
8         ok = rand(100) < SUCCESS_PROBABILITY;
9         if( ok )
10          p.msg = msg;       // copy message to parent
11        else{
12          sleep( TRANSMISSION_TIME );
13          p.msg = msg;
14        }
15    }
16  }
17 }
```

Listing 5.4: Critical section of intermediate node in application 3

5.1.3 Experimental results

In this section we present the experimental results obtained using the applications described in section 5.1.2. We compare the policies described in section 3.5 using mutexes provided by *Pthreads* but also our implementation described in section 5.1.1. We also made a comparison of *protection types* presented in section 3.4.

All experimentations were performed on a quad-core *Intel Xeon W3520* with 6GB of ram sunning *Debian GNU/Linux 5.0.8 (lenny)*. The code was compiled using *GCC 4.3.2* without any optimization flags. The execution platform used does not strongly affect our observations. The gains in execution time should be observable on any parallel execution platform.

Comparison of policies

For each application presented in section 5.1.2 above we have compared the first four policies using *fast mutexes* provided by *Pthreads* and mutexes implemented as protections in our library. Policy *Incremetal/Priority release* is excluded from the experimentations because in all cases it was equivalent to policy *Incremental/Eager*. This is due to the pattern of access to the shared variables as discussed in section 3.6.1. We detail hereafter how the applications were instantiated.

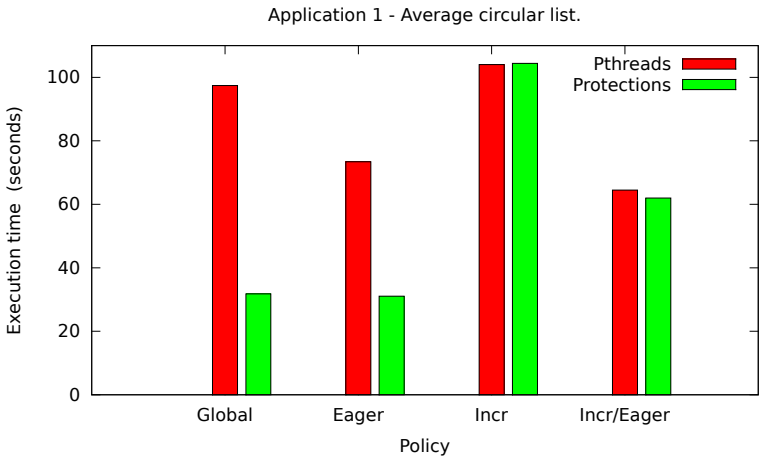
Application 1 (average circular list) we used 20 nodes connected in a circular list topology and each had to complete 20 average computations as detailed in its description. The `ackermann` function was fed with argument values 3, 10 (execution time on the specified platform ≈1sec).

Application 2 (average 2D-torus) we considered a 2D-torus of size 10×10 where each node had to perform 5 average computations. Again memory accesses were simulated by `ackermann` with values 3, 10.

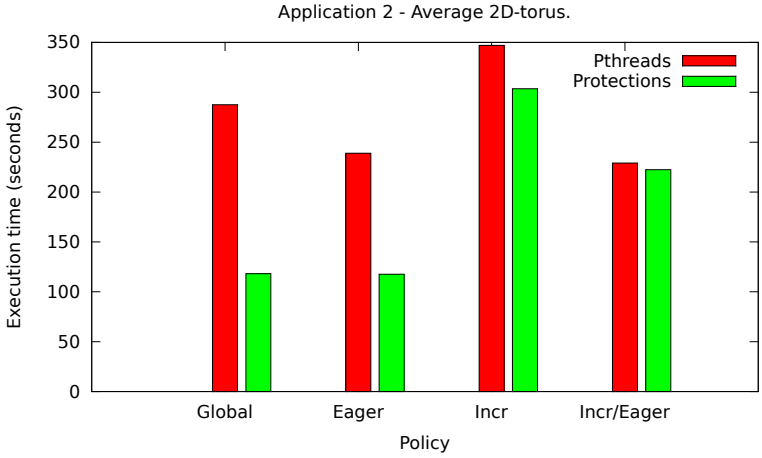
Application 3 (network communication) the topology consisted of 70 nodes distributed in 6 levels (root=1 level, leafs=1 level, intermediate=4 levels). Intermediate and leaf nodes had at most 4 parents. The termination condition for the experimentation was the root to receive 50 messages. Delay for message transmissions was set to 1sec (`TRANSMISSION_TIME = 1000;` in milliseconds) while the success probability to 80% (`SUCCESS_PROBABILITY = 80;`)

Figure 5.4 provides the plots of the execution times obtained for each application. The plots are composed by the histograms of the four policies compared, each implemented with *Pthread* and *Protection* (our implemented library) mutexes.

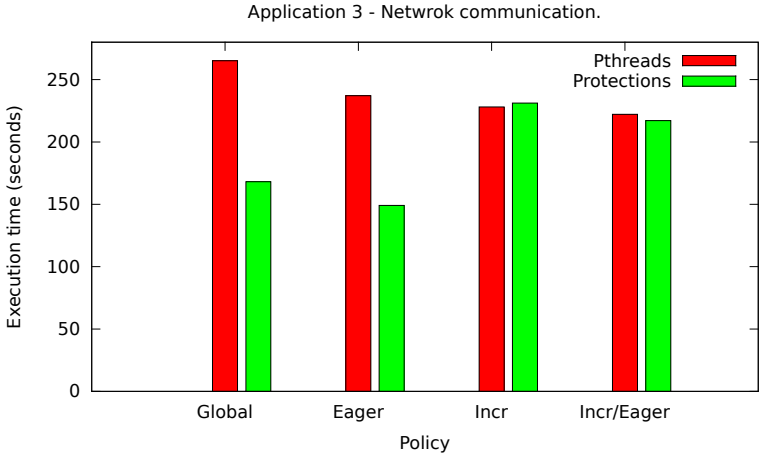
As we can observe in all cases, the *Eager release* policies perform better than the *Global* policy. This is due to releasing earlier protections which allows other threads to progress. Policy strictly *Incremental* is not efficient for applications 1 and 2. The cause of such a reduced parallelism is that each thread obtains a subset of the required protections not sufficient for completing the critical section, resulting in having a small number of threads



(a) Application 1 - circular list average



(b) Application 2 - 2D torus average



(c) Application 3 - network communication

Figure 5.4: Comparison of policies and implementation of mutexes

capable to execute in parallel. Figure 5.5 illustrates the case for application 1. The arrows denote the origin node has obtained the protection on the destination node. The dashed arrows are blocked nodes and as we can see just one node is capable of progressing.

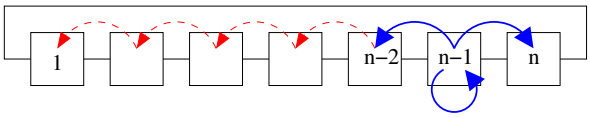


Figure 5.5: Problem of *incremental* policy for application 1

Another observation is that our implementation of mutexes reduces drastically the execution times with policies *global* and *eager*. This is due to the *atomic* acquisition of sets of variables as presented in section 5.1.1. Since in these two policies all protections are obtained at the beginning of the *critical section* the threads will never get blocked in the critical section waiting for a protection to become available.

To get a deeper insight of the policies behavior we monitored closely the execution of *application* 1 using *Pthreads* fast mutexes. The results are presented in Figure 5.6 and 5.7 which contain the execution time-line of each policy. On the vertical axis we have the number of threads while on the horizontal the time progress (in milliseconds). The different colors indicate the status of threads at a given time following this terminology:

In CS: the thread has entered its critical section. This implies it has obtained at least one of the protections required and executed an instruction on a protected variable.

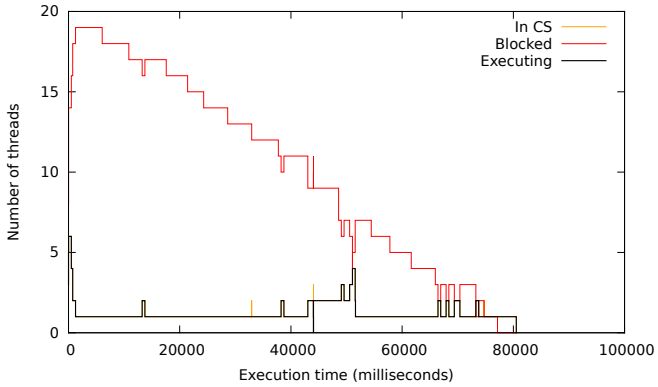
Blocked: the thread is blocked waiting on a protection. A thread can be blocked outside a critical section, if no protected instruction of the critical section was executed, or inside if it gets blocked after a protected instruction of the critical section was executed.

Executing: the thread is in the critical section and not blocked.

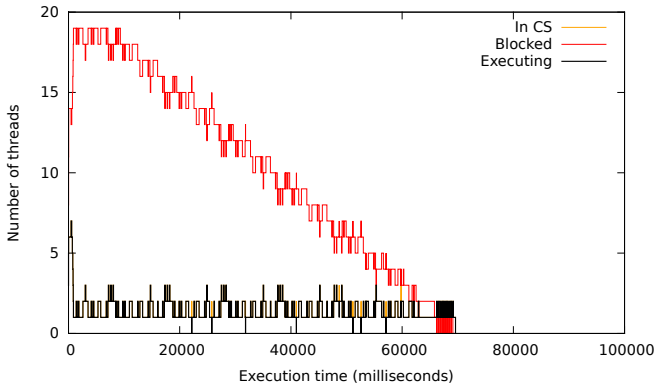
In all time-lines we observe the decrease of number of threads which is due to their termination. Furthermore since the same time scale has been used we can easily compare the execution times.

For policies *global* and *eager* the curves *In CS* and *Executing* are complementary. This is expected since in these policies all protections must be acquired prior to executing the critical section. Comparing these two, we note that *eager* policy exploits better parallelism since in general we have two threads executing.

For the incremental policies we observe a burst of parallelism which rapidly turns int a huge number of *blocked* threads. This is the behavior described in Figure 5.5 where the burst maps to most threads obtaining their left neighbor simultaneously and hence reading in parallel its value. The switch to blocked state comes right after when each node tries to obtain its own protection which is reserved by its right neighbor. Despite this phenomenon as we can note policy *Incremental/Eager* recovers rather fast and reaches rather often the parallel execution of four threads.



(a) Global policy

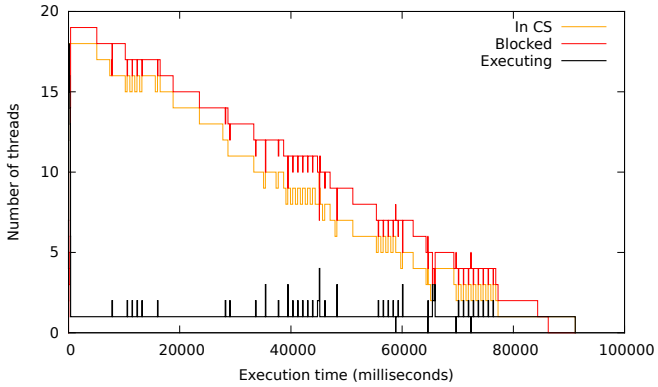


(b) Eager policy

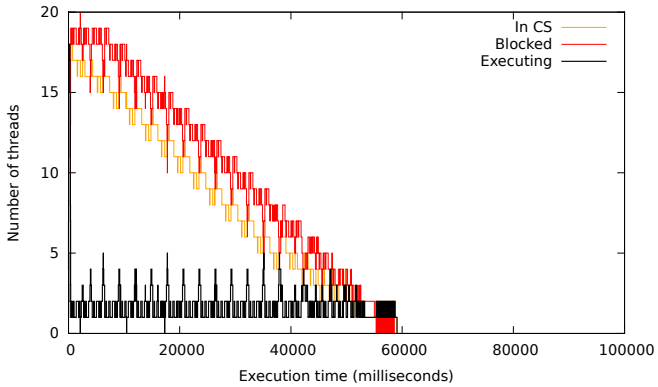
Figure 5.6: Execution of policies *global* and *eager*

Comparison of protections

The comparison between protections is made on a modified version of application 2 (average computation of a 2D-torus). The example is specially crafted such as gains of finer protections can be observed. Listing 5.5 details the critical section of a master node the figure next to it illustrates the accesses. In this version of application 2 (which no longer computes the actual average value) the neighbors accesses are the *north* and *west* and their values are uniquely read. The only variable written is that associated to the master node. Once again accesses to variables are amplified with the `ackermann` function. An *extra* `ackermann` computation has been added which prolongs the holding of write protection on master node (`C`). The extra computation promotes the write intend protection.



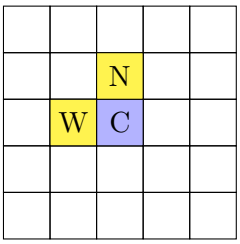
(a) Incremental policy



(b) Incremental/Eager policy

Figure 5.7: Execution of policies *incremental* and *incremental/priority release*

```
1  critical{
2    // x,y,z,ave : local variables
3    // C,N,W     : shared variables
4
5    x = N;   ackermann();
6    y = W;   ackermann();
7    z = C;   ackermann();
8    ave = (x+y+z)/3;
9    ackermann();  // Extra computation
10   C = ave; ackermann();
11 }
```



Listing 5.5: Critical section of modified 2D-torus average

The modified version of 2D-torus was executed on a torus of size 10×10 where each node had to execute its critical section presented in Listing 5.5 five times. The `ackermann` function was again fed with values 3, 10. Table 5.1 presents the results for each protection type. As we can note the more flexible *read/write* and *write intend* protections can bring significant gains. Of course, the example was specially crafted. Obtaining such impressive gains by using these protections are not straight forward.

$Mutex$	$ReadWrite$	$WriteIntend$
$8m49s$	$7m18s$	$6m26s$

Table 5.1: Execution times per protection

5.2 Offline predictive information flow

5.2.1 Proof of concept tool

As a proof-of-concept, we implemented a tool chain for our offline taint prediction analysis. The tool chain, presented in Figure 5.8, is split into an instrumentation phase (left side of figure) and execution and analysis (on the right side). First, the source files are instrumented to produce the log files. Next, the program is executed and log files are generated. The log files are sliced into arbitrarily sized epochs and taint analysis is performed as described in section 4.7.

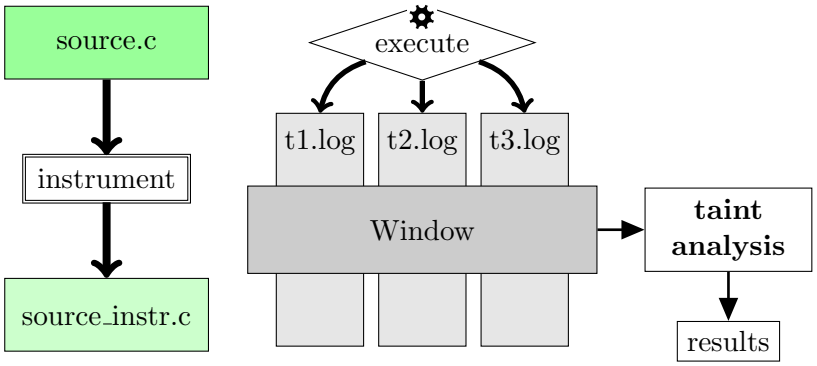


Figure 5.8: Abstract analysis framework

Source code instrumentation

The code instrumentation is implemented using the CETUS framework [DBM+09]. It is a C source-to-source compiler written in *Java* which provides some interfaces for analyzing and transforming the parsed C code. The instrumentation process consists in adding explicit logging instructions which record time-stamped information on used/defined variables of assignments. Special attention is given to keep track of variables passed as arguments into function calls and return values. Function calls related to mutex locking and un-locking are treated specially. For this proof of concept implementation not the entire ANSI C language can be instrumented. The limitations are purely syntactical and do not limit the analysis framework.

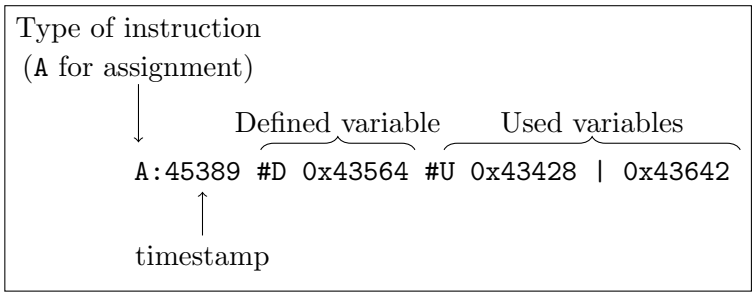
Each thread is logged in a dedicated file, so there is no need to synchronize logging instructions and thus we do not perturb much the applications execution. The time-stamping of log entries is in micro-seconds, relative to the beginning of the programs execution. The information carried by a log entry depends on the underlying instruction. In general it contains the set of *used* variables (actually their addresses on memory) and

the *defined* variable (if it exists). Hereafter we sketch the instrumentation of some basic instruction types:

- *Assignments:* the *defined* and *used* variables are clearly stated. In the right hand side of assignments there should be no function calls. For instance, the instrumentation of the assignment `x=y+z;` results into:

```
x=y+z;
fprintf(LFP,"A:%d #D %p #U %p | %p ", GET_TIME(),&x,&y,&z);
```

At execution time the log entry produced would look like that:



- *Functions:* passing arguments by value hides the dependency between the variables affected by the argument and the variable the function was called with. To overcome this problem we augment each function by adding a `void*` argument per variable in the argument list. Moreover inside the function we add a *dummy* assignment that will link the variable given as argument with the variable used inside the functions code during the offline analysis. Here is an example:

```
void f(int a){
  ...

void f(int a,void* aPT){
   fprintf(LFP,"A:%d #D %p #U %p ", GET_TIME(),&a,aPT);
   ...
```

The logging of function calls stores the time they were called, the function name and a set of variables that were passed as arguments. The only function that is time-stamped differently is `pthread_mutex_lock(..)` which timestamps the return of the function. This corresponds to the time the lock was obtained.

Log processing

Analyzing the log-files is divided into two phases. First, a slicer is used to set explicit *epoch boundaries* in the log-files. As mentioned in section 4.4.2 we use a time-based slicing. Second, the sliced log files are parsed and analyzed.

The parsing and main skeleton of the analysis are generic. They have been implemented in *Java* and are easily extendable. The parsing consists in reading from the log files time-stamped sets of used and defined variables. We read an epoch at time and feed the sliding

window analysis with it. This implies that the log files do not need to be read entirely in memory to perform the analysis. The skeleton of the analysis (i.e., the *Vertical* and *Horizontal* passes) have been interfaced such that other analyses that can benefit from that structure can be easily implemented. Notably, one would implement a new analysis by re-defining: (i) what information is held in the summary of an epoch (ii) how the analyzed property is propagated and (iii) how it is summarized.Finally, the framework also provides a lock-set analysis which identifies calls to library functions `pthread_mutex_lock` and `pthread_mutex_unlock` and encodes them as *protections* presented in section 4.8. This step should be performed *a priori* on the whole log files, since bounds of critical section may spawn over several windows.

At this time, the taint analyzer implementation makes no distinction between *strongly* and *weakly* tainted variables and can compute three types of taint propagation:

relaxed all interleaving of events in the window are accepted and kills are not taken into account;

sequential only sequentially consistent propagations are taken into account and variables killed are excluded from the summarization;

synchronized extends sequential propagation such that limitations introduced by locks are taken into account.

Visualizing taint propagations

Our tool is also capable of producing a representation of taint propagations in analyzed windows. We provide hereafter the visualization produced, during the analysis of a window with the three different types of analysis. The analysis has been slightly modified into that we produced all tainting paths that correspond to event $(18, 111, 5)$ in block $(18, 111)$. Each path is illustrated with a different color. Figure 5.9 illustrates the paths under *relaxed* analysis, Figure 5.10 under sequential and Figure 5.11 under synchronized.

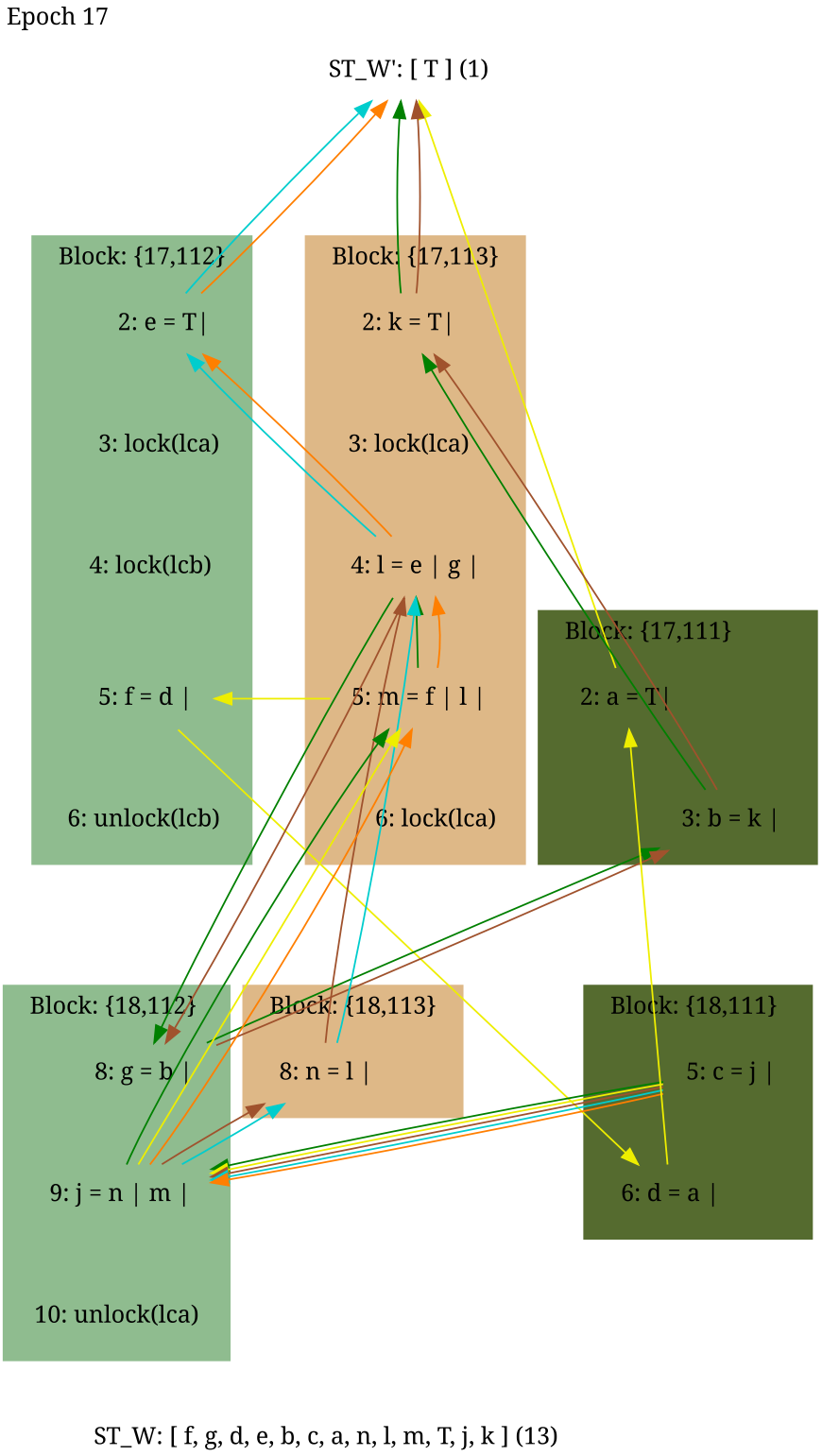


Figure 5.9: Tainting paths for $(18, 111, 5)$ under *relaxed* analysis

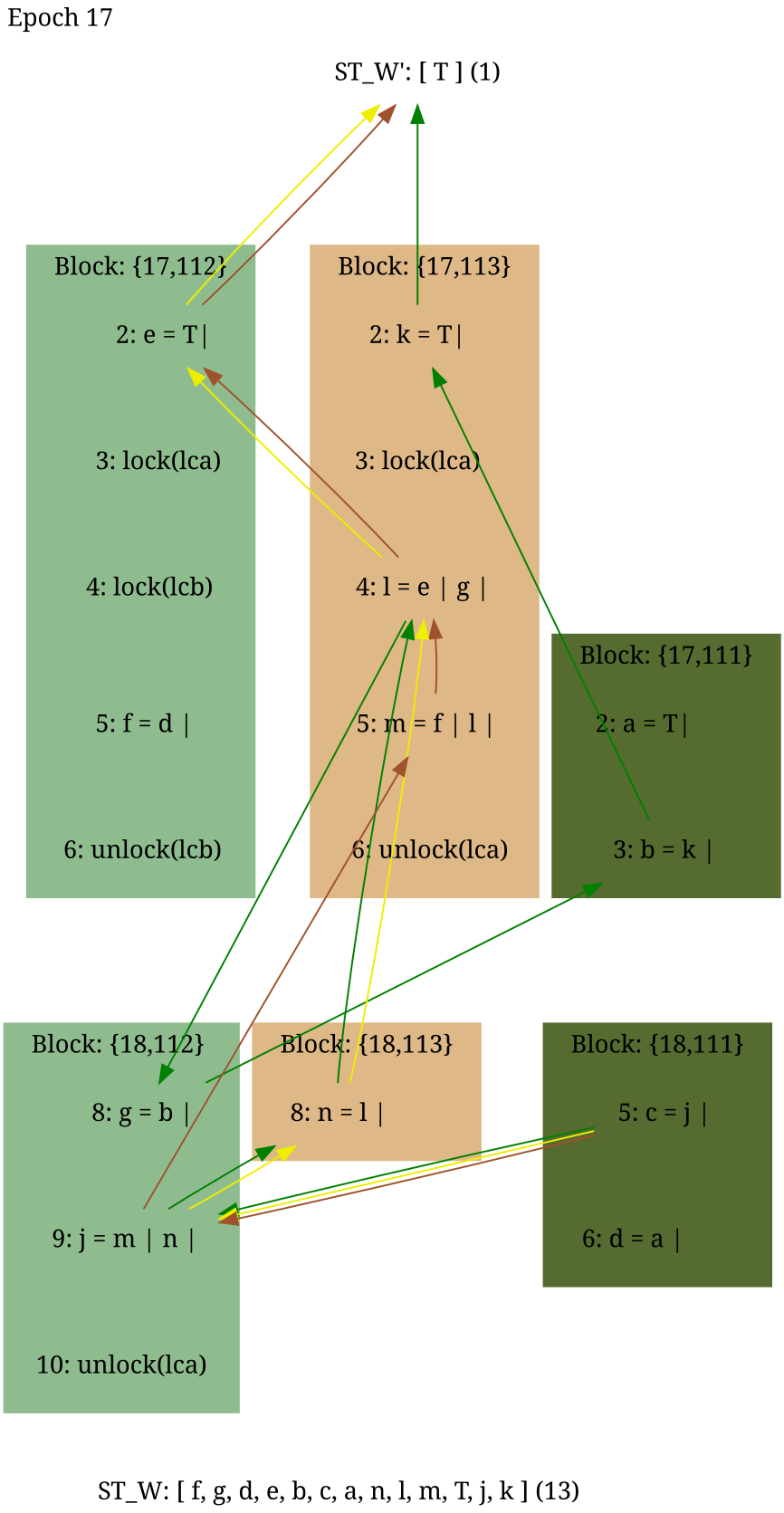


Figure 5.10: Tainting paths for $(18, 111, 5)$ under *sequential* analysis

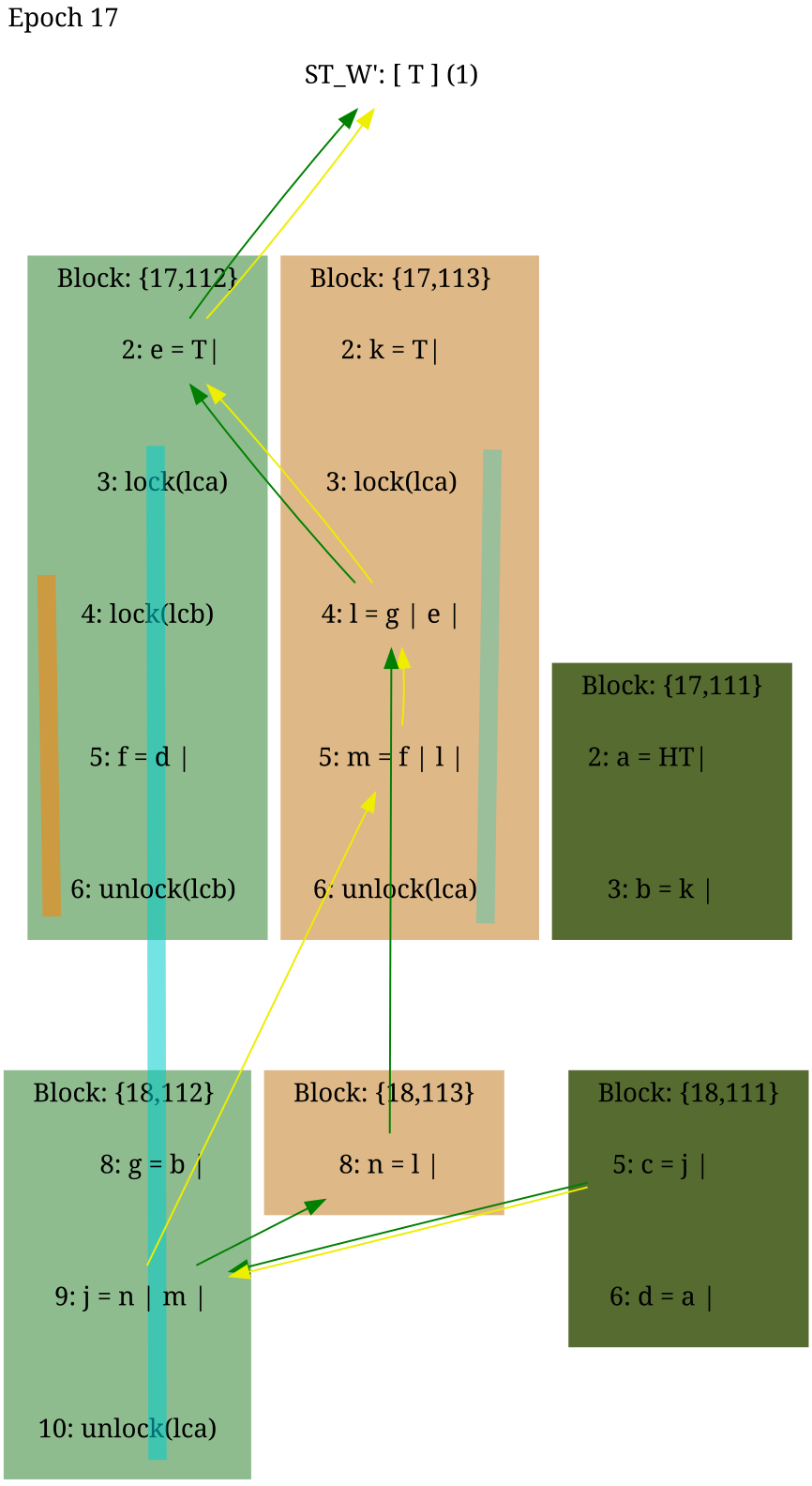


Figure 5.11: Tainting paths for $(18, 111, 5)$ under *synchronization* analysis

5.2.2 Some experimental results

To validate the framework and the analysis mechanics we initially used some toy examples (traceable manually), but rich enough to produce interesting behaviors especially regarding the epoch size. We illustrate those findings hereafter. Next,we applied the analysis on a bigger handcrafted example. In both cases, for simplicity the shared variables used are integers. The *printing* of a tainted variable is the malicious behavior we want to detect. The experimentations we carried out demonstrate: (i) the effect of epoch size on the accuracy of the analysis (ii) the reduction of false positives due to mutex support .

Thread A	Thread B	Thread C
1 `n=rand();`	1 `n=rand();`	1 `n=rand();`
2 `ack(n);`	2 `ack(n);`	2 `ack(n);`
3 `X=TAINT;`	3 `X=0;`	3 `print(X);`

The example above illustrates the code to be executed by distinctive threads. The function `ack` called is an *Ackermann* computation (time consuming) which adds non determinism in the order in which each thread executes its third instruction. Hereafter is the log produced by a parallel execution of the above program. During this execution the physical address corresponding to `X` was `&X=0x8b4` and the printed value was 0 (i.e., `X` was untainted).

thread A.log

```
A: 615 #D 0xb77     #U rand
F: 780 #F ack       #U 0xb77
A: 858 #D 0x8b4     #U 0x84f
```

thread B.log

```
A: 814  #D 0xb6f     #U rand
F: 878  #F ack       #U 0xb6f
A: 1108 #D 0x8b4     #U
```

thread C.log

```
A: 677  #D 0x24f     #U rand
F: 840  #F ack       #U 0x24f
F: 1752 #F print     #U 0x8b4
```

As we can observe in the log, Thread A taints shared variable `X` at time 858 (microseconds since the program started). Further in the execution at time 1108 Thread B un-taints `X` and consequently Thread C prints it at time 1752.

Figure 5.12 illustrates analyzing the log using two periodic partitionings. The solid and dashed horizontal lines denote the limit of epochs. Above each line/epoch we note the set of tainted variables observed by the analysis when entering that epoch. Using the left partitioning (bigger epoch size) an error is detected, since instruction `print(X)` of Thread C can precede the un-taint instruction of Thread B. On the contrary with partitioning used

on right side (small epochs size) the printing is considered safe as based on the interleaving assumptions it cannot precede the un-tainting.

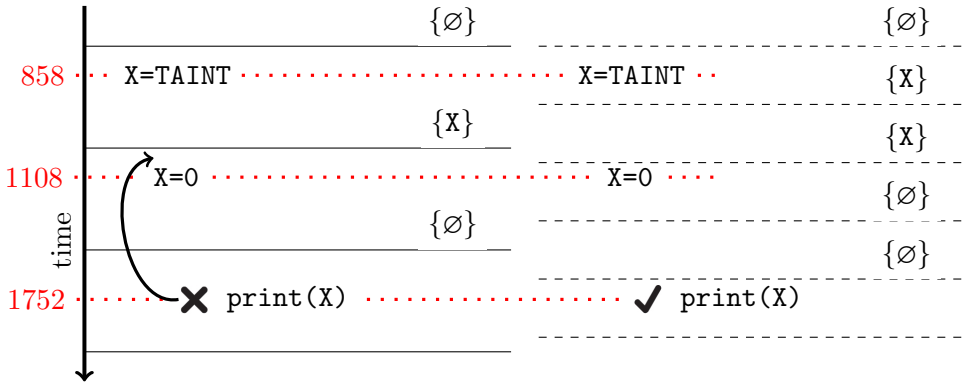


Figure 5.12: Cutting of epochs

The more complex hand-crafted example consists of a shared array of five elements which is randomly accessed by five threads. Each access is either: (i) an update with a random value (ii) an explicit taint (iii) an untaint operation followed by a print (iv) an update using another element of the array (to create longer taint dependency paths).Two variations of the application were tested. In the first one accesses are not synchronized and thus data races are likely to arise resulting into printing tainted variables. In the second, all accesses are protected using a mutex per element of the array. In this case, no errors occur since un-tainting and printing of an element are atomic. All executions are performed on a machine with 4 cores.

Simply executing the version without synchronization reveals some errors showing that tainted values can be printed. As expected, applying our analysis allows to find *more errors*. Table 5.2 presents how the size of the epoch chosen for the analysis affects the number of errors found. The second column of the table displays the number of errors observed at execution time per array element. The last columns display the number of errors detected by our analysis depending on the epoch size. As we can note, increasing this size increases the number of errors found by the analysis. Conversely, reducing too much the epoch size leads to *false negatives*, i.e., real errors are missed for epochs of size 1 μseconds (the number of errors detected by our analysis is smaller than the ones detected at runtime).

Node	Runtime errors	Epoch size in μ sec				
		100	50	20	10	1
0	1	8	6	2	2	0
1	2	11	7	4	2	1
2	0	9	3	0	0	0
3	1	5	3	1	1	1
4	0	5	1	0	0	0

Table 5.2: Errors found vs epoch size

In the example with mutex synchronization no errors occur at runtime. Table 5.3 displays the number of errors detected by our analysis depending on the epoch size. Without

	100	50	20	10
locks ignored	25	12	2	0
locks into account	4	0	0	0

Table 5.3: Using lock information in the analysis

taking mutex into account plenty of errors are found. When mutex synchronization restrictions are applied by the analysis the number of reported errors reduces but still they correspond to *false positives*, that can be eliminated by considering the diagnostics produced by the analysis. On these examples: (i) executing the instrumented version of the application introduces an overhead of about 50% (ii) log file analysis takes less than 1 second on a Intel i3 CPU @2.4GHz with 3GB of RAM.

Chapter 6

Conclusion and perspectives

In this work we addressed two topics related to the parallel execution of multi-threaded programs using shared memory. The first topic regarded the efficient implementation of *critical sections* using the *pessimistic* approach, i.e., using locks. The second topic was related to *information flow*, more precisely on predictive taint analysis. We summarize hereafter the work presented on each topic and provide some perspectives for each.

6.1 Optimizing critical sections

In this part we proposed a generic algorithm for obtaining/releasing protections necessary to provide serialization of critical sections. We provided five acquisition/release policies that guarantee serializability of critical sections and deadlock freedom. We conducted experiments showing that the policy called *global*, the one traditionally used, performed worse than policy *eager release* on all experimentations. Moreover, we implemented a library for the management of protections which allows the atomic acquisition of a set of protections. Using this library we obtained significant performance gains compared to the Posix library.

Perspectives for optimization of critical sections

Several perspectives are envisioned for this work. First, the experimentations should be extended on more realistic examples, e.g., apply to *PARSEC* benchmark [Bie11]. This would allow to strengthen the observation that the *global* policy is in general not the most performing. Making this step further necessitates automatic transformation of critical sections into lock protected portions of code. Initially, user annotations (as in [MZGB06]) could be used to specify the policy to be applied on the critical section while the locks necessary to execute each statement would be inferred automatically using one of the existing analyses in the literature. Taking this further, the selection of the most profitable policy could be automatically inferred by identifying code patterns that are more profitable for a specific policy.

Another interesting direction consists into improving the implementation of the library

for managing protections. The implementation is way from being optimal. The necessary information is stored into classic data structures and protected by a single coarse grain mutex lock. Using finer locks would allow the library to serve multiple requests in parallel. Moreover, some type of scheduling could be introduced in the processing of the requests in order to avoid *starvation*. This phenomenon can be observed if a thread requests a large number of protections which may never be available simultaneously. A straight forward solution would be to apply *FIFO* scheduling even if it reduces parallelism. Another ad-hoc solution could be to bound the number of protections to be requested atomically. This solution though gives no guarantee on avoiding starvations.

6.2 Predictive information flow analysis

In this work we focused on taint analysis, a representative information flow analysis, and proposed an efficient algorithm for *offline prediction* of taintness. During the online phase that precedes, the multithreaded program is executed without any scheduling restrictions (i.e., it is not serialized) and a log of all memory accesses is produced. Our prediction technique consists into breaking the logs into *epochs* which are then processed using a *sliding window*. Taintness is predicted locally for every window and summarized. The summary produced is used as input to the next window. The prediction algorithm uses an *iterative* method to infer taint propagations without enumerating and analyzing all plausible serializations of events in the window. We presented in details how to predict taint propagations under sequential consistency. Moreover, we included refinements to taint prediction such that untainted variables are correctly excluded from window summarizations. We further refined taint predictions by taking into account the semantics of locks. Finally, we implemented a proof of concept tool.

Perspectives for predictive information flow analysis

Identification of information flows is a central issue in all vulnerability detection tools. Existing tools either do not deal with multithreaded programs or they force them to execute sequentially. Our prediction algorithm could be incorporated in such a tool to allow extend the verdicts to a set of plausible serializations for a given parallel execution. In the context of testing it could be used in combination with a fuzzer to increase coverage and guide tests towards interesting executions. Finally, as observed by the experimentations, epoch slicing strongly affects predictions. Heuristics could be proposed for defining the minimum epoch size, or to indicate interesting events to use as slicing points.

The tool developed can be considerably improved. First, the current implementation of the source to source transformation could be extended such that more complex *C* programs can be processed. Also a more interactive interface for the analysis of windows would make back tracking of information flows more comfortable. Notably we could implement the distinction between *strong* and *weak* taintness. This would allow to exhibit a concrete trace, spanning over several windows, that propagates taintness to variables designated as *strong*.

Finally, a promising direction would be to use some dynamic binary instrumentation framework for generating the log files. This would unleash the restrictions on input programs. It would also allow producing much finer logs, since we would log events at the assembly level and not the source code level as we do now. At this level of logging it makes sense to consider other memory consistency models such as *TSO*.

Appendix A

Boolean equation systems

Hereafter we provide some background information on *boolean equation systems* (BES). The definitions and notations we introduce are from [Kei05, GK04] and are standard in the literature.

Definition A.1.1 (Boolean expression [Kei05])

Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ be a set of boolean variables. The set of boolean expressions over $\mathcal{X}$ is denoted by $B(\mathcal{X})$ and is given by the grammar:

$$\alpha ::= \bot \mid \top \mid x_i \mid \alpha \wedge \alpha \mid \alpha \vee \alpha$$

where $\bot$ stands for false, *$\top$ stands for* true *and $x_i \in \mathcal{X}$*

Definition A.1.2 (Syntax of boolean equation system [Kei05])

A boolean equation system $\mathcal{E}$ is of the form $\sigma_i x_i = \alpha_i$ where $\sigma_i \in \{\mu, \nu\}$, $x_i \in \mathcal{X}$ and $\alpha_i \in B(\mathcal{X})$

$$\mathcal{E} \equiv (\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2)\ldots(\sigma_n x_n = \alpha_n)$$

Note that:

- *all left hand sides of the equations are different*
- *all variables in the right hand side are from $\mathcal{X}$*
- *the σ sign is μ if the equation is a least fixed point or ν if it is a greatest fixed point.*

Definition A.1.3 (Boolean equation system standard form [Kei05])

A boolean equation system $\mathcal{E}$

$$\mathcal{E} \equiv (\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2)\ldots(\sigma_n x_n = \alpha_n)$$

is in standard form if, for all $i \in [1,n]$*, the right hand side expression* α_i *is of the form* $y \circ z$ *or* y *where* $\circ \in \{\wedge, \vee\}$ *and* $y, z \in \mathcal{X} \cup \{0,1\}$

Definition A.1.4 (Boolean equation system alternation depth [Kei05])

Given a boolean equation system $\mathcal{E}$

$$\mathcal{E} \equiv (\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2)\ldots(\sigma_n x_n = \alpha_n)$$

its alternation depth is the number of variables x_i *with* $1 \leq i \leq n$ *such that* $\sigma_i \neq \sigma_{i+1}$

A boolean equation system $\mathcal{E}$ *is* alternation free *if its alternation depth is zero. That is, all equations compute the same fixed point.*

Definition A.1.5 (Variable dependency graph [GK04])

Let $\mathcal{E}$ *be a* disjunctive/conjunctive *boolean equation system*

$$\mathcal{E} \equiv (\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2)\ldots(\sigma_n x_n = \alpha_n)$$

the dependency graph of $\mathcal{E}$ *is a directed graph* $G_{\mathcal{E}} = (V, E, \ell)$ *where:*

- $V = \{i | 1 \leq n\} \cup \{\bot, \top\}$ *is the set of nodes*
- $E \subseteq V \times V$ *is the set of edges such that, for all equations* $\sigma_i x_i = \alpha_i$
  - $(i,j) \in E$ *iff a variable* x_j *occurs in* α_i
  - $(i, \bot) \in E$ *iff false occurs in* α_i
  - $(i, \top) \in E$ *iff true occurs in* α_i
  - $(\bot, \bot), (\top, \top) \in E$
- $\ell : V \to \{\mu, \nu\}$ *is the node labeling function defined by* $\ell(i) = \sigma_i$ *for* $1 \leq i \leq n$, $\ell(\bot) = \mu$, *and* $\ell(\top) = \nu$.

Lemma A.1.1 (Solution of BES implies path existence [GK04])

Let $G_{\mathcal{E}} = (V, E, \ell)$ *be the variable dependency graph of a disjunctive (respectively conjunctive) boolean equation system* $\mathcal{E}$*. Let* x_i *be any variable in* $\mathcal{E}$ *and let the valuation* v *be the solution of* $\mathcal{E}$*. Then, the following are equivalent:*

1. $v(x_i) = \top$ *(respectively* $v(x_i) = \bot$*)*
2. $\exists j \in V$ *with* $\ell(j) = \nu$ *(respectively* $\ell(j) = \mu$*) such that:*
   (a) *j is reachable from i, and*
   (b) *$G_{\mathcal{E}}$ contains a cycle of which the lowest index of a node on this cycle is j*

Bibliography

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 1996.

[BH00] Bryan Buck and Jeffrey K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, November 2000.

[Bie11] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton, 2011.

[BKK10] Ulrich Bayer, Engin Kirda, and Christopher Kruegel. Improving the efficiency of dynamic malware analysis. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1871–1878, New York, NY, USA, 2010. ACM.

[BRRS10] Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. Security of multithreaded programs by compilation. *ACM Trans. Inf. Syst. Secur.*, 13(3):21:1–21:32, July 2010.

[Bru04] Derek L. Bruening. Efficient, transparent and comprehensive runtime code manipulation. Technical report, 2004.

[CCG08] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 304–315, New York, NY, USA, 2008. ACM.

[CES71] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, June 1971.

[CFC12] Maria Castillo, Federico Farina, and Alberto Cordoba. A dynamic deadlock detection/resolution algorithm with linear message complexity. In *Proceedings of the 2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, PDP '12, pages 175–179, Washington, DC, USA, 2012. IEEE Computer Society.

[CKS+08] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *Proceedings of the 35th Annual International*

Symposium on Computer Architecture, ISCA '08, pages 377–388, Washington, DC, USA, 2008. IEEE Computer Society.

[CLO07] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.

[CM09] Maximiliano Cristia and Pablo Mata. Runtime enforcement of noninterference by duplicating processes and their memories. In *WSEGI*, 2009.

[CMC+06] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The atomos transactional programming language. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 1–13, New York, NY, USA, 2006. ACM.

[CZYH06] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Proceedings of the 11th IEEE Symposium on Computers and Communications*, ISCC '06, pages 749–754, Washington, DC, USA, 2006. IEEE Computer Society.

[DBM+09] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42:36–42, 2009.

[DKK07] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 482–493, New York, NY, USA, 2007. ACM.

[DM06] Bruno Dutertre and Leonardo De Moura. The yices smt solver. Technical report, 2006.

[DSS06] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 20th international conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[EA03] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[EFJM07] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 291–296, New York, NY, USA, 2007. ACM.

[EL11] Sifakis Emmanuel and Mounier Laurent. Politiques de gestion de protections pour l'implémentation de sections critiques. RENPAR, 2011.

[EL12a] Sifakis Emmanuel and Mounier Laurent. Dynamic information-flow analysis for multi-threaded applications. *ISOLA*, 2012.

[EL12b] Sifakis Emmanuel and Mounier Laurent. Politiques de gestion de protections pour l'implémentation de sections critiques. TSI special RENPAR, 2012.

[ESKK08] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2):6:1–6:42, March 2008.

[FF00] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 219–232, New York, NY, USA, 2000. ACM.

[FFLQ08] Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. Types for atomicity: Static checking and inference for java. *ACM Trans. Program. Lang. Syst.*, 30(4):20:1–20:53, August 2008.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[GK04] Jan Groote and Misa Keinänen. Solving disjunctive/conjunctive boolean equation systems with alternating fixed points. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 436–450. Springer Berlin / Heidelberg, 2004.

[GLG12] Malay Ganai, Dongyoon Lee, and Aarti Gupta. Dtam: Dynamic taint analysis of multi-threaded programs for relevancy. In *Proceedings of the 2012 ACM SIGSOFT symposium on Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. ACM.

[GLP75] J. N. Gray, R. A. Lorie, and G. R. Putzolu. Granularity of locks in a shared data base. In *Proceedings of the 1st International Conference on Very Large Data Bases*, VLDB '75, pages 428–451, New York, NY, USA, 1975. ACM.

[GVC+10] Michelle L. Goodstein, Evangelos Vlachos, Shimin Chen, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Butterfly analysis: adapting dataflow analysis to dynamic parallel monitoring. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 257–270, New York, NY, USA, 2010. ACM.

[GVW89] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceed-*

ings of the third international conference on Architectural support for programming languages and operating systems, ASPLOS-III, pages 64–75, New York, NY, USA, 1989. ACM.

[Hav68] J. W. Havender. Avoiding deadlock in multitasking systems. *IBM Syst. J.*, 7(2):74–84, June 1968.

[HFP06] Michael W. Hicks, Jeffrey S. Foster, and P. Pratikakis. Lock inference for atomic sections. *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[HIST10] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[HLC09] Kim Hazelwood, Greg Lueck, and Robert Cohn. Scalable support for multithreaded applications on dynamic binary instrumentation systems. In *Proceedings of the 2009 international symposium on Memory management*, ISMM '09, pages 20–29, New York, NY, USA, 2009. ACM.

[HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[HMJH08] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. *Commun. ACM*, 51(8):91–100, August 2008.

[JBPT09] Ali Jannesari, Kaibin Bao, Victor Pankratius, and Walter F. Tichy. Helgrind+: An efficient dynamic race detector. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–13, Washington, DC, USA, 2009. IEEE Computer Society.

[JNPS09] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. Calfuzzer: An extensible active testing framework for concurrent programs. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 675–681, Berlin, Heidelberg, 2009. Springer-Verlag.

[KBG97] Alain Kägi, Doug Burger, and James R. Goodman. Efficient synchronization: let them eat qolb. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 170–180, New York, NY, USA, 1997. ACM.

[Kei05] Misa Keinänen. Solving boolean equation systems. Research Report A99, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, November 2005.

[Kni90] Tom Knight. Artificial intelligence at mit expanding frontiers. chapter An architecture for mostly functional languages, pages 500–519. MIT Press, Cambridge, MA, USA, 1990.

[KW10] Vineet Kahlon and Chao Wang. Universal causality graphs: a precise happens-before model for detecting bugs in concurrent programs. In *Proceedings of the 22nd international conference on Computer Aided Verification*, CAV'10, pages 434–449, Berlin, Heidelberg, 2010. Springer-Verlag.

[KZC12] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: telling the difference with portend. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 185–198, New York, NY, USA, 2012. ACM.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[LDT+12] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote Core Locking: migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the Usenix Annual Technical Conference, USENIX ATC '12*, pages 65–76, Boston, MA, USA, 2012. USENIX Association.

[LELS05] Tong Li, Carla S. Ellis, Alvin R. Lebeck, and Daniel J. Sorin. Pulse: a dynamic deadlock detection mechanism using speculative execution. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 3–3, Berkeley, CA, USA, 2005. USENIX Association.

[Lim95] B. Lim. Reactive synchronization algorithms for multiprocessors. Technical report, Cambridge, MA, USA, 1995.

[LT93] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[MC91] Sang L. Min and Jong-Deok Choi. An efficient cache-based access anomaly detection scheme. *SIGOPS Oper. Syst. Rev.*, 25(Special Issue):235–244, April 1991.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[MLH94] Peter Magnusson, Anders Landin, and Erik Hagersten. Efficient software synchronization on large cache coherent multiprocessors. Technical report, 1994.

[MMN09] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 134–143, New York, NY, USA, 2009. ACM.

[MSD10] MSDN. Priority inversion. `http://msdn.microsoft.com/en-us/library/aa915356.aspx`, 2010.

[MSQT09] Abdullah Muzahid, Dario Suárez, Shanxiang Qi, and Josep Torrellas. Sigrace: signature-based data race detection. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 337–348, New York, NY, USA, 2009. ACM.

[MZGB06] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 346–358, New York, NY, USA, 2006. ACM.

[NG09] Vijay Nagarajan and Rajiv Gupta. Architectural support for shadow memory in multiprocessors. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 1–10, New York, NY, USA, 2009. ACM.

[NKWG08] Vijay Nagarajan, Ho-Seop Kim, Youfeng Wu, and Rajiv Gupta. Dynamic information flow tracking on multicores, 2008.

[NS05] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*. The Internet Society, 2005.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[NWT+07] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the 2007 ACM SIGPLAN conference on*

Programming language design and implementation, PLDI '07, pages 22–31, New York, NY, USA, 2007. ACM.

[OC03] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '03, pages 167–178, New York, NY, USA, 2003. ACM.

[OPAGS11] Meltem Ozsoy, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Tameesh Suri. Sift: a low-overhead dynamic information flow tracking architecture for smt processors. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 37:1–37:11, New York, NY, USA, 2011. ACM.

[PFH11] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: Practical static race detection for c. *ACM Trans. Program. Lang. Syst.*, 33(1):3:1–3:55, January 2011.

[PK96] Dejan Perkovic and Peter J. Keleher. Online data-race detection via coherency guarantees. In *Proceedings of the second USENIX symposium on Operating systems design and implementation*, OSDI '96, pages 47–57, New York, NY, USA, 1996. ACM.

[Pou04] Kevin Poulsen. Tracking the blackout bug. `http://www.securityfocus.com/news/8412`, 2004.

[QWL+06] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society.

[Reg11] John Regehr. Race condition vs data race. `http://blog.regehr.org/archives/490`, 2011.

[RG02] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 5–17, New York, NY, USA, 2002. ACM.

[RGM+08] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing dynamic information flow tracking. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 35–45, New York, NY, USA, 2008. ACM.

[RVS+06] Ram Rangan, Neil Vachharajani, Adam Stoler, Guilherme Ottoni, David I. August, and George Z. N. Cai. Support for high-frequency streaming in cmps. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 259–272, Washington, DC, USA, 2006. IEEE Computer Society.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997.

[SFM10] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. Penelope: weaving threads to expose atomicity violations. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 37–46, New York, NY, USA, 2010. ACM.

[SI09] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM.

[SKSP06] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. Heapmon: a helper-thread approach to programmable, automatic, and low-overhead memory bug detection. *IBM J. Res. Dev.*, 50(2/3):261–275, March 2006.

[SM06] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, September 2006.

[SMQP09] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 253–264, New York, NY, USA, 2009. ACM.

[SMWG11] Arnab Sinha, Sharad Malik, Chao Wang, and Aarti Gupta. Predictive analysis for detecting serializability violations through trace segmentation. In Satnam Singh, Barbara Jobstmann, Michael Kishinevsky, and Jens Brandt, editors, *MEMOCODE*, pages 99–108. IEEE, 2011.

[SSP08] Prateek Saxena, R Sekar, and Varun Puranik. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 74–83, New York, NY, USA, 2008. ACM.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[Sun71] Yngve Sundblad. The ackermann function. a theoretical, computational, and formula manipulative study. *BIT Numerical Mathematics*, 11:107–119, 1971.

[SV98] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 355–364, New York, NY, USA, 1998. ACM.

[SWG92] Jaswinder P Singh, Wolf Weber, and Anoop Gupta. Splash: Stanford parallel applications for shared-memory*. Technical report, Stanford, CA, USA, 1992.

[Tan01] Andrew Tanenbaum. *Modern Operating Systems 2nd edition.* Prentice Hall, 2001.

[UMP10] Gautam Upadhyaya, Samuel P. Midkiff, and Vijay S. Pai. Using data structure knowledge for efficient lock generation and strong atomicity. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 281–292, New York, NY, USA, 2010. ACM.

[VS97] Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In *Proceedings of the 7th International Joint Conference CAAP/-FASE on Theory and Practice of Software Development*, TAPSOFT '97, pages 607–621, London, UK, UK, 1997. Springer-Verlag.

[WCGY09] Chao Wang, Swarat Chaudhuri, Aarti Gupta, and Yu Yang. Symbolic pruning of concurrent program executions. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 23–32, New York, NY, USA, 2009. ACM.

[WG12] Chao Wang and Malay Ganai. Predicting concurrency failures in the generalized execution traces of x86 executables. In *Proceedings of the Second international conference on Runtime verification*, RV'11, pages 4–18, Berlin, Heidelberg, 2012. Springer-Verlag.

[WRS] Waddington, Roy, and Schmidt. Dynamic analysis and profiling of multithreaded systems.

[WS06a] Liqiang Wang and Scott D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 137–146, New York, NY, USA, 2006. ACM.

[WS06b] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng.*, 32(2):93–110, February 2006.

[YRC05] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 221–234, New York, NY, USA, 2005. ACM.

[ZCYH05] Qin Zhao, Winnie W. Cheng, Bei Yu, and Scott Hiroshige. Dog: Efficient information flow tracing and program monitoring with dynamic binary rewriting abstract, 2005.

[ZJS+11] David (Yu) Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. Tainteraser: protecting sensitive data leaks using application-level taint tracking. *SIGOPS Oper. Syst. Rev.*, 45(1):142–154, February 2011.

[ZTZ07] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. Hard: Hardware-assisted lockset-based race detection. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 121–132, Washington, DC, USA, 2007. IEEE Computer Society.

ABSTRACT

The invasion of multi-core and multi-processor platforms on all aspects of computing makes shared memory parallel programming mainstream. Yet, the fundamental problems of exploiting parallelism efficiently and correctly have not been fully addressed. Moreover, the execution model of these platforms (notably the relaxed memory models they implement) introduces new challenges to static and dynamic program analysis. In this work we address 1) the optimization of pessimistic implementations of critical sections and 2) the dynamic information flow analysis for parallel executions of multi-threaded programs.

Critical sections are excerpts of code that must appear as executed atomically. Their pessimistic implementation reposes on synchronization mechanisms, such as mutexes, and consists into obtaining and releasing them at the beginning and end of the critical section respectively. We present a general algorithm for the acquisition/release of synchronization mechanisms and define on top of it several policies aiming to reduce contention by minimizing the possession time of synchronization mechanisms. We demonstrate the correctness of these policies (i.e., they preserve atomicity and guarantee deadlock freedom) and evaluate them experimentally.

The second issue tackled is dynamic information flow analysis of parallel executions. Precisely tracking information flow of a parallel execution is due to non-deterministic accesses to shared memory. Most existing solutions that address this problem enforce a serial execution of the target application. This allows to obtain an explicit serialization of memory accesses but incurs both an execution-time overhead and eliminates the effects of relaxed memory models. In contrast, the technique we propose allows to predict the plausible serializations of a parallel execution with respect to the memory model. We applied this approach in the context of taint analysis, a dynamic information flow analysis widely used in vulnerability detection. To improve precision of taint analysis we further take into account the semantics of synchronization mechanisms such as mutexes, which restricts the predicted serializations accordingly.

The solutions proposed have been implemented in proof of concept tools which allowed their evaluation on some hand-crafted examples.