Dataflow Analysis-Based Dynamic Parallel Monitoring

Michelle Leah Goodstein

CMU-CS-14-132
August 2014

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Todd C. Mowry, Chair
Phillip B. Gibbons
Jonathan Aldrich
Kathryn McKinley

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation under grant CNS-0720790, IIS-0713409. CNS-072079, CCF-1116898, CNS-1065112; and generous support from Intel Corporation; Google; and Intel Science and Technology Center for Cloud Computing.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Dataflow Analysis-Based Dynamic Parallel Monitoring, Dynamic Analysis, Parallel Program Monitoring, Compiler Analysis, Computer Architecture

To my family.

Abstract

Despite the best efforts of programmers and programming systems researchers, software bugs continue to be problematic. This thesis will focus on a new framework for performing dynamic (runtime) parallel program analysis to detect bugs and security exploits. Existing dynamic analysis tools have focused on monitoring sequential programs. Parallel programs are susceptible to a wider range of possible errors than sequential programs, making them even more in need of online monitoring. Unfortunately, monitoring parallel applications is difficult due to inter-thread data dependences and relaxed memory consistency models.

This thesis presents *dataflow analysis-based dynamic parallel monitoring*, a novel software-based parallel analysis framework that avoids relying on strong consistency models or detailed inter-thread dependence tracking. Using insights from dataflow analysis, our frameworks enable parallel applications to be monitored concurrently without capturing a total order of application instructions across parallel threads. This thesis has three major contributions: Butterfly Analysis and Chrysalis Analysis, as well as extensions to both enabling explicit tracking of uncertainty.

Butterfly Analysis is the first dataflow analysis-based dynamic parallel monitoring framework. Unlike existing tools, which frequently assumed sequential consistency and/or access to a total order of application events, Butterfly Analysis does not rely on strong consistency models or detailed inter-thread dependence tracking. Instead, we only assume that events in the distant past on all threads have become visible; we make no assumptions on (and avoid the overheads of tracking) the relative ordering of more recent events on other threads. To overcome the potential state explosion of considering all the possible orderings among recent events, we adapt two

techniques from static dataflow analysis, reaching definitions and available expressions, to this new domain of dynamic parallel monitoring. Significant modifications to these techniques are proposed to ensure the correctness and efficiency of our approach. We show how our adapted analysis can be used in two popular memory and security tools. We prove that our approach does not miss errors, and sacrifices precision only due to the lack of a relative ordering among recent events.

While Butterfly Analysis offers many advantages, it ignored one key source of ordering information which significantly affected its false positive rate: explicit software synchronization, and the corresponding high-level happens-before arcs. This led to the development of *Chrysalis Analysis*, which generalizes the Butterfly Analysis framework to incorporate explicit happens-before arcs resulting from high-level synchronization within a monitored program. We show how to adapt two standard dataflow analysis techniques and two memory and security lifeguards to Chrysalis Analysis, using novel techniques for dealing with the many complexities introduced by happens-before arcs. Our security tool implementation shows that Chrysalis Analysis matches the key advantages of Butterfly Analysis while significantly reducing the number of false positives, by an average of 97%.

While Chrysalis Analysis greatly improved upon Butterfly Analysis' precision, it was unable to separate potential errors due to analysis *uncertainty* from true errors. We extend both Butterfly Analysis and Chrysalis Analysis to incorporate an *uncertain* state into their metadata lattice, and provide new guarantees that true error states are now precise. We experimentally evaluate a prototype and demonstrate that we effectively isolate analysis uncertainty. We also explore possible dynamic adaptations to the presence of uncertainty.

In all cases, we have shown that our frameworks are provably guaranteed to never miss an error and sacrifice precision only due to the lack of a relative ordering among recent events, and present experimental results on performance and precision from our implementations.

Acknowledgments

Graduate school has been an amazing journey for me, and the best part of the journey has been the opportunity to meet and collaborate with fantastic people at CMU.

My advisor Todd Mowry has been a true mentor to me. I joined the Log-Based Architectures (LBA) group as a former theory student looking to transition to being a systems student. Todd always viewed my background as an asset instead of a hindrance, and gave me time, space, and instruction so that I could ramp up on computer architecture and compiler analysis. From the creation of Butterfly Analysis to the completion of this thesis, Todd has been encouraging and supportive, an advocate when necessary, an external champion of my work, and always challenging me to do my best and create the best research I could. It has been an honor to have worked and learned so much from him.

My thesis was enhanced by the feedback from my thesis committee: Todd Mowry, Phil Gibbons, Jonathan Aldrich and Kathryn McKinley. Their willingness to read and engage with thesis drafts has truly improved this document.

I was extremely lucky to join LBA at a time when it was a joint project with the Intel lablet in Pittsburgh. Phillip Gibbons, Michael Kozuch, and Shimin Chen have been extraordinary collaborators whose expertise spanned low level architecture design to theory. My work has been a balance of theory and practice, and I have been lucky enough to have collaborators whose expertise spanned the range of my thesis. Phil Gibbons worked especially closely with me on theoretical modeling and proofs. He has taught me so much about clarity and brevity in writing and presentations. Mike Kozuch has always pushed me to engage deeply with low-level proces-

sor intricacies, ensuring the assumptions I made in my abstract models matched actual processor design. Shimin Chen's expertise spanned software systems and implementation to algorithm design, and his feedback on the design of Butterfly and Chrysalis Analyses was greatly appreciated. Todd, Phil, Mike and Shimin all were deeply engaged research collaborators and mentors, whether staying up late working on paper submissions, writing letters of recommendations or providing feedback and advice during my job search. Babak Falsafi's insights and ideas were always helpful and appreciated.

Olatunji Ruwasse, Evangelos Vlachos and Theodoros Strigkos were my LBA cohort. I can't imagine graduate school without them: our shared experiences have forged lifelong friendships. We have survived hacking simulators, debugging bizarre disassembler bugs and debugging kernel deadlocks, among other mutual challenges. We have supported each other through proposals and defense, system implementation/debugging, and the paper submission/acceptance cycle. Working with Tunji, Evangelos, and Theo was a key highlight of my time at CMU. Even as we are now scattered across the globe, we have stayed in touch and continued to support each other, and I thank them for their advice and guidance as the last of the LBA-ers to defend. The addition of Vivek Seshadri, Gennady Pekhimenko and Deby Katz to our research group created excellent new friends and collaborators who I have greatly enjoyed knowing.

Outside my LBA cohort, I made many friends in CSD and SCS, among them: Arvind Ramanathan, Kami Vaniea, Michael Ashley-Rollman, Sue Ann Hong, Michael Papamichael, Kanat Tangwongsan, Charlie Garrod, Maverick Woo, Michael Dinitz, Robert Simmons, Ciera Jaspan, Mei Chen, Sam Ganzfried and James McCann, who supported and encouraged me in myriad ways, ranging from attending practice talks, giving me pep talks, or providing support going on the job market. While the transition from Intel Labs, Pittsburgh to the Intel Science and Technology Center was a difficult one, one of the "platinum linings" was the addition of the Safari research lab to the 4th floor CIC, among them: Samira Khan, Rachata Ausavarungnirun, Yoongu Kim, Justin Meza and Lavanya Subramanian, who quickly made the space into an extended

"architecture lab". Their support, particularly while finishing this thesis, has been invaluable.

While at Carnegie Mellon University, I had the privilege of serving as the Computer Science Department's Graduate Student Ombudsperson for three years. My advocacy and ability to help fellow graduate students would not have been possible without the full support I received from Mor Harchol-Balter, Srinivasan Seshan and Deborah Cavlovich. Staff members such as Sophie Park, Diana Hyde, Jennifer Landefeld, Jennifer Gabig, Sharon Burks, Marcie Baker and Catherine Copetas went over and above in providing support while I was here, whether fast reimbursements or fighting university bureaucracy on my behalf. I had many fascinating discussions with David Andersen at his "office" in the kitchen of Intel/ISTC. I learned so much from my time working with Manuel Blum and Luis von Ahn as a theory student, as well as my early collaborations with Virginia Vassilevska.

Young-Mi Hultman, Elizabeth Bales, Nathan Bales, Matthew Milcic and Rachel Milcic provided invaluable friendship from afar. My parents, Sara and Jack Goodstein, and my sister Rebecca Goodstein, have supported and cheered me on throughout my entire graduate school journey.

Writing an explicit acknowledgment section is dangerous: inevitably, someone's name and contribution is omitted by accident. To anyone whose contribution is not mentioned, I apologize and appreciate what has been done for me. While I am the author of this thesis, its existence was made possible by all the friends, family, and collaborators whose support and encouragement helped me along my journey. Thank you.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Writing correct, bug-free programs is difficult. To help address this problem, a number of tools have been developed over the years that perform *static* [22, 35, 40], *dynamic* [20, 36, 71, 82, 98], or *post-mortem* [78, 119] analysis to diagnose bugs. Static analysis is guaranteed to reason about all paths through a program; anything proven to be safe is guaranteed to be safe under any execution of the program. Static analysis does suffer from imprecision when reasoning about pointers and heap locations, whose values aren't known until runtime. Dynamic analysis benefits from access to pointer value and heap locations, but is limited to analyzing only the observed path taken by the application; dynamic analysis will generally not miss a bug in the monitored execution, but may miss bugs present in the actual application which do not occur while the application is monitored.

While static and dynamic analysis tools are generally complementary, my focus in this thesis is on *dynamic* (online) tools, which we refer to as *"lifeguards"* (because they watch over a program as it executes to make sure that it is safe). To avoid the need for source code access, lifeguards are typically implemented using either a dynamic binary instrumentation framework (e.g., Valgrind [82], Pin [71], DynamoRio [20]) or with hardware-assisted logging [23]. Lifeguards maintain shadow state to track a particular aspect of correctness as a program executes, such as its memory [80], security [84], or concurrency [98] behaviors. Most existing lifeguards

have focused on monitoring sequential programs.

As difficult as it is to write a bug-free sequential program, however, it is even more challenging to avoid bugs in parallel software, given the many opportunities for non-intuitive interactions between threads. Despite the difficulty, on modern multicore processors, increases in performance are tied to an increasing use of parallelism. To achieve this, programmers must devise parallel algorithms and write parallel software. Hence we would expect bug-finding tools such as lifeguards to become increasingly valuable as more programmers wrestle with parallel programming. Unfortunately, the way that most lifeguards have been written to date does not extend naturally to parallel software[1] due to a key stumbling block: *inter-thread data dependences*. To illustrate the complexities of parallel program analysis, it helps to begin by examining sequential program analysis, and then analyze the challenges in adapting dynamic analysis to the parallel domain.

1.1 Background: Dynamic Program Monitoring

Program monitoring performs on-the-fly checking during the execution of applications, and is an important technique for improving software reliability and security. Program monitoring tools, or *lifeguards* can be categorized according to the granularity of application events that they care about, from system-call-level [52, 91] to instruction-level [80, 82, 84, 98]. Compared to the former, the latter can obtain highly detailed dynamic information, such as memory references, for more accurate and timely bug detection. However, such fine-grained monitoring presents great challenges for system support. This thesis focuses on instruction-level lifeguards, although the results readily extend to coarser-grained settings as well.

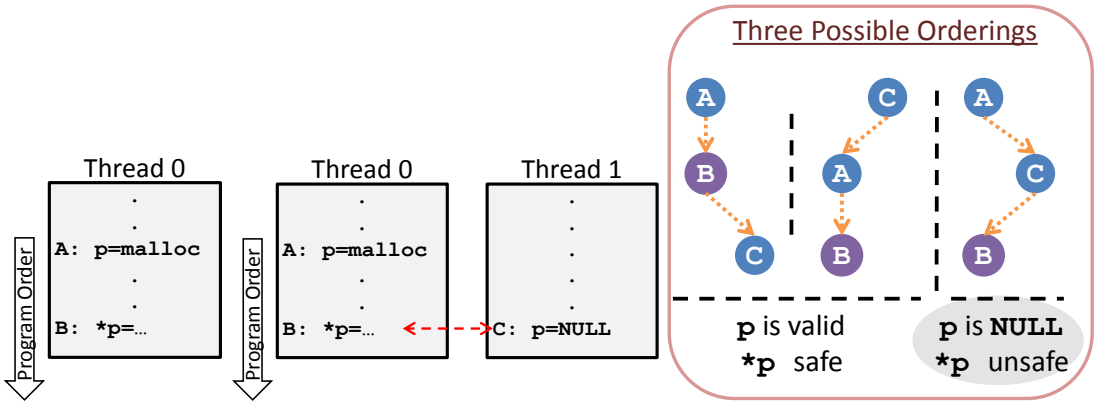
[1]For the purpose of this thesis, "parallel software" refers to software targetting a shared-memory abstraction, written in languages like C or C++, and using a threaded-style parallelism

Lifeguards: ADDRCHECK and TAINTCHECK

We describe two representative lifeguards which will be referenced throughout this thesis.

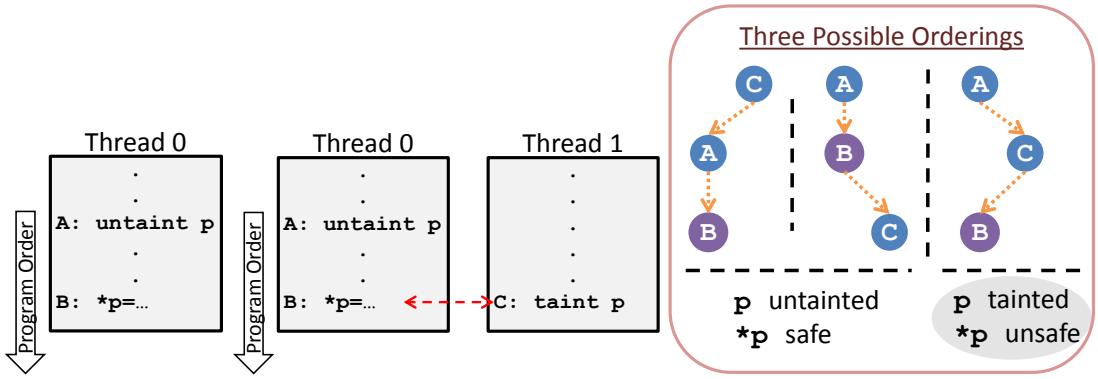
- ADDRCHECK [79] is a memory-checking lifeguard. By monitoring memory allocation calls such as `malloc` and `free`, it maintains the allocation information for each byte in the application's address space. Then, ADDRCHECK verifies whether every memory read and write accesses an allocated region of memory by reading the corresponding allocation information; that every `free` references a currently allocated region of memory; and that every `malloc` references a currently deallocated region of memory.
- TAINTCHECK [84] is a security-checking lifeguard for detecting overwrite-based security exploits (e.g., buffer overflows or printf format string vulnerabilities). It maintains metadata for every location in the application's address space, indicating whether the location is *tainted*. After a system call that receives data from the network or from an untrusted disk file, the memory locations storing the untrusted data are all marked as tainted. TAINTCHECK monitors the inheritance of the tainted state: For every executed application instruction, it computes a logical `OR` of the tainted information of all the sources to obtain the tainted information of the destination of the instruction. TAINTCHECK raises an error if tainted data is used in jump target addresses (to change the control flow), format strings, or other critical ways.

Lifeguards typically operate on shadow state, or *metadata*, that they associate with every active memory location in the program (including the heap, registers, stack, etc.). As the monitored application executes, the lifeguard follows along, instruction-by-instruction, performing an analogous operation to update the corresponding shadow state. For example, when a lifeguard that is tracking the flow of data that has been "tainted" by external program inputs [84] encounters an instruction such as "`A = B + C`", the lifeguard will look up the boolean tainted status for locations `B` and `C`, `OR` these values together, and store the result in the shadow state for `A`.



(a) Sequential Program (b) Parallel Program (c) Potential Interleavings

Figure 1.1: Analyzing parallel programs is more complicated than sequential programs (here, shown using ADDRCHECK). (a) Single threaded sequential programs can be analyzed in program order. (b) When analyzing parallel programs, it is insufficient to monitor a thread in isolation; inter-thread data dependences must be incorporated into the analysis. (c) Space of possible interleavings; which interleaving occurs affects the outcome of the analysis.



(a) Sequential (b) Parallel Program (c) Potential Interleavings

Figure 1.2: Parallel lifeguard complications extend to many lifeguards–here, illustrating TAINTCHECK. (a), (b) and (c) follow the same form as Figure 1.1, with an entirely different analysis.

When monitoring a single-threaded application, it is straightforward to think of the lifeguard as a finite state machine that is driven by the dynamic sequence of instructions from the monitored application. The order of events in this input stream is important. For single-threaded applications, it is sufficient to analyze an execution trace of the instructions in *commit order*, or the order the instructions are committed from the reorder buffer by the processor: any reordering by the processor is guaranteed to preserve all intra-thread data dependences. This is illustrated

in Figure 1.1(a) (respectively, Figure 1.2(a)), which shows a simple sequential programs that ADDRCHECK (respectively, TAINTCHECK) can easily analyze. As any processor reorderings respect intra-thread data dependences, analyzing the outcome of statement `B` in Figure 1.1(a) is straightforward: assuming the `malloc` in statement `A` returns a non-null pointer, then the dereference of `p` at `B` is safe. Likewise, in Figure 1.2(a), the dereference of `p` at `B` is safe since `p` is untainted, or marked as trusted, earlier in program order at `A`.

1.2 Inter-Thread Data Dependences Complicate Analysis

Adapting sequential dynamic analysis tools to the parallel domain is nontrivial. As an example, consider Figure 1.1(b). Thread 0 in Figure 1.1(a) and Thread 0 in 1.1(b) perform the exact same set of operations. However, in Figure 1.1(b), the application is now concurrent, and Thread 1 concurrently performs `p = NULL`. If thread 0 executes entirely before thread 1 or thread 1 executes entirely before thread 0 (the first two scenarios shown to the left in Figure 1.1(c), no error has occurred on this particular execution. However, if the two threads interleave, with the assignment of `p = NULL;` occurring after `p=malloc()` but before `*p = ...` (shown in Figure 1.1(c) on the right), then the program will experience a segmentation fault. The lifeguard cannot reason about each thread in isolation; instead, it must consider all possible interleavings and interactions between threads. This scenario is not limited to memory safety; Figure 1.2(b) illustrates a similar problem in TAINTCHECK where the different interleavings shown in Figure 1.2(c) can lead to different outcomes.

Fundamentally, analysis of parallel programs executing on a shared-address space machine is complicated by the presence of *inter-thread data dependences*. In a parallel setting, reasoning about each thread in isolation is not sufficient: interference from other threads affects analysis. In particular, without knowing the particular interleaving of threads, it can be difficult to reason about whether an error occurred on a particular dynamic run.

1.3 One Approach: Enable Dynamic Parallel Monitoring by Capturing Ordering of Application Events

The complications in analyzing Figure 1.1(b) and Figure 1.2(b) arise because the lifeguard is not aware which order of events actually occurred. One approach to enabling analysis, then, is to capture the ordering of these non-deterministic shared memory interactions, and use this information to enable dynamic program analysis of parallel programs.

1.3.1 Time Slicing

One simple solution to enable dynamic monitoring of parallel applications is simply to time slice the parallel threads on one core; by controlling which thread is scheduled on and which threads are scheduled off, its easy to infer an ordering of application events and feed this to a lifeguard. This has the advantage of being a software-only solution, with one large caveat: all application parallelism has been lost. Its important to note that one state-of-the-art dynamic analysis framework, Valgrind, actually uses timeslicing when analyzing parallel programs [1]. A key disadvantage is the loss of parallelism; all parallel threads must run on one core.

1.3.2 ParaLog

An alternative hardware-based approach treats this interference between parallel threads as a measurement problem: the goal is to capture the inter-thread data dependences and expose them to the parallel program monitor, or *lifeguard*. Our solution, ParaLog [110], utilizes hardware support to capture inter-thread data dependences and expose them to the lifeguards as happens-before arcs. A key challenge in developing ParaLog's hardware extensions was ensuring that lifeguard threads can consume these arcs online and that lifeguard threads obey these arcs when processing events from the monitored application. ParaLog is the first solution to deliver high-precision dynamic parallel program monitoring with minimal slowdown, given the proposed

hardware extensions. ParaLog supports the sequential consistency memory model, as well as the total store order memory model, two of the strongest memory models. A key advantage of ParaLog is that single-threaded lifeguards can easily be adapted to a multi-threaded environment, and lifeguard writers do not have to spend much time worrying about ordering of inter-thread data dependences.

1.4 This Thesis: Enable Dynamic Parallel Program Analysis Without Capturing Inter-Thread Data Dependences

The goal of this thesis is to answer a question: can we enable dynamic parallel program analysis without capturing a total order of application events? There are good reasons for pursuing this approach. First, solutions which do not require access to a total order of application events also do not require the specialized hardware which, at this time, does not exist on modern processors. In addition, modern processors implement *relaxed memory consistency models*. In contrast to sequential consistency, on relaxed memory consistency models, there is no guarantee a total order of application events exists that respects both read-write semantics and thread ordering, even if one instruments all coherence activity from the processor.

Motivated by the desire to find solutions that require neither hardware support nor strong consistency models, this work explores a novel monitoring approach based on dataflow analysis over windows of inter-thread interference uncertainty, called *dataflow analysis-based dynamic parallel monitoring*. Dataflow analysis-based dynamic parallel monitoring does not require capturing inter-thread data dependences, and hence avoids the need for adding hardware support for such capture. Moreover, it can be used for the weaker memory models prevalent in today's machines. Such weak memory models are not supported by ParaLog, in part because such memory models do not have a corresponding total order of application instructions across parallel threads. (In practice, such memory models could cause ParaLog to deadlock.)

1.4.1 Dataflow Analysis-Based Dynamic Parallel Monitoring

In contrast, our dataflow analysis-based parallel monitoring solution provides a generic framework to lifeguard writers that automatically reasons about concurrent interleavings, similar to how dataflow analysis provides a general platform for writing static analyses. Dataflow analysis-based parallel monitoring tolerates the lack of total ordering information across threads that occurs in today's machines while also managing to avoid the state space explosion problem. Our thread execution model represents potential concurrency within the system using *bounded windows of uncertainty*. Our key insight, inspired by *region-based analysis*, was to create a modified "closure" to automatically reason about concurrency within the uncertainty windows.

Butterfly Analysis [45] is the first dataflow-analysis based parallel monitoring framework. Much like dataflow analysis provides a general platform for writing static analyses, Butterfly Analysis is generic and easily adapted for new dynamic parallel analyses. Although inspired by dataflow analysis, Butterfly Analysis is very much a non-trivial adaptation. Moving from the static to dynamic environments involves moving from a finitely-sized control flow graph (CFG) to a (possibly infinite) dynamic run-length: we showed how analysis can proceed dynamically without waiting for the entire "dynamic CFG" to become available. We introduced new primitives to capture the effects of concurrency in the sliding windows, and new closures to avoid exploring a combinatorial explosion of interleavings. Furthermore, we showed that two canonical dataflow analyses, reaching definitions and available expressions, as well as two memory and security lifeguards based on them, are sound adaptations and never experience false negatives (missed errors, as defined by the analysis).

Butterfly Analysis supports only a very simple and regular concurrency structure of sliding windows across all threads. While Butterfly Analysis avoids the overhead of tracking inter-thread data dependences, it also ignores high-level synchronization within a program, which could lead to Butterfly Analysis believing an error existed in the program which was impossible due to synchronization. In Chrysalis Analysis [44], we showed how to generalize Butterfly Analysis

to improve precision by incorporating high-level happens-before arcs. Chrysalis Analysis supports an arbitrarily irregular and asymmetric acyclic structure within such windows. This makes the analysis problem considerably more challenging. Despite these challenges, we presented sound formalizations within the Chrysalis Analysis framework of both reaching definitions and available expressions, and showed a large improvement in precision.

Butterfly Analysis demonstrated that the dataflow analysis-based dynamic parallel monitoring approach had merit, and Chrysalis Analysis explored methods of improving precision by trading off some performance lost to the more complex analysis and thread execution model. However, in both Butterfly and Chrysalis Analysis, when a lifeguard check fails (signaling a potentially unsafe event), neither framework can automatically reason about whether the check failed due to a *true error* versus a *potential error*. Acknowledging feedback that programmers desire the ability to distinguish these cases, we modified both Butterfly and Chrysalis Analyses to incorporate *uncertainty* into their metadata lattices. The goal was two-fold: first, isolate known errors from potential errors, and give programmers a provable guarantee that any failed check of this precise state was indeed a true error. Second, by isolating true errors from potential errors, enable dynamic responses to the presence of uncertainty (e.g., dynamically resizing epochs) to recover precision when possible.

The isolation ensures that any performance overhead of a dynamic response is incurred precisely when the analysis could not reach a precise solution; neither Butterfly or Chrysalis Analysis could properly disambiguate a true error from a potential error. By dynamically adapting analysis to improve precision *only when* a potential error is encountered, the analysis only incurs the cost of the dynamic adaptation when it could potentially improve precision. It can safely avoid any dynamic adaptations when encountering a true error, as no additional amount of analysis will ever change a true error to a potential error.

Incorporating uncertainty in the metadata lattice also gives dataflow analysis-based dynamic parallel monitoring an advantage most other dynamic analyses (which frequently limit them-

selves to the particular interleaving observed) do not match: the ability to reason about *near misses*, or cases where an error did not manifest on this run, but *no synchronization prevented the buggy interleaving from occurring on a later run!* Especially when overlaid on top of Chrysalis Analysis, a "false positive" which is also a near miss provides fundamental insight about the code: while the programmer may frequently get lucky, an actual bug is present in the source code.

Beyond Data Race Detection: General Framework for Parallel Monitoring

Dataflow analysis-based dynamic parallel monitoring is a general purpose dynamic parallel analysis framework. The true power of dataflow analysis-based dynamic parallel monitoring lies in its ability to support analyses *even when the underlying application experiences a data race*. For instance, TAINTCHECK is unconcerned about data races in the monitored application, unless the data races lead to a *metadata race*, i.e. two different metadata values are possible for a given memory location. Likewise, data races in ADDRCHECK are unimportant; the only race ADDRCHECK concerns itself with is between accesses and `malloc`/`free` calls. The ability to monitor programs despite the presence of data races in the monitored application increases the power of dataflow analysis-based dynamic parallel monitoring; while concurrency-specific analyses can be overlaid onto our framework, we also support analyses that previously were limited to sequential programs, or else required extensive support to capture ordering.

1.5 Related Work

Dataflow analysis-based dynamic parallel monitoring is a new framework for monitoring parallel programs which draws from many other subfields. Limited to monitoring one thread only, it resembles traditional dynamic analysis, though with larger overheads. When considering only one window of uncertainty for a parallel application, the two passes resemble traditional static

dataflow analysis, though not on a control flow graph. One of the advantages of our approach is that it can be either a software-only solution or utilize a modest amount of hardware.

In addition, many of the primitives that allow butterfly analysis to achieve reasonable efficiency and precision, such as uncertainty epochs [8, 10, 73, 76, 106], sliding windows [68], only assuming partial ordering of events [87], and conservative analysis [25], are present in a variety of other works in the programming languages and computer architecture communities. Chrysalis analysis utilizes *vector clocks*, a well studied area [11, 24, 102] with many applications, one of which is data race detection [19, 37].

1.5.1 Platforms for Lifeguards (aka Dynamic Analysis)

Log-Based Architecture (LBA) is a hardware platform for dynamic program monitoring which offloads the lifeguard to a separate core from the application [23]. Originally, LBA was limited to a monitoring one application core with a single-threaded lifeguard. If the application had multiple threads, they had to be timesliced on the same core. In the first parallel extension of LBA, Ruwase *et al.* [96] parallelized the lifeguards but not the application, yielding a speedup in the amount of time it took a lifeguard to monitor a sequential application but not allowing the application itself to run in parallel. As mentioned earlier, Vlachos *et al.* [110] proposed ParaLog, a hardware framework which extends LBA [23] to handle parallel applications, using Flight Data Recorder (FDR) [118] as a mechanism to capture inter-thread data dependences and use them to make metadata updates deterministic. Concurrent with this thesis, Vlachos [109] later extended ParaLog to the domain of relaxed consistency models, specifically for total store order (TSO) and relaxed memory order (RMO), in a proposal named Resolve. Resolve, like ParaLog, is a hardware-based proposal, in contrast to dataflow analysis-based dynamic parallel monitoring.

LBA is only one hardware-based platform for program monitoring; DISE [27] is another. Dynamic-binary instrumentation (DBI) tools such as Valgrind [81, 83, 100], DynamoRio [20] and PIN [71, 125] can also be used to implement lifeguards, as well as solutions like Road-

Runner [39] which allow Java bytecode to be dynamically instrumented and CAB [48], a Java platform which allows analysis to be offloaded to a separate core.

While dataflow analysis-based dynamic parallel monitoring requires access to a dynamic parallel monitoring platform such as LBA or PIN, dataflow analysis-based dynamic parallel monitoring is a software framework whose correctness guarantees are independent of platform chosen and whose algorithms do not depend on that platform. The platform used to implement dataflow analysis-based dynamic parallel monitoring may affect performance (hardware is generally faster than software when it exists); for the experiments in this thesis, we used LBA.[2]

1.5.2 Parallel dataflow analyses

There have been proposals for parallel adaptations of classic dataflow problems. The adaptations focus mostly on adapting control flow graphs to reflect explicit programmer annotated parallel functions, but are often constrained in the memory consistency model, semantics, or degree of "correctness" they require in the program. In some cases [47, 103, 104], a copy-in/copy-out semantics is assumed. Sarkar's proposal [97] of using Parallel Program Graphs, or PPGs, requires deterministic or data-race free programs. Long and Clarke [66] propose a parallel dataflow extension suitable for programs using rendezvous synchronization without any shared variables. Knoop *et al.* [62, 64] propose a more general parallel adaptation, suitable for many bit vector analyses, which requires explicit parallel regions and assumes interleaving semantics. Knoop also presented specific parallel adaptations for the problems of code motion [63], partial dead code elimination [59], constant propagation [60] and demand-driven dataflow queries [61].

In contrast, dataflow analysis-based dynamic parallel monitoring is a dynamic adaptation of dataflow analysis to a parallel domain. Unlike prior attempts, dataflow analysis-based dynamic parallel monitoring cannot assume access to specialized structures or bug-free programs, be-

[2]The use of LBA for our experiments does not violate our discussion of dataflow analysis-based dynamic parallel monitoring as a software framework that does not require hardware. LBA was used to gather execution traces and insert epoch boundaries; all of this is possible with DBI and without hardware. In Chrysalis Analysis, the shim library that wraps library calls was also modified; this is again easily done within DBI.

cause it is being designed as a general framework that must work even in the presence of bugs in the monitored programs. Furthermore, dataflow analysis-based dynamic parallel monitoring is explicitly designed to work correctly on relaxed memory consistency models, and cannot require sequential consistency as a condition of the analysis behaving correctly. Finally, dataflow analysis-based dynamic parallel monitoring is a dynamic framework, which changes the structure of the data it must analyze and motivates the use of a sliding window of application events (as monitoring the entire dynamic execution is quickly intractable).

1.5.3 Systematic Testing

Dataflow analysis-based dynamic parallel monitoring in some ways resembles proposals for systematic testing [21, 74, 77], which either try to exercise all interleavings [74], or a randomized subset with probabilistic guarantees [21, 77]. Recent work also has proposed testing concurrent functions instead of exercising all interleavings [30], languages to allow programmers to guide a schedule towards a buggy interleaving [34], and combining local logging and constraint solvers to reproduce a buggy execution [54].

Unlike these systems, dataflow analysis-based dynamic parallel monitoring does not make guarantees about the correctness of its analysis for all possible interleavings. However, we do provide guarantees for all total orderings consistent with the observed partial ordering, as well as providing guarantees for many possible interleavings by observing only one. To improve coverage of dataflow analysis-based dynamic parallel monitoring in the future, one could try to combine the ideas underlying systematic testing to drive different partial orderings for dataflow analysis-based dynamic parallel monitoring to improve interleaving coverage.

1.5.4 Deterministic Multi-threading

There have been several proposals that leverage the key observation that debugging a sequential program is easier than debugging a parallel program largely due to its determinism, and propose

making multi-threading deterministic [13–15, 31, 32, 53, 56, 65, 85]. While some proposals are robust to changes in the input affecting interleavings [65], most only guarantee that the same interleaving will be seen with the exact same inputs [13, 31, 32]. In addition, achieving the best performance frequently requires relaxing the consistency model [13, 31, 32], so that while determinism has been achieved, the cost is that instructions are being deliberately reordered, potentially beyond what the hardware itself would do (or what programmers would expect when debugging). In a similar vein, there has also been work to limit allowable production interleavings to known good interleavings which survived testing [116, 120].

In contrast, dataflow analysis-based dynamic parallel monitoring is generic enough to monitor any execution, whether or not it is deterministic. It does not require the application being monitored to have been compiled with a special compiler or have access to deterministic hardware. Furthermore, if deterministic multi-threading becomes more prevalent, the insights can be used to inform dataflow analysis-based dynamic parallel monitoring when it is considering the possible interleavings. Deterministic multi-threading does not guarantee a bug-free program; in fact, dataflow analysis-based dynamic parallel monitoring, by monitoring all interleavings consistent with the observed partial ordering, has the ability to detect a buggy interleaving for an input x which may not occur in deterministic multi-threading on input x, but would occur on a slightly perturbed input x'.

1.5.5 Detect Violations of Consistency Model

A lot of work has focused on detecting violations of sequential consistency, holding that such violations are often indicative of a buggy interleaving [70, 73, 75, 92]; in some cases, the works attempt to enforce total store order (TSO) instead of sequential consistency [111], as TSO more closely matches the x86 memory consistency model [3]. However, x86 is not precisely TSO–it permits writes that are unaligned, which may be executed as multiple memory accesses where no guarantees about visibility or execution order are made [3]. In contrast, dataflow analysis-based

dynamic parallel monitoring focuses on delivering its provable guarantees *even when* the execution is not sequentially consistent; this follows from a deliberate design decision for dataflow analysis-based dynamic parallel monitoring to work on any relaxed consistency machine, as long as they provide shared memory and cache coherence.

1.5.6 Concurrent Bug Detection

There is a substantial body of work dedicated to dynamic analyses to detect concurrency bugs. Many of these analyses could themselves be expressed within the dataflow analysis-based dynamic parallel monitoring framework.

Data Race Detection

There has been substantial work focused on efficiently detecting data races [19, 25, 33, 37, 76, 94, 98, 114, 121]. Some techniques focus on efficiency over precision [98], while others prioritize both precision and efficiency [37], and some uniquely adapt dataflow analysis to perform race detection [25]. Still other proposals attempt to improve performance by probabilistically detecting data races [19], using hardware to accelerate the detection process [33] or in specialized cases, exploiting parallelism's structure [94]. People have also differentiated language level data races from low level data races [114]. Data races themselves have been shown to be much less benign than many programmers believe [38]. Furthermore, ad hoc synchronization, which can frustrate many bug detection tools (including data race detection), has also been shown to be dangerous and less effective than many programmers believe at achieving both performance and correctness [117].

Data race detection is one particular dynamic analysis; while dataflow analysis-based dynamic parallel monitoring can be used to implement a data race detector, dataflow analysis-based dynamic parallel monitoring is actually much more general and can support many analyses. Furthermore, the analyses' provable guarantees hold regardless of whether the underlying monitored

program experiences a data race.

Detecting and Diagnosing Other Concurrent Bugs

In addition to the substantial body of work on data race detection, there has also been a large body of work on detecting atomicity violations [67, 86] and deadlocks [57, 58], among others [123]. Some analyses [58, 123] combine static analysis phases with a dynamic analysis phase to achieve better results. Recent work has focused on using already-available hardware, such as performance counters [6] to detect and diagnose bugs in both sequential and concurrent programs. Later, Arulraj *et al.* [7] propose branch-tracing facilities provided by x86 to provide failure diagnoses that are suitable to be deployed in production systems. Some proposals, such as Aviso [69] and ConAir [122], not only detect failures, but also attempt to correct the problem, whether in future interleavings or by rolling back a single thread's execution.

Dataflow analysis-based dynamic parallel monitoring focuses on providing a general purpose platform for adapting analyses, such as TAINTCHECK and ADDRCHECK, to the parallel domain. One could write a deadlock detector, or a race detector, within the confines of dataflow analysis-based dynamic parallel monitoring. In some ways, dataflow analysis-based dynamic parallel monitoring is constantly checking for *interference* between threads, and then calculating how that interference affects the analysis it is currently executing. Unlike some proposals, dataflow analysis-based dynamic parallel monitoring does not attempt to fix the currently executing application.

1.6 Thesis Statement

The goal of this research is to demonstrate the following:

> *Without explicit knowledge of inter-thread data dependences, it is possible to build an efficient, software-based general framework suitable for online monitoring based on windows of uncertainty.*

1.7 Contributions

This thesis makes the following contributions:

- We propose a novel abstraction for modeling thread execution that incorporates bounded regions of uncertainty as the underlying setting for performing dynamic parallel program monitoring.
- We develop a new class of software-based general frameworks for monitoring parallel programs at runtime, called *dataflow analysis-based dynamic parallel monitoring*. Dataflow analysis-based dynamic parallel monitoring frameworks adapt forward dataflow analysis techniques to the dynamic domain and are designed to monitor parallel programs without capturing inter-thread data dependences. Dataflow analysis-based dynamic parallel monitoring is a general parallel dynamic analysis framework that delivers provable guarantees not to miss errors *even when* the monitored application experiences a data race, and is not limited to detecting concurrency-specific bugs. A large contribution of dataflow analysis-based dynamic parallel monitoring is the number of other analyses, many that previously required support to reason about inter-thread data dependences, which can now be adapted to the parallel domain.
- We introduce *Butterfly Analysis*, the first dataflow analysis-based dynamic parallel monitoring, which demonstrates how to adapt two canonical dataflow analysis problems (reaching definitions and available expressions) to the domain of dynamic parallel monitoring. We show how reaching definitions and available expressions serve as useful abstractions for adapting real-world lifeguards to the parallel domain by adapting TAINTCHECK and ADDRCHECK as layers on top of these analyses, respectively, and provide provable guarantees that our analyses miss no errors. We implement the Butterfly Analysis version of ADDRCHECK and conduct performance and sensitivity studies, demonstrating the trade-offs between improved precision and better performance.

- Inspired to further reduce the false positives present in Butterfly Analysis, we refine the thread execution model to incorporate high-level synchronization-based happens-before arcs. We generalize Butterfly Analysis to incorporate these synchronization-based arcs, creating *Chrysalis Analysis*. We show how how to generalize reaching definitions, available expressions, ADDRCHECK and TAINTCHECK within Chrysalis Analysis while maintaining all our provable guarantees, and explore the challenges of doing so with a more complicated thread execution model, and more difficult setting for updating both global and local state. We implement a TAINTCHECK prototype in both Butterfly and Chrysalis Analyses, showing that Chrysalis Analysis trades off a 1.9x slowdown (average, relative to Butterfly Analysis) for a 17.9x reduction in the number of false positives.
- We explore the root causes of "potential error" messages by introducing an *uncertain* metadata state into the reaching definitions and TAINTCHECK metadata lattices, and show how this uncertain state allows us to disambiguate true errors from possible errors. We implement the uncertainty extension to TAINTCHECK and demonstrate that all previous potential errors are now mapped to failed checks of uncertain. We investigate the impact of dynamically adapting to the presence of a failed check of uncertainty by adjusting the effective epoch size, and show that such adaptations can effectively eliminate all false positives.

1.8 Thesis Organization

The remainder of this thesis will focus on developing three dataflow analysis-based dynamic parallel monitoring frameworks: Butterfly Analysis and Chrysalis Analysis, as well as the uncertainty extensions to both. We will begin by deriving the thread execution model that underlies all frameworks in Chapter 2. Once we have the model of thread execution, we will explore how to build the first dataflow analysis-based dynamic parallel monitoring framework, Butterfly Analysis, in Chapter 3.

In Chapter 4, we will explore how augmenting the thread execution model with high-level synchronization-based happens-before arcs in Chrysalis Analysis can lead to a more complicated thread execution model (and therefore, more complex analysis) which ultimately greatly improves on Butterfly Analysis' precision.

In Chapter 5, we will show how adding an additional metadata state to explicitly track uncertainty once more increases the complexity of the analysis, but provides benefits overall by enabling isolation of true errors from potential errors. Furthermore, we will explore how dynamic adaptations in the face of a failed check of an uncertain location can lead to elimination of potential errors altogether, a large win for dataflow analysis-based dynamic parallel monitoring.

Finally, in Chapter 6, we conclude by reflecting on the contributions of our work and proposing possible future extensions.

Chapter 2

Modeling Parallel Thread Execution

Motivated by the desire to find solutions that require neither hardware support nor strong consistency models, our research explores a novel monitoring approach based on dataflow analysis over windows of inter-thread interference uncertainty. Dataflow analysis-based dynamic parallel monitor does not require capturing inter-thread data dependences, and hence avoids the need for adding hardware support for such capture. Moreover, it can be used for the weaker memory models prevalent in today's machines. Creating a software framework that can provably guarantee no missed errors requires a thread execution model suitable for proving such guarantees, where the assumptions about underlying hardware closely match modern parallel processors.

General-Purpose Lifeguard Infrastructure

Existing general-purpose support for running lifeguards can be divided into two types depending on whether lifeguards share the same processing cores as the monitored application or lifeguards run on separate cores. In the first design, lifeguard code is inserted in between application instructions using dynamic binary instrumentation in software [20, 71, 82] or micro-code editing in hardware [27]. Lifeguard functionality is performed as the modified application code executes. In contrast, the second design offloads lifeguard functionality to separate cores. An execution

trace of the application is captured at the core running the application through hardware, and shipped (via the last-level on-chip cache) on-the-fly to the core running the lifeguard for monitoring purposes [23].

We observe that lifeguards see a simple sequence of (user-level) application events regardless of whether the lifeguard infrastructure design is same-core or separate-core; the event sequence is consumed on-the-fly in the same-core design, while the trace buffer maintains any portion of the event sequence that has been collected, but not yet consumed, in the separate-core design. This observation suggests the application event sequence as the basic model for monitoring support. Using this model, we are able to abstract away unnecessary details of the monitoring infrastructure and provide a general solution that may be applied to a variety of implementations.

Most previous works studied sequential application monitoring. (A notable exception is [26], which assumes transactional memory support.) However, in the multicore era, applications increasingly involve parallel execution; therefore, monitoring support for multithreaded applications is desirable. Unfortunately, adapting existing sequential designs to handle parallel applications is non-trivial, as discussed in Chapter 1. This paper proposes a solution that does not require extensive hardware dependence-tracking mechanisms or a strong consistency model.

To begin formulating our thread execution model, we consider a model of monitoring support with multiple event sequences: one per application thread. Each sequence is processed by its own lifeguard thread[1]. The lifeguard analysis will lag behind the application execution somewhat, relying on existing techniques [23] to ensure that no real damage occurs during this (short) window.[2] As discussed in Chapter 1, event sequences do not contain detailed inter-thread dependences information.

[1]When a dynamic binary instrumentation platform such as PIN [71] is used, it is possible to use the same thread to generate and process the sequence.

[2]A lifeguard thread raising an error may interrupt the application to take corrective action [93]. Some delay between application error and application interrupt is unavoidable, due to the lag in interrupting all the application threads.

2.1 Challenges in Adapting Dataflow Analysis to Dynamic Parallel Monitoring

In the absence of detailed inter-thread dependence information, there are many possible interleavings consistent with the event sequences that lifeguards see when monitoring parallel programs.[3] Our approach is to adapt dataflow analysis—traditionally run statically at compile-time—as a dynamic run-time tool that enables us to reason about possible interleavings of different threads' executed instructions. In this section, we will motivate our design decisions, showing how simpler constructions are either too inefficient, too imprecise, or both. For ease of exposition, we will assume a sequentially consistent machine throughout this section and through Section 2.4.1. This will be relaxed in Section 2.4.

2.1.1 Naive Attempt: Adapt Control Flow Graphs (CFGs) to Dynamic Instruction-grain Monitoring

For the sake of mapping dataflow analysis onto dynamic program monitoring, the dynamic trace of events (*i.e.*, machine instructions) in dynamic monitoring is roughly analogous to the static program in traditional dataflow analysis: it is the code to be analyzed. Instead of program statements, we must analyze assembly. Unlike static source code, these sequences of events are linear (*i.e.*, there is no unresolved control flow) and have no aliasing issues. One natural approach, then, is to adapt tools and abstractions that have been developed to analyze static code to a dynamic setting, and use them to analyze dynamic traces. We initially explored adapting a control flow graph (CFG) representation to represent the order in which lifeguard analysis should proceed. A control flow graph expresses relationships between basic blocks within a program. Significantly, CFGs can represent ordered relationships between basic blocks, as well as relationships such as branches, where control flow can take different independent paths.

[3]Even on the simplest sequentially consistent machine, lifeguards do not see a single precise ordering of all application events.

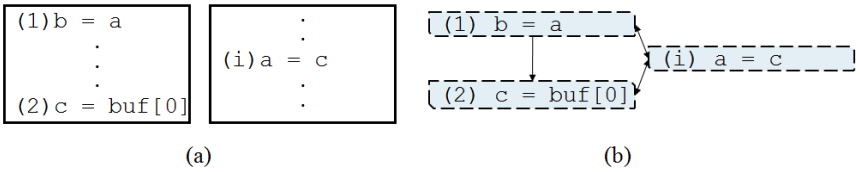


Figure 2.1: Two threads modify three shared memory locations, shown **(a)** as traces and **(b)** in a CFG. Throughout this paper, solid rectangles contain blocks of instructions, dashed hexagons contain single instructions, and "empty" blocks contain instructions that are not relevant to the current analysis.

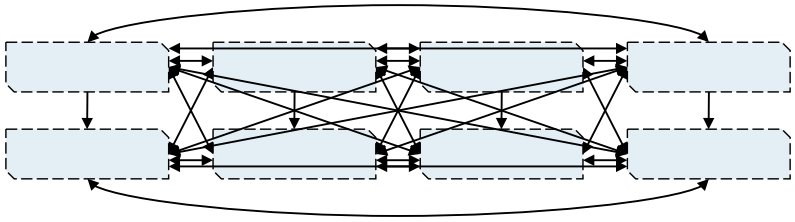


Figure 2.2: CFG of 4 threads with 2 instructions each.

Our first attempt at modeling a lack of fine-grain interthread dependence information was to assume no ordering information whatsoever between threads, even at a coarse granularity. Then, the only ordering information we could assume was that instructions in a thread execute in program order[4].

Translating these insights into a "dynamic CFG" required making nodes out of individual instructions rather than basic blocks. This allows modeling of arbitrary interleaving among instructions executed by different threads. We place directed arcs in both directions between any two instructions that could execute in parallel, and a directed arc between instructions i and $i+1$ in the same thread, indicating that the trace for a thread is followed sequentially. This yields a graph that at first glance resembled a control flow graph (see Figure 2.2); it seemed at first that enough of the structure would be similar to apply dataflow analysis. However, this approach suffers from three major problems.

Problem 1: Too many edges.

[4]Continuing the sequential consistency assumption

Figure 2.1(a) shows a very simple code example of two threads modifying three variables. Even with only three total instructions, we still require several arcs to reflect all the possible concurrency, shown in Figure 2.1(b). This may look manageable; unfortunately, adding arcs over an entire dynamic run leads to an explosion in arcs and the space necessary to keep this graph in memory. Figure 2.2 shows how quickly the number of arcs increases with only four threads, each executing two instructions. For T threads with N instructions each executing concurrently, there are $O(NT)$ edges due to the sequential nature of execution within a thread and $O((NT)^2)$ edges due to potential concurrency: each of the NT nodes has edges to all the nodes in all the other threads.

Problem 2: Arbitrarily delayed analysis.

Unlike a static control flow graph, whose size is bounded by the actual program, the dynamic run-length of a program is unbounded and potentially infinite in size if the program never halts. Since the halting problem is undecidable, analysis could not be completed until the program actually ended, because only then would the actual graph be known. This model of parallel computation quickly becomes intractable.

Problem 3: Conclusions based on impossible paths.

The third problem with this approach is that it can lead to conclusions based on impossible paths[5] through our "dynamic CFG". Recall the TAINTCHECK lifeguard described in Section 1.1. Suppose we were interested in running the TAINTCHECK lifeguard on the code in Figure 2.1(b), where `buf` has been tainted from a prior system call. Instruction 2 in Thread 1 taints `c`. Instructions (1) and (i) propagate taint from the source to their destination. According to the graph, it is valid for instruction (i) to be the immediate successor of instruction (2), implying there is a way for `a` to be tainted by inheriting taint from `c` at instruction (2). Likewise, it is valid for instruction

[5]Paths for the lifeguard analysis to follow when analyzing the application.

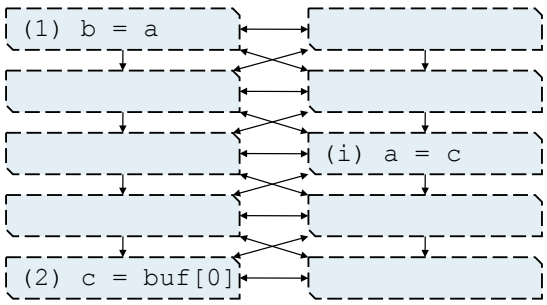


Figure 2.3: Two threads concurrently update `a`, `b` and `c`.

(1) to be the immediate successor of instruction (i), implying `b` is tainted due to `a` being tainted. However, for all three memory locations to be tainted, we must have (2) execute before (i), and (i) before (1)–contradicting the sequential consistency assumption.

2.1.2 Refinement: Bounding Potential Concurrency

We then attempted to refine our model, taking advantage of the finite amount of buffering available to current processors. Modern processors can only have a constant amount of pending instructions, typically on the order of the size of their reorder and/or store buffer, and instruction execution latency is bounded by memory access time. Combining a bounded number of instructions in flight and a bounded execution time per instruction, we can calculate that after a sufficiently long period of time, two instructions in different threads could not have executed concurrently; one must have executed *strictly before* the other.

While this intuition proved useful, it did not solve all the aforementioned problems. Even after modifying our CFG-like approach to include edges only between individual instructions that are potentially concurrent, we could still conclude that an instruction at the end of the program taints the destination of the first instruction of the first thread, by zig-zagging up from the bottom of the graph to the top. This is possible even if each instruction only has edges to three other instructions in the other thread, as depicted in Figure 2.3. Because there are still paths from the end of a thread's execution to its beginning, we can potentially conclude that every address is

tainted for almost the entire execution based on a single taint occurring at the very end.

This led us to consider restricting our dataflow analysis to only a sliding window of instructions at a time, ultimately culminating in a framework we call **Butterfly Analysis**.

2.2 Overview: Butterfly Analysis Thread Execution Model

In this section, we introduce a new model of parallel program execution, which formalizes what it means for one instruction to become globally visible *strictly before* another instruction, and shows how to group instructions into meaningful sliding windows to avoid the problems described in Section 2.1.

2.2.1 Bounding System Concurrency

We rely on a regular signal, or **heartbeat**, to be reliably delivered to all cores. For lifeguards using dynamic binary instrumentation (DBI) to monitor programs, this could be implemented using a token ring; it can also be implemented using a simple piece of hardware that regularly sends a signal to all cores. We will not assume that a heartbeat arrives simultaneously at all cores, only requiring that all cores are guaranteed to receive the signal.[6] We will use this mechanism to break traces into **uncertainty epochs**.

We do not require instantaneous heartbeat delivery, but do assume a maximum skew time for heartbeats to be delivered. By making sure that the time between heartbeats accounts for (i) memory latency for instructions involving reads or writes, (ii) time for all instructions in the reorder and store buffers to become globally visible, and (iii) reception of the heartbeat including the maximum skew in heartbeat delivery time, we can guarantee *non-adjacent* epochs (i.e., epochs that do not share a heartbeat boundary) have strict happens-before relationships.[7]

[6]The signal must be received within some guaranteed maximum latency, or else the skew time measured.

[7]This guarantee is by construction. Time between epochs is always large enough to account for the reorder buffer, store buffer, memory latency, and skew in heartbeat delivery. Instructions more than one epoch apart were already implicitly ordered, since the earlier instruction has committed, with any related store draining from the store

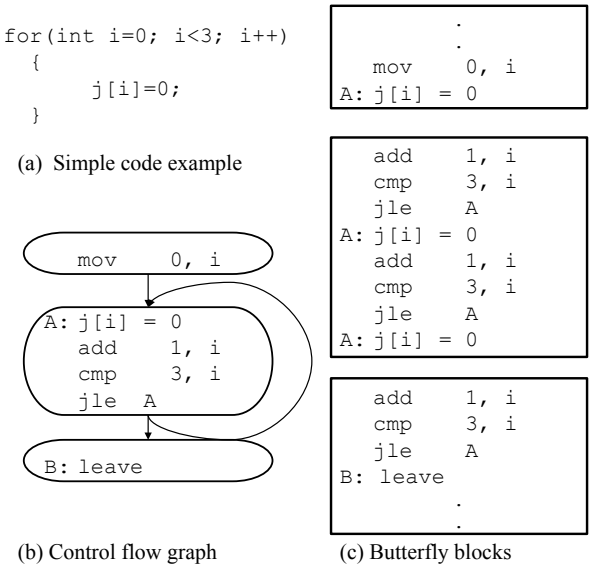


Figure 2.4: Unlike basic blocks (which are static), *butterfly blocks* contain *dynamic* instruction sequences, demarcated by heartbeats.

On the other hand, we will consider instructions in *adjacent* epochs, i.e., epochs that share a heartbeat boundary, to be *potentially concurrent* when they are not in the same thread.

An epoch contains a **block** in each thread, where a block is a series of consecutive instructions, and each block represents approximately the same number of cycles. Note that a block in our model is not equivalent to a standard basic block. As an example, the code in Figure 2.4(a) transforms into a few basic blocks, illustrated as a CFG in Figure 2.4(b), whereas Figure 2.4(c) shows blocks in our model. The epoch boundaries across threads are not precisely synchronized, and correspond to reception of heartbeats. Our model, illustrated in Figure 2.5, incorporates possible delays in receiving the heartbeat into its design. Formally, given an epoch ID l and a thread ID t, a block is uniquely defined by the tuple (l, t). A particular instruction can be specified by (l, t, i), where i is an offset from the start of block (l, t).

Our model has three main assumptions. Our first assumption will be that instructions within a thread are sequentially ordered, continuing our sequential consistency assumption from Sec-

buffer, before the later instruction is even issued. We do assume cache coherency for ordering writes to the same address.

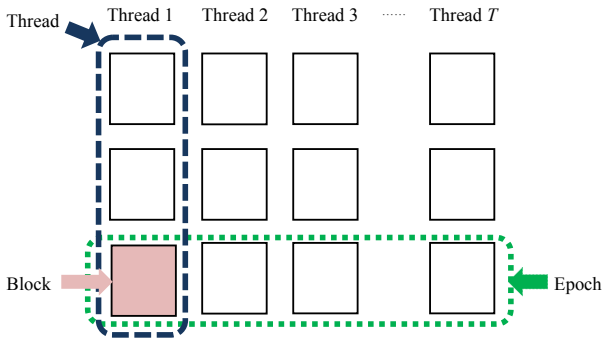
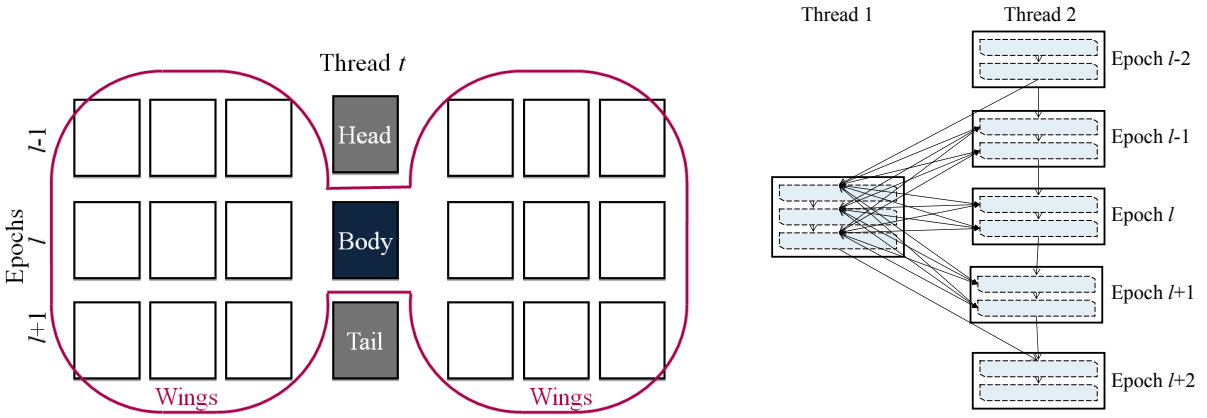


Figure 2.5: A particular block is specified by an epoch id l and thread id t. In reality, epoch boundaries will be staggered and blocks will be of differing sizes.



(a) Instructions **body** interleave with instructions in **wings**. **Head** has already executed, **tail** has not yet executed.

(b) Arcs between instructions indicate possible interleavings.

Figure 2.6: Potential concurrency modeled in butterfly analysis, shown at the **(a)** *block* and **(b)** *instruction* levels.

tion 2.1; we will later relax this assumption.

Our second assumption is that all instructions in epoch l execute (their effects are globally visible) before any instructions in epoch $l+2$, implying that any instructions in epoch l executes strictly after all instructions in epoch $l-2$.

Our third and final assumption is that instructions in block (l,t) can interleave arbitrarily with instructions in blocks of the form $(l-1,t')$, (l,t'), and $(l+1,t')$ where $t' \neq t$. The final two assumptions of this model handle the various possible delays (in receiving a heartbeat, in memory accesses, due to the reorder buffers, etc.). If an instantaneous heartbeat would have

placed an instantaneous instruction j in epoch l, our model will require that instruction j instead will always be in either epoch $l-1$, l or $l+1$.

Butterfly analysis formalizes the intuition that it may be difficult to observe orderings of nearby operations but easier to observe orderings of far apart instructions. We now motivate the term **butterfly**, which takes as parameter a block (l,t); see Figure 2.6(a). We call block (l,t) the **body** of the butterfly, $(l-1,t)$ the **head** of the butterfly and $(l+1,t)$ the **tail** of the butterfly. The head always executes before the body, which executes before the tail. For all threads $t' \neq t$, blocks $(l-1,t')$, (l,t') and $(l+1,t')$ are in the **wings** of block (l,t)'s butterfly.

2.2.2 Butterfly Framework

As described, our framework resembles a graph of parallel execution, where directed edges indicate that instruction i can be the direct predecessor of instruction j. Figure 2.6(b) illustrates this from the perspective of a block in Thread 1, epoch l.

Block $(l,1)$ has edges with arrows on both ends between its instructions and instruction in epochs $l-1$ through $l+1$ of thread 2. There is only one arrow from epochs $l-2$ and one to epoch $l+2$, indicating that the first instruction of $(l,1)$ can immediately follow the last instruction of $(l-2,2)$, and the last instruction of $(l,1)$ can be followed immediately by the first instruction of $(l+2,2)$. Overall, for T threads each with N instructions and epochs of K instructions, the graph contains $O(NKT^2)$ edges.

For our analysis to be truly useful, we must be able to guarantee that we never miss a true error condition (*false negatives*) while keeping the number of safe events that are flagged as errors (*false positives*) as close to zero as possible. While we still wish to adapt dataflow analysis techniques, we will make the final observation that behaving conservatively guarantees zero false negatives and retains the flavor of dataflow analysis. In fact, we will show that with only two passes over each block, we can reach a conclusion about metadata state with zero false negatives.

In this model, there is only a bounded degree of arbitrary interleaving: our dataflow analysis

is done on subgraphs of three contiguous epochs only. Because the analysis considers only three epochs at a time, we introduce state, not normally necessary in dataflow problems. We will call **Strongly Ordered State** (SOS) the state resulting from events that are known to have already occurred, i.e., state resulting from instructions *executed at least two epochs prior.* This state is globally shared. For each block (l, t) there is also a concept of **Local Strongly Ordered State** (LSOS), which is the SOS modified to take into account that, from the perspective of the body block (l, t), all instructions in the head of the butterfly have also executed.

2.3 Background: Cache Coherence and Relaxed Memory Consistency Models

In this section, we discuss one of the challenges facing parallel programmers: how *relaxed memory consistency models* affect program semantics by permitting shared-memory interactions that seemingly conflict with program order. One of this thesis' major contributions is its ability to support dynamic parallel monitoring on shared-memory relaxed memory consistency models so long as the machine provides *cache coherence* and does not violate intra-thread data dependences.

2.3.1 Preserving Program Ordering Within Threads

Processor pipelines are frequently taught as a simple five-stage process: instruction fetch, decode, execute, memory and write-back, where the instruction fetch stage processes instructions in program order. In reality, modern processors deploy *dynamic scheduling*, effectively reordering instructions to reduce pipeline stalls without affecting accuracy for a single thread. Dynamic scheduling leverages *instruction-level parallelism* to execute instructions Out-of-Order (OoO) while maintaining sequential semantics within a thread [50]. In order to maintain sequential semantics, processors must respect *data dependences*. By preserving intra-thread data depen-

dences, dynamic scheduling can improve single-threaded performance without affecting single-threaded correctness [50].

2.3.2 Reasoning About Orderings Across Threads on Parallel Machines

While dynamic scheduling provides sequential semantics to a single thread, on shared-memory multiprocessors instruction reordering can cause the effects of a given thread's instructions to become visible to other threads out of program order. Absent sufficient synchronization, shared-memory interactions between threads, or *inter-thread data dependences*, are non-deterministic.

Cache coherence guarantees that, for a particular memory location, there exists a hypothetical total order of all memory accesses (reads and writes) across all threads which is consistent with the order the individual processors issued the accesses and where the values returned by a read come from the most recent write to the same memory location [28].[8] There are two key properties to cache coherence: *write serialization*, meaning all writes to a single memory location are seen in the same order by all processors, and *write propagation*, meaning all writes eventually become visible (i.e., writes cannot be buffered indefinitely) [28, 51]. Cache coherence dictates that writes to a single location must propagate and become visible, but does not dictate when writes actually become visible; furthermore, the serialization property of cache coherence is limited to a single memory location.

In contrast, *memory consistency* addresses two complementary challenges: when must written values become visible to other threads via reads, and what properties hold concerning reads and writes to different memory locations [28, 51]. Memory consistency dictates which reordering operations are allowed and when writes become visible to other threads. To aid programmers, processor vendors provide ordering and synchronization primitives which disallow certain processor reorderings and can aid in writing correct programs. [4].

[8]While some processors, such as the Intel Single-Chip Cloud Computer, have shipped without providing coherence, most commodity CPUs do provide coherence [28] . The results in this thesis are limited to coherent processors.

Sequential Consistency: An Intuitive Memory Consistency Model

Perhaps the most intuitive memory consistency model is sequential consistency (SC). Sequential consistency is the closest analogue to single-processor sequential execution. The guarantee made by a SC processor is that there exists a total order of application instructions across all threads which is consistent with program order across each thread. Unfortunately, while this is the most intuitive memory consistency model for programmers to reason about, it is also one of the most constraining, as it disallows many compiler and hardware optimizations. For this reason, modern processors do not provide sequential consistency.

2.3.3 Cache Coherence

While most processors do not provide SC, they do provide cache coherence. Recall that cache coherence guarantees that there exists a hypothetical total order of all memory accesses (reads and writes) across all threads which is consistent with the order the individual cores issued the accesses and where the values returned by a read come from the most recent write to the same memory location [28]. Unlike sequential consistency, cache coherence provides guarantees only with respect to a particular memory location and the memory accesses (which can appear to become decoupled from the actual instructions) associated with that location. The ordering is only "hypothetical" because while all writes are serialized with respect to each other, reads which return the value of a particular write operation are not necessarily ordered with respect to each other, and instead ordered with respect to the immediately preceding and succeeding write operations.

Cache coherence is a property of a shared memory machine, implemented by a *cache coherence protocol*. On modern processors, it is insufficient to merely claim that a protocol is correct; cache coherent protocols are subject to formal verification using a variety of techniques [12, 16, 29, 55, 88–90, 105, 107]. Among the many techniques [88], some explicitly support *relaxed memory consistency models* [90] or heterogeneous coherence hierarchies [16],

while others verify the behavior of the interconnect [107]. The most commonly-taught family of coherence protocols, MSI/MESI [28], have undergone verification [29, 55]. While actual processor coherence protocols may be considered proprietary, companies internally perform verification to ensure correctness of their coherence protocol [12].

The MSI/MESI family of coherence protocols include different subsets of several states: modified (M), exclusive (E), shared (S), invalid (I) [28, 115]. The three most critical, hinted at by the MSI protocol, are the modified, shared and invalid states. This corresponds easily with the SWMR property (single-writer–multiple-reader) set forth by Wood *et al.* [115], which explains coherence as a per-memory-location timeline of epochs, where in each epoch either exactly one thread is a writer or many threads can be reading. Likewise, MSI guarantees that at any point in time, exactly one thread is the writer (modified) or many threads are readers (shared)[9] MESI builds upon MSI by adding an exclusive state. [10] The correspondence between MSI and SWMR should be clear; SWMR simply tracks transitions between MSI states over time, particularly between whether any thread is in the M (corresponding to an exclusive writer) versus S (corresponding to multiple readers).

It is important to note that cache coherence makes no guarantees that coherence arcs between independent locations can be combined to form a total order of application events without cycles. The interactions between independent memory location are handled by memory consistency, not coherence. Furthermore, coherence specifies that all operations to a given address can be serialized but does not specify when a given write must become visible; this is again specified by consistency [28].

[9]The invalid state indicates that while a block may be present in the cache, it has been invalidated by another core and cannot be safely read or written to without generating coherence traffic to transition into the modified or shared states.

[10]There are other members of this family, e.g., MOESI, which for simplicity are not discussed here.

2.3.4 Relaxed Consistency Models

Instead of sequential consistency, modern processors provide a range of relaxed memory consistency models [2, 46, 72, 112]. Consistency models range from more formal specifications, such as release consistency [41] to the more descriptive specifications published by processor manufacturers [2, 46, 72, 112]. The formal models [41, 101] tend to be clearer with less arbitrary edge cases, whereas manufacturers' descriptive specifications are less clear and may carve out certain behaviors with no guarantees. At the same time, manufacturer's consistency models match actual commodity processors for sale today. While some more theoretical models, such as x86-TSO [101] have been proposed, they are not a perfect match to processor semantics. For instance, while x86 appears to provide total store order (TSO), it deviates on an entire class of operations, namely, stores of size larger than 1B that are non-aligned. We easily verified that in the event of a processor issuing a write to a non-aligned word that wraps two cache lines, x86 does not even guarantee write atomicity. Compared with PowerPC and ARM, which provide very relaxed consistency models [46, 72], Intel and AMD via x86 and IA-64 provide a much stronger consistency model [2]. SPARC frequently offers a programmer a choice of consistency model: TSO, as well as progressively weaker memory models such as Partial Store Order (PSO) and Relaxed Memory Order (RMO) [112]. Programs written for RMO will work on TSO and PSO, but the opposite is not guaranteed.

Adve and Gharachorloo [4] performed a survey of shared memory consistency models, which showed that consistency models can be roughly classified along three axes. The first axis measures which memory ordering relationships (e.g., read-to-read, write-to-read, read-to-write and write-to-write) can be relaxed among non-causally related, or non-dependent, memory locations. The second axis corresponds to write visibility, in particular two conditions: (1) whether a thread can read its own write early and (2) whether a thread can read others' writes early. The third axis concerns itself with what ordering primitives the architecture makes available to programmers to prevent reordering. For instance, on strong consistency models like TSO, it is sufficient

to transform reads and writes with a sequence of "read-modify-write"; for a read, this implies that the write is writing the same value as the read; to transform a write into a read-modify-write, the value written is unaffected by the read. On more relaxed architectures such as release consistency, special *release* and *acquire* instructions are available to the programmer, and these instructions carry specific semantics.

It is the programmer's job to adequately synchronize his/her program (e.g., eliminate data races) using the available ordering primitives. While many relaxed consistency models provide sequentially consistent semantics to *data-race-free programs* [41–43], the behavior of insufficiently synchronized programs is limited only by the specification of the consistency model.

2.3.5 This Thesis: Leverage Cache Coherence and Respect of Intra-Thread Data Dependences

In this thesis, we present dataflow analysis-based dynamic parallel monitoring. One particular restriction of dataflow analysis-based dynamic parallel monitoring is that lifeguards written in this framework cannot combine metadata for two different memory locations. This restriction is inspired by the interplay of consistency models and coherence. On the one hand, how accesses to two different memory locations can be reordered depends entirely on the specific memory consistency model. On the other hand, regardless of the memory consistency model, cache coherence guarantees a total order of conflicting accesses to the same memory location. By restricting our focus to processors that (1) support cache coherence and (2) respect intra-thread data dependences, we can provide provable guarantees for our dynamic analysis framework across all relaxed consistency models, *as long as our analysis is restricted to analyzing one memory location at a time*. To meet this restriction, we will not support analyses that combine metadata for independent memory locations.

For example, an analysis that counts the number of stores to memory locations m such $m \equiv 2 \mod 3$ combines metadata for m and $m + 3$, among other locations–this analysis cannot be

supported by dataflow analysis-based dynamic parallel monitoring. In contrast, an analysis that computes the parity of the number of stores for each memory location m in the program maintains metadata separately for each location m; this analysis can be supported by dataflow analysis-based dynamic parallel monitoring. Almost every commodity processor on the market satisfies both (1) and (2) [2, 46, 72, 112].

2.4 Model of Thread Execution: Supporting Relaxed Memory Models

In the previous section, we discussed relaxed memory consistency models and cache coherence. With this background at hand, we can now finish building the thread execution model that Butterfly Analysis depends on.

Our butterfly framework is well-suited to relaxed memory models. There are relaxed assumptions on when a memory access becomes globally visible (two epochs later). There are relaxed assumptions on memory access interleavings within a sliding window. In fact, the analysis accounts for different threads possibly observing different orderings of the same accesses, e.g., two writes A and B such that thread 1 may observe A before B while thread 2 may observe B before A. We make only the weak assumptions that (i) from a given thread's view, its own intra-thread dependences are respected, and (ii) cache coherency orders writes to the same address. As discussed in Sections 3.1 and 3.3, our analysis for what happens at other threads is based on set operations; set union and intersection are both commutative operations, and set difference only becomes a problem if we change metadata before an instruction was able to read it, which will not happen given our above weak assumptions.

This suffices for our reaching definitions and available expressions analyses in Section 3.1, and hence any lifeguards based on them. However, for lifeguards such as TAINTCHECK, there may be more false positives with relaxed models than when assuming sequential consistency (as

discussed in Section 3.3.2).

2.4.1 Lifeguards As Two Pass Algorithms

We describe a two-pass algorithm for a generic lifeguard, based on the observation that starting with the global SOS as the default state, lifeguard checks can be influenced by local state and/or state produced by the wings. We split our algorithm into two passes accordingly.

In the first pass, we perform our dataflow analysis using locally available state (i.e., ignoring the wings), and produce a summary of lifeguard-relevant events (**step 1**). Next, the threads compute the meet of all the summaries produced in the wings (**step 2**). In a second pass, we repeat our dataflow analysis, this time incorporating state from the wings, and performing necessary checks as specified by the lifeguard writer (**step 3**). Finally, the threads agree on a summarization of the entire epoch's activity, and an update to the SOS is computed (**step 4**).

The lifeguard writer specifies the events the dataflow analysis will track, the meet operation, the metadata format, and the checking algorithm. Examples will be given in Section 3.3.

2.4.2 Valid Ordering

We introduce the concept of a **valid ordering** O_l, which is a total sequential ordering of all the instructions in the first l epochs, where the ordering respects the assumptions of butterfly analysis. More formally:

Definition 2.4.1. *O_l is a **valid ordering (VO)** if:*

- INCLUDES ALL INSTRUCTIONS THROUGH EPOCH l

  All instructions from epochs 0 to l are included exactly once, with no instructions from epochs $> l$ included.

- NON-ADJACENT EPOCHS ARE ORDERED *Instructions in non-adjacent epochs are ordered,* e.g.*, all instructions in epoch k occur before any instructions in epoch $k+2$ $\forall 0 \leq k \leq l$.*

- INSTRUCTIONS IN WINGS ARE CONCURRENT *Instructions in block (k, t) can interleave arbitrarily with instructions in blocks $(k-1, t')$, (k, t') and $(k+1, t')$ for any thread $t' \neq t$.*
- [IF ASSUMING SEQUENTIAL CONSISTENCY] INSTRUCTIONS IN A THREAD ARE SEQUENTIALLY ORDERED *Restricting O_l to thread t yields the trace of instructions for thread t in program order. This assumption will be ignored for relaxed memory models.*

A **path** to an instruction (block) is the prefix of a valid ordering that ends just before the instruction (the first instruction in the block, respectively).

We observe that the set of valid orderings is a superset of the possible application orderings[11]: Nearly all machines support at least cache coherency, which creates a globally consistent total order among conflicting accesses to the same location. Because our analysis considers each definition event independently, our approach has no false negatives (as argued in Section 2.4), even for relaxed memory models. This approach aligns well with guarantees of cache coherence, which only promises ordering of causally related memory locations.

We will not claim that we can construct an ordering for multiple locations simultaneously. We expect to conclude that two instruction definitions d_k and d_j both reach the end of epoch l even if the program semantics state exactly one of d_k and d_j will reach that far. We use valid orderings as a conservative approximation of what orderings a given thread could have observed.

Valid orderings are a key contribution of the thread execution model. In fact, when we provide correctness proofs in the following chapters they will almost always be in reference to valid orderings[12].

Valid Orderings Do Not Imply Sequential Consistency

There may appear to be a contradiction between the use of valid orderings in proofs throughout this thesis, such as those in later chapters, and the claim that dataflow analysis-based dynamic

[11]Note that the set of all possible application orderings does not imply that there exists any one application ordering that all threads agree on. Our use of valid orderings is to prove existential or universal guarantees. For example, if under all valid orderings an address m is `untainted`, then it must also be the case that every thread agrees m is untainted, *even if they have differing views on instruction ordering*.

[12]In Chrysalis Analysis, they will be to a refinement of valid orderings known as *valid vector orderings*.

parallel monitoring supports relaxed memory consistency models. *The use of valid orderings does not imply an assumption of sequential consistency.* In Section 2.3, we acknowledged a key restriction: lifeguards written in this framework cannot group metadata for two different memory locations, e.g., we cannot have a lifeguard that counts the number of memory locations m such $m \equiv 2 \mod 3$.

The proofs presented for reaching definitions, available expressions and ADDRCHECK reason about the metadata status for exactly one memory location at a time. As discussed in Section 2.3, *when restricting focus to a single memory location, cache coherence guarantees a hypothetical total order of reads and writes exists consistent with each thread's execution.* For instance, a definition d is only killed by a subsequent redefinition of d–mapping to the same location. An expression e is killed if any of its constituent (memory) operands has a new value written to it. Whether a memory location acquires a new value is determined independently. ADDRCHECK reasons about whether accesses/frees to a memory location m are always to allocated data; the metadata status for m depends solely on changes to m's allocation status. While valid orderings include all instructions, *each proof relies only on cache coherence and respect of intra-thread data-dependences.*[13]

The use of additional instructions, such as all instructions belonging to epoch l, within the definition of valid orderings was designed to (1) make the definition simple, (2) easily extend to TAINTCHECK and, primarily, (3) lead more naturally to summaries of epochs and (4) enable proofs about the SOS, which summarizes the effects of all prior instructions. Valid orderings are so named since they do not violate any of the fundamental assumptions of the butterfly thread execution model: all valid orderings will always enforce the ordering relation between instructions from non-adjacent epochs while considering instructions from adjacent epochs to be concurrent.

[13] ADDRCHECK additionally requires correct annotations of `free`/`malloc` calls.

TAINTCHECK**: Inheritance Complications**

The proofs and mechanisms presented for TAINTCHECK require *inheritance*, but never more than the combination of (1) respect of intra-thread data dependences and (2) cache coherence. Our reliance on cache coherence for TAINTCHECK is similar as for other lifeguards: when analysis is restricted to a single address, cache coherence orders all competing accesses to the same memory location.

The presence of *inheritance* within TAINTCHECK complicates analysis. The sequence of instructions `$r1 ← b; $r2 ← c; $r2 ← $r2 + $r1; a ← $r2;` in one thread implies `a = b + c`, meaning `a` inherits its metadata from `b` and `c`. Unlike prior lifeguards, to resolve metadata for `a`, TAINTCHECK requires knowing the metadata values of `b` and `c`. Note, however, that this metadata dependence aligns exactly with an intra-thread data-dependence: *for* `a` *to inherit from* `b` *and* `c`*,* `a` *must be data dependent on both* `b` *and* `c`*!*

TAINTCHECK introduces a `resolve` algorithm, first in Section 3.3.2 and later in Sections 4.5, 5.5 and 5.7, to resolve the metadata for `a` taking into account inheritance from parents `b` and `c`. In particular, when recursively resolving metadata for `b` and `c`, `resolve` may recursively need to resolve another thread's computation `b = d + e`. Once more, such inheritance by lifeguard metadata results directly from intra-thread data dependences within the actual application. While the `resolve` algorithm does restrict the order in which it recursively explores parents, for relaxed memory models these constraints effectively require (1) respect of ordering constraints among non-adjacent epochs and (2) never repeating the same instruction. Instructions can (deliberately) be explored out of program order, in order to support weak consistency models such as PowerPC and ARM which aggressively reorder instructions – but never violate intra-thread data dependences when doing so [46, 72].

2.5 Chapter Summary

In this chapter, we have motivated the design decisions in creating the thread execution model which will be referenced throughout this thesis. We explored why a CFG-like representation was our initial inspiration as well as the drawbacks to the naive extension of control flow graphs to the domain of dynamic parallel monitoring. We introduced the thread execution model designed for Butterfly Analysis, which bounds system concurrency without requiring access to inter-thread data dependences. We motivated how this framework avoids the pitfalls that a CFG-like representation would otherwise incur. We explored background related to shared-memory relaxed consistency models as well as cache coherence. We showed how our thread execution model was designed to support any shared-memory relaxed memory consistency model so long as it provides cache-coherence. We defined *valid orderings*, a construct tied to our thread execution model which enables the proofs, and showed how the use of valid orderings as a proof mechanism never requires more than (1) cache coherence and (2) respect of intra-thread data dependences.

In Chapter 3, which follows, we will show how to build on this thread execution model to enable Butterfly Analysis, prove correctness guarantees and experimentally evaluate a prototype implementation.

Chapter 3

Butterfly Analysis: Adapting Dataflow Analysis to Dynamic Parallel Monitoring

In Chapter 2, we explored the derivation of a thread execution model used in Butterfly Analysis. In particular, we showed how, by dividing dynamic execution into *epochs*, we could bound concurrency to a three epoch window. Within the three epoch window, we derived the *butterfly*, shown in Figure 2.6, and showed how it models concurrency in the *wings* of the butterfly while respecting that the *head* of the butterfly effectively executes before the *body* which itself effectively executes before the *tail*.[1] Even when concurrent interactions are constrained to a three epoch sliding window, the epochs are sized on the order of thousands to tens of thousands of instructions per thread (or more); exploring the entire space of potential interactions can still lead to a combinatorial explosion of interleavings that would have to be examined. In this chapter, we will explore how adapting dataflow analysis to a dynamic setting allowed us to present an efficient parallel dynamic analysis framework.

[1] Any reorderings are guaranteed to preserve intra-thread data dependences, so from a thread's perspective of its own execution, it is as though it executed in program order.

3.1 Butterfly Analysis: A Dataflow Analysis-Based Dynamic Parallel Monitoring Framework

This section presents our dataflow analysis-based dynamic parallel monitoring framework, called Butterfly Analysis. Where our derivation of the thread execution model in Chapter 2 drew on domain knowledge of both parallel computer architecture and compiler analysis, here our solution draws inspiration from a particular area of static dataflow analysis, namely region-based analysis.[2] Specifically, we adapt reaching definitions and available expressions [5], two simple forward dataflow analysis problems that exhibit a generate/propagate structure common to many other dataflow analysis problems, to a new, dynamic setting of parallel online program monitoring. Previous studies [23, 124, 125] have shown that this propagate/generate structure is a common structure for lifeguards, including lifeguards that check for security exploits, memory bugs, and data races.

In this section, we show how reaching definitions and available expressions can be formulated as generic lifeguards using butterfly analysis. In standard dataflow analysis, there are equations for calculating IN, OUT, $\mathcal{G}$ and $\mathcal{K}$ (*e.g.,* $\text{OUT} = \mathcal{G} \cup (\text{IN} - \mathcal{K})$). Our approach extends beyond these four, as discussed below. In our setting, the lifeguard's stored metadata tracks definitions or expressions, respectively, that are known to have reached epoch l. While the generic lifeguards do not define specific checks, their IN and OUT calculations provide the information useful for a variety of checks. (Section 3.3 shows how our generic lifeguards can be instantiated as ADDRCHECK and TAINTCHECK lifeguards.)

The key to our efficient analysis is that we formulate the analysis equations to fit the butterfly assumptions, as follows:

- We perform our analysis over a sliding window of three epochs rather than the entire execution trace. This not only enables our analysis to proceed as the application executes,

[2]Region-based analysis is alternatively known as interval analysis.

it also bounds the complexity of our analysis.

- We require only two passes over each epoch. The time per pass is proportional to the complexity of the checking algorithm provided by the lifeguard writer.
- We introduce state (SOS) that summarizes the effects of instructions in the distant past (i.e., all instructions prior to the current sliding window). This enables using a sliding window model without missing any errors.
- The symmetric treatment of the instructions/blocks in the wings means we can efficiently capture the effects of all the instructions in the wings. To do so, we add four new primitives: GEN-SIDE-IN, GEN-SIDE-OUT, KILL-SIDE-IN and KILL-SIDE-OUT, as defined below.

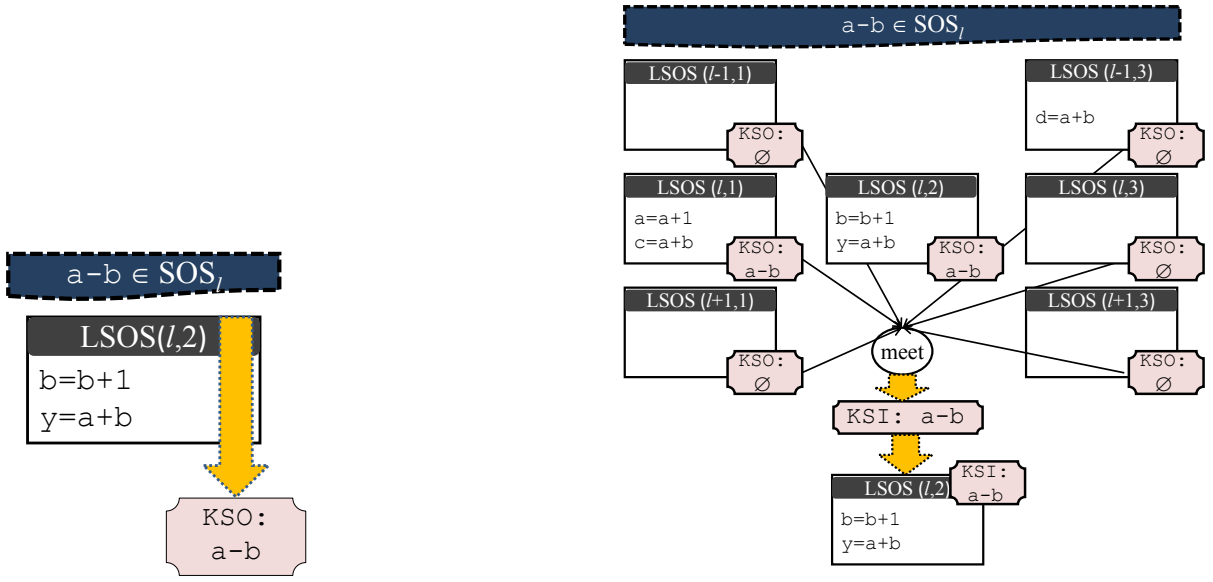
In the following sections, $\mathcal{G}_{l,t,i}$, $\mathcal{K}_{l,t,i}$, $\mathcal{G}_{l,t}$ and $\mathcal{K}_{l,t}$ refer to their sequential formulations, either over a single instruction (l,t,i) or an entire block (l,t). GEN-SIDE-OUT$_{l,t}$ will calculate the elements (definitions or expressions) block (l,t) generated that are visible when (l,t) is in the wings of a butterfly for block (l',t'). Likewise, KILL-SIDE-OUT$_{l,t}$ calculates the elements block (l,t) kills that are visible when (l,t) is in the wings for block (l',t'). GEN-SIDE-IN$_{l',t'}$ and KILL-SIDE-IN$_{l',t'}$ combine the GEN-SIDE-OUT and KILL-SIDE-OUT, respectively, of all blocks in the wings of block (l',t'). The strongly ordered state SOS$_l$, parameterized by an epoch l, will contain any elements no later than epoch $l-2$ that could reach epoch l.

3.2 Butterfly Analysis: Canonical Examples

To motivate our work, we examine Figure 3.1, a available expressions example.

Processing an Epoch

First, we note that for any sliding window of size 3, the strongly ordered states SOS$_{l-1}$, SOS$_l$ and SOS$_{l+1}$, which summarize execution through epochs $l-3$, $l-2$, and $l-1$, respectively, are available after the lifeguard has consumed events through $l-1$. This can be shown by induction,



(a) 1st Pass: Compute KILL-SIDE-OUT (KSO)

(b) 2nd Pass: Aggregate KSOs from the wings, combine into KILL-SIDE-IN

Figure 3.1: Computing KILL-SIDE-OUT and KILL-SIDE-IN in available expressions, from the perspective of the body, block $(l, 2)$, of the butterfly. Boxes with beveled edges are summaries.

as follows. The very first butterfly uses only epochs 0 and 1, and has $\text{SOS}_0 = \text{SOS}_1 = \varnothing$. After concluding all butterflies with bodies in epoch 0, we have SOS_2 as well. From then on, we inductively have the correct SOS for each epoch in the butterfly.

Now consider Figure 3.1(a). The LSOS is available for block $(l, 2)$ because the head (not shown) is available and so is SOS_l. In our first pass we discover that block $(l, 2)$ kills expression `a-b` through a redefinition of `b`. The epoch's summary need only be generated once (in available expressions, the summary contains the killed expressions), so we do not need to regenerate the summary as the block changes position in the sliding window. After the first butterfly, we are performing a first pass only on the blocks in the newest epoch under consideration; the summaries for older blocks have already been completed.

Using that information, we now examine Figure 3.1(b). This shows the entire butterfly for block $(l, 2)$. As part of step 2, summaries from all blocks in the wings (computed earlier, as argued above) are first collected and combined (represented by the circle labeled "meet"), pro-

ducing one summary for the entire wings (in available expressions, this is KILL-SIDE-IN$_{l,2}$). (Note: The meet function for KILL-SIDE-OUT is not the standard available expressions intersection, but rather computed as the union. This is explained in Section 3.2.2.) Finally, $(l, 2)$ repeats its analysis and performs checks (step 3), noting that it is not the only block to kill `a-b`.

Once the second pass is over, an epoch summary is created (step 4, not shown). In the example, epoch l witnesses the killing of expression `a-b`, as well as the generation of expression `a+b`. Any ordering of instructions in epochs $l - 1$ and l (empty blocks contain no instructions relevant to the analysis) yields $a + b$ defined at the end. Hence, `a+b` is added to SOS$_{l+2}$, and `a-b` is removed.

Note that there is a single writer for each of the data structures (one of the threads can be nominated to act as master for global objects such as the SOS), and objects are not modified after being released for reading. Hence, synchronizing accesses to the lifeguard metadata is unnecessary.

In Sections 3.2.1 and 3.2.2, we will formalize the well-known problems of reaching definitions and available expressions [5] using butterfly analysis.

Valid Ordering

Recall the definitions of *valid orderings* and *paths* from Section 2.4.2: A valid ordering O_k is a total sequential ordering of all the instructions in the first k epochs, where the ordering respects the assumptions of butterfly analysis. A path to an instruction (block) is the prefix of a valid ordering that ends just before the instruction (the first instruction in the block, respectively).

We observe that the set of valid orderings is a superset of the possible application orderings: Nearly all machines support at least cache coherency, which creates a globally consistent total order among conflicting accesses to the same location. Because our analysis considers each definition event independently, our approach has no false negatives (as argued in Section 2.4), even for relaxed memory models.

We will not claim that we can construct an ordering for multiple locations simultaneously. We expect to conclude that two instruction definitions d_k and d_j both reach the end of epoch l even if the program semantics state exactly one of d_k and d_j will reach that far. We use valid orderings as a conservative approximation of what orderings a given thread could have observed.

3.2.1 Dynamic Parallel Reaching Definitions

In Butterfly Analysis, generating a definition is global; a definition in block (l, t) is conservatively considered visible to any block (l', t') in its wings, and vice versa. Conversely, killing a definition in butterfly analysis is inherently local; it only kills the definition at a particular point in that block, making no guarantee about whether the definition may still reach by a different path or even a later redefinition in the same block. For this reason, we conservatively set $\text{KILL-SIDE-OUT}_{l,t} = \text{KILL-SIDE-IN}_{l,t} = \varnothing$ in our reaching definitions analysis, and do not rely on these primitives.

Generating and Killing At The Block Level

Let $\mathcal{G}_{l,t,i}$ be the set of definitions generated by instruction (l, t, i): $\mathcal{G}_{l,t,i} = \{d\}$ if and only if instruction (l, t, i) generates definition d, and is empty otherwise. Similarly, let $\mathcal{K}_{l,t,i}$ be the set of definitions killed by instruction (l, t, i). Define $\mathcal{G}_{l,t,(i,i)} = \mathcal{G}_{l,t,i}$ and $\mathcal{G}_{l,t,(i,j)} = \mathcal{G}_{l,t,j} \cup (\mathcal{G}_{l,t,(i,j-1)} - \mathcal{K}_{l,t,j})$ for $j > i$. We can assume a symmetric definition for $\mathcal{K}_{l,t,(i,j)}$. Then for a block (l, t) of $n + 1$ instructions, let $\mathcal{G}_{l,t} = \mathcal{G}_{l,t,(0,n)}$ and $\mathcal{K}_{l,t,} = \mathcal{K}_{l,t,(0,n)}$.

Let $\text{GEN-SIDE-OUT}_{l,t}$ represent the set of generated definitions a block (l, t) exposes to another block (l', t') anytime it is in the wings of a butterfly with body (l', t'). Because the body of the butterfly can execute anywhere in relation to its wings, we must take the union of the $\mathcal{G}_{l,t,i}$. Let $\text{GEN-SIDE-IN}_{l,t}$ represent the set of expressions visible to block (l, t) that are killed by the wings.

$$\text{GEN-SIDE-OUT}_{l,t} = \bigcup_i \mathcal{G}_{l,t,i}$$

$$\text{GEN-SIDE-IN}_{l,t} = \bigcup_{l-1 \leq l' \leq l+1} \bigcup_{t' \neq t} \text{GEN-SIDE-OUT}_{l',t'}$$

In reaching definitions, KILL-SIDE-IN = KILL-SIDE-OUT = $\varnothing$, as no block has enough information to know that along every path to a particular instruction, definition d has been killed.

Generating and Killing Across An Epoch

The concept of an epoch does not exist in standard reaching definitions. We will propose extensions to generating and killing that allow us to summarize the actions of all blocks in a particular epoch l. These definitions will enable us to define reaching an entire epoch l to mean that there is some valid ordering of the instructions in the first l epochs such that running a sequential reaching definitions analysis will conclude that d_k reaches. We calculate:

$$\mathcal{G}_l = \bigcup_t \mathcal{G}_{l,t}$$

$$\mathcal{K}_l = \bigcup_t (\mathcal{K}_{l,t} \cap (\bigcup_{t' \neq t} \mathcal{K}_{(l-1,l),t'} \cup \text{NOT-}\mathcal{G}_{(l-1,l),t'}))$$

$$\text{where } \mathcal{K}_{(l-1,l),t} = (\mathcal{K}_{l-1,t} - \mathcal{G}_{l,t}) \cup \mathcal{K}_{l,t}$$

$$\text{and NOT-}\mathcal{G}_{(l-1,l),t} = \{d_k | d_k \notin \mathcal{G}_{l-1,t} \wedge d_k \notin \mathcal{G}_{l,t}\}$$

Intuitively, the formula for $\mathcal{G}_l$ states that any particular definition that can reach the end of a block may reach the end of an epoch, because there is a valid ordering such that the instructions in block (l, t) are last. Likewise, the formula for $\mathcal{K}_l$ indicates it is harder to kill a definition d_k, as at least one block (l, t) must explicitly kill d_k and all other threads must either not generate d_k or else kill d_k (technically, not-generating and killing must span the two epochs $l - 1$ and l, as indicated in the formulas).

We define the set $\mathcal{G}(O_k)$ to be the set of definitions that, if we were to execute all instructions in order O_k, would be defined at the end of O_k. The correctness of $\mathcal{G}_l$ and $\mathcal{K}_l$ are shown by the following lemma:

Lemma 1. *If $d_k \in \mathcal{G}_l$ then there exists a valid ordering O_l such $d_k \in \mathcal{G}(O_l)$. If $d_k \in \mathcal{K}_l$ then under all valid orderings O_l, $d_k \notin \mathcal{G}(O_l)$.*

Proof. Each statement is proved independently.

If $d_k \in \mathcal{G}_l$ then there exists a valid ordering O_l such that $d_k \in \mathcal{G}(O_l)$.

If $d_k \in \mathcal{G}_l$, then there is some block (l, t) such that $d_k \in \mathcal{G}_{l,t}$, implying there exists some index i such that $d_k \in \mathcal{G}_{l,t,i} \wedge \forall j > i, d_k \notin \mathcal{K}_{l,t,j}$. Then O_l is any valid ordering where the instructions in block (l, t) are last. □

If $d_k \in \mathcal{K}_l$ then under all valid orderings O_l, $d_k \notin \mathcal{G}(O_l)$.

This follows by construction. If $d_k \in \mathcal{K}_l$ then there exists at least one thread t such that $d_k \in \mathcal{K}_{l,t}$. For all other threads $t' \neq t$ it must be the case that d_k was killed during epochs $l-1$ through l and not subsequently regenerated, or else that it was simply not generated. Either way, any possible instruction that generates d_k is followed by a kill of d_k within the same thread. So, d_k is not generated by epochs l or $l-1$, and the KILL in block (l, t) occurs strictly after any GEN in epochs $l-2$ or earlier. □

Note that while we construct valid orderings O_l for all instructions in epochs $[0, l]$, our proof only relies on the ability to order instructions that affect d.

Updating State

Any definition $d_k \in \text{SOS}_l$ was generated by an instruction that came strictly earlier than any instruction in epoch l. We require the following invariant for SOS_l:

$$d_k \in \text{SOS}_l \text{ if and only if } \exists O_{l-2} \text{ s.t. } d_k \in \mathcal{G}(O_{l-2})$$

To achieve this, we use the following rule for updating SOS:

$$\text{SOS}_l := \mathcal{G}_{l-2} \cup (\text{SOS}_{l-1} - \mathcal{K}_{l-2}) \quad \forall l \geq 2$$

$$\text{SOS}_0 = \text{SOS}_1 = \varnothing$$

Lemma 2. $\text{SOS}_l := \mathcal{G}_{l-2} \cup (\text{SOS}_{l-1} - \mathcal{K}_{l-2})$ *achieves the invariant.*

Proof. Base Case. $\text{SOS}_0 = \text{SOS}_1 = \varnothing$. According to the invariant, we wish to show $\text{SOS}_2 = \mathcal{G}(O_0) = \mathcal{G}_0$.

Any definition $d_k \in \text{SOS}_2$ must be generated by some instruction $(0, t, i)$ in block $(0, t)$, implying it is in $\mathcal{G}_{(0,t)}$ and $\mathcal{G}_0$. We can construct a valid ordering O_0 with all instructions in block $(0, t)$ last, so the invariant is satisfied.

Inductive hypothesis: If $s \leq l$, $\text{SOS}_s := \mathcal{G}_{s-2} \cup (\text{SOS}_{s-1} - \mathcal{K}_{s-2})$ achieves the invariant.

Inductive step: Consider the SOS for epoch $l+1$. It must include everything generated by epoch $l-1$, which is $\mathcal{G}_{l-1}$. Now, we must consider how many definitions $d_k \in \text{SOS}_l \wedge d_k \notin \text{SOS}_{l+1}$, which are precisely those definitions d_k such that for all valid orderings, epoch $l-1$ kills d_k. This is exactly what $\mathcal{K}_{l-1}$ calculates; the elements of $\mathcal{K}_{l-1}$ are precisely those that should be removed from the SOS_l when creating SOS_{l+1}. This yields the equation: $\text{SOS}_{l+1} = \mathcal{G}_{l-1} \bigcup (\text{SOS}_l - \mathcal{K}_{l-1})$. □

Recall that the Local Strongly Ordered State for a block (l, t), denoted $\text{LSOS}_{l,t}$, represents the SOS_l augmented to include instructions in the head that were already processed. The invariant required for the LSOS is:

$$d_k \in \text{LSOS}_{l,t} \text{ iff } \exists \text{ valid ordering } O \text{ of instructions in epochs } [0, l-2] \text{ and block } (l-1, t) \text{ s.t. } d_k \in \mathcal{G}(O)$$

To achieve this, we use the following LSOS update rule:

$$\text{LSOS}_{l,t} = \mathcal{G}_{l-1,t} \cup (\text{SOS}_l - \mathcal{K}_{l-1,t}) \cup \{d_k | d_k \in \text{SOS}_l \wedge d_k \in \mathcal{K}_{l-1,t} \wedge \exists t' \neq t \text{ s.t. } d_k \in \mathcal{G}_{l-2,t'}\}$$

Let $\text{LSOS}_{l,t,k}$ denote the LSOS after k instructions have executed.

$$\text{LSOS}_{l,t,k} = \begin{cases} \text{LSOS}_{l,t} \text{ if } k = 0 \\ \mathcal{G}_{l,t,k-1} \cup (\text{LSOS}_{l,t,k-1} - \mathcal{K}_{l,t,k-1}) \text{ ow} \end{cases}$$

This is the standard $\text{OUT} = \mathcal{G} \cup (\text{IN} - \mathcal{K})$ formula, with $\text{LSOS}_{l,t,k-1}$ acting as IN and $\text{LSOS}_{l,t,k}$ as OUT.

Calculating In and Out

Let $\text{IN}_{l,t,0} = \text{IN}_{l,t}$ represent the set of definitions that could possibly reach the beginning of block (l, t). $\text{IN}_{l,t}$ should be the union of the set of valid definitions along all possible paths to instruction $(l, t, 0)$. Let $\text{IN}_{l,t,i}$ be the set of definitions that reach instruction (l, t, i). We have:

$$\text{IN}_{l,t} = \text{GEN-SIDE-IN}_{l,t} \bigcup \text{LSOS}_{l,t}$$

$$\text{IN}_{l,t,i} = \text{GEN-SIDE-IN}_{l,t} \cup \text{LSOS}_{l,t,i}$$

Let $\text{OUT}_{l,t,i}$ and $\text{OUT}_{l,t}$ be the sets of definitions that are still defined after executing instruction (l, t, i) or block (l, t), respectively:

$$\text{OUT}_{l,t,i} = \mathcal{G}_{l,t,i} \bigcup (\text{IN}_{l,t,i} - \mathcal{K}_{l,t,i})$$

$$\text{OUT}_{l,t} = \mathcal{G}_{l,t} \bigcup (\text{IN}_{l,t} - \mathcal{K}_{l,t})$$

Using reaching definitions as a lifeguard, we have now shown how to compute the checks, the OUT computation.

Applying the Two-Pass Algorithm

We can now set our parameters for the two-pass algorithm proposed in Section 2.4.1. For step 1, our local computations are $\mathcal{G}_{l,t}$, $\mathcal{K}_{l,t}$ and $\text{LSOS}_{l,t}$. These are used for our checking algorithm. The summary information is $\text{GEN-SIDE-OUT}_{l,t}$. For step 2, the meet function is $\cup$, calculated over the GEN-SIDE-OUT from the wings, to get $\text{GEN-SIDE-IN}_{l,t}$. We then use $\text{GEN-SIDE-IN}_{l,t}$ to perform our second pass of checks (step 3). Finally, we use $\mathcal{G}_l$ and $\mathcal{K}_l$ to update the SOS (step 4).

3.2.2 Dynamic Parallel Available Expressions

An expression e reaches a block (l, t) only if there is no path to the block that kills e, i.e., no valid ordering in which e is killed before the first instruction of the block, as available expressions is a must analysis. In such cases, there is no need to recompute the expression. However, if any path to the block kills e, then there is no guarantee that e is precomputed and we must recompute it

in block (l, t). With reaching definitions, d_k reaches a particular point p if in at least one valid ordering d_k reaches p; in available expressions, e_k only reaches p if in all valid orderings e_k reaches p. This gives some intuition that $\mathcal{K}$ in available expressions behaves like $\mathcal{G}$ in reaching definitions, and likewise $\mathcal{G}$ in available expressions behaves like $\mathcal{K}$ in reaching definitions.

Let $\mathcal{G}_{l,t,i}$ be the set of expressions generated by instruction (l, t, i): $\mathcal{G}_{l,t,i} = \{e_k\}$ if and only if instruction (l, t, i) generates expression e_k, and is empty otherwise. Similarly, let $\mathcal{K}_{l,t,i}$ be the set of expressions killed by instruction (l, t, i). We calculate $\mathcal{G}_{l,t}$ and $\mathcal{K}_{l,t}$ as usual.

Let KILL-SIDE-OUT$_{l,t}$ represent the set of killed expressions a block (l, t) exposes to another block (l', t') anytime it is in the wings of a butterfly with body (l', t'). Because the body of the butterfly can execute anywhere in relation to its wings, we must take the union of the $\mathcal{K}_{l,t,i}$. Let KILL-SIDE-IN$_{l,t}$ represent the set of expressions visible to block (l, t) that are killed by the wings.

$$\text{KILL-SIDE-OUT}_{l,t} = \bigcup_i \mathcal{K}_{l,t,i}$$

$$\text{KILL-SIDE-IN}_{l,t} = \bigcup_{l-1 \leq l' \leq l+1} \bigcup_{t' \neq t} \text{KILL-SIDE-OUT}_{l',t'}$$

In available expressions, GEN-SIDE-IN $=$ GEN-SIDE-OUT $= \varnothing$ for the same reason that KILL-SIDE-IN $=$ KILL-SIDE-OUT $= \varnothing$ in reaching definitions; no block has enough information to know that every path to a particular instruction has generated a particular expression.

The properties we desire for $\mathcal{G}_l$ and $\mathcal{K}_l$ are roughly the opposite of those from reaching definitions.

$$\mathcal{K}_l = \bigcup_t \mathcal{K}_{l,t}$$

$$\mathcal{G}_l = \bigcup_t (\mathcal{G}_{l,t} \cap (\bigcup_{t' \neq t} \mathcal{G}_{(l-1,l),t'} \cup \text{NOT-}\mathcal{K}_{(l-1,l),t'}))$$

$$\text{where } \mathcal{G}_{(l-1,l),t} = (\mathcal{G}_{l-1,t} - \mathcal{K}_{l,t}) \cup \mathcal{G}_{l,t}$$

$$\text{and NOT-}\mathcal{K}_{(l-1,l),t} = \{e_k | e_k \notin \mathcal{K}_{l-1,t} \wedge e_k \notin \mathcal{K}_{l,t}\}$$

The correctness of $\mathcal{K}_l$ and $\mathcal{G}_l$ follows along the lines of the proof of Lemma 1, with the roles of $\mathcal{G}$ and $\mathcal{K}$ reversed.

Updating State

The SOS has the same equation and update rule as described in Section 3.2.1. The LSOS has a slightly different form, reflecting the different roles $\mathcal{G}$ and $\mathcal{K}$ play. In available expressions, an expression only reaches an instruction (l, t, i) if it has been defined along all paths to the instruction. So, if $e_k \in \text{SOS}_l \wedge e_k \notin \mathcal{K}_{l-1,t}$, then $e_k \in \text{LSOS}_{l,t}$ because e_k is calculated along all paths. However, if $e_k \in \mathcal{K}_{l-1,t}$ then at least one path exists where the expression is not defined. If $e_k \notin \text{SOS}_l$, the only way that $e_k \in \text{LSOS}_{l,t}$ is if it is defined by the head ($e_k \in \mathcal{G}_{l-1,t}$) and no other thread t' ever kills e_k in epoch $l-2$; otherwise, because the head can interleave with epoch $l-2$, there is a possible path where e_k is killed before the body executes. This leads to:

$$\text{LSOS}_{l,t} = \left(\mathcal{G}_{l-1,t} - \bigcup_{t' \neq t}\{e_k | e_k \in \mathcal{K}_{l-2,t'}\}\right) \cup (\text{SOS}_l - \mathcal{K}_{l-1,t})$$

$\text{LSOS}_{l,t,k}$ has the same update rule as stated in Section 3.2.1.

Calculating In and Out

Let $\text{IN}_{l,t,i}$ be the set of inputs that reach instruction i in thread t and epoch l. Let $\text{IN}_{l,t,0} = \text{IN}_{l,t}$ represent the set of expressions that could possibly reach the beginning of block (l, t). $\text{IN}_{l,t}$ should be the intersection of the set of valid expressions of all possible paths to instruction $(l, t, 0)$:

$$\text{IN}_{l,t} = \text{LSOS}_{l,t} - \text{KILL-SIDE-IN}_{l,t}$$
$$\text{IN}_{l,t,i} = \text{LSOS}_{l,t,i} - \text{KILL-SIDE-IN}_{l,t}$$

Let $\text{OUT}_{l,t,i}$ be the set of expressions that are still defined after executing instruction i in thread t and epoch l, and $\text{OUT}_{l,t}$ represent the set of expressions still defined after all instructions in the block have executed. Then:

$$\text{OUT}_{l,t,i} = \mathcal{G}_{l,t,i} \bigcup (\text{IN}_{l,t,i} - \mathcal{K}_{l,t,i})$$
$$\text{OUT}_{l,t} = \mathcal{G}_{l,t} \bigcup (\text{IN}_{l,t} - \mathcal{K}_{l,t})$$

Applying the Two-Pass Algorithm

The parameters for the two-pass algorithm are similar to those in Section 3.2.1. In step 1, we again calculate $\mathcal{G}_{l,t}$, $\mathcal{K}_{l,t}$ and $\text{LSOS}_{l,t}$, but now the summary information is $\text{KILL-SIDE-OUT}_{l,t}$. Step 2 uses $\cup$ for the meet function but calculates over all the $\text{KILL-SIDE-OUT}_{l',t'}$ in the wings, to get $\text{KILL-SIDE-IN}_{l,t}$, which is then used to perform our second pass of checks (step 3). Finally, we use $\mathcal{G}_l$ and $\mathcal{K}_l$ to update SOS_l (step 4).

3.3 Implementing Lifeguards With Butterfly Analysis

Now that we have shown how to use butterfly analysis with basic dataflow analysis problems, we extend it to two lifeguards. For each of these lifeguards, we will show that we lose some *precision* (i.e., experience some false positives) but never have any false negatives; compared against any valid ordering, we will catch all errors present in the valid ordering but potentially flag some safe events as errors. Our main contribution will be parallel adaptations of ADDRCHECK and TAINTCHECK that do not need access to inter-thread dependences and are supported on even relaxed memory models, as long as those memory models respect intra-thread dependences and provide cache coherence. For each lifeguard, we also show how to update the metadata using dataflow analysis.

3.3.1 AddrCheck

ADDRCHECK [79], as described in Section 1.1, checks accesses, allocations and deallocations as a program runs to make sure they are safe. In the sequential version, this is straightforward; writing to unallocated memory is an error. In butterfly analysis, one thread can allocate memory before another writes, but if these operations are in adjacent epochs, the operations are potentially concurrent. In ADDRCHECK, a false positive occurs when the program behaves safely but the lifeguard believes it has seen an invalid sequence of malloc, free, and memory access events.

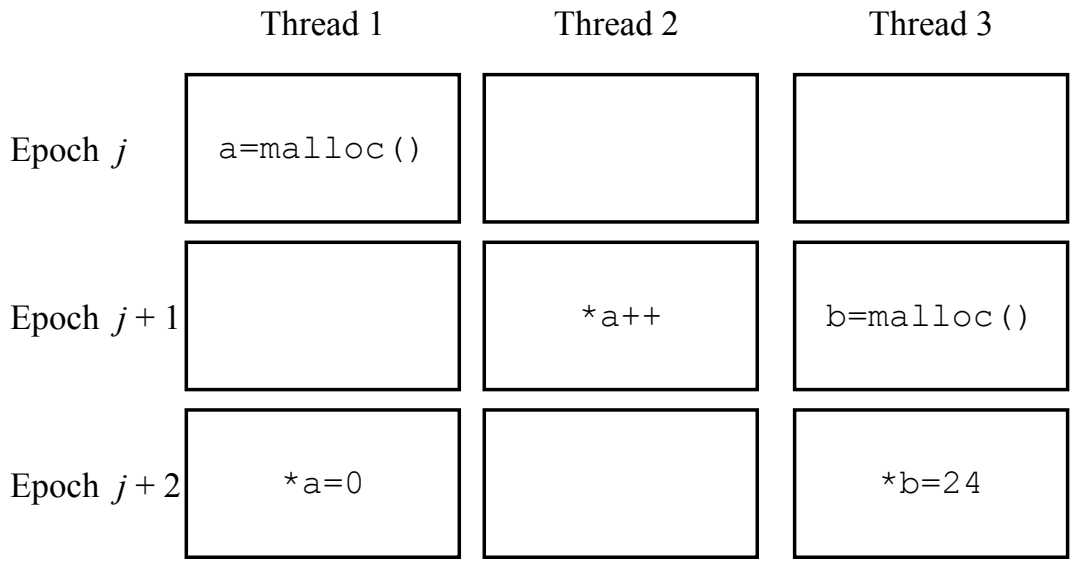


Figure 3.2: ADDRCHECK examples of interleavings between allocations and accesses. There is a potentially concurrent access to `a` by Thread 2 during its allocation by Thread 1, but the allocation of `b` by Thread 3 is *isolated* from other threads.

We describe ADDRCHECK as an adaptation of available expressions, associating allocations with $\mathcal{G}$ and deallocations with $\mathcal{K}$. We chose available expressions because we want to guarantee zero false negatives; for all valid orderings, we want to know whether an access to memory location m is an access to allocated memory, and always detect accesses to unallocated memory regions. Let $\mathcal{G}_{l,t,i} = \{m\}$ if and only if instruction (l, t, i) allocates memory location m and otherwise $\varnothing$. Likewise, $\mathcal{K}_{l,t,i} = \{m\}$ if and only if instruction (l, t, i) deallocates memory location m and otherwise $\varnothing$. $\mathcal{G}_{l,t}$, $\mathcal{K}_{l,t}$, $\mathcal{G}_l$, $\mathcal{K}_l$, SOS_l and $\text{LSOS}_{l,t}$ use the equations and update rules from Section 3.2.2.

Checking Algorithm

Our checking algorithm needs to be more sophisticated than available expressions' use of IN and OUT. A naive calculation would not detect that a location had been freed twice, since set difference does not enforce that $B \subseteq A$ before performing $A - B$. Modifying the checks is straightforward, though.

There are two basic conditions we wish to ensure. First, any time an address is being accessed (either read or write) or deallocated, we wish to know that the address is definitely allocated. Secondly, any time an address is being allocated, we wish to know that the address is definitely deallocated. Examining these two conditions in detail, we find that each has two parts.

In the first case (i.e., when we wish to ensure that an address is definitely *allocated*), it suffices to ensure that the address appears allocated within our thread and that no other thread is concurrently allocating or deallocating this address. Symmetrically, to check that an address is definitely *deallocated*, it suffices if the address appears deallocated within the given thread with no other thread concurrently allocating or deallocating the address. The general implication of these two rules is that whenever the metadata states changes from allocated to deallocated (or vice versa), any concurrent (i.e., in the *wings*) read, write, allocate or deallocate is problematic. This is analogous to a race on the metadata state.

Consider Figure 3.2. When thread 2 accesses `a` in epoch $j+1$, `a` does not appear allocated yet, because its allocation will not be reflected in the SOS for another epoch. However, when thread 3 allocates `b` in epoch $j+1$, it appears deallocated within the thread and no other thread is accessing it. The subsequent access to `b` in epoch $j+2$ is also safe because it is within the same thread, even though the allocation is not yet reflected in the SOS.

More formally, we split the checking into two parts. We first verify that any address we accessed or deallocated appeared to be allocated within our thread, and any address we allocated appeared deallocated in our thread. These checks can be resolved by checking that an access or deallocation (allocation) to memory location x at instruction (l,t,i) is contained (not contained, respectively) in the $\text{LSOS}_{l,t,i}$.

Next, we want to ensure that allocations and deallocations were *isolated* from any other concurrent thread. This occurs during the second pass, using the summaries created in the first pass. For ADDRCHECK, the summary is $s_{l,t} = (\mathcal{G}_{l,t}, \mathcal{K}_{l,t}, \text{ACCESS}_{l,t})$, where $\text{ACCESS}_{l,t}$ contains

all addresses that block (l, t) accessed. Combining the wing summaries yields:

$$S_{l,t} = (\bigcup_{\text{wings}} \mathcal{G}_{l',t'}, \bigcup_{\text{wings}} \mathcal{K}_{l',t'}, \bigcup_{\text{wings}} \textsc{Access}_{l',t'}).$$

To verify isolation, we check that the following set is empty ($s_{l,t}$ is abbreviated as s, and $S_{l,t}$ as S):

$$((s.\mathcal{G}_{l,t} \cup s.\mathcal{K}_{l,t}) \bigcap (S.\mathcal{G}_{l,t} \cup S.\mathcal{K}_{l,t})) \bigcup$$
$$(s.\textsc{Access}_{l,t} \bigcap (S.\mathcal{G}_{l,t} \cup S.\mathcal{K}_{l,t})) \bigcup$$
$$(S.\textsc{Access}_{l,t} \bigcap (s.\mathcal{G}_{l,t} \cup s.\mathcal{K}_{l,t}))$$

and otherwise flag an error.

Theorem 3. *Any error detected by the original* AddrCheck *on a valid execution ordering for a given machine (obeying intra-thread dependences and supporting cache coherence) will also be flagged by our butterfly analysis.*

Proof. Observe that the original AddrCheck detects errors that occur pairwise between operations (i.e., allocations, accesses, and deallocations) on the same address. It is therefore sufficient to restrict our analysis to pairs of instructions involving the same address.

We consider any memory consistency model that respects intra-thread dependences and supports cache coherence. Suppose there is an execution E of the monitored program on a machine supporting that model such that one of the pairwise error conditions is violated for an address x, e.g., there is an access to x after it is deallocated. Let $E|x$ be the subsequence of E consisting of all instructions involving x. By the assumptions of the butterfly analysis, there is a valid ordering O such that $O|x$, the subsequence of O consisting of all instructions involving x, is identically $E|x$. Because butterfly analysis considers O among the many possible valid orderings, and checks for all combinations of pairwise errors for locations, it too will flag an error. □

3.3.2 TaintCheck

TAINTCHECK [84] tracks the propagation of taint through a program's execution; if at least one of the two sources is tainted, then the destination is considered tainted. When extending TAINTCHECK to work using butterfly analysis, we extend this conservative assumption as follows. If some valid ordering O causes an address x to appear tainted at instruction (l, t, i), we conclude that (l, t, i) taints x even if it does not taint x under any other valid ordering. We modify reaching definitions to accommodate TAINTCHECK. In this setting, a false negative refers to concluding that data is untainted when it is actually tainted, whereas a false positive refers to believing data to be tainted when it is actually untainted.

Unfortunately, adapting TAINTCHECK to butterfly analysis is not as simple as adapting ADDRCHECK. TAINTCHECK has an additional method of tracking information called *inheritance*. Consider a simple assignment `a:=b+1`. If we already know that `b` is tainted, then `a` is tainted via propagation, and can be calculated using IN and OUT. If `b` is a shared global variable whose taint status is unknown to the thread executing this instruction, then `a` inherits the same taint status as `b`.

In order to efficiently compute taint status while handling inheritance, we will use a SSA-like scheme that assigns unique tuples (l, t, i) instead of integers. We also define a function `loc()` that given an SSA numbering (l, t, i) returns x, where x is the location being written to by instruction (l, t, i). Our metadata are transfer functions between SSA-numbered variables and their taint status, with $\bot$ as taint and $\top$ as untaint. We will use $x_{l,t,i}$ as an abbreviation for the case where $x = \texttt{loc}(l, t, i)$. The SOS will only contain addresses believed to be tainted. Then:

$$\mathcal{G}_{l,t,i} = \begin{cases} (x_{l,t,i} \leftarrow \bot) & \text{if } (l,t,i) \equiv \texttt{taint}(x) \\ (x_{l,t,i} \leftarrow \top) & \text{if } (l,t,i) \equiv \texttt{untaint}(x) \\ (x_{l,t,i} \leftarrow \{a\}) & \text{if } (l,t,i) \equiv x := \texttt{unop}(a) \\ (x_{l,t,i} \leftarrow \{a,b\}) & \text{if } (l,t,i) \equiv x := \texttt{binop}(a,b) \end{cases}$$

If we know that the last write to a was $\bot$ in a block, we can short-circuit the `unop` and `binop` calculations, concluding $(x_{l,t,i} \leftarrow \bot)$. This resembles propagation in reaching definitions.

Let $S = \{\top, \bot, \{a\}, \{a,b\} | \exists \text{memory locations } a, b\}$. In other words, S represents the set of all possible right-hand values in our mapping. We define the set $\mathcal{K}_{l,t,i} = \{(x_{l,t,j} \leftarrow s) | s \in S, j < i, \text{ and } \texttt{loc}(l,t,j) = \texttt{loc}(l,t,i)\}$. In TAINTCHECK, GEN-SIDE-OUT$_{l,t}$, KILL-SIDE-OUT$_{l,t}$, GEN-SIDE-IN$_{l,t}$, KILL-SIDE-IN$_{l,t}$, $\mathcal{G}_{l,t}$ and $\mathcal{K}_{l,t}$ all function identically as defined in Section 3.2.1 for reaching definitions.

Checking Algorithm

The main difference between TAINTCHECK and reaching definitions is the checking algorithm. Given an instruction-level transfer function $(x \leftarrow s)$, a location $y_{l,t,i}$ is a **parent** of x if $\exists z_{l',t',i'} \in s$ such that $\texttt{loc}(l,t,i) = \texttt{loc}(l',t',i')$. We will say instruction (l,t,i) occurs **strictly before** instruction (l',t',i'), if one of three conditions hold. First, if $l \leq l' - 2$. The other two cases only apply if the memory model is sequentially consistent. If $l = l', t = t'$ and $i < i'$, or if $t = t'$ and $l < l'$, then (l,t,i) occurs strictly before (l',t',i'). We denote this as $(l,t,i) < (l',t',i')$.

Algorithm 1 presents a function `resolve` that takes a particular transfer function of the form $(x_{l,t,i} \leftarrow s)$ and a set of transfer functions T. Intuitively, `resolve` resembles depth first search on a graph. Parents are replaced with their predecessors recursively until we run out of transfer functions or reach a termination condition, whichever happens first.

We consider two variants of `resolve`: one for sequential consistency and one for more

Algorithm 1 TAINTCHECK `resolve` Algorithm

Input: $(x_{l,t,i} \leftarrow s), T$
Extracts the list of parents of $x_{l,t,i}$: $\{y_0, y_1, \ldots, y_k\}$ using the `loc` function
for all y_j a parent of $x_{l,t,i}$ **do**
  Search for rules of the form $(y_j \leftarrow s') \in T$
  Replaces y_j with all the parents of y_j in s', subject to a termination condition
  if any parent of y_j is $\bot$ **then**
    Terminate with the rule $(x_{l,t,i} \leftarrow \bot)$.
  else if any parent of y_j is $\top$ **then**
    Drop it from the list of parents, and continue
Postcondition: Either $(x_{l,t,i} \leftarrow s)$ converges to $(x_{l,t,i} \leftarrow \bot)$, or s becomes empty. If s is empty, conclude $(x_{l,t,i} \leftarrow \top)$.

relaxed models. Under sequential consistency, it makes sense to enforce sequential execution within each thread. To do so, we associate t counters of the form (l, t, i) with each parent. We only allow a replacement for a parent y with $z_{l',t',i'}$ if (l', t', i') occurs strictly before the counter at position t' associated with y. If so, we update the counter to reflect the new (l', t', i') value, and continue. If y is replaced with multiple predecessors, we follow the same procedure for each predecessor. This forces the ordering of instructions implied by the checking algorithm to always be in sequential order when restricted to a particular thread t.

If we do not have sequential consistency, we must relax the `resolve` algorithm's termination condition while still preventing false negatives. The issue is that a sequence of assignments causing x to inherit from y can exist, but depend on an assignment occurring in the wings; in Figure 2.1(b), executing (2) before (i) before (1) is legal on some relaxed memory models [4]. By disallowing a parent to eventually be replaced by itself we prevent infinite loops, because there are only a bounded number of potential parents; it will not guarantee that the ordering that taints memory location x is actually valid. This resembles iteration as performed in dataflow analysis to resolve loops.

Theorem 4. *If* `resolve` *returns* $(x_{l,t,i} \leftarrow \top)$*, then there is no valid ordering of the first* $l + 1$ *epochs such that* x *is* $\bot$ *at instruction* (l, t, i)*. Therefore, any error detected by the original* TAINTCHECK *on a valid execution ordering for a given machine (obeying intra-thread depen-*

dences and supporting cache coherence) will also be flagged by our butterfly analysis.

Proof. We begin by assuming sequential consistency and the associated termination condition, and then show how the proof holds on relaxed memory models with their associated termination condition.

Sequential Consistency: Suppose there was a valid ordering of the first $l+1$ epochs such that $x \leftarrow \perp$ at instruction (l, t, i). That implies there exists a sequence of $k+1$ transfer functions $\hat{f}$ such that the associated instructions in order would taint x.

Restricting $\hat{f}$ to functions from a particular thread t will produce a subsequence, potentially empty, that is still ordered, so we will not have violated the sequential consistency assumption. This shows $\hat{f}$ is a legitimate sequence of parents to follow, so we would conclude $(x \leftarrow \perp)$.

Relaxed Memory Models: Once more, if there is a valid ordering O of the first $l+1$ epochs such that $x \leftarrow \perp$ at instruction (l, t, i), there exists a sequence of $k+1$ transfer functions $\hat{f}$ such that the associated instructions in order would taint x.

By virtue of the fact that O is a valid ordering, restricting the $\hat{f}$ to functions from a particular thread t will show that each transfer function is associated with a unique instruction from thread t (no duplicates or repeats), even if the subsequence violates program order. This shows that, under the relaxed termination condition, $\hat{f}$ is a legitimate sequence of parents to follow, so we would conclude $(x \leftarrow \perp)$. □

Reducing False Positives

Suppose we are trying to resolve $(a_{2,2,1} \leftarrow b)$, and in the wings of the butterfly are transfer functions $(b_{1,3,1} \leftarrow r)$ and $(r_{3,1,1} \leftarrow \perp)$. Under either of the proposed termination conditions, it is still possible to conclude instruction $(a \leftarrow \perp)$. However, for $(a \leftarrow \perp)$ to occur, then instruction $(3, 1, 1)$ must execute before instruction $(1, 3, 1)$, a direct violation of our butterfly assumptions (epoch 1 always executes before epoch 3).

To reduce the number of false positives, the resolution of checks takes place in two phases. In the first phase, a block (l, t) can use any transfer function from epochs $l-1$ or l in `resolve`. In the second phase, only transfer functions from $l+1$ and l can be used by `resolve`. If in the first phase, we conclude $\perp$ for a location x, that location remains $\perp$ throughout the second phase. The correctness of this optimization is supported by the following lemma.

Lemma 5. *If there exists a valid ordering O among 3 consecutive epochs such that x is tainted then*

(1) x is tainted via an interleaving of the first 2 epochs;

(2) x is tainted via an interleaving of the last 2 epochs; or

(3) there exist a predecessor y of x such that y is tainted in the first two epochs and there exists a path from x to y in the last two epochs using only transfer functions from the last two epochs.

Proof. A valid ordering of 3 epochs that taints x might taint x when restricted only to (1) the first two epochs, or (2) restricted only to the last 2 epochs. The final case is when all three epochs are used to taint x. In this case, there can be no interleaving between the first and third epochs, because all instructions in the first epoch must commit before any instructions in the third epoch begin. As the first epoch cannot taint x directly and neither can the third epoch (this would put us into cases 1 or 2) then it must be the case that some predecessor of x is tainted by an interleaving of the first epoch with some of the second epoch, and then that there is a valid interleaving between the remaining instructions in the second epoch with the third epoch such that x inherits from y. This is conservatively handled by (3). □

SOS and LSOS

Instead of transfer functions the SOS and LSOS will track locations believed to be tainted. Once again, TAINTCHECK is slightly more complicated than reaching definitions. We can conclude that a variable is tainted in epoch l based on an interleaving with epoch $l+1$. Consider Figure 3.3. If we do not commit `a` to the SOS before beginning a butterfly for block $(j+2, 2)$ we may

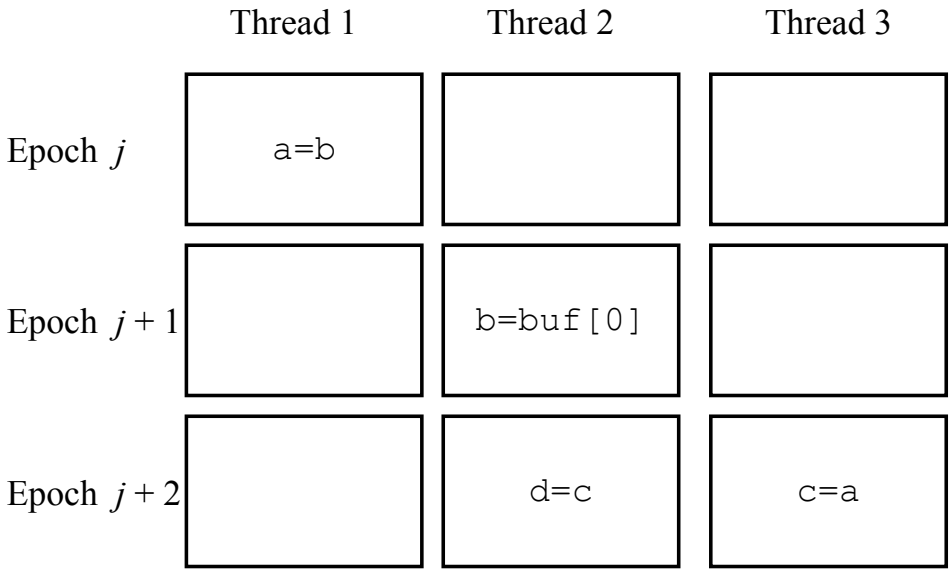


Figure 3.3: Updating the SOS is nontrivial for TAINTCHECK. By the end of epoch $j+1$, a has been tainted, but the SOS may need to be updated before blocks in epoch $j+2$ begin butterfly analysis.

conclude that d is untainted, even though there is a path where d is tainted. If we consider a to be tainted before beginning epoch $j+2$, though, there is no guarantee the instruction that taints a has actually already executed. However, considering an address to be tainted early is merely imprecise, while considering an address to be tainted too late violates our guarantees.

Define the function LASTCHECK(x, l, t) to be the last metadata value returned by resolve for location x while checking block (l, t). This is not the same as recomputing resolve for x at the end of the block. Rather, it is similar to computing the difference between the LSOS at the end of the block and the LSOS at the beginning. If x was assigned to in block (l, t), then LASTCHECK(x, l, t) will return $\top$ or $\bot$; otherwise, it returns $\varnothing$. We can extend this definition to LASTCHECK$(x, (l-1, l), t)$ which will tell us whether the LASTCHECK spanning two epochs $l-1$ and l tainted, untainted, or merely propagated x. In our SOS, we will track only those variables x we believe are tainted, and will use LASTCHECK to do so. We define

$$\mathcal{G}_l = \bigcup_t \{x | \text{LASTCHECK}(x, l, t) = \bot\}$$
$$\mathcal{K}_l = \bigcup_t \{x | \text{LASTCHECK}(x, l, t) = \top \wedge$$

$$(\forall t' \neq t, \textsc{LastCheck}(x, (l-1, l), t) = \top \vee$$
$$\textsc{LastCheck}(x, (l-1, l), t) = \varnothing)\}$$

This is an almost identical formulation to reaching definitions; the difference is that we use LASTCHECK to change our metadata format from transfer functions to tainted addresses. SOS_l and $\text{LSOS}_{l,t}$ use the update rules for reaching definitions. We claim the following conditions hold for the SOS:

Condition 3.3.1. *If there exists a valid ordering O_s of the first $l-2$ epochs such that x is tainted in O_s then $x \in \text{SOS}_{l-2}$.*

Condition 3.3.2. *If $x \in \text{SOS}_{l-2}$, then there exists at least one thread t such that t assigns to x and believes a valid ordering of the first $l-2$ epochs exists that taints x.*

The first condition is identical to reaching definitions. The second condition addresses imprecision due to our reliance on the checking algorithm. Analogous conditions hold for the LSOS.

3.4 Evaluation of A Butterfly Analysis Prototype

To demonstrate the practicality of butterfly analysis and to identify areas where further optimizations may be helpful, we now present our experimental evaluation of our initial implementation of butterfly analysis within a parallel monitoring framework.

3.4.1 Experimental Setup

While the generality of butterfly analysis makes it applicable to a wide variety of dynamic analysis frameworks (including software-only frameworks based upon binary instrumentation [20, 71, 82]), we chose to build upon the *Log-Based Architectures* (LBA) [23] framework in our experiments due to its relatively low run-time overheads. With LBA, each application thread is monitored by a dedicated lifeguard thread running concurrently on a separate processor on the same CMP. The LBA hardware captures a dynamic instruction log per application thread and passes

Table 3.1: Simulator and Benchmark Parameters

Simulation Parameters	
Cores	{4,8,16} cores
Pipeline	1 GHz, in-order scalar, 65nm
Line size	64B
L1-I	64KB, 4-way set-assoc, 1 cycle latency
L1-D	64KB, 4-way set-assoc, 2 cycle latency
L2	{2,4,8}MB, 8-way set-assoc, 4 banks, 6 cycle latency
Memory	512MB, 90 cycle latency
Log buffer	8KB

Application	Suite	Input Data Set
BARNES	Splash-2	16384 bodies
FFT	Splash-2	$m = 20$ (2^{20} sized matrix)
FMM	Splash-2	32768 bodies
OCEAN	Splash-2	Grid size: 258×258
BLACKSCHOLES	Parsec 2.0	16384 options (`simmedium`)
LU	Splash-2	Matrix size: 1024×1024, $b = 64$

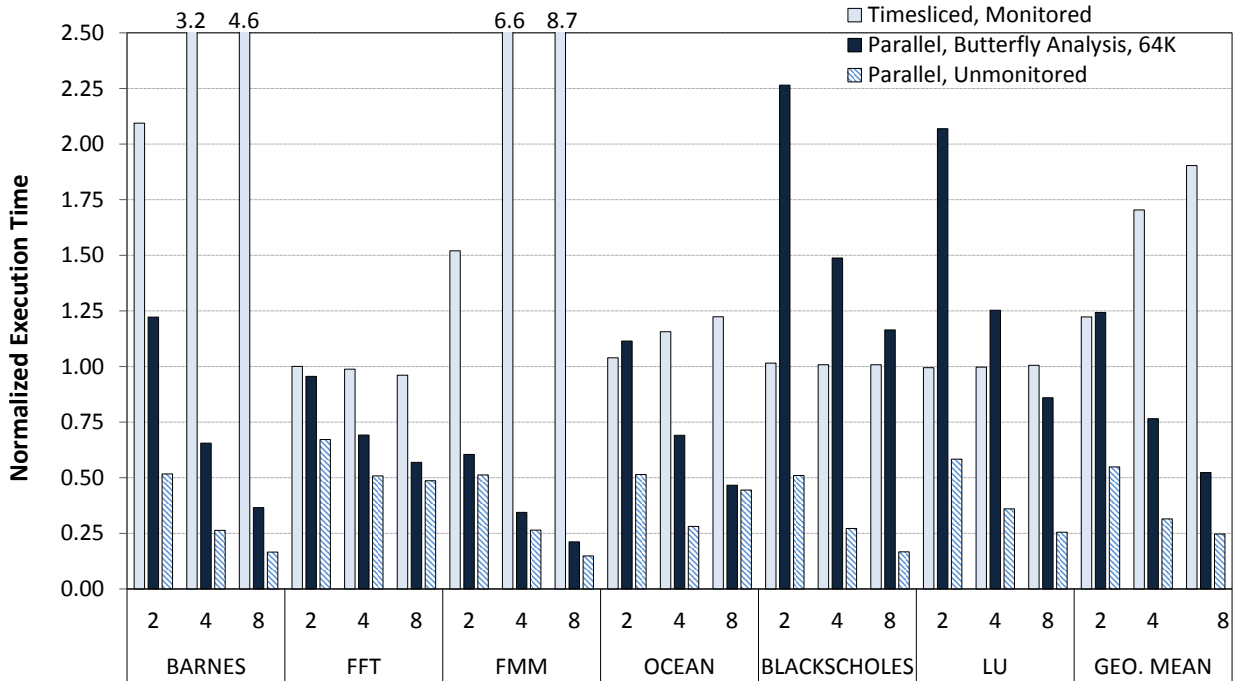


Figure 3.4: Relative performance, normalized to sequential, unmonitored execution time. X-axis: number of application threads.

it (via the L2 cache) to the corresponding lifeguard thread. When lifeguard processing is slower than the monitored application (as is the case in our experiments), the monitored application stalls whenever the log buffer is full; hence our performance results show lifeguard processing time, which is equivalent to application execution time including such log buffer stalls.

Because the LBA [23] hardware support is not available on existing machines, we simulated the LBA hardware functionality (including log capture and event dispatch) on a shared-memory CMP system using the Simics [108] full-system simulator. Although the LBA hardware is simulated, the full software stack for butterfly analysis is executed faithfully in our experiments. Table 3.1 shows the parameters for our machine model as well as the benchmarks that we monitored (taken from Splash-2 [113] and Parsec 2.0 [17]).

For our lifeguard, we implemented a parallel, heap-only version of ADDRCHECK–based upon [79]—using butterfly analysis as described in Section 3.3.1. The LBA logging mechanism makes it easy to generate and communicate heartbeats: we simply insert heartbeat markers into the log after h instructions have occurred per thread,[3] where h equals 8K or 64K instructions in our experiments. We use the *metadata-TLB* and *idempotent filtering*[4] accelerators from LBA [23], and we filter out stack accesses.

3.4.2 Experimental Results

We now evaluate the performance and accuracy of our butterfly-analysis-based ADDRCHECK lifeguard compared with the current state-of-the-art.

Performance Analysis. Because lifeguards involve additional processing and no direct performance benefit for the monitored application, the performance question is how much they slow

[3]In practice, we issue heartbeats after hn instructions are executed by the application, where n is the number of application threads, without enforcing uniformity of execution across threads. In the worst case, one thread will execute hn instructions while the rest will execute 0. We maintain the invariant that at least h cycles have passed on each core. Butterfly analysis does not require balanced workloads within an epoch.

[4]For idempotent filtering, we flushed the filters at the end of each epoch so that events are only filtered within (and never across) epochs.

down performance relative to unmonitored execution. Figure 3.4 shows the performance impact of butterfly analysis, where the y-axis is execution time normalized to the given application running sequentially on a single thread without monitoring (hence shorter bars are faster). We show performance with 2, 4, and 8 application threads,[5] as labeled on the x-axis. Within each set of bars, we show three cases: (i) *"Timesliced Monitoring"*, the current state of the art where all application threads are interleaved on one core, and are monitored by a sequential lifeguard (running on a separate core); (ii) *"Parallel, Monitoring,"* which is with our butterfly analysis; and (iii) *"Parallel, No Monitoring,"* which is the application running in parallel without any monitoring (as a point of comparison).

As we observe in Figure 3.4, when monitoring only *two* application threads, the performance of butterfly analysis relative to the state-of-the-art timesliced approach is mixed: it is significantly better for BARNES and FMM, comparable for FFT and OCEAN, and significantly worse for BLACKSCHOLES and LU. A key advantage of butterfly analysis relative to timesliced analysis, however, is that the analysis itself can enjoy parallel speedup with additional threads. Hence as the scale increases to *eight* application (and lifeguard) threads, butterfly analysis outperforms timesliced analysis in five of six cases, and in four of those cases by a wide margin. In the one case where timesliced outperforms butterfly analysis with eight threads (i.e., BLACKSCHOLES), one can observe in Figure 3.4 that butterfly analysis is speeding up well with additional threads, but it has not quite reached the crossover point with eight threads.

While our butterfly approach offers the advantage of exploiting parallel threads to accelerate lifeguard analysis, the implementation of our current prototype has the disadvantage that it performs more work per monitored instruction than the timesliced approach. For example, in the first pass, our current implementation executes roughly 7-10 instructions for each monitored load and store instruction simply to record it for the second pass, above and beyond performing the same first-pass checks as traditional ADDRCHECK. We believe that this overhead is not fun-

[5]Recall that LBA uses a total of $2k$ cores to run an application with k threads, since k additional cores are used to run the lifeguard threads.

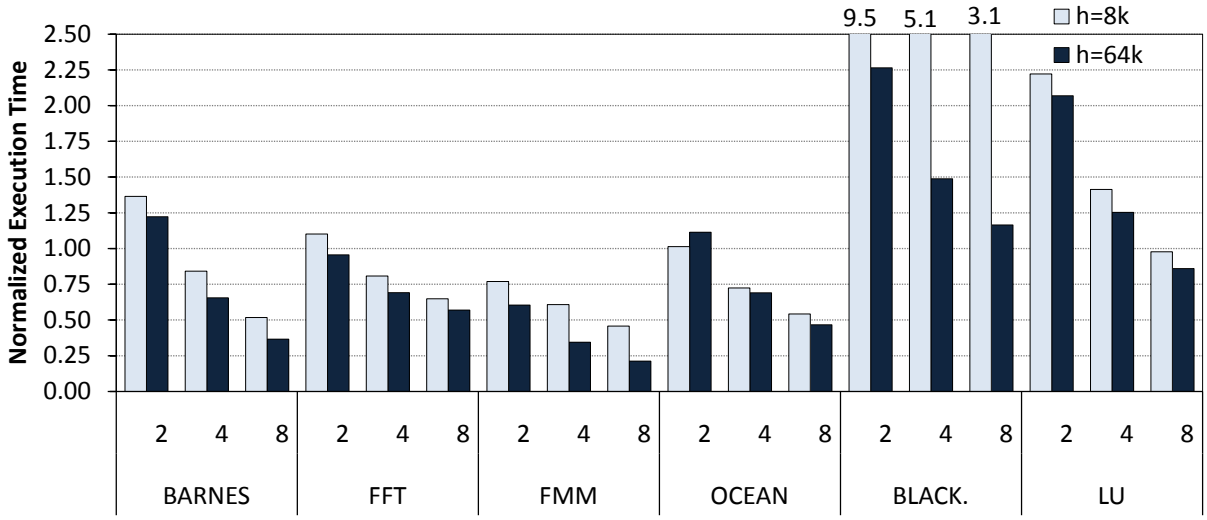


Figure 3.5: Performance sensitivity analysis with respect to epoch size. Results shown for h=8K and 64K.

damental to butterfly analysis, and that it could be significantly reduced by caching parts of our first-pass analysis and reusing it when the same monitored code is revisited. (We plan to explore this possibility in future work.) Despite the inefficiencies of our initial prototype, we observe in Figure 3.4 that it offers compelling performance advantages relative to the state-of-the-art due to its ability to exploit parallelism.

Sensitivity of Performance and Accuracy To Epoch Size. A key parameter in butterfly analysis is the *epoch size*, which dictates the granularity of our concurrency analysis. We now explore how this parameter affects lifeguard performance and accuracy.

Figure 3.5 shows the impact of epoch size on performance. We show two epoch sizes (h): 8K and 64K instructions. (Note that the epoch size in Figure 3.4 was h=64K.) As we see in Figure 3.5, in nearly all cases (i.e., everything except the two and four thread cases for OCEAN), the performance improves with a larger epoch size. Intuitively, this makes sense because the fixed costs of analysis per epoch—including barrier stalls after each pass—are amortized over a larger number of instructions. To understand what happened in OCEAN, let us first consider the

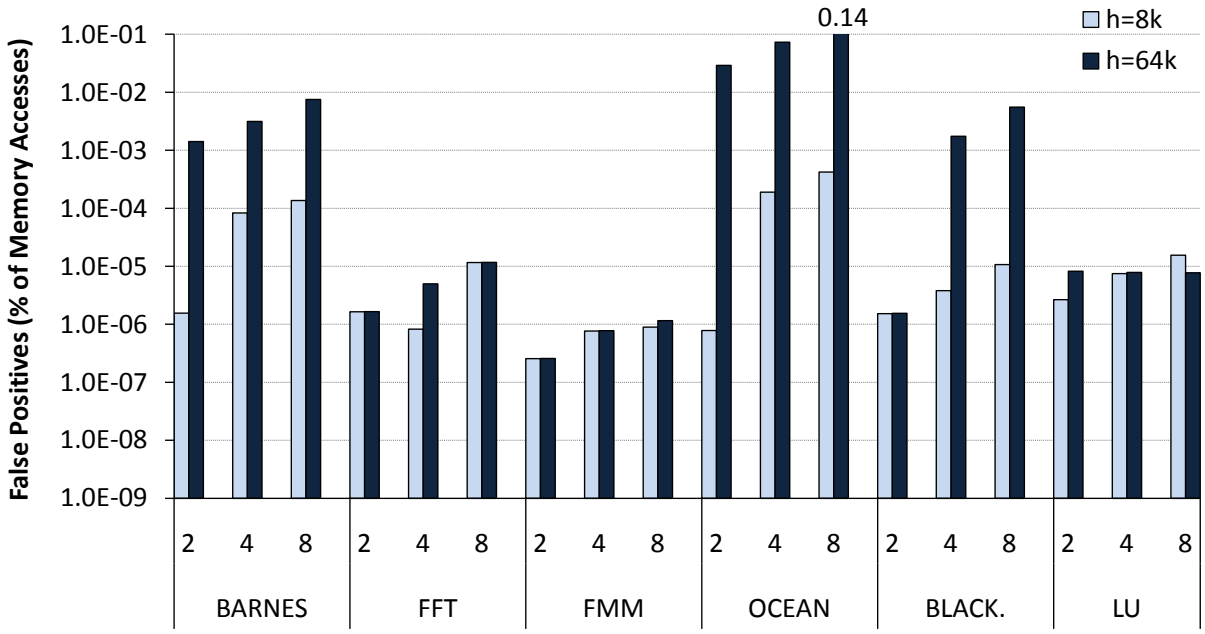


Figure 3.6: Precision sensitivity to epoch size. Results shown for h=8K and 64K. Y-axis: false positives as percentage of memory accesses, shown on a logscale.

impact of epoch size on *accuracy*.

While the advantage of a larger epoch size is better performance, Figure 3.6 shows that the disadvantage is an increase in the false positive rate of the analysis.[6] (Recall that false negatives are impossible with butterfly analysis.) In a number of cases (e.g., FFT, FMM, and LU), the false positive rate did not increase significantly when the epoch size increased from 8K to 64K instructions, but in other cases it did increase by orders of magnitude. In fact, the increase in the false positive rate for OCEAN helps explain why its performance degraded with a larger epoch size: false positives are expensive to process in ADDRCHECK, and in OCEAN they increased enough to offset the savings in amortized overhead. Aside from OCEAN, the false positive rates remain below 0.01% of memory accesses even with the larger epoch size. With the smaller epoch size, all programs have false positive rates well below 0.001% of memory accesses. Overall, we observe that the epoch size is a knob that can be tuned to trade off performance versus accuracy

[6]Recall that for ADDRCHECK, a false positive refers to the lifeguard mistaking a safe event (e.g., an access to allocated memory) for an unsafe event (e.g., an access to unallocated memory).

(subject to a minimum size, as discussed in Section 2.2.1), and that there are reasonable epoch sizes that offer both high performance and high accuracy.

3.5 Chapter Summary

In this chapter, we have presented a new approach to performing dynamic monitoring of parallel programs that requires little or no hardware support: all that we require is a simple heartbeat mechanism, which can be implemented entirely in software. Inspired by dataflow analysis, we have demonstrated how our new *butterfly analysis* approach can be used to implement an interesting lifeguard that outperforms the current state-of-the-art approach (i.e., timeslicing) while achieving reasonably low false-positive rates. The key tuning knob in our framework is the *epoch size*, which can be adjusted to trade off performance versus accuracy.

Chapter 4

Chrysalis Analysis: Incorporating Synchronization Arcs in Dataflow-Analysis-Based Parallel Monitoring

While the dataflow-based approach of Butterfly Analysis offers many advantages (including a theoretically sound framework with no missed errors), one of the limitations of the original framework was that it ignored explicit software synchronization. Because Butterfly Analysis considers all possible instruction interleavings within each window of uncertainty, lifeguards can conservatively report an error based on a hypothetical ordering that can never arise due to synchronization operations. In Figure 4.1, for example, because the ordering `untaint(p)`, `taint(p)`, `*p` cannot be ruled out under Butterfly Analysis, a lifeguard would report the dereference of a possibly tainted pointer. Such *false positives* represent an additional burden on the application developer as error reports must be analyzed, true errors may be missed if they are lost in a large number of false positive messages, and developers may abandon the tool if the work required to process the false reports exceeds the tool's benefits.

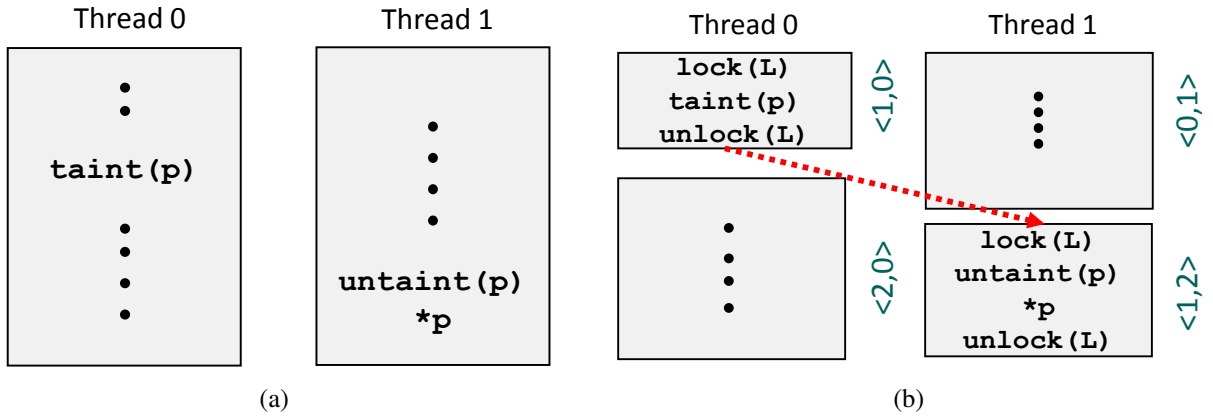


Figure 4.1: (a) Butterfly Analysis ignores synchronization arcs (such as from locks), and hence views the `taint(p)` and `*p` as racing if they are close in time, even if the source code resembles (b). (b) Chrysalis Analysis eliminates such false positives by dynamically capturing explicit synchronization arcs. (Note that `[un]taint(p)` indicates an application operation that would cause the lifeguard to [un]set the "tainted" metadata value for `p`.)

Chrysalis Analysis: Adding Happens-Before Arcs to Butterfly Analysis

In this chapter, we propose and evaluate *"Chrysalis Analysis,"* which is an extension of Butterfly Analysis that takes into account the dynamic happens-before constraints resulting from explicit software synchronization, thereby reducing the number of erroneous false positives, as illustrated in Figure 4.1(b). Integrating happens-before relationships into the Butterfly Analysis framework while retaining the elegance and efficiency of the original framework was a major challenge, due to the irregularities that this introduced, as will be described in detail in Section 4.1. This required generalizing the dataflow analysis mechanisms in the original framework (which were based on simple sliding windows spanning all the threads) to handle all the complexities introduced by partial orderings induced by happens-before arcs between pairs of threads.

Table 4.1 compares Chrysalis Analysis with Butterfly Analysis and ParaLog [110]. Compared to ParaLog, Chrysalis Analysis does not require special hardware support or a mechanism for tracking inter-thread memory dependences, and it can also handle weak memory consistency models. Compared to Butterfly Analysis, Chrysalis Analysis offers improved precision by not reporting errors that are precluded by software synchronization. In fact, when monitoring data-race-free parallel programs, the precision of Chrysalis Analysis should be comparable to ParaLog

Table 4.1: Comparison of Parallel Program Monitoring Solutions

	Implementation requirements		Analysis supports	Monitoring precision	
	Hardware support	Inter-thread dependence tracking	Weak models of consistency	General programs	Data-race-free programs
ParaLog [110]	Yes	Yes	TSO only	high	high
Chrysalis Analysis	*No*	*No*	*Yes*	*good*	*high*
Butterfly Analysis [45]	No	No	Yes	fair	fair

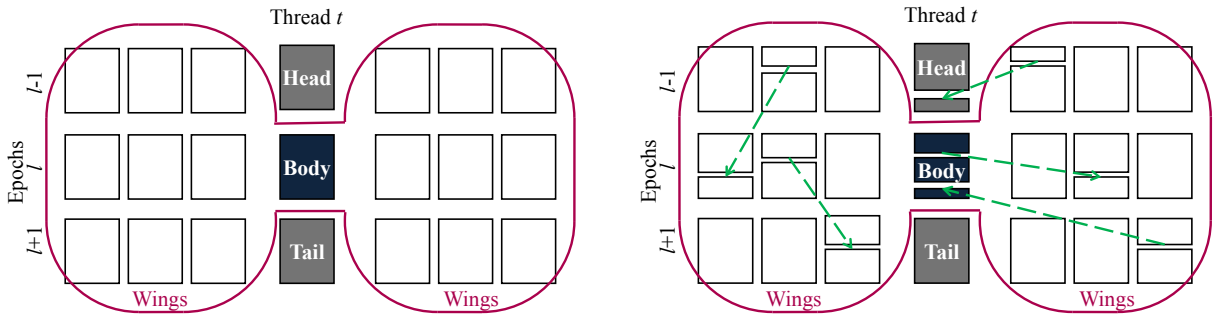
(because all valid orderings are accurately captured via happens-before arcs).[1] There is a trade-off, though, in obtaining this improved precision, as Chrysalis Analysis is somewhat slower than Butterfly Analysis.

Contributions

This chapter makes the following contributions:

- We propose *Chrysalis Analysis*, which builds upon Butterfly Analysis to model the *happens-before* arcs from explicit synchronization, thereby increasing precision. While Butterfly Analysis supported only a very simple and regular concurrency structure of sliding windows across all threads, Chrysalis Analysis supports an arbitrarily irregular and asymmetric acyclic structure within such windows (making the analysis problem considerably more challenging).
- We present (sound) formalizations in the Chrysalis Analysis framework for reaching definitions, available expressions, and two well-studied lifeguards.
- In contrast to work presented in the prior chapter, we implemented a far more challenging lifeguard (TAINTCHECK, requiring not only dataflow analysis but also *inheritance* analysis whereby a single instruction is itself a transfer function) in both the Butterfly and Chrysalis Analysis frameworks to evaluate their precision.

[1] We assume explicit synchronization such as locks and barriers (tracked by Chrysalis Analysis) are used to prevent races.



(a) Butterfly Analysis (b) Chrysalis Analysis

Figure 4.2: (a) Butterfly Analysis divides thread execution into *epochs*. A *block* is a thread-epoch pair. (b) Chrysalis Analysis incorporates high-level synchronization events by dividing blocks into *subblocks* based on the happens-before arcs (shown as dashed arrows) resulting from such events.

- Our experimental results demonstrate a factor of 17.9x reduction in the number of false positives, while slowing down the lifeguard by an average of 1.9x.

4.1 Overview of Chrysalis Analysis

In this section, we introduce Chrysalis Analysis. We begin by motivating the utility of our analysis using simple examples, as well as showcasing the challenges we faced in generalizing Butterfly Analysis. Then, we introduce the new primitives that enable Chrysalis Analysis. Finally, we illustrate some of the major challenges in generalizing Butterfly Analysis to produce Chrysalis Analysis, namely maintaining global state and updating local state.

4.1.1 Adding Happens-Before Arcs: A Case Study

We begin with a few examples that illustrate the utility of our new primitives as well as the challenges that lie ahead.

Consider Figure 4.3(a). This shows a single thread's execution, where in one block it issues (an instruction that serves to) `untaint(p)` and in the next `*p`. Any single-threaded analysis will conclude that at the point where `*p` is dereferenced, `p` is untainted because only one thread was executing and that thread untainted `p` prior to dereferencing it as a pointer.

Now consider Figure 4.3(b). Here, we see Thread 1 issuing `untaint(p); unlock(L)`. After Thread 1 unlocks `L`, Thread 0 is the next thread to acquire `L` (`lock(L)`) and then issues `*p`. In this case, the happens-before relationship is not due to intra-thread data dependences but rather the synchronization on lock `L`. However, from the perspective of a lifeguard monitoring this program, Thread 1 issued `untaint(p)` before Thread 0 issued `*p`; since no instructions in Thread 0 prior to `*p` conflict, this analysis should be identical to case (a). This is true for Chrysalis Analysis, but not Butterfly Analysis.

Finally, consider Figure 4.3(c), which contains a data race. Thread 1 is still untainting `p` and releasing lock `L` immediately before Thread 0 dereferences `p`. However, we also see that block b_1 in Thread 0, prior to issuing `lock(L)` in block b_2, issued `taint(p)` without holding lock `L`. To know whether the dereference of `p` is safe in Thread 0, the analysis needs to know whether `p` was tainted at the program point immediately prior. It knows that two things happened before the dereference: `p` was tainted by Thread 0, and `p` was untainted by Thread 1. However, *the ordering between these two operations remains unknown!* Figure 4.3(c) illustrates that extra happens-before information is not a panacea for all causes of imprecision. In this case, the lifeguard must behave conservatively, since the analysis cannot determine whether the `taint(p)` occurs before or after the `untaint(p)`.

While incorporating happens-before arcs can significantly improve precision, it spoils the key ingredient underlying Butterfly Analysis, namely, that the simple "butterfly" of Figure 4.2(a) captured all the ordering information known to the lifeguard. Each of the steps in Butterfly Analysis' two-pass analysis exploits this regular structure in a fundamental way; and with happens-before arcs, this regularity no longer exists (see Figure 4.2(b)). Including additional partial ordering information, while making the analysis more precise, also makes the analysis much more difficult!

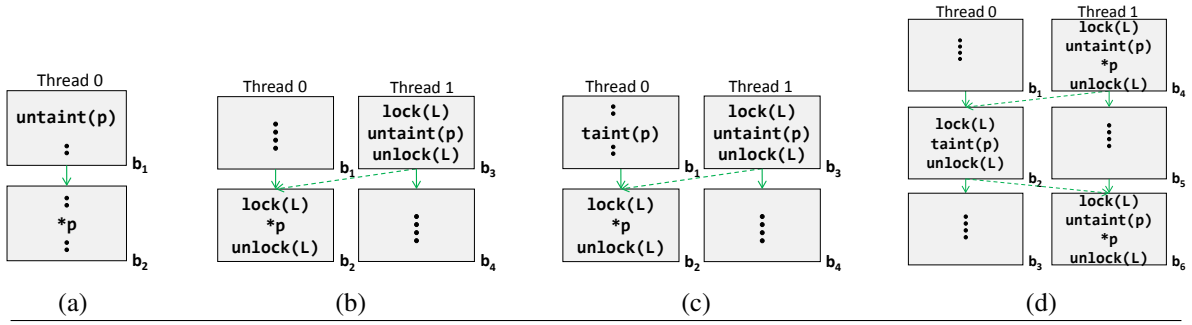


Figure 4.3: TAINTCHECK examples for dereferencing a pointer `p`. (a) In single-threaded execution, respect of intra-thread dependences implies `p` is untainted. (b) The synchronization between Thread 1 and Thread 0 means block b_3 executes before block b_2. Since b_1 does not assign to `p`, `p` is untainted. This is a win for Chrysalis Analysis. (c) Similar to (b), but now Thread 0 issues `taint(p)` in b_1 concurrently with b_3. Conservative analysis means `p` must be treated as tainted. (d) Another win for Chrysalis Analysis. Each dereference of `p` in b_4 and b_6 is guaranteed to only see `p` as untainted.

4.1.2 Maximal Subblocks

Consider again Figures 4.3(b) and (c). In both cases, what allowed us to analyze these figures was the fact that the traces were divided at each `lock` or `unlock` call. This allowed us to reason that an entire section of the trace *happened before* another section. This motivates the following definition, based on the synchronization events currently tracked by our analysis. An execution trace for a thread is partitioned into *maximal subblocks* (*subblocks* for short) by breaking the trace (i) after the last instruction of an epoch, (ii) after each `send`, `unlock`, or `barrier-wait` call, and (iii) whenever the following instruction is a `receive` or `lock` call.[2] This subdivision enables us to reason that entire maximal subblocks must have occurred before, after, or concurrent with other maximal subblocks. Note that the maximal subblocks are precisely the irregularity mentioned earlier. The number and size of maximal subblocks per thread, and per epoch, is based on the frequency and types of synchronization used by the application. This is illustrated in Figures 4.1(b) and 4.2(b).

Despite the introduction of subblocks, for ease of comparison, we will adopt the notation of Butterfly Analysis by referring to instructions based on their offset within blocks. Namely,

[2]In the special case where the first instruction of an epoch is a `receive` or `lock` call, the first maximal subblock is considered empty and the second maximal subblock begins with the `receive` or `lock`, respectively.

(l, t, i) refers to the ith instruction in epoch l of thread t's trace.

Next, we discuss how to determine, given two subblocks, b and b', if b executed before, after, or concurrent with b'.

4.1.3 Testing Ordering Among Subblocks

To test ordering between maximal subblocks, each maximal subblock b has an associated vector clock $v(b)$. Vector clocks have been used in many other works [11, 19, 24, 37, 102]; they are a natural distributed clock primitive. We are using them here to label individual subblocks based on synchronization events and epoch boundaries.[3] The addition of vector clocks and subblocks transforms the Butterfly Analysis diagram, shown in Figure 4.1(a), into Figure 4.1(b).

We modify the standard vector clock algorithm slightly. Let n be the number of threads. If $v(b)[i]$ is the ith position in vector clock $v(b)$, then for $0 \leq i < n$ we initially set $v(b)[j] = 1$ if $j = i$ and $v(b)[j] = 0$ otherwise. For example, the second thread in a 3-thread system would begin with vector clock $< 0, 1, 0 >$. Consider a `send` (equivalently, `unlock L`) from thread j to thread k. If v_j is thread j's current vector clock and v_k is thread k's current vector clock, then thread j will first bind v_j to $v_{\texttt{send}}$, which k will later receive. Then, $v_j[j]$ is incremented, and a new maximal subblock begins in thread j. When thread k processes the associated `receive` (equivalently, `lock L`), it will set $v_k[i] = \max\{v_{\texttt{send}}[i], v_k[i]\}$ for $0 \leq i < n$ and then increment $v_k[k]$. The `receive` instruction in thread k begins a new maximal subblock.

On a barrier wait (assuming all n threads participate) let

$$\forall i, 0 \leq i < n, v_{bar}[i] = v_i[i].$$

[3]We make the typical assumption that, even on relaxed memory consistency models, synchronization events such as `lock`, `unlock` and `barrier-wait` always carry an associated memory fence. Such a fence implies, for example, that all the effects of instructions before an `unlock` complete before the lock is released [18].

Then, thread j updates its vector clock to be:

$$\forall i \neq j, v_j[i] = v_{bar}[i], \quad v_j[j]{+}{+}$$

This is easily extended to work for a barrier of $n' < n$ threads: Update $v_j[i]$ only if both threads i and j were among the n' threads participating in the barrier.

It makes sense to include ordering information that Butterfly Analysis provides. After the second epoch, we can update vector clocks at epoch boundaries, since the ordering is always known when instructions are separated by at least an epoch. Epoch l treats a snapshot of the vector clocks available at the end of epoch $l-2$ as a barrier (call them v^*_{bar}). While v^*_{bar} is calculated the same as v_{bar}, the update rule differs slightly:

$$\forall i \neq j, v_j[i] = \max\{v^*_{bar}[i], v_j[i]\}, \quad v_j[j]{+}{+}$$

We can compare two vector clocks, v and v', by comparing their components. Vector clock v *happens before* vector clock v' if $\forall i, v[i] \leq v'[i] \wedge \exists j$ s.t. $v[j] < v'[j]$. We indicate this relationship using $v < v'$ or equivalently, $v' > v$. If $v \not< v'$ and $v' \not< v$, then v and v' label *concurrent* maximal subblocks; we will denote this as $v \sim v'$.

We will use this terminology loosely. For instance, if (l, t, i) is an instruction in maximal subblock b, then we might talk about $v(l, t, i)$ or $v(b)$, which are equivalent vector clocks.

4.1.4 Reasoning About Partial Orderings

Some new complexities arise when trying to reason about all the partial orderings which are consistent with Chrysalis Analysis. For instance, there can be an upwards arc from epoch $l+1$ into epoch l, where a subblock in epoch $l+1$ happens before a different subblock in epoch l. This is illustrated in Figure 4.2(b), where the last subblock in the body happens after a subblock in the rightmost thread in the wings. Once all the instructions in epoch l have executed, we also know

that some of the instructions in epoch $l+1$ have also executed, because there exists a maximal subblock in epoch l which happens after a maximal subblock in epoch $l+1$. We call l^+ the **extended epoch of** l, and define it to include all instructions in epoch l, as well as all instructions $(l+1,t,i)$ in epoch $l+1$ for which there is some subblock b in epoch l where $v(l+1,t,i) < v(b)$.

We introduce the concept of a *valid vector ordering*, which extends Butterfly Analysis' *valid ordering* to incorporate happens-before arcs. A valid ordering is any total sequential ordering of instructions consistent with both the intra-thread dependences and the ordering of instructions in non-adjacent epochs. This is too broad a set of orderings for Chrysalis Analysis to use because it includes orderings that violate the happens-before arcs captured by our vector clocks. To capture the more restricted set of orderings, we define valid vector orderings:

Definition 4.1.1. *O_l is a **valid vector ordering (VVO)** if:*

- BACKWARDS COMPATIBLE

  O_l restricted to epochs $[0,l]$ and ignoring happens-before arcs is a valid ordering.

- INCLUDES ALL INSTRUCTIONS THROUGH EPOCH l^+

  All instructions from epochs 0 to l are included exactly once, as well as those instructions from epoch $l+1$ that belong to l^+, with no instructions from epochs $> l+1$ included.

- RESPECTS HAPPENS-BEFORE

  If $v(l,t,i) < v(l',t',i')$, instruction (l,t,i) appears before instruction (l',t',i') in O_l.

4.1.5 Challenge: Maintaining Global State

As in Butterfly Analysis, Chrysalis Analysis requires global state because the analysis only proceeds over a sliding window of execution. Butterfly Analysis made simplifying assumptions that allowed it to symmetrically reason about each butterfly in parallel. For instance, consider the traces depicted in Figure 4.3(d). In Butterfly Analysis, the dashed happens-before arcs (from synchronization) are ignored. Suppose this all occurred within the same epoch l. Both subblocks b_4 and b_6 would conservatively reason that p was tainted before the dereference, as it was possi-

ble that the `taint(p)` occurred between the `untaint(p)` and `*p`. However, that `taint(p)` is issued only once, whereas Thread 1 issues `untaint(p)` twice! It was impossible for both dereferences of `p` to be against a tainted pointer, and also have `p` tainted at the end of epoch l. However, Butterfly Analysis would have summarized this epoch with `p` tainted.

Once Chrysalis Analysis adds the happens-before (dashed) arcs in Figure 4.3(d), it is clear not only that each `*p` is to untainted data, but also that at the end of the epoch, `p` is untainted. This requires epoch-level summarization to consider the happens-before arcs. Summarizing an "epoch" in Chrysalis Analysis will also mean summarizing the extended epoch. Suppose alternatively that subblocks b_2 and b_3 were instead in epoch $l+1$, while all other subblocks were in epoch l as before; b_2 now belongs to l^+, and we still conclude that after all instructions in l^+ have executed, `p` is untainted and we update the global state (SOS) accordingly.

4.1.6 Challenge: Updating Local State

Once again, consider Figure 4.3(d). As stated earlier, the second pass for Butterfly Analysis was entirely in parallel, because there were no additional happens-before arcs. Chrysalis Analysis wishes to improve on this imprecision. As the second pass of the analysis proceeds, at each entry point to a new maximal subblock Chrysalis Analysis must wait for all of its direct parents (those subblocks with a happens-before arc pointing to it) to finish their second pass before beginning. This makes the analysis aware, for example, that b_2's taint of `p` was prior to b_6, and hence b_6's subsequent issue of `untaint(p)` made `*p` safe.

To do this analysis correctly, it is important to note that each direct parent takes on a role analogous to the head in Butterfly Analysis. However, now we can have multiple "heads", arising either due to explicit synchronization or intra-thread sequential semantics. The parents are unlikely to be ordered themselves. Just as in Figure 4.3(c), we will need to conservatively reason about instructions that executed prior to a subblock. This will lead to our treating the local state (LSOS) more as a dataflow problem and less as pure state. We will conduct a "meet" at subblock

entry points of the incoming GEN and KILL sets we define in the sections that follow.

4.2 Reaching Definitions

In this section, we will show how to extend Reaching Definitions, a classical dataflow analysis problem, to Chrysalis Analysis. We will begin by showing how to define generating and killing definitions at the instruction, subblock and epoch level, as well as showing how to compute the Side-In and Side-Out primitives. Then we will show how to update both the Strongly Ordered State (SOS) and the Local Strongly Ordered State (LSOS), and present the two-pass algorithm for reaching definitions. Throughout the section, we will prove key properties of our definitions, showing that Chrysalis Analysis will not "miss an error" (meaning, it will never claim a definition d does not reach a program point p when there was a way for that to happen). Extensions of Chrysalis Analysis to Available Expressions, another classical dataflow analysis problem, and ADDRCHECK, a memory lifeguard, appear in Sections 4.3 and 4.4, respectively. Later in Section 4.5 we will give an example lifeguard, TAINTCHECK, based on reaching definitions.

4.2.1 Gen and Kill equations

We begin by defining GEN and KILL at all granularities, represented by $\mathcal{G}$ and $\mathcal{K}$, respectively.

Instruction-Level

Let $\mathcal{G}_{l,t,i} = \{d\}$ if (l,t,i) generates d. Let $\mathcal{K}_{l,t,i} = \{d | (l,t,i) \text{ kills } d\}$.

Maximal Subblock-Level

We often wish to refer to a maximal subblock b as $(l,t,(i,j))$, meaning it is composed of instructions (l,t,i) through (l,t,j). If $b = (l,t,(i,j))$ then:

$$\mathcal{G}_b = \mathcal{G}_{l,t,(i,j)} = \mathcal{G}_{l,t,j} \cup (\mathcal{G}_{l,t,(i,j-1)} - \mathcal{K}_{l,t,j})\ \mathcal{K}_b = \mathcal{K}_{l,t,(i,j)} = \mathcal{K}_{l,t,j} \cup (\mathcal{K}_{l,t,(i,j-1)} - \mathcal{G}_{l,t,j})$$

with $\mathcal{G}_{l,t,(i,i)} = \mathcal{G}_{l,t,i}$ and $\mathcal{K}_{l,t,(i,i)} = \mathcal{K}_{l,t,i}$ as base cases to the recursion. These are the standard flow equations for GEN and KILL, defined now over maximal subblocks.

Side-Out and Side-In (Per Subblock)

In generalizing Butterfly Analysis' treatment of Side-Out and Side-In, we must take into account the additional information provided by the vector clocks. It now makes sense to consider Side-Out and Side-In per maximal subblock b, rather than per block (l, t). In the event that there are no happens-before arcs, the subsequent equations are equivalent to the original Butterfly Analysis equations for Side-Out and Side-In. Let b and b' be maximal subblocks, where $b = (l, t, (j, k))$. Then the new equations for GEN-SIDE-OUT and GEN-SIDE-IN are:

$$\text{GEN-SIDE-OUT}_b = \bigcup_{j \le i \le k} \mathcal{G}_{l,t,i}$$

$$\text{GEN-SIDE-IN}_b = \bigcup_{\{b' | v(b') \sim v(b)\}} \text{GEN-SIDE-OUT}_{b'}$$

Epoch-Level

We will define three useful sets, AFTER, MB and NOT-BEFORE, and use them to define $\mathcal{G}_l$ and $\mathcal{K}_l$ for an epoch l. If b is a maximal subblock in l^+ (the extended epoch of l), then:

$$\text{MB}_l = \{b | b \text{ is a maximal subblock in epoch } l\}.$$

$$\text{AFTER}_b = \{b' | b' \in \text{MB}_{l^+} \text{ and } v(b) < v(b')\}$$

$$\text{NOT-BEFORE}_b = \{b' | b' \in (\text{MB}_{l-1} \cup \text{MB}_{l^+}) \wedge (v(b) \sim v(b') \vee v(b) < v(b'))\}$$

$$\mathcal{G}_l = \bigcup_{\{b | b \in \text{MB}_{l^+}\}} \left(\mathcal{G}_b - \bigcup_{b' \in \text{AFTER}_b} \mathcal{K}_{b'}\right)$$

$$\mathcal{K}_l = \bigcup_{\{b | b \in \text{MB}_{l^+}\}} \left(\mathcal{K}_b - \bigcup_{\{b' | b' \in \text{NOT-BEFORE}_b\}} \mathcal{G}_{b'}\right)$$

Lemma 6. *If there exists a valid vector ordering (VVO) O_l of the instructions in l^+ such that $d \in \mathcal{G}(O_l)$ then $d \in \mathcal{G}_l$.*

Proof. $d \in \mathcal{G}(O_l)$ implies that there exists an instruction $(\hat{l}, t, i)$, $\hat{l} \in l^+$, in the total order O_l such

that $(\hat{l}, t, i)$ generates d and no subsequent instruction kills d. Let b be the maximal subblock containing $(\hat{l}, t, i)$. By the definition of VVO, there is no instruction (l', t', i') that kills d such that either $v(\hat{l}, t, i) < v(l', t', i')$ or $(\hat{l}, t, i)$ is before (l', t', i') in the same block b. Thus, $d \in \mathcal{G}_b$ (by construction) and $d \notin \bigcup_{b' \in \text{AFTER}_b} \mathcal{K}_{b'}$, implying $d \in \mathcal{G}_l$. □

Lemma 7. *If $d \in \mathcal{K}_l$, then no valid vector ordering O of the instructions in epochs $l-1$ through l^+ exists such that $d \in \mathcal{G}(O)$.*

Proof. If $d \in \mathcal{K}_l$, then by definition there exists a maximal subblock b such that $d \in \mathcal{K}_b$ and for all maximal subblocks b' such that $v(b) \sim v(b')$ or $v(b) < v(b')$, $d \notin \mathcal{G}_{b'}$. Let (l, t, k) be the last instruction in b that kills d; $d \in \mathcal{K}_b$ implies that no instruction in b after (l, t, k) generates d. (Due to data dependences, kills and generates of d are strictly ordered within a subblock.)

Consider any VVO O of the instructions in epochs $l-1$ through l^+. By the definition of VVO, the only instructions following (l, t, k) in O that can kill or generate d are those belonging to any maximal subblock b' that is concurrent or occurs strictly after b. As argued above, $d \notin \mathcal{G}_{b'}$, implying either (i) b' never generates d or (ii) any generation of d in b' is followed by a subsequent kill of d also in b', which would be reflected in O. Thus, any generation of d in O either occurs strictly before (l, t, k), or else is followed by a kill of d; either way, d does not reach the end of O. Hence, $d \notin \mathcal{G}(O)$. □

4.2.2 Strongly Ordered State

As in Butterfly Analysis, Chrysalis Analysis uses the epoch-level summaries $\mathcal{G}_l$ and $\mathcal{K}_l$ to compute the Strongly Ordered State (SOS). This equation is unchanged from Butterfly Analysis; all the changes are in the generalization of $\mathcal{G}_l$ and $\mathcal{K}_l$.

$$\text{SOS}_0 = \text{SOS}_1 = \varnothing$$

$$\text{SOS}_l = \mathcal{G}_{l-2} \cup (\text{SOS}_{l-1} - \mathcal{K}_{l-2}) \; \forall l \geq 2$$

The following theorem proves that if there exists any VVO such that a definition d reaches the end of l epochs, then it will be in SOS_{l+2}.

Theorem 8. *If there exists a valid vector ordering O_l of the instructions in epochs $[0, l^+]$ such that $d \in \mathcal{G}(O_l)$ then $d \in \text{SOS}_{l+2}$.*

Proof. Our proof will proceed by induction on l. In the base case of $l = 0$, we have $\text{SOS}_{l+2} = \mathcal{G}_0$ by an application of Lemma 6. Now assume that the lemma is true for all $l < k$, and show for $l = k$. Suppose $d \in \mathcal{G}(O_l)$. As in Lemma 6, by the definition of VVO, there exists an instruction $(\tilde{l}, \tilde{t}, \tilde{i})$ in O_l generating d such that no subsequent instruction kills d. In particular, $\forall (l', t', i')$ where $v(\tilde{l}, \tilde{t}, \tilde{i}) < v(l', t', i')$, $d \notin \mathcal{K}_{l',t',i'}$. There are two cases:

$\tilde{l} \geq l$: Let b be the maximal subblock containing instruction $(\tilde{l}, \tilde{t}, \tilde{i})$. No subsequent instruction in b can kill d. Thus, $d \in \mathcal{G}_b$ and b belongs to l^+, which implies $d \in \mathcal{G}_l$. Hence, $d \in \text{SOS}_{l+2}$ by definition.

$\tilde{l} < l$: Because, as argued above, there is no kill ordered after $(\tilde{l}, \tilde{t}, \tilde{i})$, we have that $d \notin \mathcal{K}_l$ and there exists a VVO O_{l-1} of the instructions in epochs $[0, (l-1)^+]$ such that $d \in \mathcal{G}(O_{l-1})$. Applying the inductive hypothesis, we have that $d \in \text{SOS}_{l+1}$. Thus, $d \in \text{SOS}_{l+1} - \mathcal{K}_l$, implying $d \in \text{SOS}_{l+2}$.

□

4.2.3 Local Strongly Ordered State

Once we have computed the SOS, the next step is to calculate the Local Strongly Ordered State (LSOS). As mentioned in Section 4.1.6, we face a few challenges in generalizing Butterfly Analysis' LSOS update rule. Butterfly Analysis proposed an equation for calculating the LSOS from the body (block (l, t)) based on the SOS, and the $\mathcal{G}$ and $\mathcal{K}$ from the head (block $(l-1, t)$):

$$\textit{Butterfly Analysis: } \text{LSOS}_{l,t} = \mathcal{G}_{l-1,t} \cup (\text{SOS}_l - \mathcal{K}_{l-1,t}) \cup$$
$$\{d | d \in \text{SOS}_l \wedge d \in \mathcal{K}_{l-1,t} \wedge \exists t' \neq t \text{ s.t. } d \in \mathcal{G}_{l-2,t'}\}$$

This rule took advantage of a specialized structure that does not hold in Chrysalis Analysis. First, there was only one head, or direct predecessor, for any block, so $\mathcal{G}_{l-1,t}$ and $\mathcal{K}_{l-1,t}$ are easily directly referenced. Second, removing a definition $d \in \mathcal{K}_{l-1,t}$ from SOS_l was incorrect if another thread t' had actually generated d in epoch $l-1$ or $l-2$; only $l-2$ had to be directly added back in, because everything in epoch $l-1$ was part of the $\text{GEN-SIDE-IN}_{l,t}$. The union of the final set fixed the accuracy of the LSOS, but at the price of a deviation from the standard $\text{OUT} = \text{GEN} \cup (\text{IN} - \text{KILL})$ formulation.

Chrysalis Analysis must anticipate the possibility of a more generalized structure, where subblocks have multiple direct parents, which may all execute before a particular subblock b, but not necessarily be totally ordered amongst themselves. This is illustrated in Figure 4.3(c), where b_1 and b_3 both occur before b_2 but the taint status of $\mathtt{p}$ is uncertain (conservatively, tainted) before b_2. Dataflow analysis provides a natural way of handling such effects: the meet operator.

Our solution for Chrysalis Analysis will involve representing the local differences applied to the SOS as transfer functions. We will use $\text{IN}^{\mathcal{G}}$ and $\text{OUT}^{\mathcal{G}}$ for GEN difference, and $\text{IN}^{\mathcal{K}}$ and $\text{OUT}^{\mathcal{K}}$ for KILL. Furthermore, we will present a way of calculating the LSOS from the SOS that will use the $\text{OUT} = \text{GEN} \cup (\text{IN} - \text{KILL})$ structure, where $\text{IN} = \text{SOS}_l$ and $\text{OUT} = \text{LSOS}_b$, without involving extra sets.

The rest of the section focuses on the program point immediately before a maximal subblock b. For instructions on the "interior" of b, we can use standard dataflow analysis techniques to update the intermediate state, *i.e.,* $\text{LSOS}_{l,t,i+1} = \mathcal{G}_{l,t,i} \cup (\text{LSOS}_{l,t,i} - \mathcal{K}_{l,t,i})$. We only require a more general solution to the entry points of maximal subblocks. The meet operator ($\bigsqcap$) for reaching definitions in Chrysalis Analysis is union ($\bigcup$).

LSOS: Representing GEN As Transfer Functions

The LSOS transfer functions focus on the sliding window of epochs $l-1$ through $l+1$. Let $\text{MB}_{[l_1,l_2]} = \bigcup_{l_1 \le l_i \le l_2} \text{MB}_{l_i}$. Then, we define $\text{HB}(b)$:

$$\text{HB}(b) = \{b' | v(b') < v(b) \text{ where } \left(b' \in \text{MB}_{[l-1,l+1]}\right) \vee (b' \in \text{MB}_{l-2} \wedge \exists b'' \in \text{MB}_{l-1} \text{ such that } v(b'') < v(b'))\}$$

for maximal subblocks b and b'. Note that this captures all maximal subblocks b' that happen before b and are within the 3 epoch sliding window, as well as including those subblocks in epoch $l-2$ that have predecessors in epoch $l-1$.

For the LSOS, we define $\mathcal{G}_b$ and $\mathcal{K}_b$ to be the standard dataflow formulations of $\mathcal{G}$ and $\mathcal{K}$ over a maximal subblock b, restricted to the sliding window of epochs $l-1$ through $l+1$. Define $\text{pred}(b)$ to be the set of maximal subblocks $b' \in \text{HB}(b)$ such that either (i) b' is the immediate predecessor of b in thread t or (ii) the first instruction of b receives a send from (equivalently, locks an unlock by) the last instruction of b'. Barriers are an all-to-all send/receive. Then we can define the OUT and IN formulas for GEN:

$$\text{OUT}_b^{\mathcal{G}} = \mathcal{G}_b \cup (\text{IN}_b^{\mathcal{G}} - \mathcal{K}_b)$$

$$\text{IN}_b^{\mathcal{G}} = \begin{cases} \varnothing & \text{if } b \text{ is a thread's 1st subblock at level } l-1 \\ \bigsqcap_{b' \in \text{pred}(b)} \text{OUT}_{b'}^{\mathcal{G}} & \text{otherwise} \end{cases}$$

The following establishes the correctness of this formulation:

Lemma 9. *If there exists a valid vector ordering O of the instructions in $\text{HB}(b)$ such that $d \in \mathcal{G}(O)$ then $d \in \text{IN}_b^{\mathcal{G}}$.*

Proof. Suppose $\text{HB}(b)$ is not empty and $d \in \mathcal{G}(O)$. As in earlier proofs, the definition of VVO implies that there exists in O an instruction (l', t', j') that generates d in a maximal subblock b', such that no subsequent instruction kills d, and in particular, $\forall$ maximal subblocks $\tilde{b} \in \text{HB}(b)$ with $v(b') < v(\tilde{b})$, we have $d \notin \mathcal{K}_{\tilde{b}}$. It follows that $d \in \mathcal{G}_{b'}$, and hence $d \in \text{OUT}_{b'}^{\mathcal{G}}$. Moreover, since $d \notin \mathcal{K}_{\tilde{b}}$ and $\tilde{b}$ is not the first subblock for its thread at level $l-1$, we have that $d \in \text{OUT}_{\tilde{b}}^{\mathcal{G}}$ for all such $\tilde{b}$.

If $b' \in \text{pred}(b)$, then by the definition of $\text{IN}_b^{\mathcal{G}}$ and the fact that b is not the first subblock for its

thread at level $l-1$, we have $d \in \text{IN}_b^{\mathcal{G}}$. If $b' \notin \text{pred}(b)$, then $b' \in \text{HB}(b)$ implies that there exists a $\tilde{b} \in \text{pred}(b)$ such that $v(b') < v(\tilde{b})$. As argued above, $d \in \text{OUT}_{\tilde{b}}^{\mathcal{G}}$, and hence $d \in \text{IN}_b^{\mathcal{G}}$. □

$\text{IN}_b^{\mathcal{G}}$ captures the set of local GEN differences to reflect in the LSOS at the entry point to b, i.e., definitions from instructions that executed before b but may not be in the SOS.

LSOS: Representing KILL As Transfer Functions

The formula for $\text{OUT}_b^{\mathcal{K}}$ is similar to the formula for $\text{OUT}_b^{\mathcal{G}}$:

$$\text{OUT}_b^{\mathcal{K}} = \mathcal{K}_b \cup (\text{IN}_b^{\mathcal{K}} - \mathcal{G}_b).$$

Recall that the meet function $\bigsqcap$ is still union, even though we are combining kill sets.

In defining $\text{IN}_b^{\mathcal{K}}$, it helps to have the following set:

$$\text{DEL-INK}_{b'} = \{b'' | (v(b'') \sim v(b')) \vee ((v(b') < v(b'')) \wedge (v(b) \not< v(b'')))\}$$

$$\text{IN}_b^{\mathcal{K}} = \begin{cases} \varnothing & \text{if } b \text{ is a thread's 1st subblock at level } l-1 \\ \bigsqcap_{b' \in \text{pred}(b)} (\text{OUT}_{b'}^{\mathcal{K}} - & \\ \quad (\cup_{b'' \in \text{DEL-INK}_{b'}} \text{GEN-SIDE-OUT}_{b''})) & \text{otherwise} \end{cases}$$

Lemma 10. *If $d \in \text{IN}_b^{\mathcal{K}}$ then $\forall$ valid vector orderings O composed solely of all instructions from maximal subblocks b' such that $v(b') < v(b)$, $d \notin \mathcal{G}(O)$.*

Proof. If $d \in \text{IN}_b^{\mathcal{K}}$ then $\exists b' \in \text{pred}(b)$ such that $d \in \text{OUT}_{b'}^{\mathcal{K}}$ and $\forall b''$ such that b'' is concurrent with b' or b'' occurs after b' but not after b, $d \notin \text{GEN-SIDE-OUT}_{b''}$.

Consider any O, restricted to subblocks that occur before b. It must have a nonempty suffix S beginning with an instruction (l', t', i') that is the last kill of d in b'. By the definition of VVO, the remaining instructions in S must either be concurrent with b' or happen after—precisely the set encapsulated by $\text{DEL-INK}_{b'}$. By construction, if $d \in \text{OUT}_{b'}^{\mathcal{K}} - (\bigcup_{b'' \in \text{DEL-INK}_{b'}} \text{GEN-SIDE-OUT}_{b''})$ then

$d \notin$ GEN-SIDE-OUT$_{b''}$ for all $b'' \in$ DEL-INK$_{b'}$, meaning no later instruction in S can define d. Thus, since $d \in \mathcal{K}(S)$ for a nonempty suffix S implies $d \in \mathcal{K}(O)$, we have that $d \notin \mathcal{G}(O)$. □

Creating LSOS

We now have all the building blocks we need to adjust the formula for calculating LSOS at subblock entry points. If $b = (l, t, (i, j))$ is a maximal subblock, let LSOS$_b$ indicate the LSOS at the entry to block b, namely, LSOS$_{l,t,i}$. Then LSOS$_b$ = IN$_b^{\mathcal{G}} \cup$ (SOS$_l$ − IN$_b^{\mathcal{K}}$).

Theorem 11. *If* $\exists$ *a valid vector ordering* O *of the instructions from epochs* $[0, (l-2)^+]$ *and* HB(b) *such that* $d \in \mathcal{G}(O)$*, then* $d \in$ LSOS$_b$.

Proof. The proof follows from a straightforward extension of Lemma 9. Instead of limiting ourselves to an ordering of instructions in HB(b), we consider all instructions from epochs $[0, (l-2)^+]$ and HB(b). Then if the instruction (l', t', i') generating d has $l' > l - 2$, Lemma 9 dominates. Otherwise, it is still the case the d is not in a later kill set (represented by IN$_b^{\mathcal{K}}$). If we restrict the ordering to the first $[0, (l-2)^+]$ epochs, this is the same as the proof that $d \in$ SOS$_l$ (Theorem 4.2.2), so $d \in$ SOS$_l$ − IN$^{\mathcal{K}}{}_b$. □

4.2.4 In and Out Functions

We now consider what each instruction will compute for its IN and OUT functions. For an instruction (l, t, i) that belongs to maximal subblock $b = (l, t, (j, j'))$:

$$\text{IN}_{l,t,i} = \text{GEN-SIDE-IN}_b \bigcup \text{LSOS}_{l,t,i}$$

$$\text{OUT}_{l,t,i} = \mathcal{G}_{l,t,i} \bigcup (\text{IN}_{l,t,i} - \mathcal{K}_{l,t,i})$$

4.2.5 Applying the Two-Pass Algorithm

Reaching Definitions in Chrysalis Analysis, like in Butterfly Analysis, is implemented as a two-pass algorithm. In the first pass, $\mathcal{G}_b$, $\mathcal{K}_b$ and GEN-SIDE-OUT$_b$ are calculated. Once all threads

finish the first pass, the second pass can begin. The second pass must respect the happens-before arcs; if subblock b' in thread t' is an immediate predecessor of subblock b in thread t, thread t cannot calculate $\text{IN}_b^{\mathcal{G}}$ or $\text{IN}_b^{\mathcal{K}}$ until the second pass of b' has completed and $\text{OUT}_b^{\mathcal{G}}$ and $\text{OUT}_b^{\mathcal{K}}$ are available. This means the $LSOS_b$ cannot be computed until just before the start of the second pass for subblock b. Once all threads have completed the second pass, it is safe to update the SOS.

4.3 Available Expressions

The adaptation of Available Expressions to Chrysalis Analysis proceeds in a similar manner to Reaching Definitions, with the roles of $\mathcal{G}$ and $\mathcal{K}$ reversed. We will show the equations for $\mathcal{G}$ and $\mathcal{K}$ at all granularities, or state when they are equivalent to Reaching Definitions. Due to the strong similarity, when proofs are equivalent up to refactoring $\mathcal{G}$ and $\mathcal{K}$, references to the corresponding proofs are provided instead of duplicating them.

4.3.1 Gen and Kill equations

Instruction Level

Let $\mathcal{G}_{l,t,i} = \{e\}$ if instruction (l, t, i) generates expression e. Let $\mathcal{K}_{l,t,i} = \{e | (l, t, i) \text{ kills } e\}$ (e.g., (l, t, i) may redefine e's terms).

Maximal Subblock-Level

The equations for dataflow at the maximal subblock level are equivalent to those of Reaching Definitions.

Kill-Side-Out/Kill-Side-In

We define KILL-SIDE-OUT$_b$, the analog of GEN-SIDE-OUT$_b$ in Reaching Definitions. As in Reaching Definitions, we now have KILL-SIDE-OUT$_b$ and KILL-SIDE-IN$_b$ at the maximal subblock level. For maximal subblock $b = (l, t, (j, k))$:

$$\text{KILL-SIDE-OUT}_b = \bigcup_{j \leq i \leq k} \mathcal{K}_{l,t,i}$$
$$\text{KILL-SIDE-IN}_b = \bigcup_{\{b' | v(b') \sim v(b)\}} \text{KILL-SIDE-OUT}_{b'}$$

Epoch-Level

In defining the epoch-level sets necessary for Available Expressions, we reuse the sets MB$_l$, AFTER$_b$, and NOT-BEFORE$_b$ defined in Section 4.2.1. For an epoch l and maximal subblocks $b \in \text{MB}_{l^+}$, the epoch-level summaries are:

$$\mathcal{K}_l = \bigcup_{\{b | b \in \text{MB}_{l^+}\}} \left(\mathcal{K}_b - \bigcup_{b' \in \text{AFTER}_b} \mathcal{G}_{b'} \right)$$
$$\mathcal{G}_l = \bigcup_{\{b | b \in \text{MB}_{l^+}\}} \left(\mathcal{G}_b - \bigcup_{\{b' | b' \in \text{NOT-BEFORE}_b\}} \mathcal{K}_{b'} \right)$$

Once more, the roles of $\mathcal{K}$ and $\mathcal{G}$ are reversed compared to Reaching Definitions. This correspondence is evident in the correctness results we obtain for available expressions. The proofs to Lemmas 12 and 13 are quite similar to those of Lemmas 7 and 6, respectively, with the roles of $\mathcal{G}$ and $\mathcal{K}$ reversed. They are provided for completeness.

Lemma 12. *If $e \in \mathcal{G}_l$ then $\forall$ valid vector orderings O of the instructions in $l - 1$ through l^+, $e \in \mathcal{G}(O)$.*

Proof. Suppose $e \in \mathcal{G}_l$. Then there must exist an instruction (l, t, k) in a maximal subblock b such that $e \in \mathcal{G}_b$ and for all subblocks b' such that $v(b) \sim v(b')$ or $v(b) < v(b')$, $e \notin \mathcal{K}_{b'}$. This follows from $e \in \mathcal{G}_l$. Consider any VVO O.

Consider the *suffix* of O beginning with instruction (l, t, k). By definition of VVO, the only instructions that can follow (l, t, k) are other instructions in b (while respecting data dependences), and instructions belonging to any maximal subblock b' which is concurrent with or

strictly after b. We have shown that for such b', $e \notin \mathcal{K}_{b'}$, implying either b' never kills e or any kill of e in b' is followed by a subsequent generate of e also in b'. Applying the definition of VVO, if a generate of e in b' is followed by a kill of e in b', this would be reflected in O. In particular, it is also reflected in the suffix beginning with (l, t, k). Thus, for the suffix of O beginning with (l, t, k), if e is killed at all, it is guaranteed to be followed by a generate of e. So any kill of e in O either occurs strictly before (l, t, k) or else is followed by a generate of e. Either way, e reaches the end of O, thus $e \in \mathcal{G}(O)$. □

Lemma 13. *If $\exists$ a valid vector ordering O of the instructions in l^+ such that $e \in \mathcal{K}(O)$ then $e \in \mathcal{K}_l$.*

Proof. First, there must exist an instruction (l, t, i) in O such that (l, t, i) kills e and no subsequent instruction in O generates e. This follows from $e \in \mathcal{K}(O)$. This implies there is no instruction (l', t', i') such that $e \in \mathcal{G}_{(l',t',i')}$ and $v(l, t, i) < v(l', t', i')$ (by definition of VVO). Then let b be the maximal subblock containing (l, t, i). We know that $e \in \mathcal{K}_b$ (by construction) and that $e \notin \bigcup_{b' \in \text{AFTER}_b} \mathcal{G}_{b'}$, so $e \in \mathcal{K}_l$. □

4.3.2 Strongly Ordered State

The equation for computing SOS_l is unchanged from Reaching Definitions (Section 4.2.2); the differences are captured in the new equations for $\mathcal{G}_l$ and $\mathcal{K}_l$. However, the correctness result we prove varies slightly compared to Reaching Definitions, reflecting the differences between showing the existence of an interleaving where a property holds (Reaching Definitions) and showing that a property holds across all interleavings (Available Expressions).

Theorem 14. *If $e \in \text{SOS}_{l+2}$ then for all valid vector orderings O_l of instructions in epochs $[0, l^+]$, $e \in \mathcal{G}(O_l)$.*

Proof. Our proof will proceed by induction on l. In the base case of $l = 0$, we have $e \in \text{SOS}_2 = \mathcal{G}_0$. Applying Lemma 12 proves the base case. Now assume the lemma is true for all $l < j$, and

show for $l = j$. Suppose $e \in \text{SOS}_{l+2}$. Then either $e \in \mathcal{G}_l$ or $e \in \text{SOS}_{l+1} - \mathcal{K}_l$.

$e \in \mathcal{G}_l$: We need a slight generalization of Lemma 12. Consider any VVO O_l of epochs $[0, l^+]$. Again, there exists an instruction (l, t, k) in a maximal subblock b, $e \in \mathcal{G}_{l,t,k}$, such that $\forall b'$ where $v(b) \sim v(b')$ or $v(b) < v(b')$, $e \notin \mathcal{K}_{b'}$. We again consider the suffix of O_l beginning at (l, t, k), and reach the same conclusion as Lemma 12. So $e \in \mathcal{G}(O_l)$.

$e \in \text{SOS}_{l+1} - \mathcal{K}_l$: Both $e \in \text{SOS}_{l+1}$ and $e \notin \mathcal{K}_l$ hold. Consider any VVO O_l of epochs $[0, l^+]$. Let O' be the restriction of O_l to epochs $[0, (l-1)^+]$. Applying the inductive hypothesis, we know that $e \in \mathcal{G}(O')$. There must be some instruction (l', t, k) in O' such $e \in \mathcal{G}_{l',t,k}$ and no instruction after (l', t, k) in O' kills e. We now return to O_l, and consider the suffix of O_l beginning with (l', t, k). The difference in the two suffixes must be solely made up of instructions from l^+.

By the contrapositive of Lemma 13, we know that no VVO O'' of instructions in l^+ can kill e. It follows that no suffix of O'' can kill e. Integrating a suffix of O'' which does not kill e with the suffix of O' beginning with (l', t, k) (also composed of instructions which do not kill e) cannot kill e, so the suffix beginning at (l', t, k) in O_l must generate e; thus $e \in \mathcal{G}(O_l)$.

□

4.3.3 Local Strongly Ordered State

LSOS: Representing KILL As Transfer Functions

The $\text{OUT}_b^{\mathcal{K}}$/$\text{IN}_b^{\mathcal{K}}$ sets in Available Expressions strongly correspond to $\text{OUT}_b^{\mathcal{G}}$/$\text{IN}_b^{\mathcal{G}}$ in Reaching Definitions. As in Reaching Definitions, the meet operator ($\bigsqcap$) for Available Expressions is union($\cup$). These represent the expressions which should be killed in the LSOS_b as compared to the SOS_l.

$$\text{OUT}_b^{\mathcal{K}} = \mathcal{K}_b \cup (\text{IN}_b^{\mathcal{K}} - \mathcal{G}_b)$$

$$\text{IN}_b^{\mathcal{K}} = \begin{cases} \varnothing & \text{if } b \text{ is a thread's 1st subblock at level } l-1 \\ \bigsqcap_{b' \in \text{pred}(b)} \text{OUT}_{b'}^{\mathcal{K}} & \text{otherwise} \end{cases}$$

In our correctness result, we reuse the notation $\text{HB}(b)$ as defined in Section 4.2.3.

Lemma 15. *If $\exists$ a valid vector ordering O of the instructions in $\text{HB}(b)$ such that $e \in \mathcal{K}(O)$ then $e \in \text{IN}_b^{\mathcal{K}}$.*

The proof for Lemma 15 is essentially identical to Lemma 9, with the roles of $\mathcal{K}$ and $\mathcal{G}$ reversed.

LSOS: Representing GEN As Transfer Functions

The $\text{IN}_b^{\mathcal{G}}$/$\text{OUT}_b^{\mathcal{G}}$ sets in Available Expression strongly correspond to $\text{IN}_b^{\mathcal{K}}$/$\text{OUT}_b^{\mathcal{K}}$ in Reaching Definitions. These represent the expressions which should be added to the LSOS_b as compared to SOS_l.

$$\text{OUT}_b^{\mathcal{G}} = \mathcal{G}_b \cup (\text{IN}_b^{\mathcal{G}} - \mathcal{K}_b)$$

$$\text{DEL-ING}_{b'} = \{b'' | (v(b'') \sim v(b')) \vee ((v(b') < v(b'')) \wedge (v(b) \not< v(b'')))\}$$

$$\text{IN}_b^{\mathcal{G}} = \begin{cases} \varnothing & \text{if } b \text{ is a thread's 1st subblock at level } l-1 \\ \bigsqcap_{b' \in \text{pred}(b)} (\text{OUT}_{b'}^{\mathcal{G}} - & \\ \quad (\cup_{b'' \in \text{DEL-IN}\mathcal{K}_{b'}} \text{KILL-SIDE-OUT}_{b''})) & \text{otherwise} \end{cases}$$

Lemma 16. *If $e \in \text{IN}_b^{\mathcal{G}}$ then $\forall$ valid vector orderings O composed solely of all instructions from maximal subblocks b' such that $v(b') < v(b)$, $e \in \mathcal{G}(O)$.*

The proof for Lemma 16 is essentially identical to the proof of Lemma 10, with the roles of $\mathcal{G}$ and $\mathcal{K}$ reversed.

Creating LSOS

As with the SOS, the equation for the LSOS is unchanged compared to Reaching Definitions:

$\text{LSOS}_b = \text{IN}_b^{\mathcal{G}} \cup (\text{SOS}_l - \text{IN}_b^{\mathcal{K}})$. The differences have been folded into the equations for $\text{IN}_b^{\mathcal{G}}$ and $\text{IN}_b^{\mathcal{K}}$.

Theorem 17. *If $e \in \text{LSOS}_b$, then $\forall$ valid vector orderings O of the instructions from epochs $[0, (l-2)^+]$ and $\text{HB}(b)$, $e \in \mathcal{G}(O)$.*

Proof. Suppose $e \in \text{LSOS}_b$. Then either $e \in \text{IN}_b^{\mathcal{G}}$ or $e \in \text{SOS}_l - \text{IN}_b^{\mathcal{K}}$.

$e \in \text{IN}_b^{\mathcal{G}}$: Then $e \in \mathcal{G}(O)$ by application of Lemma 16.

$e \in \text{SOS}_l - \text{IN}_b^{\mathcal{K}}$: In this case, $e \in \text{SOS}_l$ and $e \notin \text{IN}_b^{\mathcal{K}}$. Applying Theorem 14, we know that for every VVO O' of instructions in epochs $[0, (l-2)^+]$, $e \in \mathcal{G}(O')$. The contrapositive of Lemma 15 implies that no VVO composed solely of instructions in $\text{HB}(b)$ will kill e. Finally, we note that if a sequence of instructions does not kill an expression e, then interleaving that sequence of instructions with an O' which generates e (while maintaing all properties of a VVO) must still generate e. Thus, $e \in \mathcal{G}(O)$.

□

Applying the Two-Pass Algorithm

Available Expressions is also implemented as a two pass algorithm. In the first pass, $\mathcal{G}_b$, $\mathcal{K}_b$ and KILL-SIDE-OUT_b are calculated. When all threads complete the first pass, threads begin the second pass. During the second pass, threads wait for their predecessors b' to compute $\text{OUT}_{b'}^{\mathcal{G}}$ and $\text{OUT}_{b'}^{\mathcal{K}}$ so they can compute the $\text{IN}_b^{\mathcal{G}}$, $\text{IN}_b^{\mathcal{K}}$ and ultimately LSOS_b. The KILL-SIDE-IN_b can be computed on demand during the second pass. Finally, after all threads complete the second pass, the SOS is updated.

4.4 AddrCheck

As in Butterfly Analysis, we model ADDRCHECK on available expressions. Allocations will "generate" the memory location "expression", and deallocations kill such "expressions". Let

$\mathcal{G}_{l,t,i} = \{m\}$ if and only if instruction (l,t,i) allocates memory location m and otherwise $\varnothing$. Likewise, $\mathcal{K}_{l,t,i} = \{m\}$ if and only if instruction (l,t,i) deallocates memory location m and otherwise $\varnothing$. $\mathcal{G}_b$, $\mathcal{K}_b$, $\mathcal{G}_l$, $\mathcal{K}_l$, $\text{IN}_b^{\mathcal{G}}$, $\text{IN}_b^{\mathcal{K}}$, SOS and LSOS all take their form from the Available Expressions template. In addition, ADDRCHECK tracks a unified read/write set called ACCESS.

We extend the two modes of checking, local and isolation, introduced by Butterfly Analysis. Local checking verifies that any address that was accessed or deallocated in a thread's maximal subblock was locally allocated at the start of the subblock; it also verifies that any address that was allocated in a thread's maximal subblock was locally deallocated at the start of the subblock. Isolation checking ensures that these local checks do not miss interference by another thread; for example, if Thread 1 believes address m to be locally allocated, but was unaware it had been recently freed by Thread 2.

Local Checks

As in Butterfly Analysis, local checks are resolved via LSOS lookups. The formulas for the LSOS generalize in the same ways the formulas for available expressions were generalized.

Isolation Checks and Summaries

Isolation checks again utilize a *summary*, which is now for a maximal subblock b instead of a block (l,t). A summary is represented as $s_b = (\mathcal{G}_b, \mathcal{K}_b, \text{ACCESS}_b)$, where ACCESS_b contains all addresses that subblock b read or wrote.

We create a "side-in" summary:

$$S_b = (\bigcup_{\{b' | v(b') \sim v(b)\}} \mathcal{G}_{b'}, \bigcup_{\{b' | v(b') \sim v(b)\}} \mathcal{K}_{b'}, \bigcup_{\{b' | v(b') \sim v(b)\}} \text{ACCESS}_{b'}).$$

To verify isolation, we check that the following set is empty:

$$((s_b.\mathcal{G}_{l,t} \cup s_b.\mathcal{K}_{l,t}) \bigcap (S_b.\mathcal{G}_{l,t} \cup S_b.\mathcal{K}_{l,t})) \bigcup$$

$$(s_b.\text{ACCESS}_{l,t} \bigcap (S_b.\mathcal{G}_{l,t} \cup S_b.\mathcal{K}_{l,t})) \bigcup$$
$$(S_b.\text{ACCESS}_{l,t} \bigcap (s_b.\mathcal{G}_{l,t} \cup s_b.\mathcal{K}_{l,t}))$$

and otherwise flag an error.

Theorem 18. *Any error detected by the sequential* ADDRCHECK *on a valid vector ordering O for a given machine will also be flagged by Chrysalis Analysis.*

Proof. The correctness proof follows the lines of the Theorem 3 in Chapter 3 for Butterfly Analysis. Observe that ADDRCHECK detects errors based on a pairwise interactions between operations (i.e., allocations, accesses and frees). Suppose there was an execution E such that a sequential ADDRCHECK would have caught an error on memory location x. Represent this execution as E, with $E|x$ the execution restricted to operations utilizing x. By the assumptions of Chrysalis Analysis, a VVO O must exist such that $O|x$, is equivalent to $E|x$. Since Chrysalis Analysis will take O into account, it will also catch the error. □

4.5 TaintCheck

As in Butterfly Analysis, we build the Chrysalis Analysis extension of TAINTCHECK on top of reaching definitions. TAINTCHECK presents a unique challenge, as it incorporates not only dataflow but also *inheritance*. Instead of definitions, expressions, or addresses, a particular $\mathcal{G}_{l,t,i}$ is actually a transfer function. An instruction can either taint a memory location x, untaint a memory location x, be a unary operation on some location a or a binary operation on locations a and b. More formally, as in Butterfly Analysis, define $\mathcal{G}_{l,t,i}$ as:

$$\mathcal{G}_{l,t,i} = \begin{cases} (x_{l,t,i} \leftarrow \bot) & \text{if } (l,t,i) \equiv \texttt{taint}(x) \\ (x_{l,t,i} \leftarrow \top) & \text{if } (l,t,i) \equiv \texttt{untaint}(x) \\ (x_{l,t,i} \leftarrow \{a\}) & \text{if } (l,t,i) \equiv x := \texttt{unop}(a) \\ (x_{l,t,i} \leftarrow \{a,b\}) & \text{if } (l,t,i) \equiv x := \texttt{binop}(a,b) \end{cases}$$

When $x := \texttt{unop}(a)$, we say x *inherits (metadata) from* a and likewise $x := \texttt{binop}(a, b)$ indicates x *inherits (metadata) from* a *and* b. We use the set $S = \{\top, \bot, \{a\}, \{a, b\} | a, b$ are memory locations$\}$ to represent the set of all possible right-hand values in our mapping. We will also utilize the function `loc()` that given (l, t, i) returns x, where x is the destination location in instruction (l, t, i). As in Butterfly Analysis, $\mathcal{K}_{l,t,i}$ takes the form:

$$\mathcal{K}_{l,t,i} = \{(x_{l,t,j} \leftarrow s) | s \in S, j < i, \texttt{loc}(l, t, j) = \texttt{loc}(l, t, i)\}$$

$\mathcal{G}_b$, $\mathcal{K}_b$, GEN-SIDE-OUT$_b$ and GEN-SIDE-IN$_b$ all follow the reaching definitions template, but now track transfer functions instead of actual states. However, the LSOS and SOS still need to be states, as in reaching definitions. In general, we would like GEN to track $\bot$ and KILL to track $\top$. To convert between transfer functions and actual metadata, Butterfly Analysis introduced a `resolve`, or checking, algorithm: $\texttt{resolve}(m, l, t, i)$ takes a memory location m which is the destination of instruction (l, t, i) and returns either $\top$ or $\bot$.

Resolving Transfer Functions to Taint Metadata

TAINTCHECK requires resolving potential inheritance relationships when the ordering between concurrent instructions is unknown. In Section 3.3.2, we introduced an algorithm for "resolving" inheritance by recursively evaluating transfer functions in the wings, subject to two termination conditions: one for sequential consistency and one for relaxed memory models. The addition of vector clocks naturally prunes the search space any taint resolution algorithm has to explore: we associate the vector clock with each predecessor and verify that the current path of vector clocks is a VVO. The `resolve` algorithm is shown in Algorithm 2. Our `resolve` algorithm takes an input a tuple (m, l, t, i) and a set T of transfer functions, and returns the taint status of m at instruction (l, t, i). For brevity within `resolve`, $\texttt{loc}(y_i)$ will refer to the destination of the instruction associated with y_i.

We define a *proper predecessor* of $x_{l,t,i} \leftarrow s$ to be any $y_{l',t',i'} \leftarrow s'$ such that $\texttt{loc}(l', t', i') \in s$, $s \in S$ and $v(l, t, i) \not< v(l', t', i')$.

Algorithm 2 TAINTCHECK `resolve`(m, l, t, i) ALGORITHM

Input: $m, (l, t, i), T$
Initialize $P(m, l, t, i)$ as the list of proper predecessors of $(x_{l,t,i} \leftarrow s)$ from T: $\{(y_0 \leftarrow s_0), \ldots, (y_k \leftarrow s_k)\}$, where $\texttt{loc}(y_i) \in s$.
for all $(y_j \leftarrow s_j) \in P(m, l, t, i)$ **do**
  if $s_j = \bot$ **then**
    Terminate with the rule $(x_{l,t,i} \leftarrow \bot)$.
  else if $s_j = \top$ **then**
    Remove the $(y_j \leftarrow \top)$ from $P(m, l, t, i)$, and continue
  Add the proper predecessors $(y'_{i'} \leftarrow s'_{i'}) \in T$ of $(y_j \leftarrow s_j)$ to $P(m, l, t, i)$, subject to a termination condition and verification that following these new arcs does not violate VVO rules.
Postcondition: Either $(x_{l,t,i} \leftarrow s)$ converges to $(x_{l,t,i} \leftarrow \bot)$, or $P(m, l, t, i)$ becomes empty. If $P(m, l, t, i)$ is empty, conclude $(x_{l,t,i} \leftarrow \top)$.

The sequential consistency and relaxed memory consistency models termination conditions are maintained.[4] Now, `resolve` will only replace a predecessor $y_j \leftarrow s$ with a new predecessor $y_{j'} \leftarrow s'$ if, using vector clocks, the instruction associated with $y_{j'}$ *does not occur after* the instruction associated with y_j.

Converting Transfer Functions Into Metadata

We will use the function LASTCHECK(x, b), introduced in Butterfly Analysis, which represents the last taint status returned when resolving the metadata of location x in subblock b, modified to summarize a subblock b instead of a block (l, t). If x was the destination for an instruction in subblock b, then LASTCHECK(x, b) will return $\top$ or $\bot$; otherwise, it returns $\varnothing$. This serves as a proxy for $\mathcal{G}_b$ or $\mathcal{K}_b$ whenever we require states, not transfer functions.

[4]The sequential consistency termination condition required ensuring that a sequence of proper predecessor replacement operations, when restricted to a single thread, only allowed replacement if the associated instruction occurred earlier within the thread; the relaxed memory consistency termination condition disallowed a predecessor to eventually be replaced by itself. These are the same termination conditions presented in Section 3.3.2.

Extracting GEN-SIDE-OUT into State

We introduce a new set DIDTAINT_b for a maximal subblock b, which includes memory location m if there exists an instruction (l, t, i) contained within b for which m was the destination, and $\texttt{resolve}(m, l, t, i)$ returned $\perp$. More formally:

$$\text{DIDTAINT}_b = \{m | \exists (l, t, v), i \leq v \leq j, m = \texttt{loc}(l, t, v) \wedge \texttt{resolve}(m, l, t, v) \leftarrow \perp\}$$

If the `resolve` function for a location m ever returns $\perp$, $m \in \text{DIDTAINT}_b$. This set is now used in place of the GEN-SIDE-OUT whenever we need an actual state versus transfer functions.

At this point, we have almost completed the adaptation of TAINTCHECK into Chrysalis Analysis. We have $\mathcal{G}_b$, $\mathcal{K}_b$, GEN-SIDE-OUT and GEN-SIDE-IN, and when necessary can move between transfer functions and actual states.

Complications: Calculating $\text{IN}_b^{\mathcal{K}}$

One complication arises that was not an issue for Butterfly Analysis. Butterfly Analysis had a special case for updating the LSOS, which used the head. Because the head always executed before the body, LASTCHECK for the head was always available. In Section 4.2.3, our generalization of the update rules for the LSOS requires access to GEN-SIDE-OUT_b as states, not transfer functions. We require them before we begin the second pass over subblock b, but they are not guaranteed to be available until after each thread completes its entire second pass. Recall the equations:

$$\text{DEL-INK}_{b'} = \{b'' | (v(b'') \sim v(b')) \vee ((v(b') < v(b'')) \wedge (v(b) \not\sim v(b'')))\}$$
$$\text{IN}_b^{\mathcal{K}} = \bigsqcap_{b' \in \text{pred}(b)} \left(\text{OUT}_{b'}^{\mathcal{K}} - \left(\bigcup_{b'' \in \text{DEL-INK}_{b'}} \text{GEN-SIDE-OUT}_{b''} \right) \right)$$

At first, it seems paradoxical. However, the situation is salvageable after making a key observation. If $b'' \in \text{DEL-INK}_{b'}$ and $v(b'') \not\sim v(b)$, then $v(b'') \sim v(b)$. (To see this, suppose instead that $v(b'') > v(b)$. Because $b' \in \text{pred}(b)$, we have $v(b') < v(b)$, which implies $v(b') < v(b'')$. But then by the definition of DEL-INK, $v(b) \not\sim v(b'')$, a contradiction.) In other words, when we first need to calculate $\text{IN}_b^{\mathcal{K}}$ before the second pass over block b, if not all of the actual $\text{DIDTAINT}_{b''}$

are available, at least the transfer functions are; and we will use them in the `resolve` process. As long as our `resolve` process is accurate, our second pass will still be accurate.

However, there is another use of $\text{IN}_b^{\mathcal{K}}$ and $\text{OUT}_b^{\mathcal{K}}$, namely, seeding the next epoch's initial $\text{IN}^{\mathcal{K}}$ for the same thread. To fix this, we make a second observation: All of the subblocks in epoch l^+ had correct and complete DIDTAINT sets available once their second pass was completed. Starting from the initial seed values of $\text{IN}^{\mathcal{K}}$ for the first subblock of epoch l in each thread, we can recompute $\text{IN}^{\mathcal{K}}$ and $\text{OUT}^{\mathcal{K}}$ for all subblocks so that the next sliding window can proceed. While not every subblock from epoch $l+1$ will have a DIDTAINT set ready, all subblocks (and their GEN-SIDE-OUT) in epoch $l+1$ remain available in the next sliding window.

Updating State

Once we have LASTCHECK_b and DIDTAINT_b as the state proxies for $\mathcal{G}_b$, $\mathcal{K}_b$, GEN-SIDE-OUT_b and GEN-SIDE-IN_b, we can calculate $\mathcal{G}_l, \mathcal{K}_l$ and SOS_l, using the same SOS update rules as reaching definitions. With $\text{IN}_b^{\mathcal{K}}$, $\text{OUT}_b^{\mathcal{K}}$, $\text{IN}_b^{\mathcal{G}}$ and $\text{OUT}_b^{\mathcal{G}}$, we can compute the LSOS.

Theorem 19. *If* `resolve` *returns* $(x_{l,t,i} \leftarrow \top)$*, then there is no valid vector ordering of the instructions in epochs* $[0, (l+1)+]$ *such that* x *is* $\bot$ *at instruction* (l, t, i)*.*

Proof. Suppose there were a VVO such that $x_{l,t,i} \leftarrow \bot$ at instruction (l, t, i). This implies a finite sequence of transfer functions $\hat{f}$ such that the associated instructions in order would (a) taint x and (b) obey all vector orderings. Our `resolve` algorithm will follow all valid vector orderings, so it would have discovered the $x_{l,t,i} \leftarrow \bot$ and returned $\bot$, a contradiction. □

It follows that any error detected by the original TAINTCHECK on a valid execution ordering for a given machine (with a memory model that at least obeys intra-thread dependences and supports cache coherence) will also be flagged by Chrysalis Analysis.

4.6 Evaluation and Results

We now present our preliminary experimental evaluation of TAINTCHECK comparing the precision and performance of our TAINTCHECK implementation in Chrysalis Analysis to our implementation in Butterfly Analysis. Both the Chrysalis Analysis and Butterfly Analysis implementations of TAINTCHECK are new for this work.

4.6.1 Experimental Setup

Chrysalis Analysis, like Butterfly Analysis, is general purpose and can be implemented using a variety of dynamic analysis frameworks, including those based on binary instrumentation [20, 71, 82]. We built Chrysalis Analysis on top of the *Log-Based Architectures* (LBA) framework [23]. In LBA, every application thread is monitored by a dedicated lifeguard thread running on a core distinct from the application; as the application executes, a dynamic instruction trace is captured and transported to the lifeguard through a log that resides in the last-level on-chip cache. LBA itself is modeled using the Simics [108] full-system simulator.

We implemented a word-granularity version of TAINTCHECK in both Chrysalis and Butterfly Analyses. Some conservative assumptions were made in the `resolve` algorithm, such as setting a threshold for how many predecessors (set at 1024) we will follow before cutting off the `resolve` algorithm and conservatively tainting the destination. We tested our implementation on four Splash-2 benchmarks [113], shown in Table 4.6.1, where we synthetically tainted the benchmarks' input data. We tested two different configurations for both Butterfly and Chrysalis Analysis, shown in Table 4.3. To capture happens-before arcs, we leveraged the wrapper for shared library calls used by LBA [23] and generated vector clocks on the producer side, which were communicated to the lifeguard cores via the log.

[5]We used LBA to generate and communicate epoch boundaries, inserting epoch boundaries after hn instructions had been executed by the entire application, where n is the number of application threads; h was set at 8K for these experiments.

Table 4.2: Splash-2 [113] benchmarks used in evaluation

Benchmark	Inputs
BARNES	512 bodies
FFT	$m = 14$ (2^{14} sized matrix)
FMM	512 bodies
LU	Matrix size: 128×128, $b = 16$

Table 4.3: Simulator Parameters used in evaluation

Simulation Parameters	
Cores	{4, 8} cores
Application/Lifeguard	{2/2, 4/4} Threads
L1-I, L1-D	64KB
L2	{2MB, 4MB}
Epoch boundaries	Insert every 8K instructions/thread (on average)[5]

Table 4.4: Potential errors reported by our lifeguard. Two configurations are shown, each with a Butterfly and Chrysalis implementation.

	4-core		8-core	
	Butterfly	Chrysalis	Butterfly	Chrysalis
BARNES	13	1	38	0
FFT	3	0	9	0
FMM	62	0	93	12
LU	5	0	10	0
Total	83	1	150	12

Our prior work on Butterfly Analysis, presented in Chapter 3, focused on ADDRCHECKas a lifeguard, assumed pointers to memory in the benchmarks tested were properly allocated and thus any reported errors were false positives. In contrast, our new TAINTCHECK tool reports *potential errors*. Due to the introduction of synthetic tainting, true positives are possible if taint flows from a synthetically tainted address to a jump target, for example. Accordingly, we treat the total number of potential errors reported as a ceiling for false positives encountered. The reduction in potential errors in our results comparing the Butterfly and Chrysalis precision numbers is entirely due to Butterfly reporting false positives due to the lack of happens-before arcs. The current Chrysalis implementation can only report potential errors, and not distinguish between false and true positives.

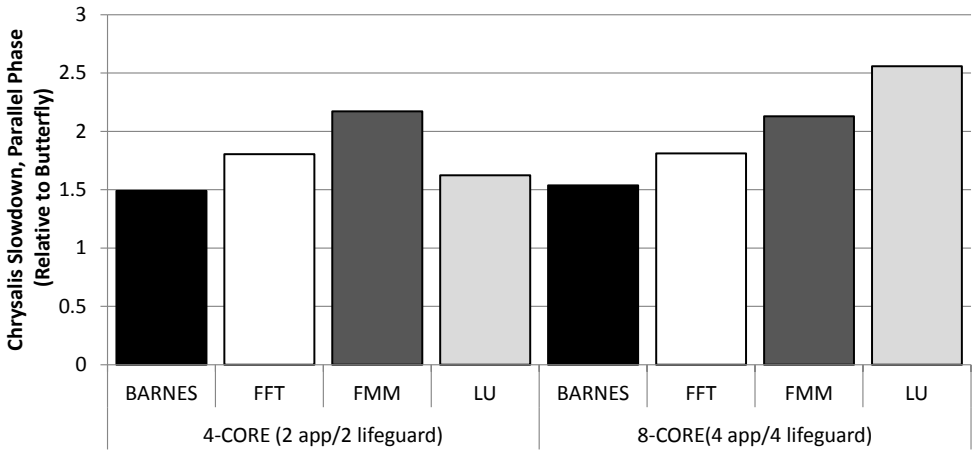


Figure 4.4: Chrysalis Analysis, normalized to Butterfly Performance.

4.6.2 Results

The primary motivation for Chrysalis Analysis was improved precision relative to Butterfly Analysis. As shown in Table 4.4, precision in TAINTCHECK improved significantly compared to Butterfly Analysis for all benchmarks and both configurations (4- and 8-core). Some false positives were possible in our implementation of TAINTCHECK as we made several conservative decisions in both the Chrysalis and Butterfly Analysis implementations, such as fixing a threshold for exploring predecessors in `resolve`, tracking taint status at word (instead of byte) granularity, and using synthetic tainting. Conservative decisions made in situations such as Figure 4.1(c), when there is not enough ordering information to precisely determine taint status, could also lead to a memory address being falsely tainted.

For the 4-core configuration, Chrysalis Analysis reports only one potential error across all benchmarks, on the BARNES run. The equivalent Butterfly Analysis 4-core BARNES run has 13 potential errors. On the 8-core configuration, the only Chrysalis Analysis run to report potential errors is FMM. Its Butterfly Analysis counterpart has 93 potential errors, compared to 12 for Chrysalis, approximately a 7.8x reduction in false positives. The potential errors in the 8-core Chrysalis FMM run correspond primarily to starting and exiting a thread as well as `pthread_mutex_lock`. Note that the implementation of the high-level synchronization prim-

itives that we capture cannot themselves be protected by the same high-level synchronization, so we may miss some arcs that would prevent races. *Over all benchmarks and configurations, Chrysalis Analysis improved precision by a factor of 17.9x relative to Butterfly Analysis.*

Next, we examine the performance overheads of Chrysalis Analysis relative to Butterfly Analysis. Across all benchmarks, the slowdowns range from 1.5x to less than 2.6x. Over all benchmarks and configurations, the geometric mean slowdown is approximately 1.9x. This is not an unreasonable tradeoff; an average of less than two-fold slowdown in exchange for a drastic improvement in precision. For BARNES, FFT, and FMM, the slowdowns remain fairly constant when comparing the 8-core configuration to the 4-core configuration, indicating that the Chrysalis Analysis implementation is scaling at the same rate as the Butterfly Analysis tool. There is an increase in overhead for the 8-core LU, but even in this case its overheads are still less than 2.6x.

Because our prototype of Chrysalis Analysis was intended to be a proof-of-concept rather than a highly tuned piece of software, we believe that the results shown in Figure 4.4 are conservative.

4.7 Related Work

This work significantly extends our previous work on Butterfly Analysis presented in Chapter 3, making use of vector clocks to track synchronization events. Vector clocks [11, 24, 102] have been used in a number of data race detectors [19, 37, 38, 76, 99, 121]. For example, Flanagan and Freund proposed *FastTrack* [37], which primarily uses a compact representation to detect data races but still uses vector clocks to track `lock` and `unlock` operations. *FastTrack* achieves precision similar to full vector-clock based methods and performance similar to LockSet [98]. Muzahid *et al.* [76] divide thread execution into epochs to form a data race detector based on signatures, and use vector clocks to determine happens-before relationships. In contrast, Chrysalis Analysis is not simply a data race detector, but a general dataflow analysis framework for imple-

menting a broad range of sophisticated lifeguards.

4.8 Chapter Summary

To retain the advantages of Butterfly Analysis while reducing the number of false positives, we have proposed and evaluated *Chrysalis Analysis*, which incorporates happens-before information from explicit software synchronization. Our implementation of the TAINTCHECK lifeguard demonstrates that Chrysalis Analysis reduces the number of false positives by 17.9x while increasing lifeguard overhead by an average of 1.9x.

Chapter 5

Explicitly Modeling Uncertainty to Improve Precision and Enable Dynamic Performance Adaptations

In Chapter 4, we presented Chrysalis Analysis [44], a generalization of Butterfly Analysis [45] which incorporates high-level synchronization-based happens-before arcs. Chrysalis Analysis dramatically improved Butterfly Analysis' precision–on average, approximately $17.9x$–at a cost of approximately $1.9x$ average slowdown relative to Butterfly Analysis. However, of the remaining potential errors reported, Chrysalis Analysis did not offer any way to differentiate true errors, or to explore the root cause of the error being flagged.

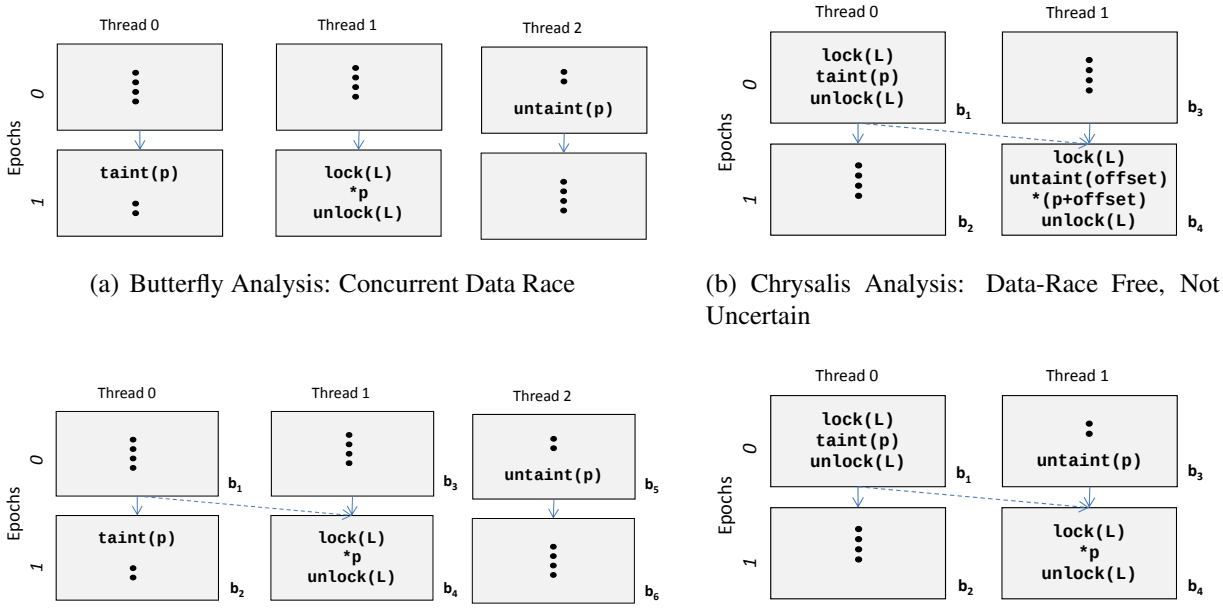
In adapting analyses such as TAINTCHECK to dataflow-analysis based dynamic parallel monitoring, one formerly precise state `taint` became a *conservative* metadata state; a failed check of `tainted` memory could now indicate a true error or a false positive due to conservative analysis; it was impossible to tell. The `untaint` state remained precise, which allowed for the theoretical guarantees that no errors were ever missed. However, programmers are interested in more than just potential errors. It is much more useful to be able to differentiate between true and potential errors.

With Chaotic Butterfly (and similarly, Chaotic Chrysalis), the former precise metadata state of `taint` is restored as precise once more. A new state of `uncertain` is introduced to the metadata lattice. The purpose of the uncertain state is to explicitly capture the effects of conservative analysis. Consider a dynamic information flow tracking lifeguard such as TAINTCHECK [84], which has two metadata states: `taint` and `untaint`. In both Butterfly Analysis and Chrysalis Analysis, the `untaint` state is *precise*, meaning that an `untaint` judgment for memory location `m` implies that under all possible orderings consistent with the observed partial ordering, `m` is always `untainted`. The `taint` state, on the other hand, is *conservative*. If at least one ordering of instructions would have `taint m`, then `m` is considered `tainted`.

However, anytime the analysis had to behave conservatively, we assumed the worst case and returned `taint`. Thus, when the analysis reported a potential error, it could not distinguish between a "true error" and false positives due to conservative analysis. Incorporating an explicit uncertain state allows these conservative judgments to be isolated into an uncertain state. Thus, the `taint` state becomes precise: if the analysis reports `m` is tainted, then under all possible orderings, `m` is tainted. The `uncertain` state captures the cases where all orderings do not lead to a single unified judgment of either `taint` or `untaint`, as well as any other cases where the analysis makes a conservative judgment. We can still guarantee that no true error is ever missed; however, it can now fall into a bucket of either a true positive or a potential error (when the analysis does not have enough information to know if the error actually manifested).

Uncertainty Examples

Consider Figure 5.1. In Figure 5.1(a), an example of uncertainty within Butterfly Analysis is shown. The `untaint(p)` in Thread 2, epoch 0 and the `taint(p)` in Thread 0, epoch 1 are both considered *concurrent* with respect to the dereference `*p` in Thread 1 epoch 1–leading Thread 1 to conclude the metadata status of `p` is *uncertain* at the dereference point. Likewise, casting this example into Chrysalis Analysis, as shown in Figure 5.1(c) does not necessarily



(a) Butterfly Analysis: Concurrent Data Race

(b) Chrysalis Analysis: Data-Race Free, Not Uncertain

(c) Chrysalis Analysis: Concurrent Data Race

(d) Chrysalis Analysis: Prior Data Race

Figure 5.1: (a) Uncertainty in Butterfly Analysis, with a data race in the wings. The different orderings of `taint(p)` in Thread 0 and `untaint(p)` in Thread 2, relative to the dereference `*p` in Thread 1, causes the metadata state to be *uncertain*. (c) Recasting the example from (a) into Chrysalis Analysis doesn't guarantee that the data race will resolve, and the metadata state for `*p` is still uncertain. (d) An example in Chrysalis Analysis which incorporates a data race which is known to resolve before the dereference `*p` in b_4. (b) An example which deceptively resembles (c) and (d)–here a data race free program dereferences `*(p+offset)`, where `p` is `tainted` and `offset` is `untainted`. According to the rules of TAINTCHECK, `*(p+offset)` is treated as `tainted`

resolve the uncertainty of `p`'s metadata state. Finally, Figure 5.1(d) demonstrates a canonical tension arising within Chrysalis Analysis: both the `taint(p)` and `untaint(p)` are resolved before the `*p`, but their ordering relative to each other is still unknown, leading to an uncertain metadata state for `p`. In Figure 5.1(b), a data race free program is shown where the dereference's safety derives from `p`, which is `tainted`, as well as from `offset`, which is `untainted`. Unlike prior examples, `*(p+offset)` is considered `tainted` and not `uncertain`, as the taint status for `p` is definitive.

Note that it makes sense to incorporate uncertainty into both Butterfly Analysis and Chrysalis Analysis. If a programmer is only concerned about true errors (those which must have occurred), and if Butterfly Analysis paired with explicit uncertainty can isolate true errors from potential

errors well, then the performance advantage of Butterfly Analysis may make it the better choice. For this reason, we have incorporated an `uncertain` state into both the Butterfly Analysis and Chrysalis Analysis theoretical formulations.

5.1 "Subtyping" Uncertainty: Tracking Causes of Uncertainty

From a theoretical perspective, uncertainty is treated as only one state within the metadata lattice, which is the new $\bot$; our theoretical results will only have one `uncertain` state. However, once we have proven guarantees about the uncertain state, we can further "subtype" it to track the cause of the conservative judgment. In one case, the state of memory location m may be considered `uncertain` if two different potential orderings are consistent with the observed partial order, with one of them leading to `taint` and the other leading to `untaint`. In another case, heuristics introduced into the analysis (such as thresholds for how long one should search a graph for paths to `tainted` or `untainted` ancestors) may lead a search to be cut off early. These two types of uncertainty have different root causes: in one case, while the error may not have occurred, nothing prevents it from occurring on another run. In the other, a change of threshold may lead to a more precise analysis, which is useful information for an end-user to have.

5.2 Overview of Uncertainty

Incorporating uncertainty into dataflow-analysis based dynamic parallel monitoring requires a few key changes. First, a new state must be introduced to capture uncertainty. Next, all the dataflow equations must be updated. In the setting of TAINTCHECK, this means that the equations from when an address is `tainted` to separate true taint from potentially tainted; potentially tainted addresses will now be considered `uncertain`. To understand how and where these equations must change, it helps to examine different scenarios where uncertainty arises.

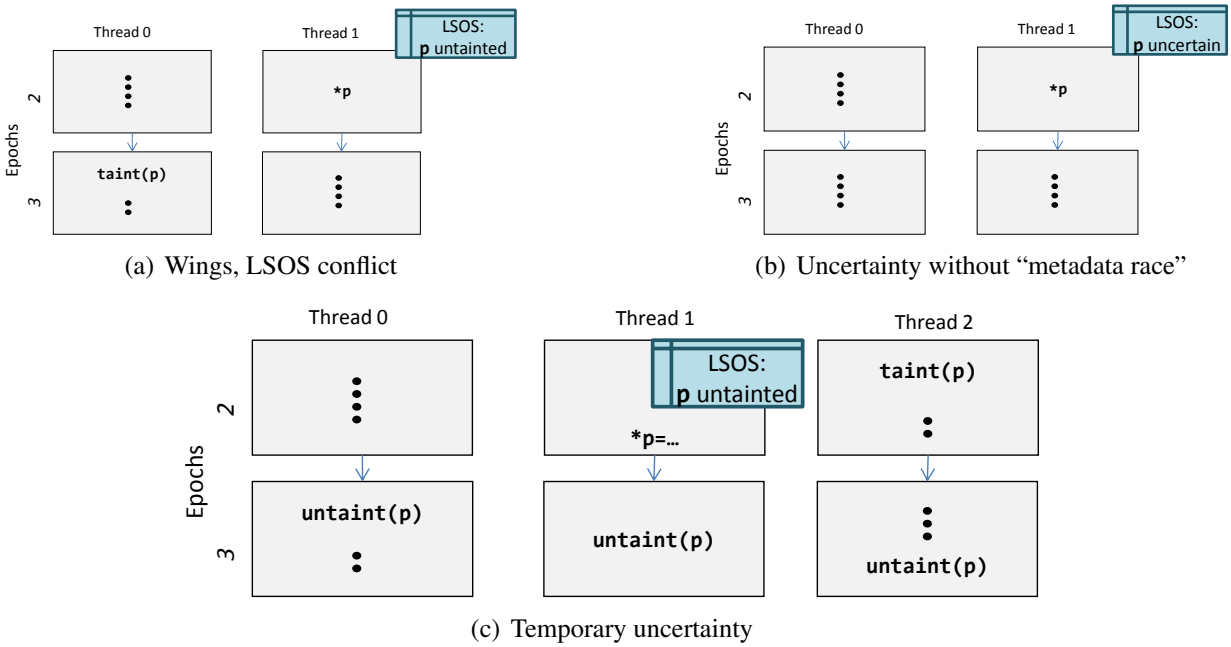


Figure 5.2: Additional examples of uncertainty, shown in Butterfly Analysis. In (a), thread 0 issues `taint(p)` concurrently with Thread 1 issuing `*p`; however, Thread 1 is uncertain whether it read the tainted value or the untainted value reflected in the LSOS. In (b), even without concurrent threads altering the metadata status for `p`, it is still possible for Thread 1 to consider the metadata for `p` uncertain if the uncertainty arose earlier in analysis. Finally, (c) illustrates that uncertainty can arise temporarily; at the end of epoch 3, it is clear that `p` is untainted, but in epoch 2, it is not possible for Thread 1 to know whether it read a `tainted` or `untainted` version of `p`.

5.2.1 Uncertainty Examples: Scenarios where uncertainty arises

Consider Figure 5.1(a), depicting a threads 0 and 2 racing on pointer `p`; Thread 0 issues `taint(p)` whereas Thread 2 issues `untaint(p)`, shown in Butterfly Analysis. Thread 1 is dereferencing `p`. Note that the uncertainty does not arise merely because threads 0 and 2 are racing while writing values to `p`; it arises because they are *concurrent* and the writes lead to conflicting metadata values: in this case, `taint` and `untaint`. We will call that a *metadata race*. This same situation is possible in Chrysalis Analysis, if the program is not data race free, as shown in Figure 5.1(c).

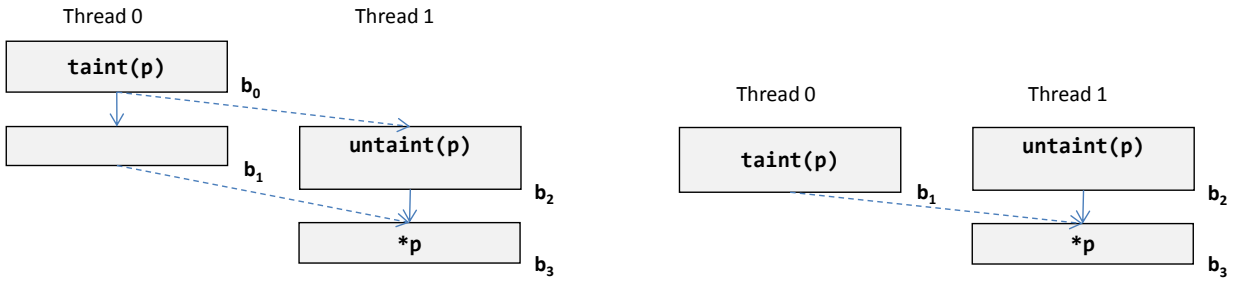
Its also not necessary that the metadata race occur simultaneously with the access. For instance, in Figure 5.1(d), the metadata race is guaranteed to have resolved before Thread 1 issues `*p`. However, *simply knowing that the race has resolved does not mean the analysis knows how*

it has resolved. Figures 5.2(a) and 5.2(b) illustrate how uncertainty does not even necessarily arise from a metadata race. In Figure 5.2(a), the LSOS indicates that `p` is `untainted`, whereas Thread 0 in epoch 3 issues `taint p`. Based on the thread execution model, we know that instructions which are represented by the LSOS have already committed and thereby cannot have been issued concurrently with instructions in epoch 3. However, Thread 1 is not sure whether the `taint(p)` by Thread 0 occurs before or after it; if the `taint(p)` occurs after, then it must have read the `p: untainted` value from its LSOS but if it occurs before, Thread 1 sees `p: tainted`. Thus, despite these two events being ordered, it is still the case that Thread 1 is `uncertain` of the actual metadata value for `p`. Figure 5.2(b) reflects that uncertainty can propagate into the SOS and into the LSOS, leaving a check by Thread 0 for the metadata of `p` in `*p` to return `uncertain` even when no other thread is concurrently changing the metadata status for `p`. Finally, Figure 5.2(c) illustrates how the metadata value for `p`, while uncertain for Thread 1 in epoch 2, will be clarified by the end of epoch 3 after each thread completes the epoch by issuing `untaint(p)`.

5.2.2 Challenge: Dataflow Analysis Does Not Preserve Timing

Both Butterfly and Chrysalis Analysis are built on a foundation of dataflow analysis, adapted to run dynamically at runtime and using a different thread execution model than the standard control flow graph. Dataflow analysis tracks flow of information without the context of where and when it flowed from. While Chrysalis Analysis required a slight deviation from pure dataflow to incorporate the happens-before arcs, an even larger deviation will be necessary for correctly tracking uncertainty. Where before it might have been sufficient to return conservative state and continue analysis, now our goal is to keep the largest precise states as possible without admitting error. This leads to complexities, especially in the context of Chrysalis Analysis.

Consider Figure 5.3(a) and Figure 5.3(b). From the perspective of subblock b_1 in both figures, the `taint(p)` reaches the end of the subblock. However, for the analysis to be as precise



(a) Taint before untaint before dereference

(b) Concurrent taint, untaint

Figure 5.3: Incorporating uncertainty into Chrysalis Analysis requires tightening the equations for `taint`, `untaint` and `uncertain` propagation. In both (a) and (b), `p` is `tainted` at the end of b_1. However, in (a) `p` should be considered `untainted` before b_3 begins execution, since b_2 executes after b_0 and before b_1; in (b) `p` should be considered `uncertain`. In standard dataflow analysis, calculating IN uses what reaches the end of blocks b_1 and b_2 without considering when it was generated or the ordering of the generates.

as possible, we must differentiate between these two cases at the input to subblock b_3: in Figure 5.3(a), `p` should be considered `untainted` whereas in Figure 5.3(b), it should be considered `uncertain`. To accomplish this precision will require further refinements to the dataflow equations to take into account the ordering information that pure dataflow cannot capture.

5.2.3 Challenge: Non-Binary Metadata Complications

Binary metadata makes performing checks straightforward. In traditional TAINTCHECK, if something is not `tainted`, then it is `untainted`. In reaching definitions, a definition either may reach a program point or it must not. Increasing the number of metadata states, from two to three[1], increases the complexity of computing the meet operation and transfer functions, as well as the complexity of communicating and storing the metadata.

Metadata Datastructures

The metadata complications are nonintuitive. As dataflow analysis-based dynamic parallel monitoring has such a large reliance on set operations such as union and intersection, particularly in

[1] or more, depending on how many types of uncertainty are tracked

calculating the SIDE-OUT, SIDE-IN, and global state update, the metadata format which at first seems natural (storing a metadata state per memory location) is less ideal than the reverse: storing the memory locations that correspond to a given state in a set. With only two metadata states, the reversal of the mapping does not overly complicate set membership queries or insertion/deletion operations. With increasing numbers of sets, however, the set membership, insertion and deletion operations become increasingly complication.

5.2.4 Challenge: Non-Identical Meet Operation and Transfer Functions

Likewise, with only two metadata states, there are only four mappings the meet operation and transfer functions had to track. With $n \geq 3$ metadata states, there are $\binom{n}{2}$ mappings. Furthermore, for lifeguards like TAINTCHECK, the meet and transfer functions are no longer equivalent. Consider the statement `*(p+offset)`. If `p` maps to `taint` and `offset` to `untaint`, the transfer function for TAINTCHECK dictates that the jump target `p+offset` is considered tainted (*e.g., Figure 5.1(b)*). On the other hand, TAINTCHECK will consider `*p` to be `uncertain` if the wings show two different taint values possible for `p` (*e.g., Figure 5.1(c)*), because the *meet* of `taint` and `untaint` is `uncertain`, whereas the transfer function $f_{\text{transfer}}(\texttt{taint}, \texttt{uncertain}) = \texttt{taint}$.

TaintCheck Predecessor Resolution Complications

Both Butterfly Analysis and Chrysalis Analysis were able to leverage the fact that the meet and transfer functions were identical when resolving the taint status of predecessor. If the predecessor relationship were a graph, Butterfly and Chrysalis Analysis could perform a breadth first search, and easily short circuit if a `taint` value were ever encountered: any encounter of `taint` always mapped the destination address to `taint`. Now, the analysis must still perform a meet of the different metadata values it sees in the wings, but then feed those values into a transfer function whenever it is analyzing a statement of the form `a := b + c`, which implies a depth first

search where meet and transfer functions are carefully applied at the right stages.

5.2.5 Challenge: State Computations Increasingly Complicated

While Chrysalis Analysis added an element of increased complexity to local state computations, adding uncertainty layers yet more complexity. First, for both Butterfly Analysis and Chrysalis Analysis, if an address x was considered tainted at the beginning of the subblock, then some predecessor believed that x was potentially tainted (others may have disagreed, but at least one agreed). Now, two predecessors can each believe x is in a precise state, where one predecessor believes x is tainted and the other believes x is untainted. Instead of using either of those states, the local state will have to map x to uncertain. While this example shows the complication for LSOS, an analogous situation arises for updating the SOS.

5.3 Leveraging Uncertainty

Separating true errors from potential errors is not the only benefit of incorporating uncertainty into dataflow-analysis based dynamic parallel monitoring. By being able to differentiate between true errors and cases where the analysis lacks sufficient information to make a precise judgment, we can attempt to dynamically increase the information available to the analysis and hopefully yield a precise judgment. Furthermore, explicitly tracking uncertainty enables dynamic decisions between faster and slower paths; if a fast but conservative pass leads to an uncertain result, a slower but more precise result pass can be rerun. Depending on the frequency, these dynamic adaptations give us the hope of improving both precision and performance.

5.3.1 One Dynamic Adaptation: Dynamically Adjusting Epoch Sizes To Balance Performance and Precision

One way to react to uncertainty is to dynamically reshape epoch boundaries. Recall from Chapter 2 that epoch size is bounded from below but not from above; it needs to be large enough to account for buffering in the system (*e.g.,* reorder buffer, store buffer, maximum memory access latency) but no upper bound is specified. With Butterfly Analysis, we showed that larger epoch sizes corresponded to better performance but worse precision than smaller epoch sizes; dynamically adapting epoch sizes offers the chance to achieve precision equivalent to smaller epochs with performance similar to larger epochs.

There is one big challenge to dynamically adapting epoch sizes; epoch boundaries are emitted on the application side based on the number of retired application instructions. By the time the lifeguard encounters an uncertain result where it would desire more ordering information, it is too late to extract that information from the application trace; uncertainty is not encountered until the second pass of analysis, so at least one more epoch of instructions (and potentially more) have already been retired by the application, and the boundaries of where it is safe to subdivide an epoch are unclear to the lifeguard.

Insight: Always emit small epochs, frequently ignore heartbeats

The insight which enables dynamic epoch resizing is simple: just as there is no ceiling to epoch size, there also is no requirement that a lifeguard thread honor each heartbeat it encounters, *so long as all lifeguard threads elide the same heartbeats*. In essence, only honoring odd heartbeats, or heartbeats that are multiples of 7, is simple making each individual epoch that much larger.

Armed with this insight, the path to enabling dynamic epoch resizing is clear: always emit heartbeats sized for small epochs on the application side, but ignore many of these epoch boundaries on the lifeguard side. When a lifeguard thread encounters a potential error that it requires

more information to resolve, coordinate a rollback to the smaller epochs (whose boundaries are already correctly in the log) and restart analysis; the goal of this work is to recover the precision of the smaller epoch size with the performance of the larger (simulated) epoch size.

5.4 Reaching Definitions

Both Butterfly Analysis and Chrysalis Analysis are examples of *dataflow-analysis-based dynamic parallel monitoring*: their theoretical formulations are adaptations of (traditionally static) dataflow analysis to a dynamic parallel monitoring setting. As such, the addition of an `uncertain` state to Butterfly Analysis and Chrysalis Analysis involves adapting the dataflow equations. For our purposes, we will begin with calculating reaching definitions, a canonical dataflow analysis, and show how to add an `uncertain` state to the metadata lattice.

It may be slightly more natural to consider using constant propagation as our model rather than reaching definitions; it would be equivalent to an instruction yielding Not-A-Constant(NAC) in constant propagation. While the uncertainty formulations of Butterfly Analysis and Chrysalis Analysis could be expressed in terms of constant propagation, we have continued to use reaching definitions for ease of comparison to the original formulations. However, note that moving forward, both GEN and KILL are fully precise ("must") states, as compared to prior formulations in Chapters 3 and 4.

5.4.1 Butterfly Analysis: Incorporating Uncertainty

To provide full generality for analyses such as TAINTCHECK, we will assume the existence of a magic instruction whose effects are uncertain; a definition d may or may not be generated This eases proper modeling of *inheritance* in TAINTCHECK.

I will assume that we represent the dataflow primitives GEN and KILL as $\mathcal{G}$ and $\mathcal{K}$, respectively. I will represent uncertainty("maybe-generated") as $\mathcal{M}$.

Recall the structure of Butterfly Analysis, as represented in Figure 4.2(a). Butterfly Analysis divides thread execution into *epochs*, sized to ensure that instructions in *non-adjacent epochs* (or epochs that do not share a boundary) do not interleave. This reduces the window of concurrency Butterfly Analysis must reason about to a three-epoch window. Within that window, a *block* is specified by a thread-epoch pair. The **head**, **body**, **tail** and **wings** are illustrated in Figure 4.2(a).

5.4.2 Gen and Kill equations

We will use $\mathcal{M}$ to represent the set of definitions in an uncertain (or maybe-generated) state, to reduce confusion with untaint and union.

Instruction-Level

. If an instruction generates d, kills d or marks d uncertain, then d is respectively in $\mathcal{G}_{l,t,i}$, $\mathcal{K}_{l,t,i}$ or $\mathcal{M}_{l,t,i}$.

Block-level

We use the standard dataflow representations for generation/kill/uncertainty across a set of consecutive instructions (l,t,i) through (l,t,j) modified to incorporate uncertainty.

$$\mathcal{G}_{l,t,(i,i)} = \mathcal{G}_{l,t,i} \qquad \forall j > i,\ \ \mathcal{G}_{l,t,(i,j)} = \mathcal{G}_{l,t,j} \bigcup (\mathcal{G}_{l,t,(i,j-1)} - (\mathcal{K}_{l,t,j} \cup \mathcal{M}_{l,t,j})).$$

$$\mathcal{K}_{l,t,(i,i)} = \mathcal{K}_{l,t,i} \qquad \forall j > i,\ \ \mathcal{K}_{l,t,(i,j)} = \mathcal{K}_{l,t,j} \bigcup (\mathcal{K}_{l,t,(i,j-1)} - (\mathcal{G}_{l,t,j} \cup \mathcal{M}_{l,t,j})).$$

$$\mathcal{M}_{l,t,(i,i)} = \mathcal{M}_{l,t,i} \qquad \forall j > i,\ \ \mathcal{M}_{l,t,(i,j)} = \mathcal{M}_{l,t,j} \bigcup (\mathcal{M}_{l,t,(i,j-1)} - (\mathcal{K}_{l,t,j} \cup \mathcal{G}_{l,t,j})).$$

Recall that a block in Butterfly Analysis represents a sequence of consecutive instructions belonging to epoch l and thread t, represented as block (l,t). We represent generation/kill/marking uncertain across a block (l,t) which contains $n+1$ instructions as:

$$\mathcal{G}_{l,t} = \mathcal{G}_{l,t,(0,n)}$$

$$\mathcal{K}_{l,t} = \mathcal{K}_{l,t,(0,n)}$$

$$\mathcal{M}_{l,t} = \mathcal{M}_{l,t,(0,n)}$$

Side-Out and Side-In

We will use the intermediate sets $\text{ALL}^{\mathcal{G}}_{l,t}$, $\text{ALL}^{\mathcal{K}}_{l,t}$ and $\text{ALL}^{\mathcal{M}}_{l,t}$ to compute the side-in and side-out.

$$\text{ALL}^{\mathcal{G}}_{l,t} = \bigcup_i \mathcal{G}_{l,t,i}$$
$$\text{ALL}^{\mathcal{K}}_{l,t} = \bigcup_i \mathcal{K}_{l,t,i}$$
$$\text{ALL}^{\mathcal{M}}_{l,t} = \bigcup_i \mathcal{M}_{l,t,i}$$

Given these intermediate sets, we can calculate the side-out:

$$\text{GSO}_{l,t} = \text{ALL}^{\mathcal{G}}_{l,t} - (\text{ALL}^{\mathcal{K}}_{l,t} \cup \text{ALL}^{\mathcal{M}}_{l,t})$$
$$\text{KSO}_{l,t} = \text{ALL}^{\mathcal{K}}_{l,t} - (\text{ALL}^{\mathcal{G}}_{l,t} \cup \text{ALL}^{\mathcal{M}}_{l,t})$$
$$\text{MSO}_{l,t} = \text{ALL}^{\mathcal{M}}_{l,t} \cup (\text{ALL}^{\mathcal{G}}_{l,t} \cap \text{ALL}^{\mathcal{K}}_{l,t})$$

This follows from the behavior of the wings. For example, any block which both generates and kills a definition d will create an uncertain result in the minds of any block for which it is in the wings, thus those results are moved from GSO to MSO. We can use a similar trick of intermediate sets $\text{WING}^{\mathcal{G}}_{l,t}$, $\text{WING}^{\mathcal{K}}_{l,t}$ and $\text{WING}^{\mathcal{M}}_{l,t}$ to calculate the side-in:

$$\text{WING}^{\mathcal{G}}_{l,t} = \bigcup_{\{t' \neq t\}} \bigcup_{\{l' | l-1 \leq l' \leq l+1\}} \text{GSO}_{l',t'}$$
$$\text{WING}^{\mathcal{K}}_{l,t} = \bigcup_{\{t' \neq t\}} \bigcup_{\{l' | l-1 \leq l' \leq l+1\}} \text{KSO}_{l',t'}$$
$$\text{WING}^{\mathcal{M}}_{l,t} = \bigcup_{\{t' \neq t\}} \bigcup_{\{l' | l-1 \leq l' \leq l+1\}} \text{MSO}_{l',t'}$$

$$\text{GSI}_{l,t} = \text{WING}^{\mathcal{G}}_{l,t} - (\text{WING}^{\mathcal{K}}_{l,t} \cup \text{WING}^{\mathcal{M}}_{l,t})$$
$$\text{KSI}_{l,t} = \text{WING}^{\mathcal{K}}_{l,t} - (\text{WING}^{\mathcal{G}}_{l,t} \cup \text{WING}^{\mathcal{M}}_{l,t})$$
$$\text{MSI}_{l,t} = \text{WING}^{\mathcal{M}}_{l,t} \cup (\text{WING}^{\mathcal{G}}_{l,t} \cap \text{WING}^{\mathcal{K}}_{l,t})$$

5.4.3 Incorporating Uncertainty Into Strongly Ordered State

Summarizing an epoch

We begin with the must-kill and must-generate equations to summarize an epoch, as we are trying to represent generate and kill as precisely as possible. The inclusion of the prior epoch to account for potential interference is extended from Butterfly Analysis. We use standard composition of transfer functions (*e.g.*,$\mathcal{G}_{(l-1,l),t} = \mathcal{G}_{l,t} \bigcup (\mathcal{G}_{l-1,t} - (\mathcal{K}_{l,t} \cup \mathcal{M}_{l,t}))$ and symmetrically for $\mathcal{M}_{(l-1,l),t}$ and $\mathcal{K}_{(l-1,l),t}$).

$$\mathcal{K}_l = \bigcup_t \left(\mathcal{K}_{l,t} - \left(\bigcup_{t' \neq t} [\mathcal{G}_{(l-1,l),t'} \cup \mathcal{M}_{(l-1,l),t'}] \right) \right)$$

$$\mathcal{G}_l = \bigcup_t \left(\mathcal{G}_{l,t} - \left(\bigcup_{t' \neq t} [\mathcal{K}_{(l-1,l),t'} \cup \mathcal{M}_{(l-1,l),t'}] \right) \right)$$

Intuitively, a definition d is killed in an epoch if at least one subblock (l,t) kills d and for all concurrent subblocks in epochs $[l-1,l]$, none have a net effect of definitely generating or possibly generating d. $\mathcal{G}_l$ and $\mathcal{K}_l$ are symmetric because we are looking to achieve equivalent precision for these two sets in this work.

Another way of viewing the definition of $\mathcal{K}_l$ (symmetrically, $\mathcal{G}_l$) is by observing when definitions $d \notin \mathcal{K}_l$. By this definition, $\exists t$ such that $x \notin \mathcal{K}_{l,t} \wedge (x \in \mathcal{M}_{(l-1,l),t} \vee x \in \mathcal{G}_{(l-1,l),t})$ then $x \notin \mathcal{K}_l$.

If, instead, we wanted to represent may-kill and may-generate, we would represent may-kill as $\bigcup_t \mathcal{K}_{l,t}$ and may-generate as $\bigcup_t \mathcal{G}_{l,t}$. In fact, for Butterfly Analysis, we previously used may-generate and must-kill.

To accurately capture the uncertainty within an epoch, we need to separate out and account for separate sources of uncertainty. First, the difference between may-kill and must-kill, or may-generate and must-generate, captures the fact that orderings may exist in which two different outcomes are possible, when looking at all instructions in the epoch. Second, we must capture

any organic uncertainty (e.g., a check within the wings returns "uncertain", which persists to the end of some block (l, t).

$$\mathcal{M}_l = (\bigcup_t \mathcal{M}_{l,t}) \cup \left(\bigcup_t \bigcup_{\{t' | t \neq t'\}} \left(\left[K_{l,t} \cap (\mathcal{G}_{(l-1,l),t'} \cup \mathcal{M}_{(l-1,l),t'}) \right] \cup \left[G_{l,t} \cap (\mathcal{K}_{(l-1,l),t'} \cup \mathcal{M}_{(l-1,l),t'}) \right] \right) \right)$$

Mutual Exclusion

By construction, we have that $\mathcal{M}_{l,t}$, $\mathcal{G}_{l,t}$ and $\mathcal{K}_{l,t}$ are pairwise mutually exclusive sets. Now we want to show that the epoch summarizations $\mathcal{G}_l, \mathcal{M}_l, \mathcal{K}_l$ are again pairwise mutually exclusive. We do this by showing that $x \in K_l \Rightarrow x \notin \mathcal{M}_l \wedge x \notin \mathcal{G}_l$ and then that $x \in \mathcal{M}_l \Rightarrow x \notin \mathcal{K}_l$. As our generate and kill formulations are now symmetric, we get two other implications by swapping the roles of $\mathcal{G}$ and $\mathcal{K}$, completing the proof.

Lemma 20. *If $x \in \mathcal{K}_l$ then $x \notin \mathcal{M}_l$ and $x \notin \mathcal{G}_l$.*

Proof. For both proofs, we will use $x \notin \mathcal{G}_{(l-1,l),t}$ implies $x \notin \mathcal{G}_{l,t}$ (and likewise for all the compositions over 2 epochs). This follows by construction.

First, we will show $x \in \mathcal{K}_l \Rightarrow x \notin \mathcal{G}_l$**.** If $x \in \mathcal{K}_l$, that implies there exists t such that $x \in \mathcal{K}_{l,t}$ and $\forall t' \neq t$, $x \notin \mathcal{G}_{(l-1,l),t'} \wedge x \notin \mathcal{M}_{(l-1,l),t'}$; it follows that $x \notin \mathcal{G}_{l,t'} \wedge x \notin \mathcal{M}_{l,t'}$. This follows by definition of $\mathcal{K}_l$. Furthermore, as $\mathcal{K}_{l,t} \cap \mathcal{G}_{l,t} = \varnothing$, $x \notin \mathcal{G}_{l,t}$. So $\forall t, x \notin \mathcal{G}_{l,t}$ and therefore $x \notin \mathcal{G}_l$ by definition of $\mathcal{G}_l$.

Now, we will show that $x \in \mathcal{K}_l \Rightarrow x \notin \mathcal{M}_l$**.** If $x \in \mathcal{K}_l$, then again there exists t such that $x \in \mathcal{K}_{l,t}$ and $\forall t' \neq t$, $x \notin \mathcal{G}_{(l-1,l),t'} \wedge x \notin \mathcal{M}_{(l-1,l),t'}$; it follows that $x \notin \mathcal{G}_{l,t'} \wedge x \notin \mathcal{M}_{l,t'}$. As before, $x \notin \mathcal{M}_{l,t}$ as $\mathcal{K}_{l,t} \cap \mathcal{M}_{l,t} = \varnothing$ and likewise $x \notin \mathcal{G}_{l,t}$. So $x \notin \bigcup_t \mathcal{M}_{l,t}$, as I have shown $x \notin \mathcal{M}_{l,t'} \forall t'$.

Furthermore: I have shown $\forall t, x \notin \mathcal{G}_{l,t}$ so $x \notin \bigcup_t \bigcup_{\{t' | t \neq t'\}} \left[G_{l,t} \cap (\mathcal{K}_{(l-1,l),t'} \cup \mathcal{M}_{(l-1,l),t'}) \right]$.

Finally: As $\forall t', x \notin \mathcal{G}_{(l-1,l),t'} \cup \mathcal{M}_{(l-1,l),t'}$ (including $t' = t$), then for any t, $x \notin \bigcup_{t' \neq t} (\mathcal{G}_{(l-1,l),t'} \cup \mathcal{M}_{(l-1,l),t'})$. Thus, $x \notin \bigcup_t \bigcup_{\{t' | t \neq t'\}} \left[K_{l,t} \cap (\mathcal{G}_{(l-1,l),t'} \cup \mathcal{M}_{(l-1,l),t'}) \right]$. So $x \notin \mathcal{M}_l$. □

The following lemma is presented without proof; it follows by symmetry of $\mathcal{G}_l$ and $\mathcal{K}_l$.

Lemma 21. *If $x \in \mathcal{G}_l$ then $x \notin \mathcal{M}_l$ and $x \notin \mathcal{K}_l$.*

For a full proof of mutual exclusivity, it remains to show that $x \in \mathcal{M}_l$ implies $x \notin \mathcal{K}_l$ (and by symmetry, $x \notin \mathcal{G}_l$).

Lemma 22. *If $x \in \mathcal{M}_l$ then $x \notin \mathcal{K}_l$ and $x \notin \mathcal{G}_l$.*

Proof by cases. It will suffice to show only that $x \in \mathcal{M}_l \Rightarrow x \notin \mathcal{K}_l$; we get $x \notin \mathcal{G}_l$ by symmetry.

If $x \in \mathcal{M}_l$, then there $\exists t$ such that at least one of the following cases holds:

(1) $x \in \mathcal{M}_{l,t}$ *or*

(2) $x \in \bigcup_{t' \neq t} \mathcal{K}_{l,t} \cap (\mathcal{G}_{(l-1,l),t'} \cup \mathcal{M}_{(l-1,l),t'})$ *or*

(3) $x \in \bigcup_{t' \neq t} \mathcal{G}_{l,t} \cap (\mathcal{K}_{(l-1,l),t'} \cup \mathcal{M}_{(l-1,l),t'})$

I will show $x \in \mathcal{M}_l$ implies $x \notin K_l$ for each case.

Case 1: $x\mathcal{M}_{l,t}$ implies $x \notin \mathcal{K}_{l,t}$ and $x \notin \mathcal{G}_{l,t}$. Furthermore, $x \in \mathcal{M}_{(l-1,l),t}$. Thus, $\forall t' \neq t, x \notin \mathcal{K}_{l,t'} - (\bigcup_{t'' \neq t'}(\mathcal{G}_{(l-1,l),t''} \cup \mathcal{M}_{(l-1,l),t''}))$. Thus $x \notin \mathcal{K}_l$.

Case 2: If $x \in \bigcup_{t' \neq t} \mathcal{K}_{l,t} \cap (\mathcal{G}_{(l-1,l),t'} \cup \mathcal{M}_{(l-1,l),t'})$ then there must $\exists t' \neq t$ such that $x \in \mathcal{K}_{l,t}$ and **(a)** $x \in \mathcal{G}_{(l-1,l),t'}$ or **(b)** $x \in \mathcal{M}_{(l-1,l),t'}$.

First, we will show that for all $t'' \neq t'$, $x \notin \mathcal{K}_{l,t''} - \left(\bigcup_{\hat{t} \neq t''}[\mathcal{G}_{(l-1,l),\hat{t}} \cup \mathcal{M}_{(l-1,l),\hat{t}}]\right)$

(a) If $x \in \mathcal{G}_{(l-1,l),t'}$ then $\forall t'' \neq t', x \notin \mathcal{K}_{l,t''} - \mathcal{G}_{(l-1,l),t'} \Rightarrow x \notin \mathcal{K}_{l,t''} - \left(\bigcup_{\hat{t} \neq t''}[\mathcal{G}_{(l-1,l),\hat{t}} \cup \mathcal{M}_{(l-1,l),\hat{t}}]\right)$.

Likewise, if $x \in \mathcal{M}_{(l-1,l),t'}$, then $\forall t'' \neq t', x \notin \mathcal{K}_{l,t''} - \mathcal{M}_{(l-1,l),t'} \Rightarrow x \notin \mathcal{K}_{l,t''} - \left(\bigcup_{\hat{t} \neq t''}[\mathcal{G}_{(l-1,l),\hat{t}} \cup \mathcal{M}_{(l-1,l),\hat{t}}]\right)$.

Now we must show this also holds for $t'' = t'$, namely, $x \notin \mathcal{K}_{l,t'} - \left(\bigcup_{\hat{t} \neq t'}[\mathcal{G}_{(l-1,l),\hat{t}} \cup \mathcal{M}_{(l-1,l),\hat{t}}]\right)$.

(b) If $x \in \mathcal{G}_{(l-1,l),t'}$ then $x \notin \mathcal{K}_{l,t'}$; and likewise **(b)** if $x \in \mathcal{M}_{(l-1,l),t'}$, $x \notin \mathcal{K}_{l,t'}$. Therefore, applying the formula for $\mathcal{K}_l$, $x \notin \mathcal{K}_l$.

Case 3: If $x \in \mathcal{G}_{l,t}$, then $\forall t'' \neq t, x \notin \mathcal{K}_{l,t''} - \mathcal{G}_{(l,l-1),t}$. Also, if $x \in \mathcal{G}_{l,t}$ then $x \notin \mathcal{K}_{l,t}$ so $x \notin \mathcal{K}_{l,t} - A$ for any A. Thus, $x \notin \mathcal{K}_l$.

By symmetry, we get that $x \in \mathcal{M}_l$ implies $x \notin \mathcal{G}_l$. □

Combining Lemmas 20,21 and 22 yields that all three sets have pairwise empty intersections.

Invariants for Epoch Summaries

Lemma 23. *If $d \in \mathcal{K}_l$ then for all valid orderings O of instructions in epochs $[l-1, l]$, $d \in \mathcal{K}(O)$.*

Proof. If $d \in \mathcal{K}_l$, then there exists thread t such that $d \in \mathcal{K}_{l,t} - \left(\bigcup_{t' \neq t}[\mathcal{G}_{(l-1,l),t'} \cup M_{(l-1,l),t'}]\right)$. Let (l, t, i) be the last instruction in block (l, t) to kill d. Consider any valid ordering O of instructions in epochs $[l-1, l]$, and let O' be the suffix of O beginning with (l, t, i). It can only be followed by later instructions in block (l, t) or by other threads' instructions in epochs $[l-1, l]$.

Since $d \notin \left(\bigcup_{t' \neq t}[\mathcal{G}_{(l-1,l),t'} \cup M_{(l-1,l),t'}]\right) \forall t' \neq t$, then in particular any gen or "uncertain" by a thread t' must be followed in the same thread by a subsequent kill which is the final "operation" on d as $d \notin \mathcal{G}_{(l-1,l),t'}$ and $d \notin \mathcal{M}_{(l-1,l),t'}$. Thus, any generate or uncertain following instruction (l, t, i) is itself followed by a kill, meaning $d \in \mathcal{K}(O')$ and thus that $d \in \mathcal{K}(O)$. □

By a symmetrical proof, the following lemma also holds:

Lemma 24. *If $d \in \mathcal{G}_l$ then for all valid orderings O of instructions in epochs $[l-1, l]$, $d \in \mathcal{G}(O)$.*

The following invariant holds for the uncertain state:

Lemma 25. *If at least one of the following three conditions holds:*

(1) $\exists$ valid ordering O of instructions in epoch l such that $d \in \mathcal{M}(O)$ or

(2) $\exists$ a valid ordering O of instructions in epochs $[l-1, l]$ and $\exists$ thread t such that $d \in \mathcal{G}_{l,t}$ and $d \notin \mathcal{G}(O)$ or

(3) $\exists$ a valid ordering O of instructions in epochs $[l-1, l]$ and $\exists$ thread t such that $d \in \mathcal{K}_{l,t}$ and $d \notin \mathcal{K}(O)$

then $d \in \mathcal{M}_l$.

Proof by cases. I will show how if any of the three conditions holds, then $d \in \mathcal{M}_l$.

Case 1: If $\exists$ valid ordering O of instructions in epoch l such that $d \in \mathcal{M}(O)$, then there must exist t such that $d \in \mathcal{M}_{l,t}$. This implies $d \in \bigcup_t \mathcal{M}_{l,t}$ which further implies $d \in \mathcal{M}_l$.

Case 2: Assume $\exists$ valid ordering O of instructions in epochs $[l-1, l]$ such that $d \notin \mathcal{G}(O)$ and $\exists t$ such that $d \in \mathcal{G}_{l,t}$. Let (l, t, i) be the last instruction in (l, t) to generate d. Consider the suffix of O' of O beginning with (l, t, i). It must end with $d \in \mathcal{M}(O')$ or $d \in \mathcal{K}(O')$. [This follows by $d \notin \mathcal{G}(O)$; if $d \in \mathcal{G}(O')$ then $d \in \mathcal{G}(O)$ and since the first instruction in O' generates d, something later in O' must reverse that effect.] So there must be at least one other thread that either considers d in the uncertain state or kills d and does not later generate d. Consider the last such (kill or "uncertain") operation in O'. Let the thread performing the operation be t'. Then $d \in \mathcal{K}_{(l-1,l),t'} \cup M_{(l-1,l),t'}$ and $d \in \mathcal{G}_{l,t}$ so $d \in \mathcal{G}_{l,t} \cap (\mathcal{K}_{(l-1,l),t'} \cup M_{(l-1,l),t'})$, and applying properties of union, $d \in \mathcal{M}_l$.

Case 3: Follows by symmetry with case (2).

□

Strongly Ordered State Equations

There will now be three flavors of SOS. This is a change from Butterfly Analysis, where we only specifically tracked generated metadata and implicitly tracked the other state. The equations that follow hold for $l \geq 2$; for $l = 0$ or $l = 1$, they are all identically empty, as in Butterfly Analysis.

$\text{SOS}_l^{\mathcal{G}} = \mathcal{G}_{l-2} \cup (\text{SOS}_{l-1}^{\mathcal{G}} - (\mathcal{K}_{l-2} \cup \mathcal{M}_{l-2}))$.

$\text{SOS}_l^{\mathcal{K}} = \mathcal{K}_{l-2} \cup (\text{SOS}_{l-1}^{\mathcal{K}} - (\mathcal{G}_{l-2} \cup \mathcal{M}_{l-2}))$.

$\text{SOS}_l^{\mathcal{M}} = \mathcal{M}_{l-2} \cup (\text{SOS}_{l-1}^{\mathcal{M}} - (\mathcal{G}_{l-2} \cup \mathcal{K}_{l-2}))$.

Invariants for Strongly Ordered State Formulations

Lemma 26. *If one of the following is true:*

(1) $\exists$ *valid ordering* O *of instructions in epochs* $[0, l-2]$ *such that* $d \in \mathcal{M}(O)$ OR

(2) $\exists$ *valid ordering* O *of instructions in epochs* $[0, l-2]$ *such that* $d \notin \mathcal{G}(O)$ *and* $\exists$ *thread* t *such that* $d \in \mathcal{G}_{l-2,t}$ OR

(3) $\exists$ *valid ordering* O *of instructions in epochs* $[0, l-2]$ *such that* $d \notin \mathcal{K}(O)$ *and* $\exists$ *thread* t *such that* $d \in \mathcal{K}_{l-2,t}$ OR

(4) (Propagation) $\exists\, l' < l-2$ *such that cases (1), (2), or (3) applies to instructions in epochs* $[0, l']$ *and* $\forall l''$ *such that* $l-2 \geq l'' > l'$, $d \notin (\mathcal{G}_{l''} \cup \mathcal{K}_{l''})$

then $d \in SOS_l^{\mathcal{M}}$.

Proof by cases (using induction). We will consider each case in turn.

Case 1: Base case: $l = 2$. If there exists a valid ordering O of instructions in epoch 0, then there is some last instruction $(0, t, i)$ which marks d as uncertain where no later instruction in O generates or kills d. Then $d \in \mathcal{M}_{0,t} \Rightarrow d \in \bigcup_t M_{0,t} \Rightarrow d \in \mathcal{M}_0 \Rightarrow d \in \text{SOS}_l^{\mathcal{M}}$.

Inductive hypothesis: assume condition (1) of the lemma is true for $l < k$, and then show it holds for $l = k$.

Inductive step: Let (l', t, i) be the last instruction in O such that $d \in \mathcal{M}_{l',t,i}$. There are three subcases, where $l' = l-2$, $l' = l-3$ or $l' < l-3$.

case(a): $l' = l-2$. Then $d \in \mathcal{M}_{l-2,t}$ which implies $d \in \bigcup_t \mathcal{M}_{l-2,t}$ which implies $d \in \mathcal{M}_{l-2}$, so $d \in \text{SOS}_l^{\mathcal{M}}$.

case(b): $l' = l-3$: Let $O_{[0,l-3]}$ be the restriction of O to epochs $[0, l-3]$, then $d \in \mathcal{M}(O_{[0,l-3]})$ and by the inductive hypothesis, $d \in \text{SOS}_{l-1}^{\mathcal{M}}$. Let $O_{[l-3,l-2]}$ be the restriction of O to instructions in epochs $[l-3, l-2]$. This must include (l', t, i), which will still be the last instruction such that $d \in \mathcal{M}_{l',t,i}$. Since the relative ordering of instructions is preserved, $d \in \mathcal{M}(O_{[l-3,l-2]})$. By the contrapositives of Lemmas 24 and 23, $d \notin \mathcal{K}_{l-2} \wedge d \notin \mathcal{G}_{l-2}$, so $d \in \text{SOS}_{l-1}^{\mathcal{M}} - (\mathcal{K}_{l-2} \cup \mathcal{G}_{l-2})$ and therefore $d \in \text{SOS}_l^{\mathcal{M}}$.

case(c): $l' < l-3$: Proceeds similarly to case (b). Let $O_{[0,l-3]}$ be the restriction of O to epochs $[0, l-3]$. As in case (b), $d \in \mathcal{M}(O_{[0,l-3]})$ and applying the inductive hypothesis, $d \in \text{SOS}_{l-1}^{\mathcal{M}}$. As (l', t, i) is the last instruction in O such that $d \in \mathcal{M}_{l',t,i}$, it cannot be the case that any instruction in epoch $l-2$ would

kill or generate d, as that would by necessity be ordered after (non-adjacent epochs) and no later instruction in O has marks d as uncertain, which would contradict $d \in \mathcal{M}(O)$. If no instruction in $l-2$ kills or generates d, then $d \notin \mathcal{K}_{l-2} \wedge d \notin \mathcal{G}_{l-2}$ so $d \in \text{SOS}_{l-1}^{\mathcal{M}} - (\mathcal{K}_{l-2} \cup \mathcal{G}_{l-2}) \Rightarrow d \in \text{SOS}_{l}^{\mathcal{M}}$.

Case 2: Base Case: $l = 2$. If $d \in \mathcal{G}_{0,t}$ but $\exists$ valid ordering O of instructions in epoch 0 such that $d \notin \mathcal{G}(O)$, then there must be some other thread t' which either kills d or marks it as uncertain, so $d \in (\mathcal{M}_{0,t'} \cup \mathcal{K}_{0,t'})$ and thus $d \in \mathcal{G}_{0,t} \cap (\mathcal{M}_{0,t'} \cup \mathcal{K}_{0,t'})$, so $d \in \mathcal{M}_0$ and thus $d \in \text{SOS}_2^{\mathcal{M}}$.

Inductive hypothesis: we assume condition (2) of the lemma is true for $l < k$ and show it is true for $l = k$.

Inductive Step: The proof is a straightforward extension of the proof of Lemma 25, case (2). Here, the valid ordering O spans epochs $[0, l-2]$, but we can still examine the suffix beginning with the last generate of d at instruction $(l-2, t, i)$. Such an instruction is guaranteed to exist because $d \in \mathcal{G}_{l-2,t}$. The only instructions that can follow $(l-2, t, i)$ in O are again limited to those in epochs $l-3$ or $l-2$. Let O' be the suffix of O beginning immediately after instruction $(l-2, t, i)$. By the argument in Lemma 25, case (2), $d \in \mathcal{M}_{l-2}$ which implies $d \in \text{SOS}_{l}^{\mathcal{M}}$.

Case 3: Follows by symmetry from Case 2.

Case 4: Suppose any of cases (1), (2) or (3) apply to instructions in epochs $[0, l']$. Then, as we have shown in each of the three prior cases, $d \in \text{SOS}_{l'+2}^{\mathcal{M}}$. Applying the fact that $d \notin \mathcal{G}_{l'+1} \cup \mathcal{K}_{l'+1}$, $d \in \text{SOS}_{l'+2}^{\mathcal{M}} - (\mathcal{G}_{l'+1} \cup \mathcal{K}_{l'+1})$ which implies $d \in \text{SOS}_{l'+3}^{\mathcal{M}}$; this holds through $d \in \text{SOS}_{l-1}^{\mathcal{M}} - (\mathcal{G}_{l-2} \cup \mathcal{K}_{l-2})$, which shows that $d \in \text{SOS}_{l}^{\mathcal{M}}$. □

Lemma 27. *If $d \in SOS_l^{\mathcal{G}}$ then $\forall$ valid orderings O of instructions in epochs $[0, l-2]$, $d \in \mathcal{G}(O)$.*

Proof by induction. Base Case: $l = 2$. Then $d \in \text{SOS}_2^{\mathcal{G}} = \mathcal{G}_0$ according to our equations. We can apply Lemma 24. [One caveat: no epoch before epoch 0 so we ignore the non-existent "prior" epoch].

We will assume the lemma is true for $l < k$, and show it holds for $l = k$. There are two cases. If $d \in \text{SOS}_l^{\mathcal{G}}$, then $d \in \mathcal{G}_{l-2}$ or $d \in \text{SOS}_{l-1}^{\mathcal{G}} - (\mathcal{K}_{l-2} \cup \mathcal{M}_{l-2})$.

Case 1: If $d \in \mathcal{G}_{l-2}$ then $\exists t$ such that $d \in \mathcal{G}_{l-2,t}$ and $\forall t' \neq t$, $d \notin \mathcal{K}_{(l-3,l-2),t'} \wedge d \notin \mathcal{M}_{(l-3,l-2),t'}$. Let $(l-2,t,i)$ be the last instruction in $(l-2,t)$ to generate d. Consider any arbitrary valid ordering O, and let O' be the suffix of O beginning with instruction $(l-2,t,i)$. All instructions in O' must be from epochs $[l-3,l-2]$ (as instructions in $l' < l-3$ executed strictly before any instruction in epoch $l-2$). Let $O'_{t'}$ be the restriction of O' to any $t' \neq t$. It follows from $d \notin \mathcal{M}_{(l-3,l-2),t'} \wedge d \notin \mathcal{K}_{(l-3,l-2),t'}$ that for any $t' \neq t$ that $d \notin \mathcal{M}(O'_{t'}) \wedge d \notin \mathcal{K}(O'_{t'})$. In addition, since $d \in \mathcal{G}_{l-2,t}$ and $(l-2,t,i)$ is the last gen of d in the block, no later instructions in $(l-2,t)$ generate or mark as uncertain d. Combining all these interleavings will not change the final status of d; thus, $d \in \mathcal{G}(O')$ and therefore $d \in \mathcal{G}(O)$.

Case 2: If $d \in \text{SOS}_{l-1}^{\mathcal{G}} - (\mathcal{K}_{l-2} \cup \mathcal{M}_{l-2})$ then in particular $d \in \text{SOS}_{l-1}^{\mathcal{G}}$ and $d \notin \mathcal{K}_{l-2} \cup \mathcal{M}_{l-2}$. Consider any arbitrary valid ordering O of instructions in epochs $[0,l-2]$. Let $O_{[0,l-3]}$ be O restricted to instructions in epochs $[0,l-3]$. By the inductive hypothesis, $d \in \mathcal{G}(O_{[0,l-3]})$. There must exist instruction (l',t,i) in $O_{[0,l-3]}$ which is the last generate of d. Let O' be the suffix of O beginning with instruction (l',t,i). We need to show that $d \in \mathcal{G}(O')$, which implies that $d \in \mathcal{G}(O)$.

Let $O'_{[0,l-3]}$ be the restriction of O' to instructions in $[0,l-3]$. Then, $d \in \mathcal{G}(O'_{[0,l-3]})$; this follows from the inductive hypothesis, as $d \in \mathcal{G}(O_{[0,l-3]})$ and $O'_{[0,l-3]}$ is simply the suffix of $O_{[0,l-3]}$ beginning with instruction (l',t,i). Our proof now proceeds by contradiction. We will consider if $d \notin \mathcal{G}(O')$ and obtain a contradiction.

Let O'_{l-2} be the restriction of O' to instructions in epoch $l-2$. If $d \notin \mathcal{G}(O')$, then the instructions in O'_{l-2} either killed d or marked d as uncertain (by our previous observation; no instructions in $[0,l-3]$ which occurred after (l',t,i) could have done so). If $d \in \mathcal{K}(O'_{l-2})$ then there must exist t' such that $d \in \mathcal{K}_{l-2,t'}$. By our construction, d is

in the may-kill set for epoch $l-2$. In particular, definitions in the may-kill set either belong to $\mathcal{K}_{l-2}$ or $\mathcal{M}_{l-2}$, as shown earlier. This contradicts $d \notin \mathcal{K}_{l-2} \cup \mathcal{M}_{l-2}$. Similarly, if $d \in \mathcal{M}(O'_{l-2})$ then $\exists t'$ such that $d \in \mathcal{M}_{l-2,t'}$ which implies $d \in \mathcal{M}_{l-2}$, which contradicts $d \notin \mathcal{M}_{l-2}$.

Thus, $d \in \mathcal{G}(O')$, and $d \in \mathcal{G}(O)$.

□

The following lemma is provided without proof, but it follows by symmetry with Lemma 27.

Lemma 28. *If $d \in SOS_l^{\mathcal{K}}$ then $\forall$ valid orderings O of instructions in epochs $[0, l-2]$, $d \in \mathcal{K}(O)$.*

5.4.4 Calculating local state

We can again revisit the must-kill and must-gen versus may-kill and may-gen summaries for the head. Recall that we must take into account interference of other threads in epoch $l-2$ when applying summaries from the head $(l-1,t)$ to the LSOS for block (l,t).

For block (l,t), the may-$\{$kill, gen$\}$ formulations for the head are simply $\mathcal{K}_{l-1,t}$ and $\mathcal{G}_{l-1,t}$.

The must-$\{$kill,gen$\}$ formulations for the head of block (l,t) are simply:

$$\mathcal{G}^*_{l-1,t} = \mathcal{G}_{l-1,t} - \bigcup_{t' \neq t} (\mathcal{K}_{l-2,t'} \cup \mathcal{M}_{l-2,t'})$$

$$\mathcal{K}^*_{l-1,t} = \mathcal{K}_{l-1,t} - \bigcup_{t' \neq t} (\mathcal{G}_{l-2,t'} \cup \mathcal{M}_{l-2,t'})$$

Lemma 29. *If $d \in \mathcal{K}^*_{l-1,t}$ then $\forall$ valid orderings O of instructions in epoch $l-2$ and instructions in block $(l-1,t)$, $d \in \mathcal{K}(O)$.*

Proof. If $d \in \mathcal{K}^*_{l-1,t}$ then $d \in \mathcal{K}_{l-1,t}$, so there is some instruction $(l-1,t,i)$ in block $(l-1,t)$ which kills d and is not followed by a generate or an operation that marks d as uncertain. Consider the instructions in epoch $l-2$. The instructions in block $(l-2,t)$ happen before $(l-1,t)$ (applying intra-thread dependences). This leaves the instructions which are concurrent with

those in $(l-1,t)$ (in the range we are considering), which consists of instructions in subblocks $(l-2,t')\forall t' \neq t$. But $\forall t' \neq t$, $d \notin G_{l-2,t'} \cup \mathcal{M}_{l-2,t'}$. So for any subblock $(l-2,t')$, it either also kills d or else does nothing to d. Thus, all valid orderings O of instructions in $l-2$ with instructions in block $(l-1,t)$ have $d \in \mathcal{K}(O)$. □

The following lemma is presented without proof; it follows by symmetry to Lemma 29.

Lemma 30. *If $d \in \mathcal{G}^*_{l-1,t}$ then $\forall$ valid orderings O of instructions in epoch $l-2$ and instructions in block $(l-1,t)$, $d \in \mathcal{G}(O)$.*

In representing what a block marks as uncertain, we want to include everything the head $(l-1,t)$ marked as uncertain as well as anything the head may-but-not-must have {generated, killed}.

$$\mathcal{M}^*_{l-1,t} = \mathcal{M}_{l-1,t} \cup \left(\mathcal{G}_{l-1,t} \cap (\bigcup_{t' \neq t} \mathcal{K}_{l-2,t'} \cup \mathcal{M}_{l-2,t'}) \right) \cup \left(\mathcal{K}_{l-1,t} \cap (\bigcup_{t' \neq t} \mathcal{G}_{l-2,t'} \cup \mathcal{M}_{l-2,t'}) \right)$$

Similarly, the next lemma is presented without proof but strongly resembles Lemma 25.

Lemma 31. *If at least one of the following three conditions holds:*

(1) $d \in \mathcal{M}_{l-1,t}$ OR

(2) $\exists$ *a valid ordering O of instructions in epoch $l-2$ and block $(l-1,t)$ such that $d \notin \mathcal{G}(O)$ and $d \in \mathcal{G}_{l,t}$* OR

(3) $\exists$ *a valid ordering O of instructions in epoch $l-2$ and block $(l-1,t)$ such that $d \notin \mathcal{K}(O)$ and $d \in \mathcal{K}_{l,t}$*

then $d \in \mathcal{M}^*_{l-1,t}$.

Note that $\mathcal{G}^*_{l,t} \subseteq \mathcal{G}_{l,t}$ and $\mathcal{K}^*_{l,t} \subseteq \mathcal{K}_{l,t}$, while $\mathcal{M}^*_{l,t} \supseteq \mathcal{M}_{l,t}$. Essentially, anything from the head $(l-1,t)$ which marks something as generate (killed) must not have been concurrent with another thread t' that either killed or marked uncertain (generated or marked uncertain) in epoch $l-2$, so the precise sets get smaller. When we observe such potentially concurrent and interfering events,

we add them to $\mathcal{M}^*$, so the uncertain set grows.

Then we express the LSOS formulas as:

$$\text{LSOS}^{\mathcal{G}}_{l,t} = \mathcal{G}^*_{l-1,t} \cup (\text{SOS}^{\mathcal{G}}_l - (\mathcal{K}^*_{l-1,t} \cup \mathcal{M}^*_{l-1,t}))$$

$$\text{LSOS}^{\mathcal{K}}_{l,t} = \mathcal{K}^*_{l-1,t} \cup (\text{SOS}^{\mathcal{K}}_l - (\mathcal{G}^*_{l-1,t} \cup \mathcal{M}^*_{l-1,t}))$$

$$\text{LSOS}^{\mathcal{M}}_{l,t} = \mathcal{M}^*_{l-1,t} \cup (\text{SOS}^{\mathcal{M}}_l - (\mathcal{K}^*_{l-1,t} \cup \mathcal{G}^*_{l-1,t}))$$

These formulas closely mirror how we summarized an epoch.

LSOS Invariants

Lemma 32. *If $d \in$ LSOS$^{\mathcal{G}}_{l,t}$ then $\forall$ valid orderings O of instructions in epochs $[0, l-2]$ and instructions in block $(l-1, t)$, $d \in \mathcal{G}(O)$.*

Proof. If $d \in \text{LSOS}^{\mathcal{G}}_{l,t}$, then $d \in \mathcal{G}^*_{l-1,t}$ or $d \in \text{SOS}^{\mathcal{G}}_l - (\mathcal{K}^*_{l-1,t} \cup \mathcal{M}^*_{l-1,t})$. If $d \in \mathcal{G}^*_{l-1,t}$, then $d \in \mathcal{G}_{l-1,t}$ and for all threads $t' \neq t$, $d \notin \bigcup_{t' \neq t}(\mathcal{K}_{l-2,t'} \cup \mathcal{M}_{l-2,t'})$. Within any ordering O, instructions from the head can only interleave with instructions in epoch $l-2$, and the net effect of blocks in $l-2$ is neither to generate nor mark uncertain d, so $d \in \mathcal{G}(O) \forall O$.

If $d \in \text{SOS}^{\mathcal{G}}_l - (\mathcal{K}^*_{l-1,t} \cup \mathcal{M}^*_{l-1,t})$, then d must have been generated in epoch $l-2$ or earlier. We know that $d \notin \mathcal{M}_{l-1,t}$, or we would have $d \in \mathcal{M}^*_{l-1,t}$ (contradiction). If $d \in \mathcal{K}_{l-1,t}$, then it either ends up in $\mathcal{K}^*_{l-1,t}$ or $\mathcal{M}^*_{l-1,t}$ (contradiction). So the head cannot have killed or marked d as uncertain. Thus, interleaving the instructions in the head with any ordering of $[0, l-2]$ must still generate d.

□

Lemma 33. *If $d \in$ LSOS$^{\mathcal{K}}_{l,t}$ then $\forall$ valid orderings O of instructions in epochs $[0, l-2]$ and instructions in block $(l-1, t)$, $d \in \mathcal{K}(O)$.*

Proof symmetric to Lemma 32.

Lemma 34. *If one of the following is true:*

(1) $\exists$ *valid ordering* O *of instructions in epochs* $[0, l-2]$ *and block* $(l-1, t)$ *such that* $d \in \mathcal{M}(O)$ OR

(2) $\exists$ *valid ordering* O *of instructions in epochs* $[0, l-2]$ *and block* $(l-1, t)$ *such that* $d \notin \mathcal{G}(O)$ *and* $d \in \mathcal{G}_{l-1,t}$ OR

(3) $\exists$ *valid ordering* O *of instructions in epochs* $[0, l-2]$ *and block* $(l-1, t)$ *such that* $d \notin \mathcal{K}(O)$ *and* $d \in \mathcal{K}_{l-1,t}$ OR

(4) (Propagation) One of the four conditions in Lemma 26 holds and $d \notin (\mathcal{K}^*_{l-1,t} \cup \mathcal{G}^*_{l-1,t})$.

then $d \in LSOS^{\mathcal{M}}_{l,t}$.

Proof. We will proceed by cases.

Case 1: Proceed as in Case 1 of Lemma 26. Consider the valid ordering O, and let (l', t, i) be the last instruction which marks d as uncertain in O. The cases break down where $l' = l - 1$ and thus this instruction is in $(l-1, t)$ implying that $d \in \mathcal{M}_{l-1,t}$ and thus $d \in \mathcal{M}^*_{l-1,t} \Rightarrow d \in \text{LSOS}^{\mathcal{M}}_{l,t}$.

Otherwise, $l' \leq l - 2$; we can apply Case 1 of Lemma 26 to show that $d \in \text{SOS}^{\mathcal{M}}_{l}$. Finally, if all valid orderings O of instructions in epoch $l-2$ and block $(l-1, t)$ were to show that d was generated (or killed) we'd always have a suffix that generated (or killed) d, contradicting the existence of an ordering where $d \in \mathcal{M}(O)$. Therefore, we can apply the contrapositives of Lemmas 29 and 30 to yield that $d \notin \mathcal{K}^*_{l-1,t} \cup \mathcal{G}^*_{l-1,t}$, showing that $d \in (\text{SOS}^{\mathcal{M}}_{l} - (\mathcal{K}^*_{l-1,t} \cup \mathcal{G}^*_{l-1,t}))$ and therefore $d \in \text{LSOS}^{\mathcal{M}}_{l,t}$.

Case 2: By the contrapositive of Lemma 30, $d \notin \mathcal{G}^*_{l-1,t}$ (else all suffixes O' of O would show $d \in \mathcal{G}(O')$ which would imply $d \in \mathcal{G}(O)$ – contradiction). However, $d \in \mathcal{G}_{l-1,t}$. Therefore, $d \in \mathcal{G}_{l-1,t} - \mathcal{G}^*_{l-1,t} = \mathcal{G}_{l-1,t} \cap (\bigcup_{t' \neq t} \mathcal{K}_{l-2,t'} \cup \mathcal{M}_{l-2,t'}))$, which implies that $d \in \mathcal{M}^*_{l-1,t}$ and therefore that $d \in \text{LSOS}^{\mathcal{M}}_{l,t}$.

Case 3: By symmetry with Case 2.

Case 4: As Lemma 26 holds, then $d \in \text{SOS}_l^{\mathcal{M}}$. Furthermore, $d \notin (\mathcal{K}^*_{l-1,t} \cup \mathcal{G}^*_{l-1,t})$. If $d \in \text{SOS}_l^{\mathcal{M}} - (\mathcal{K}^*_{l-1,t} \cup \mathcal{G}^*_{l-1,t})$ then $d \in \text{LSOS}^{\mathcal{M}}_{l,t}$. □

5.5 TaintCheck with Uncertainty in Butterfly Analysis

Our formulation of TAINTCHECK to include uncertainty will bear strong resemblance to those introduced in Sections 3.3.2 and 4.5.

5.5.1 First Pass: Instruction-level Transfer Functions and Calculating Side-Out

Unlike Butterfly Analysis, which captured the first pass transfer functions in $\mathcal{G}_{l,t,i}$, we will separate them, using $\mathcal{T}_{l,t,i}$ to be the transfer function associated with instruction (l,t,i):

$$\mathcal{T}_{l,t,i} = \begin{cases} (x_{l,t,i} \leftarrow \bot) & \text{if } (l,t,i) \equiv \texttt{taint}(x) \\ (x_{l,t,i} \leftarrow \top) & \text{if } (l,t,i) \equiv \texttt{untaint}(x) \\ (x_{l,t,i} \leftarrow \{a\}) & \text{if } (l,t,i) \equiv x := \texttt{unop}(a) \\ (x_{l,t,i} \leftarrow \{a,b\}) & \text{if } (l,t,i) \equiv x := \texttt{binop}(a,b) \end{cases}$$

When $x := \texttt{unop}(a)$, we say x *inherits (metadata) from* a and likewise $x := \texttt{binop}(a,b)$ indicates x *inherits (metadata) from* a *and* b. We use the set S:

$$S = \{\texttt{taint}, \texttt{untaint}, \texttt{uncertain}, \{a\}, \{a,b\} | a, b \text{ are memory locations}\}$$

to represent the set of all possible right-hand values in our mapping. We will also utilize the function `loc()` that given (l,t,i) returns x, where x is the destination location in instruction (l,t,i).

At the end of the first pass, blocks in the wings will exchange the TRANSFER-SIDE-OUT (TSO) and create the TRANSFER-SIDE-IN (TSI). The TRANSFER-SIDE-OUT of the transfer functions for a block (l,t) is calculated as:

Algorithm 3 TAINTCHECK TRANSFER(s_1, s_2) Algorithm

Input: $s_1, s_2 \in \{\texttt{taint}, \texttt{untaint}, \texttt{uncertain}\}$
if $s_1 ==$ `taint` or $s_2 ==$ `taint` **then**
  return `taint`
else if $s_1 ==$ `uncertain` or $s_2 ==$ `uncertain` **then**
  return `uncertain`
else
  //$s_1 ==$ `untaint` and $s_2 ==$ `untaint`
  return `untaint`

Algorithm 4 TAINTCHECK MEET(s_1, s_2) Algorithm

Input: $s_1, s_2 \in \{\texttt{taint}, \texttt{untaint}, \texttt{uncertain}\}$
if $s_1 ==$ `taint` and $s_2 ==$ `taint` **then**
  return `taint`
else if $s_1 ==$ `untaint` and $s_2 ==$ `untaint` **then**
  return `untaint`
else
  //either $s_1 ==$ `untaint` and $s_2 ==$ `taint`
  //or $s_1 ==$ `taint` and $s_1 ==$ `untaint`
  //or either $s_1 ==$ `uncertain` or $s_2 ==$ `uncertain`
  return `uncertain`

$\text{TSO}_{l,t} = \bigcup_i \mathcal{T}_{l,t,i}$

5.5.2 Between Passes: Calculating Side-In

Likewise, the TRANSFER-SIDE-IN of the transfer functions for a block (l, t) is the union of the TSO of the wings:

$\text{TSI}_{l,t} = \bigcup_{l-1 \leq l' \leq l+1} \bigcup_{t' \neq t} \text{TSO}_{l',t'}$

Despite the notational difference, the TSO and TSI calculated in this section are identical to those in Section 3.3.2, there called GSO and GSI.

5.5.3 Resolving Transfer Functions to Metadata

To convert between transfer functions and actual metadata in TAINTCHECK, we previously introduced an algorithm to resolve potential inheritance relationships when the ordering between

concurrent instructions is unknown, called `resolve`. It had two return values: `taint` and `untaint`. We refine this algorithm so that for an instruction (l, t, i) which writes to destination m, `resolve` now has three possible return values: `taint`, `untaint` or `uncertain`.

As previously described, `resolve` worked by recursively evaluating transfer functions in the wings, subject to termination conditions. The addition of uncertainty introduces complexity because the TRANSFER (Algorithm 3) and MEET (Algorithm 4) functions are no longer identical. TAINTCHECK defines a destination address m to be tainted if either of its sources m_1 or m_2 is `tainted`, *e.g.,* `transfer`(m_1, m_2) `= taint` if m_1 is tainted. Prior to explicitly tracking uncertainty, if $m \leftarrow m_1$ and there were two possibilities in the wings, one where m_1 was tainted and one where m_1 was untainted, we previously had to behave conservatively and consider m tainted, since `meet(taint, untaint) = taint`. While this conservative analysis impacted precision, it enabled a performance optimization: we could organize the `resolve` function as resembling a breadth first search of a "parent graph"; if an edge ever led to `taint`, then the destination location was considered tainted.

Challenge: Meet Function Not Identical to Transfer Function

This is no longer the case. Consider Algorithm 3, the algorithm for TRANSFER. If at least one of its inputs is `tainted` then it returns `taint`. In contrast, the MEET algorithm, shown in Algorithm 4, only returns `taint` if both s_1 and $_2$ evaluate to `taint`. This difference means we can no longer implicitly short circuit when one tainted input is processed. Furthermore, there is no new safe value to short-circuit on; one `uncertain` input not sufficient for TRANSFER to return `uncertain` whereas MEET will always return uncertain if either input is `uncertain`.

Note: there is a difference between the instruction-level transfer functions, here represented as $\mathcal{T}_{l,t,i}$ and the TRANSFER function. The instruction-level transfer functions specific a destination address and a source, where the source is either a raw metadata value, a single memory location or a pair of memory locations. In contrast, the TRANSFER function operates slowly on

pairs of metadata values. The TRANSFER function is used as a subroutine within the `resolve` algorithm, whose purpose is to convert inheritance relations capture by TAINTCHECK into metadata values.

The transfer function is applied when evaluating a statement of the form $x = m_1 + m_2$. In contrast, the MEET operation is used when it is unclear which metadata status a thread will read. For example, we always must consider whether a thread will read its own local value (reflecting LSOS metadata status), or a value from the wings (reflecting a need to resolve the wings). If there are different values in the wings, the MEET conservatively calculates the "worst" of what the thread sees.

One advantage of our new, more precise formulations of `taint` (respectively, `untaint`) is that `resolve` can only return `taint` (`untaint`) at instruction (l, t, i) if all possible interleavings show $m = \texttt{loc}(l, t, i)$ is `tainted` (`untainted`) at (l, t, i). We will formally prove this in Theorem 37.

A New Resolve Algorithm Incorporating Uncertainty

Algorithm 5 contains a pseudocode implementation of `resolve` for the uncertain Butterfly Analysis version of TaintCheck. It now resembles depth first search. The initial call is to $\texttt{resolve}((s, (l, t, i), T)$, which invokes the recursive $\texttt{do\_resolve}(s, orig\_tid, (l, t, i), T, H)$ until exhaustion.[2] Algorithm 6 contains a pseudocode implemention of `do_resolve`.

Our `resolve` algorithm takes an input a tuple $(s, (l, t, i))$ and a set T of transfer functions in the wings, and returns the taint status of m at instruction (l, t, i), where $m = \texttt{loc}(l, t, i)$ and $\mathcal{T}_{l,t,i} = (m \leftarrow s)$. Note that to determine the new metadata value for m, we call resolve on s; the current metadata value for m is irrelevant unless m serves as one of its own parents, in which case either $s = \{m\}$ or $s = \{m, b\}$ for some other memory location b.

[2]In practice, to curtail long searches and bound memory usage, our implementation sets thresholds for how long exploration of potential predecessors will continue, and explicitly tracks any early returns from exploration in a separate state called `heuristic`.

Algorithm 5 TAINTCHECK `resolve`$(s, (l,t,i), T)$ Algorithm

Input: $s \in S$, (l,t,i): instruction, T: set of transfer functions in wings
if $s ==$ `taint` or $s ==$ `untaint` or $s ==$ `uncertain` **then**
  return s
else if $s == \{a\}$ for memory location a **then**
  a_state_{LSOS} = metadata state of m in LSOS
  $a\_state_{\text{WING}} =$ **do_resolve**$(m, t, (l,t,i), T, (l,t,i))$
  $resolve\_state =$ **meet**$(a\_state_{\text{LSOS}}, a\_state_{\text{WING}})$
  return $resolve\_state$
else
  //$s == \{a, b\}$ for memory locations a, b
  //resolve metadata for a
  a_state_{LSOS} = metadata state of a in LSOS
  $a\_state_{\text{WING}} =$ **do_resolve**$(a, t, (l,t,i), T, (l,t,i))$
  $a\_state_{\text{MEET}} =$ **meet**$(a\_state_{\text{LSOS}}, a\_state_{\text{WING}})$
  //resolve metadata for b
  b_state_{LSOS} = metadata state of b in LSOS
  $b\_state_{\text{WING}} =$ **do_resolve**$(b, t, (l,t,i), T, (l,t,i))$
  $b\_state_{\text{MEET}} =$ **meet**$(b\_state_{\text{LSOS}}, m_2\_state_{\text{WING}})$
  //apply transfer function
  $resolve\_state =$ **transfer**$(a\_state_{\text{MEET}}, m_2\_state_{\text{MEET}})$
  return $resolve\_state$

We define a *proper predecessor* of $x_{l,t,i} \leftarrow s$ to be any $y_{l',t',i'} \leftarrow s'$ such that $\texttt{loc}(l', t', i') \in s$, $s \in S$ and where (l', t', i') executing before (l, t, i) does not violate any valid ordering rules of the prior instructions in H.[3]. We denote the set of proper predecessors for $x_{l,t,i} \leftarrow s$ where $\texttt{loc}(l, t, i) = m$ by $P(m, (l, t, i), T, H)$. For brevity within `resolve`, $\texttt{loc}(y_i)$ will refer to the destination of the instruction associated with y_i.

Note the cascading roles of MEET and TRANSFER; metadata state between the wings and the LSOS is subject to a MEET operation. When a memory location has two parents, their metadata status is subject to a TRANSFER function. Finally, keeping separate counters for the number of times `taint`, `untaint` and `uncertain` are encountered (which could also be boolean values) allows a final MEET calculation of all the metadata values possible via the wings.

[3] Among other requirements, valid ordering rules preclude an instruction repeating itself

Algorithm 6 TAINTCHECK `do_resolve`$(m, orig\_tid, (l,t,i), T, H)$ Algorithm

Input: m: current destination address, $orig\_tid$: original thread, (l,t,i): current instruction, T: set of transfer functions in wings, H: history of previously considered instructions
if $m ==$ `taint` or $m ==$ `untaint` or $m ==$ `uncertain` **then**
  return m
$num\_taint = num\_untaint = num\_uncertain = 0$
for all $(y_{(l',t',i')} \leftarrow s_j) \in P(m, (l,t,i), T, H)$ **do**
  if $s_j ==$ `taint` or $s_j ==$ `untaint` or $s_j ==$ `uncertain` **then**
    return s_j
  else if $s_j == a$ for memory location a **then**
    a_state_{LSOS} = metadata state of a in LSOS
    $a\_state_{\text{WING}} =$ **do_resolve**$(a, orig\_tid, (l',t',i'), T, (l,t,i) :: H)$
    $resolve\_state =$ **meet**$(a\_state_{\text{LSOS}}, a\_state_{\text{WING}})$
    Increment counter of $\{num\_taint, num\_untaint, num\_uncertain\}$ that matches $resolve\_state$
  else
    //resolve metadata for a
    //$s_j = \{a, b\}$ for memory locations a, b
    a_state_{LSOS} = metadata state of a in LSOS
    $a\_state_{\text{WING}} =$ **do_resolve**$(a, orig\_tid, (l',t',i'), T, (l,t,i) :: H)$
    $a\_state_{\text{MEET}} =$ **meet**$(a\_state_{\text{LSOS}}, a\_state_{\text{WING}})$
    //resolve metadata for b
    b_state_{LSOS} = metadata state of b in LSOS
    $b\_state_{\text{WING}} =$ **do_resolve**$(b, orig\_tid, (l',t',i'), T, (l,t,i) :: H)$
    $b\_state_{\text{MEET}} =$ **meet**$(b\_state_{\text{LSOS}}, a\_state_{\text{WING}})$
    //apply transfer function
    $resolve\_state =$ **transfer**$(a\_state_{\text{MEET}}, b\_state_{\text{MEET}})$
    Increment counter of $\{num\_taint, num\_untaint, num\_uncertain\}$ that matches $resolve\_state$
//all proper predecessors have recursively been explored
if $num\_uncertain > 0$ or ($num\_taint > 0$ and $num\_untaint > 0$) **then**
  return `uncertain`
else if $num\_taint > 0$ **then**
  //$num\_untaint == 0$
  return `taint`
else
  //$num\_taint == 0$
  return `untaint`

5.5.4 Second Pass: Representing TaintCheck as an Extension of Reaching Definitions

The `resolve` function enables us to move from transfer functions for individual instructions to metadata for locations. This enables us to express TAINTCHECK as an extension of reaching definitions. The second pass of TAINTCHECK performs checks and resolve the transfer function $\mathcal{T}_{l,t,i} = m \leftarrow s$ to a metadata value for destination address m. We can therefore define:

$$\mathcal{G}_{l,t,i} = \begin{cases} m & \texttt{resolve}(s,(l,t,i),\text{SIDE-IN}) \leftarrow \texttt{taint} \\ \varnothing & \text{otherwise} \end{cases}$$

$$\mathcal{K}_{l,t,i} = \begin{cases} m & \texttt{resolve}(s,(l,t,i),\text{SIDE-IN}) \leftarrow \texttt{untaint} \\ \varnothing & \text{otherwise} \end{cases}$$

$$\mathcal{M}_{l,t,i} = \begin{cases} m & \texttt{resolve}(s,(l,t,i),\text{SIDE-IN}) \leftarrow \texttt{uncertain} \\ \varnothing & \text{otherwise} \end{cases}$$

The block equations for $\mathcal{G}_{l,t}$, $\mathcal{K}_{l,t}$ and $\mathcal{M}_{l,t}$ follow immediately once we have defined $\mathcal{G}_{l,t,i}$, $\mathcal{K}_{l,t,i}$ and $\mathcal{M}_{l,t,i}$, respectively. The motivation for allowing an instruction which might "mark d uncertain" is now clear – our `resolve` function can return `uncertain`.

Note that this new definition of $\mathcal{G}_{l,t}$, $\mathcal{K}_{l,t}$ and $\mathcal{M}_{l,t}$ (separating the transfer functions from the state calculations) naturally encapsulates what we previously expressed using LASTCHECK.

There are some situations where we might need GSI, KSI or MSI as state. We define

$$\text{ALL}_{l,t}^{\texttt{taint}} = \{m | \exists i \text{ s.t. } \mathcal{T}_{l,t,i} = (m \leftarrow s) \wedge \texttt{resolve}(s,(l,t,i),\text{SIDE-IN}) \leftarrow \texttt{taint}\ \}$$

$$\text{ALL}_{l,t}^{\texttt{untaint}} = \{m | \exists i \text{ s.t. } \mathcal{T}_{l,t,i} = (m \leftarrow s) \wedge \texttt{resolve}(s,(l,t,i),\text{SIDE-IN}) \leftarrow \texttt{untaint}\ \}$$

$$\text{ALL}_{l,t}^{\texttt{uncertain}} = \{m | \exists i \text{ s.t. } \mathcal{T}_{l,t,i} = (m \leftarrow s) \wedge \texttt{resolve}(s,(l,t,i),\text{SIDE-IN}) \leftarrow \texttt{uncertain}\ \}$$

$$\text{ALL}_{l,t}^{\mathcal{G}} = \text{ALL}_{l,t}^{\texttt{taint}}$$

$$\text{ALL}_{l,t}^{\mathcal{K}} = \text{ALL}_{l,t}^{\texttt{untaint}}$$

$$\text{ALL}_{l,t}^{\mathcal{M}} = \text{ALL}_{l,t}^{\texttt{uncertain}}$$

Then the equations for $\text{GSO}_{l,t}$, $\text{KSO}_{l,t}$ and $\text{MSO}_{l,t}$ immediately follow, as do those for $\text{WING}_{l,t}^{\mathcal{G}}$,

$\text{WING}^{\mathcal{K}}_{l,t}$ and $\text{WING}^{\mathcal{M}}_{l,t}$ and thus those for $\text{GSI}_{l,t}$, $\text{KSI}_{l,t}$ and $\text{MSI}_{l,t}$. We now have all the necessary building blocks to calculate the epoch summaries $\mathcal{G}_l$, $\mathcal{K}_l$ and $\mathcal{M}_l$. The SOS and LSOS equations all immediately follow, as do their proofs (indeed, all earlier proofs follow as well).

Lemma 35. *If* $\texttt{resolve}(s, (l,t,i), \text{TSI}_{l,t})$ *returns* `untaint` *for location* $m = \texttt{loc}(l,t,i)$ *at instruction* (l,t,i)*, then under all valid orderings of the first* $l+1$ *epochs,* m *is* `untainted` *at instruction* (l,t,i).

Proof. If $s =$ `untaint`, then this is trivially true. Likewise, if $s =$ `uncertain` or $s =$ `taint`, $\texttt{resolve}(s, (l,t,i), \text{TSI}_{l,t,i})$ will never return `untaint`. It remains to show this is true if $s = \{a\}$ or $s = \{a, b\}$. We will begin by making a simplifying assumption, namely $s = \{a\}$, and then show how to generalize.

The `resolve` algorithm has two components. First, `resolve` looks up the metadata state of a in the LSOS. If a is `tainted` or `uncertain`, `resolve` cannot return `untaint`. Therefore, the LSOS must have a as `untainted`. We can apply Lemma 33 so that every valid ordering of instructions in epochs $[0, l-2]$ and block $(l-1, t)$ must have a `untainted`.

It will suffice to show that all interleavings of instructions in the wings with (l,t,i) either lead to a being `untainted` immediately before instruction (l,t,i), or else no instruction in the wings modifies a. If no instruction modifies a, then the claim is shown. Otherwise, we observe that `resolve` explores all proper predecessors (and thus all valid interleavings) of instructions in the wings, so it will not miss an ordering of instructions in the wings that could potentially lead to a being `tainted` or `uncertain`. Finally, we observe that by applying the MEET function can never return `untaint` if it observes a potential interleaving that leads to either `uncertain` or `taint`. If all instructions which happen before and which occur concurrent with (l,t,i) `untaint` a, then a is `untainted` under all valid orderings. So, m must be `untainted` when it inherits from a at (l,t,i).

To generalize, consider instead if $s = \{a, b\}$. Before returning, `resolve` performs $\text{TRANSFER}(a\_state_{\text{MEET}}, b\_state_{\text{MEET}})$, which will only return `untaint` if both a and b are un-

tainted, so each of a and b must meet the same criteria outlined above: LSOS reflects `untaint`, and all interleavings of instructions of the wings show that both a and b are `untainted` immediately before (l, t, i). Therefore, when m inherits from a and b at instruction (l, t, i), m inherits `untaint` under all possible interleavings.

□

Lemma 36. *If* $\texttt{resolve}(s, (l, t, i), \text{TSI}_{l,t})$ *returns* `taint` *for location* $m = \texttt{loc}(l, t, i)$ *at instruction* (l, t, i)*, then under all valid orderings of the first* $l + 1$ *epochs,* m *is* `tainted` *at instruction* (l, t, i).

Proof. Proof proceeds in a similar manner to Lemma 35, with a few key differences. If $s =$ `taint`, the statement is trivially true. If $s =$ `untaint` or $s =$ `uncertain`, `resolve` will not return `taint`. We will again begin by making a simplifying assumption, that $s = \{a\}$, and then show how it holds with $s = \{a, b\}$.

Lemma 32 guarantees that every valid ordering of instructions in epochs $[0, l - 2]$ and block $(l - 1, t)$ must have m `tainted`. As in Lemma 35, `resolve` explores all proper predecessors, and will not miss an ordering of instructions in the wings that could potentially lead to . Once more, the MEET function cannot return `taint` if it observes a potential interleavings of instructions in the wings that leads to `untaint` or `uncertain`. The proof when $s = \{a\}$ completes in the same manner as Lemma 35.

When $s = \{a, b\}$, the proof deviates slightly from Lemma 35. $\text{TRANSFER}(a\_state_{\text{MEET}}, b\_state_{\text{MEET}})$ returns `taint` if at least one of a_state_{MEET} or b_state_{MEET} returns `taint`. This implies that under all interleavings, at least one of a or b is always guaranteed to be tainted before (l, t, i) executes. By the definition of TAINTCHECK, this implies that when m inherits from a and b at instruction (l, t, i), m inherits `taint` under all possible interleavings. □

Theorem 37. *Any error detected by the original* TAINTCHECK *on a valid execution ordering for a given machine (obeying intra-thread dependences and supporting cache coherence) will*

also be flagged by our butterfly analysis as either `tainted` *or* `uncertain`*. Furthermore, any failed check of a* `tainted` *address is an error the original* TAINTCHECK *would discover under all valid execution orderings for a given machine. Thus, any potential false positives derive from failed checks of* `uncertain`*.*

Proof. First, if there exists a valid execution with a failed check of `taint`, then there exists a valid ordering of the first $l + 1$ epochs such that m is tainted at instruction (l, t, i), and by the contrapositive of Lemma 35, `resolve` will not return `untaint` for m at (l, t, i). So, m will either be `tainted` or marked `uncertain`. The second statement follows directly from Lemma 36. If everything marked as `taint` is a true error, and nothing marked by `untaint`is ever an error, then all false positives must flow from a failed check of `uncertain`. □

Incorporating Thresholds

In practice, rather than allowing Algorithms 5 and 6 to run until exhaustion, we incorporated a threshold that cut off exploration. This was present in our prior implementation from Chapter 4 as well; whenever we encountered the threshold, we conservatively returned `taint`. Now, with the addition of uncertainty, we were able to explicitly track the uncertainty that arises due to encountering this threshold, which we have titled `heuristic`. This is explicitly tracked separately from `uncertain` in our evaluation section.

5.6 Chrysalis Analysis: Incorporating Uncertainty

Chrysalis Analysis' structure, as depicted in Figure 4.2(b), strongly resembles that of Butterfly Analysis. The main difference is the asymmetry introduced by dividing blocks into *maximal sub-blocks* based on happens-before arcs from high-level synchronization. The richer representation of Chrysalis Analysis allows it to achieve higher precision. However, the asymmetry introduced

by incorporating arcs and happens-before information adds additional complexity to standard Chrysalis Analysis. Incorporating uncertainty further increases the complexity.

Vector Clocks and Maximal Subblocks

Chrysalis analysis uses *maximal subblocks* instead of blocks; the boundaries of a maximal subblock are formed by the beginning or ending of the containing block (l, t), as well as the location of any SND or RCV instructions. For expediency, we will occasionally refer to a maximal subblock b as "subblock" as shorthand for "maximal subblock". We will use $v(b)$ to indicate the vector clock associated with maximal subblock b. The relation $v(b) < v(b')$ indicates that subblock b has a vector clock that happened before subblock b'; equivalently, subblock b executed before subblock b'. $v(b) \sim v(b')$ indicates that b and b' executed concurrently.

5.6.1 Gen and Kill Equations

Instruction-level

Chrysalis Analysis uses the same instruction-level equations as Butterfly Analysis.

Subblock-level

We will use the notation $\mathcal{G}_b, \mathcal{K}_b$ and $\mathcal{M}_b$ to represent the $\mathcal{G}, \mathcal{K}, \mathcal{M}$ primitives across a maximal subblock b. If subblock $b = (l, t, (i, j))$, then the dataflow representations for subblock b are:

$$\mathcal{G}_b = \mathcal{G}_{l,t,(i,j)}$$
$$\mathcal{K}_b = \mathcal{K}_{l,t,(i,j)}$$
$$\mathcal{M}_b = \mathcal{M}_{l,t,(i,j)}$$

Side-Out and Side-In

For a given maximal subblock $b = (l, t, (j, k))$, we first calculate $\text{ALL}_b^{\mathcal{G}}$, $\text{ALL}_b^{\mathcal{K}}$ and $\text{ALL}_b^{\mathcal{M}}$:

$$\begin{aligned}
\text{ALL}_b^{\mathcal{G}} &= \bigcup_{\{(l,t,i)|j \le i \le k\}} \mathcal{G}_{l,t,i} \\
\text{ALL}_b^{\mathcal{K}} &= \bigcup_{\{(l,t,i)|j \le i \le k\}} \mathcal{K}_{l,t,i} \\
\text{ALL}_b^{\mathcal{M}} &= \bigcup_{\{(l,t,i)|j \le i \le k\}} \mathcal{M}_{l,t,i}
\end{aligned}$$

We can then present the various Side-out equations:

$$\begin{aligned}
\text{GSO}_b &= \text{ALL}_b^{\mathcal{G}} - (\text{ALL}_b^{\mathcal{K}} \cup \text{ALL}_b^{\mathcal{M}}) \\
\text{KSO}_b &= \text{ALL}_b^{\mathcal{K}} - (\text{ALL}_b^{\mathcal{G}} \cup \text{ALL}_b^{\mathcal{M}}) \\
\text{MSO}_b &= \text{ALL}_b^{\mathcal{M}} \cup (\text{ALL}_b^{\mathcal{G}} \cap \text{ALL}_b^{\mathcal{K}})
\end{aligned}$$

As in Butterfly Analysis, we use the $\text{WING}_b^{\mathcal{G}}$, $\text{WING}_b^{\mathcal{K}}$ and $\text{WING}_b^{\mathcal{M}}$ intermediate sets before calculating side-in:

$$\begin{aligned}
\text{WING}_b^{\mathcal{G}} &= \bigcup_{\{b'|v(b)\sim v(b')\}} \text{GSO}_{b'} \\
\text{WING}_b^{\mathcal{K}} &= \bigcup_{\{b'|v(b)\sim v(b')\}} \text{KSO}_{b'} \\
\text{WING}_b^{\mathcal{M}} &= \bigcup_{\{b'|v(b)\sim v(b')\}} \text{MSO}_{b'}
\end{aligned}$$

and then use these sets to calculate side-in:

$$\begin{aligned}
\text{GSI}_b &= \text{WING}_b^{\mathcal{G}} - (\text{WING}_b^{\mathcal{K}} \cup \text{WING}_b^{\mathcal{M}}) \\
\text{KSI}_b &= \text{WING}_b^{\mathcal{K}} - (\text{WING}_b^{\mathcal{G}} \cup \text{WING}_b^{\mathcal{M}}) \\
\text{MSI}_b &= \text{WING}_b^{\mathcal{M}} \cup (\text{WING}_b^{\mathcal{G}} \cap \text{WING}_b^{\mathcal{K}})
\end{aligned}$$

5.6.2 Incorporating Uncertainty into Strongly Ordered State

Notation

We introduce notation to represent extended epochs that includes not only subblocks from epoch $l+1$ which execute before subblocks in epoch l, but also subblocks in epoch $l-1$ which are known to execute after at least one subblock in epoch l. Let $l^{\mp}$ denote the *bilaterally extended*

epoch, defined as:

$$l^{\mp} = l^{+} \cup \{b | b \in \text{MB}_{l-1} \wedge \exists b' \in \text{MB}_{l} \text{ s.t. } v(b') < v(b)\}$$

or equivalently:

$$l^{\mp} = \text{MB}_{l} \cup \{b | b \in \text{MB}_{l-1} \wedge \exists b' \in \text{MB}_{l} \text{ s.t. } v(b') < v(b)\} \cup \{b | b \in \text{MB}_{l+1} \wedge \exists b' \in \text{MB}_{l} \text{ s.t. } v(b) < v(b')\} s$$

Summarizing an Epoch

As in Section 5.4.3, we will begin by specifying the must-kill and must-generate sets, as well as may-kill and may-generate sets, across an epoch. The must-{kill, gen} sets will be $\mathcal{K}_l$ and $\mathcal{G}_l$, respectively, while the difference between may- and must- {kill, gen} will be folded into $\mathcal{M}_l$.

We begin with must-kill over an epoch:

$$\mathcal{K}_l = \bigcup_{b \in l^{+}} \left(\left(\mathcal{K}_b - \bigcup_{\{b' | b' \in \text{MB}_{l^{\mp}}\}} (\mathcal{G}_{b'} \cup \mathcal{M}_{b'}) \right) - \bigcup_{\{b'' | v(b) \sim v(b''), b'' \in \text{MB}_{[l-1, l^{+}]}\}} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right)$$

We could equivalently express this (using the fact that $(A - B) - C = A - (B \cup C)$ in set theory):

$$\mathcal{K}_l = \bigcup_{b \in l^{+}} \left(\mathcal{K}_b - \bigcup_{\{b' | v(b) < v(b') \vee v(b) \sim v(b'), b' \in \text{MB}_{[l-1, l^{+}]}\}} (\mathcal{G}_{b'} \cup \mathcal{M}_{b'}) \right)$$

but it will be more useful to have the first formulation when calculating the difference between may-kill and must-kill.

Likewise, the must-gen set is:

$$\mathcal{G}_l = \bigcup_{b \in l^+} \left(\left(\mathcal{G}_b - \bigcup_{\{b' | b' \in \text{MB}_{l^\mp}\}} (\mathcal{K}_{b'} \cup \mathcal{M}_{b'}) \right) - \bigcup_{\{b'' | v(b) \sim v(b''), b'' \in \text{MB}_{[l-1,l^+]}\}} (\mathcal{K}_{b''} \cup \mathcal{M}_{b''}) \right)$$

If we simply wanted the may-$\{$kill, gen$\}$ sets, those would be:

$$\text{may-}\mathcal{K}_l = \bigcup_{b \in l^+} \left(\mathcal{K}_b - \bigcup_{\{b' | v(b) < v(b'), b' \in \text{MB}_{[l-1,l^+]}\}} (\mathcal{G}_{b'} \cup \mathcal{M}_{b'}) \right)$$

$$\text{may-}\mathcal{G}_l = \bigcup_{b \in l^+} \left(\mathcal{G}_b - \bigcup_{\{b' | v(b) < v(b'), b' \in \text{MB}_{[l-1,l^+]}\}} (\mathcal{K}_{b'} \cup \mathcal{M}_{b'}) \right)$$

There are contrasts and similarities to the formulas in Section 5.4.3. First, this is not just $\bigcup_b \mathcal{K}_b$; we must subtract off the effects of subblocks which *definitely* occurred later, even in the may-kill set. However, as in Section 5.4.3, the difference between the two formulas is that we do not subtract off the effects of concurrent blocks. This motivates our formulation for $\mathcal{M}_l$:

$$\begin{aligned}
\mathcal{M}_l = \bigcup_{b \in l^+} & \left[\mathcal{M}_b - \bigcup_{\{b' | v(b) < v(b'), b' \in \text{MB}_{l^\mp}\}} (\mathcal{G}_{b'} \cup \mathcal{K}_{b'}) \right] \\
& \cup \left[\left(\mathcal{K}_b - \bigcup_{\{b' | v(b) < v(b'), b' \in \text{MB}_{[l-1,l^+]}\}} (\mathcal{G}_{b'} \cup \mathcal{M}_{b'}) \right) \cap \bigcup_{\{b'' | v(b) \sim v(b''), b'' \in \text{MB}_{[l-1,l^+]}\}} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right] \\
& \cup \left[\left(\mathcal{G}_b - \bigcup_{\{b' | v(b) < v(b'), b' \in \text{MB}_{[l-1,l^+]}\}} (\mathcal{K}_{b'} \cup \mathcal{M}_{b'}) \right) \cap \bigcup_{\{b'' | v(b) \sim v(b''), b'' \in \text{MB}_{[l-1,l^+]}\}} (\mathcal{K}_{b''} \cup \mathcal{M}_{b''}) \right] \\
& - (\mathcal{G}_l \cup \mathcal{K}_l)
\end{aligned}$$

Once again, $\mathcal{M}_l$ is primarily composed of three parts. First, if a subblock in l^+ marks d as uncertain, and no strictly later subblock b' generates or kills d (as net effect of its subblock) then $d \in \mathcal{M}_l$. Also, if at least one subblock b kills (generates) d, and no strictly later subblock

generates (kills) d or marks it as uncertain (as net effect) but a concurrent subblock b' either generates (kills) or marks it as uncertain, then $d \in \mathcal{M}_l$. To make mutual exclusion obvious, we also subtract the $\mathcal{G}_l \cup \mathcal{K}_l$, which is not circular as neither $\mathcal{G}_l$ nor $\mathcal{K}_l$ depend on $\mathcal{M}_l$.

Mutual Exclusion

Mutual exclusion will follow by construction. As M_l subtracts both $\mathcal{G}_l$ and $\mathcal{K}_l$, it follows that $\mathcal{M}_l \cap \mathcal{G}_l = \mathcal{M}_l \cap \mathcal{K}_l = \varnothing$. Lemmas 38 and 39 (proved in the next subsection) will provide mutual exclusion between $\mathcal{G}_l$ and $\mathcal{K}_l$ – as it cannot be true that, for all VVO O defined over identical ranges, $d \in \mathcal{G}(O)$ and $d \in \mathcal{K}(O)$.

Invariants for Epoch Summaries

Lemma 38. *If $d \in \mathcal{K}_l$ then for all valid vector orderings O of instructions in epochs $[l-1, l^+]$, $d \in \mathcal{K}(O)$.*

Proof. If $d \in \mathcal{K}_l$ then there exists maximal subblock $b \in \text{MB}_{l^+}$ such that $d \in \mathcal{K}_b$, then (1) for all subblocks b' such that $v(b) < v(b'), b' \in \text{MB}_{[l-1,l^+]}$, $d \notin \mathcal{G}_{b'}$ and $d \notin \mathcal{M}_{b'}$ and (2) for all subblocks b'' such that $v(b) \sim v(b''), b'' \in \text{MB}_{[l-1,l^+]}$, $d \notin \mathcal{G}_{b''}$ and $d \notin \mathcal{M}_{b''}$.

Consider any valid vector ordering O. Let (l', t, i) be the last instruction in b to kill d. Let O' be the suffix of O beginning with instruction (l', t, i). The only instructions that can follow (l', t, i) are those that are concurrent or after (l', t, i). However, all such instructions must belong to a subblock b' which is either concurrent with b or ordered after b; in either case, $d \notin \mathcal{G}_{b'}$ and $d \notin \mathcal{M}_{b'}$ so even if an instruction in b' generates or marks d as uncertain, there must be a later instruction within b' which kills d. Thus, $d \in \mathcal{K}(O')$ and $d \in \mathcal{K}(O)$. □

Lemma 39. *If $d \in \mathcal{G}_l$ then for all valid vector orderings O of instructions in epochs $[l-1, l^+]$, $d \in \mathcal{G}(O)$.*

Symmetric to Lemma 38.

Lemma 40. *If at least one of the following three conditions holds:*

(1) $\exists$ *valid vector ordering* O *of instructions in epoch* $l^{\mp}$ *such that* $d \in \mathcal{M}(O)$ *and the last instruction to mark* d *as uncertain occurs in epoch* l^{+} *or*

(2) $\exists$ *subblock* $b \in \text{MB}_{l^+}$ *such that* $d \in (\mathcal{G}_b - \bigcup_{\{b' | v(b) < v(b'), b' \in \text{MB}_{l^{\mp}}\}} (\mathcal{M}_{b'} \cup \mathcal{K}_{b'}))$ *and* $\exists$ *a valid vector ordering* O *of instructions in epochs* $[l-1, l^+]$ *such that* $d \notin \mathcal{G}(O)$ *or*

(3) $\exists$ *subblock* $b \in \text{MB}_{l^+}$ *such that* $d \in (\mathcal{K}_b - \bigcup_{\{b' | v(b) < v(b'), b' \in \text{MB}_{l^{\mp}}\}} (\mathcal{M}_{b'} \cup \mathcal{G}_{b'}))$ *and* $\exists$ *a valid vector ordering* O *of instructions in epochs* $[l-1, l^+]$ *such that* $d \notin \mathcal{K}(O)$

then $d \in \mathcal{M}_l$.

Proof by cases. As in Lemma 25, the proof proceeds by cases.

1. $\exists O$ of instructions in $l^{\mp}$ such that $d \in \mathcal{M}(O)$ and the last instruction to mark d as uncertain belongs to l^+ implies $\exists (l', t, i)$ (with $l' = l$ or $l' = l+1$) such that $d \in \mathcal{M}_{l',t,i}$ and no later instruction in O kills or generates d. Let b be the subblock containing (l', t, i). Since no later instruction gen or kills d in O (and therefore in b), $d \in \mathcal{M}_b$. Also, $\forall b'$ such that $v(b) < v(b')$, $d \notin (\mathcal{G}_{b'} \cup \mathcal{K}_{b'})$, so $d \in \mathcal{M}_b - \bigcup_{\{b' | v(b) < v(b'), b' \in \text{MB}_{l^{\mp}}\}} (\mathcal{G}_{b'} \cup \mathcal{K}_{b'})$. It remains to show that $d \notin \mathcal{G}_l \cup \mathcal{K}_l$. This follows as the suffix beginning with (l', t, i) is a valid suffix for some valid vector ordering O' of instructions in epochs $[l-1, l^+]$, so $d \in \mathcal{M}(O')$ and by applying the contrapositives of Lemmas 38 and 39, we get that $d \notin \mathcal{G}_l \cup \mathcal{K}_l$.
2. Let (l', t, i) be the last instruction in b to generate d. Consider the suffix O' of O beginning with instruction (l', t, i). It must hold that $d \in \mathcal{M}(O')$ or $d \in \mathcal{K}(O')$ [follows by $d \notin \mathcal{G}(O')$, since the first instruction of O' generates d]. So there must be at least one other thread $t' \neq t$ that either considers d as uncertain or kills d. Consider the last such operation in O', and label its subblock b'. As this is the last operation in O' to mark d uncertain or kill d, it must also be the last instruction in b' to touch d (by definition, no later instruction will gen, kill or mark d uncertain). Then $d \in (\mathcal{K}_{b'} \cup \mathcal{M}_{b'})$. By construction $v(b) \not< v(b')$ [otherwise, we have a contradiction]. Likewise, $v(b') \not< v(b)$ [otherwise, we contradict O a VVO] so we must have that $v(b') \sim v(b)$, which implies $d \in (\mathcal{G}_b - \bigcup_{\{b' | v(b) < v(b')\}} (\mathcal{M}_{b'} \cup \mathcal{K}_{b'})) \cap (\bigcup_{\{b'' | v(b) \sim v(b'')\}} (\mathcal{M}_{b'} \cup \mathcal{K}_{b'}))$.

It remains to show that $d \notin \mathcal{G}_l \cup \mathcal{K}_l$. $d \notin \mathcal{G}_l$ follows by construction. As $d \in (\mathcal{G}_b - \bigcup_{\{b' | v(b) < v(b'), b' \in \text{MB}_{l^{\mp}}\}} (\mathcal{M}_{b'} \cup \mathcal{K}_{b'}))$, it follows that $\forall b' \in \text{MB}_{[l-1,l^+]}$ such that $v(b) < v(b')$, $d \notin \mathcal{K}'_b$. If $d \in \mathcal{K}_l$, then $\exists \hat{b}$ such that $d \in \mathcal{K}_{\hat{b}}$ and $\forall b' \in \text{MB}_{[l-1,l]}$ such that $v(\hat{b}) < v(b')$, $d \notin \mathcal{G}_{b'} \cup \mathcal{M}_{b'}$. We will consider all possible ordering relations between b and $\hat{b}$, show that each reaches a contradiction, and thus that $d \notin \mathcal{K}_l$.

If $v(b) < v(\hat{b})$, then this contradicts $d \in (\mathcal{G}_b - \bigcup_{\{b' | v(b) < v(b'), b' \in \text{MB}_{l^{\mp}}\}} (\mathcal{M}_{b'} \cup \mathcal{K}_{b'}))$. If $v(\hat{b}) < v(b)$, then this contradicts $d \in \mathcal{K}_l$ (as a subblock after $\hat{b}$ generates d). Likewise, if $v(\hat{b}) \sim v(b)$, then this contradicts $d \in \mathcal{K}_l$ and thus $d \in \mathcal{M}_l$.

3. Follows by symmetry from case (2).

□

Strongly Ordered State Equations

Our modified equations for the SOS will take exactly the same form as in Section 5.4.3:

$\text{SOS}_l^{\mathcal{G}} = \mathcal{G}_{l-2} \cup (\text{SOS}_{l-1}^{\mathcal{G}} - (\mathcal{K}_{l-2} \cup \mathcal{M}_{l-2}))$.

$\text{SOS}_l^{\mathcal{K}} = \mathcal{K}_{l-2} \cup (\text{SOS}_{l-1}^{\mathcal{K}} - (\mathcal{G}_{l-2} \cup \mathcal{M}_{l-2}))$.

$\text{SOS}_l^{\mathcal{M}} = \mathcal{M}_{l-2} \cup (\text{SOS}_{l-1}^{\mathcal{M}} - (\mathcal{G}_{l-2} \cup \mathcal{K}_{l-2}))$.

SOS Invariants

We will use very similar invariants as in Section 5.4.3, only modified to take into account valid vector orderings (VVO) versus valid orderings (VO).

Lemma 41. *If $d \in \text{SOS}_l^{\mathcal{K}}$ then $\forall$ valid vector orderings $\hat{O}$ of instructions in epochs $[0, (l-2)^+], d \in \mathcal{K}(\hat{O})$.*

Proof. By induction. Proof has similar structure to Lemma 27, but generalizes to use valid vector orderings.

Base cases: $l < 2$ then $\text{SOS}_l^{\mathcal{K}} = \varnothing$; trivially true. $l = 2$, then $\text{SOS}_2^{\mathcal{K}} = \mathcal{K}_0$, and we can apply Lemma 38 for epoch 0.

For the inductive hypothesis, we will assume the statement holds for $k < l - 2$ and show it holds for $k = l - 2$. Then, if $d \in \text{SOS}_l^{\mathcal{K}}$, there are two cases: either (1) $d \in \mathcal{K}_{l-2}$ or (2) $d \in \text{SOS}_{l-1}^{\mathcal{K}} - (\mathcal{G}_{l-2} \cup \mathcal{M}_{l-2})$.

Case 1: If $d \in \mathcal{K}_{l-2}$, then Lemma 38 applies. We assume the notation in that proof. In particular, we can use the same construction, and observe that any suffix O' of O (O consisting of instruction in epochs $[l-3, (l-2)^+]$ as per the proof) beginning with the instruction (l', t, i) must have $d \in \mathcal{K}(O')$, so every such suffix O' has $d \in \mathcal{K}(O')$.

Now consider an arbitrary VVO $\hat{O}$ of instructions in epochs $[0, (l-2)^+]$. It suffices to observe that each $\hat{O}$ has a suffix $\hat{O}'$ beginning with the same instruction (l', t, i), and where all subsequent instructions must belong to epochs $[l-3, (l-2)^+]$. It follows that $\hat{O}'$ must have $d \in \mathcal{K}(\hat{O}')$ and thus $d \in \mathcal{K}(\hat{O})$.

Case 2: Begin with any VVO $\hat{O}$ of instructions in epochs $[0, (l-2)^+]$. Let $\hat{O}_{[0,l-3]}$ be the restriction of $\hat{O}$ to instructions from epochs $[0, l-3]$. By the inductive hypothesis, $d \in \mathcal{K}(\hat{O}_{[0,l-3]})$. Let instruction (l', t, i) be the last kill of d in $\hat{O}_{[0,l-3]}$, and let O' be the suffix of $\hat{O}$ beginning with instruction (l', t, i). It will suffice to show that $d \in \mathcal{K}(O')$.

Let $O'_{[0,l-3]}$ be the suffix of O' restricted to instructions from epochs $[0, l-3]$. By construction and the inductive hypothesis, $d \in \mathcal{K}(O'_{[0,l-3]})$. Our proof now proceeds by contradiction. We will consider if $d \notin \mathcal{K}(O')$ and obtain a contradiction.

Let $O'_{(l-2)^+}$ be the restriction of O' to instructions from extended epoch $(l-2)^+$. If $d \notin \mathcal{K}(O')$, then there exists a later instruction in $O'_{(l-2)^+}$ which either generated d or marked d as uncertain with no subsequent instruction killing d (by our previous observation; no instructions in $[0, l-3]$ which occurred after (l', t, i) could have done so). This means either **(a)** $d \in \mathcal{M}(O'_{l-2})$ or **(b)** $d \in \mathcal{G}(O'_{l-2})$.

(a): Suppose $d \in \mathcal{M}(O'_{l-2})$; let (l'', t'', i'') be the last instruction to mark d uncertain.

Let b be the maximal subblock which contains (l'', t'', i''). It must follow that $d \in \mathcal{M}_b - \bigcup_{\{b' | v(b) < v(b'), b' \in \text{MB}_{(l-2)^+}\}} (\mathcal{G}_{b'} \cup \mathcal{K}_{b'})$; otherwise, some instruction guaranteed to execute after (l'', t'', i'') would kill or generate d, contradicting $d \in \mathcal{M}(O'_{l-2})$. But then $d \in \mathcal{M}_{l-2}$; contradiction.

(b): Suppose $d \in \mathcal{G}(O'_{l-2})$; as in case (a), let (l'', t'', i'') be the last instruction to generate d. Let b be the maximal subblock which contains (l'', t'', i''). It must follow that $d \in \mathcal{G}_b - \bigcup_{\{b' | v(b) < v(b'), b' \in \text{MB}_{(l-2)^+}\}} (\mathcal{M}_{b'} \cup \mathcal{K}_{b'})$; otherwise, some instruction guaranteed to execute after (l'', t'', i'') would kill or mark d as uncertain, contradicting $d \in \mathcal{G}(O'_{l-2})$. But then, by construction, $d \in \mathcal{M}_{l-2}$ or $d \in \mathcal{G}_{l-2}$; either way, we achieve a contradiction. So, $d \in \mathcal{K}(O')$, and therefore $d \in \mathcal{K}(\hat{O})$ for any $\hat{O}$, completing the proof.

□

Lemma 42. *If $d \in SOS_l^{\mathcal{G}}$ then $\forall$ valid vector orderings $\hat{O}$ of instructions in epochs $[0, (l-2)^+], d \in \mathcal{G}(\hat{O})$.*

This follows by symmetry with Lemma 41.

Lemma 43. *If one of the following is true:*

(1) $\exists$ valid vector ordering O of instructions in epochs $[0, (l-2)^+]$ such that $d \in \mathcal{M}(O)$ OR

(2) $\exists$ valid vector orderings O, O' of instructions in epochs $[0, (l-2)^+]$ such that $d \notin \mathcal{G}(O')$ but $d \in \mathcal{G}(O)$, where the last generate of d in O occurs in epoch $(l-2)^+$ OR

(3) $\exists$ valid vector orderings O, O' of instructions in epochs $[0, (l-2)^+]$ such that $d \notin \mathcal{K}(O')$ but $d \in \mathcal{K}(O)$, where the last kill of d in O occurs in epoch $(l-2)^+$ OR

(4) (Propagation) $\exists\, l' < l-2$ such that cases (1), (2), or (3) applies to instructions in epochs $[0, l']$ and $\forall l''$ such that $l - 2 \geq l'' > l'$, $d \notin (\mathcal{G}_{l''} \cup \mathcal{K}_{l''})$

then $d \in SOS_l^{\mathcal{M}}$.

Proof. We consider each case in turn.

Case 1: Let (l', t, i) be the last instruction in O which marks d as uncertain. There are two cases: **(a)** $l' \in (l-2)^+$ (e.g., $l' = l-2$ or $l' = l-1$) and **(b)** $l' < l-2$

(a): follows by Lemma 40, case (1) – the restriction of O to $l^{\mp}$ must also have $d \in \mathcal{M}(O)$, so $d \in \mathcal{M}_{l-2}$ which implies $d \in \text{SOS}_l^{\mathcal{M}}$.

(b): Let O^P be the prefix of O such that the last instruction of O^P is (l', t, i) and let O^S be the suffix of O beginning with the instruction immediately following (l', t, i). If $d \in \mathcal{M}_{l'} - \bigcup_{l-2 \geq \tilde{l} > l'} (\mathcal{K}_{\tilde{l}} \cup \mathcal{G}_{\tilde{l}})$ then $d \in \text{SOS}_l^{\mathcal{M}}$ (unpacking the recursion). By construction, $d \notin \mathcal{K}(O^S) \wedge d \notin \mathcal{G}(O^S)$. This implies that any kill or gen in O^S would have to be followed by an instruction which marks d uncertain; however, no such instruction can exist since (l', t, i) is the last instruction to mark d uncertain. Thus no instructions in O^S can generate or kill d, which implies $\forall \tilde{l} > l'$, $d \notin \mathcal{K}_{\tilde{l}} \wedge d \notin \mathcal{G}_{\tilde{l}}$. Therefore, $d \in \text{SOS}_l^{\mathcal{M}}$.

Case 2: Given that $d \in \mathcal{G}(O)$ and that the last generate of d in O occurs in epoch $(l-2)^+$, we know that there exists $b \in \text{MB}_{(l-2)^+}$ such that $\forall b' \in \text{MB}_{[0,(l-2)^+]}$ such that $v(b) < v(b')$, $d \notin \mathcal{K}_{b'} \cup \mathcal{M}_{b'}$. [Applying the rules of valid vector orderings, we can restrict this to $b' \in \text{MB}_{(l-2)^{\mp}]}$.] This follows by definition of a VVO. Therefore, $d \in \mathcal{G}_b - \bigcup_{\{b' | v(b) < v(b'), b' \in \text{MB}_{(l-2)^{\mp}}\}} \mathcal{K}_{b'} \cup \mathcal{M}_{b'}$. By definition, this implies that $d \in \mathcal{G}_{l-2}$ or $d \in \mathcal{M}_{l-2}$.

Let (l', t, i) be the last generate of d in b. Let O'' be the suffix of O' beginning with instruction (l', t, i); as $d \notin \mathcal{G}(O')$, then $d \notin \mathcal{G}(O'')$. Let $O'_{[l-3,(l-2)^+]}$ be O' restricted to instructions from epochs $[l-3, (l-2)^+]$. Note that O'' is a valid suffix of $O'_{[l-3,(l-2)^+]}$ as well, since $l' \in (l-2)^+$. So $d \notin \mathcal{G}(O'_{[l-3,(l-2)^+]})$. Applying Lemma 39, $d \notin \mathcal{G}_{l-2}$. Therefore, $d \in \mathcal{M}_{l-2}$, which implies $d \in \text{SOS}_l^{\mathcal{M}}$.

Case 3: Holds by symmetry with Case 2.

Case 4: Suppose any of Cases (1), (2) or (3) holds for instructions in epochs $[0, l'^+]$. Then as we have shown in all these cases, $d \in \text{SOS}_{(l'+2)^+}^{\mathcal{M}}$. As in Lemma 26, we can apply that $\forall l''$ s.t. $l-2 \geq l'' > l'$, $d \notin (\mathcal{G}_{l''} \cup \mathcal{K}_{l''})$ to show that $d \in \text{SOS}_{(l'+2)^+}^{\mathcal{M}} - (\mathcal{G}_{l'+1} \cup \mathcal{K}_{l'+1}) \Rightarrow$

$d \in \text{SOS}^{\mathcal{M}}_{l'+3}$; this holds through $d \in \text{SOS}^{\mathcal{M}}_{l-1} - (\mathcal{G}_{l-2} \cup \mathcal{K}_{l-2})$, completing the proof that $d \in \text{SOS}^{\mathcal{M}}_{l}$.

□

5.6.3 Calculating Local State

Useful Ordering Relations

I will define two sets $\mathcal{U}$ and $\mathcal{V}$, and their union $\mathcal{X}$, which partition the subblocks b'' which execute before a subblock $b \in \text{MB}_l$ and either concurrently with or after a predecessor b' of b, $b' \in \text{MB}_{[l-2,l+1]}$:

$$\mathcal{U}(b,b') = \{b''|v(b'') < v(b) \wedge v(b') < v(b'') \wedge b'' \in \text{MB}_{[l-2,l+1]}\}$$

$$\mathcal{V}(b,b') = \{b''|v(b'') < v(b) \wedge v(b') \sim v(b'') \wedge b'' \in \text{MB}_{[l-2,l+1]}\}$$

$$\mathcal{X}(b,b') = \mathcal{U}(b,b') \cup \mathcal{V}(b,b') = \{b''|v(b'') < v(b) \wedge v(b'') \not< v(b') \wedge b'' \in \text{MB}_{[l-2,l+1]}\}$$

For a given subblock b and immediate predecessor $b' \in \text{pred}(b)$, b'' is in $\mathcal{U}(b,b')$ if and only if b'' executes strictly before b and strictly after b'. Likewise, $b'' \in \mathcal{V}(b,b')$ if and only if b'' executes strictly before b and but concurrently with b'. $\mathcal{X}(b,b')$ captures the set of all subblocks b'' which execute before b and which do not execute before b'.

In and Out: Subblock Level

We can now present the equations for OUT^0_b, IN^0_b, OUT^1_b, IN^1_b, OUT^ψ_b and IN^ψ_b. As in Chrysalis Analysis, the meet operator($\bigsqcap$) is union ($\bigcup$).

$$\textsc{Out}_b^0 = \mathcal{K}_b \cup (\textsc{In}_b^0 - (\mathcal{G}_b \cup \mathcal{M}_b))$$

$$\begin{aligned}\textsc{In}_b^0 &= \bigsqcap_{b' \in \text{pred}(b)} \left[\textsc{Out}_{b'}^0 - \bigcup_{b'' \in \mathcal{X}(b,b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right] \\ &= \bigsqcap_{b' \in \text{pred}(b)} \left[\left(\textsc{Out}_{b'}^0 - \bigcup_{b'' \in \mathcal{U}(b,b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right) - \left(\bigcup_{b'' \in \mathcal{V}(b,b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right) \right]\end{aligned}$$

Note that $\textsc{In}_b^0$ represents the set of definitions that must be killed prior to executing b. The may-kill set is slightly different: $\bigsqcap_{b' \in \text{pred}(b)} \left[\textsc{Out}_{b'}^0 - \bigcup_{b'' \in \mathcal{U}(b,b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right]$. The difference between the must-kill and may-kill sets will again be folded into our uncertain formulation, and this difference can be represented as:

$$\bigsqcap_{b' \in \text{pred}(b)} \left[\left(\textsc{Out}_{b'}^0 - \bigcup_{b'' \in \mathcal{U}(b,b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right) \cap \left(\bigcup_{b'' \in \mathcal{V}(b,b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right) \right]$$

We can likewise define $\textsc{Out}_b^1$ and $\textsc{In}_b^1$.

$$\textsc{Out}_b^1 = \mathcal{G}_b \cup (\textsc{In}_b^1 - (\mathcal{K}_b \cup \mathcal{M}_b))$$

$$\begin{aligned}\textsc{In}_b^1 &= \bigsqcap_{b' \in \text{pred}(b)} \left[\textsc{Out}_{b'}^1 - \bigcup_{b'' \in \mathcal{X}(b,b')} (\mathcal{K}_{b''} \cup \mathcal{M}_{b''}) \right] \\ &= \bigsqcap_{b' \in \text{pred}(b)} \left[\left(\textsc{Out}_{b'}^1 - \bigcup_{b'' \in \mathcal{U}(b,b')} (\mathcal{K}_{b''} \cup \mathcal{M}_{b''}) \right) - \left(\bigcup_{b'' \in \mathcal{V}(b,b')} (\mathcal{K}_{b''} \cup \mathcal{M}_{b''}) \right) \right]\end{aligned}$$

Let $\textsc{Kill-In}_b = \bigcup_{\left\{\hat{b} | \hat{b} \in \text{MB}_{[l-1^{\mp}, l+1]} \wedge v(\hat{b}) < v(b)\right\}} \mathcal{K}_{\hat{b}} - \bigcup_{\left\{b'' \in \mathcal{X}(b,\hat{b})\right\}} \mathcal{G}_{b''} \cup \mathcal{M}_{b''}$. We will show that $\textsc{In}_b^0$ and $\textsc{Kill-In}_b$ are equivalent, which aids in later proofs. While $\textsc{In}_b^0$ resembles the dataflow equations introduced in Butterfly and Chrysalis Analyses, $\textsc{Kill-In}_b$ provides a much more amenable for proving invariants.

Lemma 44. $\textsc{In}_b^0 = \textsc{Kill-In}_b$.

The proof will require proving that $\mathcal{X}(b,\hat{b}) = \mathcal{X}(b,b') \cup \mathcal{X}(b',\hat{b})$ when $v(\hat{b}) < v(b') < v(b)$.

$$\begin{aligned}
\mathcal{X}(b,\hat{b}) &= \{b''|v(b'') < v(b) \wedge v(b'') \not< v(\hat{b})\} \\
&= \{b''|v(b'') < v(b) \wedge v(b'') \not< v(\hat{b}) \wedge v(b'') < v(b')\} \\
&\quad \cup \{b''|v(b'') < v(b) \wedge v(b'') \not< v(\hat{b}) \wedge v(b'') \not< v(b')\} \\
v(b') < v(b) \wedge v(b'') < v(b') &\Rightarrow v(b'') < v(b) \\
v(\hat{b}) < v(b') \wedge v(b'') \not< v(b') &\Rightarrow v(b'') \not< v(\hat{b}) \\
&= \{b''|v(b'') \not< v(\hat{b}) \wedge v(b'') < v(b')\} \cup \{b''|v(b'') < v(b) \wedge v(b'') \not< v(b')\} \\
&= \mathcal{X}(b',\hat{b}) \cup \mathcal{X}(b,b')
\end{aligned}$$

Proof of Lemma 44 by Induction. Induction will proceed based on the number of predecessor subblocks (not just immediate predecessors) in the window.

Base case: If there are no predecessors, then $\text{IN}_b^0 = \text{KILL-IN}_b = \varnothing$ trivially. With one predecessor subblock b' of b, then $\text{IN}_b^0 = \mathcal{K}_{b'} - \bigcup_{\{b'' \in \mathcal{X}(b,b')\}} \mathcal{G}_{b''} \cup \mathcal{M}_{b''} = \text{KILL-IN}_b$ because $\bigcup_{\{b'' \in \mathcal{X}(b,b')\}} = \varnothing$.

For the inductive hypothesis, we will assume the statement is true for $k < n$ predecessors. As the inductive step, we will show it holds if b has n predecessors.

We begin with the definition of IN_b^0, and expand it using the definition of $\text{OUT}_{b'}^0$ for an immediate predecessor b' of b:

$$\begin{aligned}
\text{IN}_b^0 &= \bigsqcap_{b' \in \text{pred}(b)} \left[\text{OUT}_{b'}^0 - \bigcup_{b'' \in \mathcal{X}(b,b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right] \\
&= \left[\bigcup_{b' \in \text{pred}(b)} \mathcal{K}_{b'} - \bigcup_{b'' \in \mathcal{X}(b,b')} \mathcal{G}_{b''} \cup \mathcal{M}_{b''} \right] \cup \left[\bigcup_{b' \in \text{pred}(b)} \text{IN}_{b'}^0 - (\mathcal{G}_{b'} \cup \mathcal{M}_{b'} \cup \bigcup_{b'' \in \mathcal{X}(b,b')} \mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right]
\end{aligned}$$

We can apply the inductive hypothesis to $\text{IN}^0_{b'}$:

$$= \left[\bigcup_{b' \in \text{pred}(b)} \mathcal{K}_{b'} - \bigcup_{b'' \in \mathcal{X}(b,b')} \mathcal{G}_{b''} \cup \mathcal{M}_{b''} \right] \cup \left[\bigcup_{b' \in \text{pred}(b)} \text{KILL-IN}_{b'} - (\mathcal{G}_{b'} \cup \mathcal{M}_{b'} \cup \bigcup_{b'' \in \mathcal{X}(b,b')} \mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right]$$

Now, we focus on expanding the right side term:

$$\bigcup_{b' \in \text{pred}(b)} \text{KILL-IN}_{b'} - (\mathcal{G}_{b'} \cup \mathcal{M}_{b'} \cup \bigcup_{b'' \in \mathcal{X}(b,b')} \mathcal{G}_{b''} \cup \mathcal{M}_{b''})$$

$$= \bigcup_{b' \in \text{pred}(b)} \left[\bigcup_{\{\hat{b} | \hat{b} \in \text{MB}_{[l-1,(l+1)^+]} \wedge v(\hat{b}) < v(b')\}} \mathcal{K}_{\hat{b}} - \bigcup_{b'' \in \mathcal{X}(b',\hat{b}} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right] - \left(\mathcal{G}_{b'} \cup \mathcal{M}_{b'} \cup \bigcup_{b'' \in \mathcal{X}(b,b')} \mathcal{G}_{b''} \cup \mathcal{M}_{b''} \right)$$

$$= \bigcup_{b' \in \text{pred}(b)} \left[\bigcup_{\{\hat{b} | \hat{b} \in \text{MB}_{[l-1^{\mp},(l+1)^+]} \wedge v(\hat{b}) < v(b')\}} \mathcal{K}_{\hat{b}} - \left(\mathcal{G}_{b'} \cup \mathcal{M}_{b'} \bigcup_{b'' \in \mathcal{X}(b',\hat{b}) \cup X(b,b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right) \right]$$

Apply: when $v(\hat{b}) < v(b') < v(b), \mathcal{X}(b,b') \cup X(b',\hat{b}) = \mathcal{X}(b,\hat{b})$ – shown earlier

$$= \bigcup_{b' \in \text{pred}(b)} \left[\bigcup_{\{\hat{b} | \hat{b} \in \text{MB}_{[l-1^{\mp},(l+1)^+]} \wedge v(\hat{b}) < v(b')\}} \mathcal{K}_{\hat{b}} - \left(\mathcal{G}_{b'} \cup \mathcal{M}_{b'} \bigcup_{b'' \in \mathcal{X}(b,\hat{b})} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right) \right]$$

Since $v(\hat{b}) < v(b') < v(b), b' \in \mathcal{X}(b,\hat{b})$

$$= \bigcup_{b' \in \text{pred}(b)} \left[\bigcup_{\{\hat{b} | \hat{b} \in \text{MB}_{[l-1^{\mp},(l+1)^+]} \wedge v(\hat{b}) < v(b')\}} \mathcal{K}_{\hat{b}} - \left(\bigcup_{b'' \in \mathcal{X}(b,\hat{b})} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right) \right]$$

We can return to the equation for IN_b^0:

$$= \left[\bigcup_{b' \in \text{pred}(b)} \mathcal{K}_{b'} - \bigcup_{b'' \in \mathcal{X}(b,b')} \mathcal{G}_{b''} \cup \mathcal{M}_{b''} \right] \cup \left[\bigcup_{b' \in \text{pred}(b)} \text{KILL-IN}_{b'} - (\mathcal{G}_{b'} \cup \mathcal{M}_{b'} \cup \bigcup_{b'' \in \mathcal{X}(b,b')} \mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right]$$

$$= \left[\bigcup_{b' \in \text{pred}(b)} \mathcal{K}_{b'} - \bigcup_{b'' \in \mathcal{X}(b,b')} \mathcal{G}_{b''} \cup \mathcal{M}_{b''} \right] \cup$$

$$\left[\bigcup_{b' \in \text{pred}(b)} \left[\bigcup_{\{\hat{b} | \hat{b} \in \text{MB}_{[l-1^{\mp},(l+1)^+]} \wedge v(\hat{b}) < v(b')\}} \mathcal{K}_{\hat{b}} - \left(\bigcup_{b'' \in \mathcal{X}(b,\hat{b})} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right) \right] \right]$$

$$= \bigcup_{b' \in \text{pred}(b)} \left[\bigcup_{\{\hat{b} | \hat{b} \in \text{MB}_{[l-1^{\mp},(l+1)^+]} \wedge (\hat{b} = b' \vee v(\hat{b}) < v(b'))\}} \mathcal{K}_{\hat{b}} - \left(\bigcup_{b'' \in \mathcal{X}(b,\hat{b})} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right) \right]$$

$$= \bigcup_{\{\hat{b} | \hat{b} \in \text{MB}_{[l-1^{\mp},(l+1)^+]} \wedge \exists b' \in \text{pred}(b) \text{ s.t. } (\hat{b} = b' \vee v(\hat{b}) < v(b'))\}} \mathcal{K}_{\hat{b}} - \left(\bigcup_{b'' \in \mathcal{X}(b,\hat{b})} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right)$$

Note: $\{\hat{b} | \hat{b} \in \text{MB}_{[l-1^{\mp},(l+1)^+]} \wedge \exists b' \in \text{pred}(b) \text{ s.t. } (\hat{b} = b' \vee v(\hat{b}) < v(b'))\} =$

$$\{\hat{b} | \hat{b} \in \text{MB}_{[l-1^{\mp},(l+1)^+]} \text{ s.t. } v(\hat{b}) < v(b)\}$$

$$= \bigcup_{\{\hat{b} | \hat{b} \in \text{MB}_{[l-1^{\mp},(l+1)^+]} \text{ s.t. } v(\hat{b}) < v(b)\}} \mathcal{K}_{\hat{b}} - \left(\bigcup_{b'' \in \mathcal{X}(b,\hat{b})} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right)$$

$= \text{KILL-IN}_b$.

□

Lemma 45. *If $d \in \text{IN}_b^0$ then $\forall$ valid vector orderings O of instructions in subblocks $\tilde{b} \in \text{MB}_{[l-2,l+1]}$, such that $v(\tilde{b}) < v(b)$, $d \in \mathcal{K}(O)$.*

Proof. If $d \in \text{IN}_b^0$ then $d \in \text{KILL-IN}_b$ which implies that $\exists \hat{b}$ where $v(\hat{b}) < v(b)$ and $\hat{b} \in \text{MB}_{[l-1^{\mp},l+1]}$ such that $d \in \mathcal{K}_{\hat{b}} - \bigcup_{b'' \in \mathcal{X}(b,\hat{b})} \mathcal{G}_{b''} \cup \mathcal{M}_{b''}$. In other words, there exists some maximal subblock $\hat{b}$ such that all subblocks b'' which are concurrent with $\hat{b}$ or after $\hat{b}$ and also before b do not have $d \in \mathcal{G}_{b''} \cup \mathcal{M}_{b''}$. Let (l', t', i') be the last kill of d in $\hat{b}$. For any VVO O, consider the suffix O_s beginning with (l', t', i'). The only other instructions which follow (l', t', i') must

belong to a subblock b'' which both execute before b and after or concurrent with $\hat{b}$. Furthermore, none of them have $d \in \mathcal{G}_{b''} \cup \mathcal{M}_{b''}$ so any instruction that generates or marks d as uncertain must be followed within the same subblock with a kill of d. Thus, $d \in \mathcal{K}(O_s) \Rightarrow d \in \mathcal{K}(O)$. □

Lemma 46. *If* $d \in \text{IN}_b^1$ *then* $\forall$ *valid vector orderings* O *of instructions in subblocks* $\tilde{b} \in \text{MB}_{[l-2,l+1]}$ *such that* $v(\tilde{b}) < v(b)$, $d \in \mathcal{G}(O)$.

Proof. Follows by symmetry to Lemma 45. □

Finally, we define OUT_b^ψ and IN_b^ψ. Recall that $\mathcal{X}(b,b') = \mathcal{U}(b,b') \cup \mathcal{V}(b,b')$. As before, OUT_b^ψ resembles OUT_b^1 and OUT_b^0:

Likewise, IN_b^ψ is quite similar to its counterpart $\mathcal{M}_l$, which summarizes the effect of an extended epoch l^+:

$$
\begin{aligned}
\text{OUT}_b^\psi =& \mathcal{M}_b \cup (\text{IN}_b^\psi - (\mathcal{K}_b \cup \mathcal{G}_b)) \\
\text{IN}_b^\psi =& \bigsqcap_{b' \in \text{pred}(b)} \left[\text{OUT}_{b'}^\psi - \bigcup_{b'' \in \mathcal{U}(b,b')} (\mathcal{K}_{b''} \cup \mathcal{G}_{b''}) \right] \\
& \cup \left[\left(\text{OUT}_{b'}^0 - \bigcup_{b'' \in \mathcal{U}(b,b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right) \bigcap_{b'' \in \mathcal{V}(b,b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right] \\
& \cup \left[\left(\text{OUT}_{b'}^1 - \bigcup_{b'' \in \mathcal{U}(b,b')} (\mathcal{K}_{b''} \cup \mathcal{M}_{b''}) \right) \bigcap_{b'' \in \mathcal{V}(b,b')} (\mathcal{K}_{b''} \cup \mathcal{M}_{b''}) \right] \\
& - \left(\text{IN}_b^1 \cup -\text{IN}_b^0 \right)
\end{aligned}
$$

Lemma 47. *If at least one of the following three conditions holds:*

(1) $\exists b' \in pred(b)$ *such that* $d \in \text{OUT}_{b'}^\psi$ *and for all* $b'' \in \mathcal{U}(b,b')$, $d \notin (\mathcal{K}_{b''} - \mathcal{G}_{b''})$ OR

(2) $\exists$ *valid vector orderings* O, O' *of instructions in* $\dot{b} \in \text{MB}_{[(l-1)^\mp, l+1]}$ *where* $v(\dot{b}) < v(b)$ *such that* $d \in \mathcal{K}(O)$ *but* $d \notin \mathcal{K}(O')$, *with the last kill of* d *in* O *occurring in subblock* $\hat{b} \in \text{MB}_{[(l-1)^\mp, l+1]}$ OR

(3) $\exists$ *valid vector orderings* O, O' *of instructions in* $\dot{b} \in \text{MB}_{[(l-1)^{\mp}, l+1]}$ *where* $v(\dot{b}) < v(b)$ *such that* $d \in \mathcal{G}(O)$ *but* $d \notin \mathcal{G}(O')$*, with the last gen of* d *in* O *occurring in subblock* $\hat{b} \in \text{MB}_{[(l-1)^{\mp}, l+1]}$

then $d \in \text{IN}_b^{\psi}$.

Proof. We consider each case in turn.

Case (1) This corresponds trivially with part of the definition.

Case (2) It will suffice to show the following three facts:

- $d \notin \text{IN}_b^0$: By the contrapositive of Lemma 45.
- $d \notin \text{IN}_b^1$: As $d \in \mathcal{K}_{\hat{b}} - \bigcup_{b'' \in \mathcal{U}(b,\hat{b})}(\mathcal{G}_{b''} \cup \mathcal{M}_{b''})$, no subblock which occurs after $\hat{b}$ generates or marks d uncertain. Therefore, any $\tilde{b} \in \text{MB}_{[(l-1)^{\mp}, l+1]}$ such that $v(\tilde{b}) < v(b)$, either occurs before or concurrently with $\hat{b}$; by definition, then, $d \notin \text{IN}_b^1$.
- $\exists b' \in \text{pred}(b)$ s.t. either $d \in \text{OUT}_{b'}^{\psi} - \bigcup_{b'' \in \mathcal{U}(b,b')}(\mathcal{G}_{b''} \cup \mathcal{K}_{b''})$ or $d \in (\text{OUT}_{b'}^0 - \bigcup_{b'' \in \mathcal{U}(b,b')}(\mathcal{G}_{b''} \cup \mathcal{M}_{b''})) \bigcap_{b'' \in \mathcal{V}(b,b')}(\mathcal{G}_{b''} \cup \mathcal{M}_{b''})$: Proof follows.

We will show inductively that in this situation, $\exists b' \in \text{pred}(b)$ such that either $d \in \text{OUT}_{b'}^{\psi} - \bigcup_{b'' \in \mathcal{U}(b,b')}(\mathcal{G}_{b''} \cup \mathcal{K}_{b''})$ or $d \in \left(\text{OUT}_{b'}^0 - \bigcup_{b'' \in \mathcal{U}(b,b')}(\mathcal{G}_{b''} \cup \mathcal{M}_{b''})\right) \bigcap_{b'' \in \mathcal{V}(b,b')}(\mathcal{G}_{b''} \cup \mathcal{M}_{b''})$, which will imply $d \in \text{IN}_b^{\psi}$.

Base case: Two immediate predecessors (necessarily unordered), one kills d and the other either marks d uncertain or generates d. Then:

$d \in \left(\text{OUT}_{\hat{b}}^0 - \bigcup_{b'' \in \mathcal{U}(b,\hat{b})}(\mathcal{G}_{b''} \cup \mathcal{M}_{b''})\right) \bigcap_{b'' \in \mathcal{V}(b,\hat{b})}(\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \Rightarrow d \in \text{IN}_b^{\psi}$.

For our inductive hypothesis, we will assume this holds for $k < n$ predecessors (not necessarily immediate) and show it holds for $k = n$.

Since $d \in \mathcal{K}(O)$, with the last kill of d in O occurring in $\hat{b}$, it must be the case that $d \in \mathcal{K}_{\hat{b}} - \bigcup_{b'' \in \mathcal{U}(b,\hat{b})}(\mathcal{G}_{b''} \cup \mathcal{M}_{b''})$. This follows by definition of valid vector ordering. Further, there cannot be a b'' such that $v(\hat{b}) < v(b'')$ where any instruction in b'' kills d; this would contradict the last kill of d in O occurring in $\hat{b}$.

If $\hat{b} \in \text{pred}(b)$, then:

$d \in \left(\text{OUT}^0_{\hat{b}} - \bigcup_{b'' \in \mathcal{U}(b,\hat{b})}(\mathcal{G}_{b''} \cup \mathcal{M}_{b''})\right) \bigcap_{b'' \in \mathcal{V}(b,\hat{b})}(\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \Rightarrow d \in \text{IN}^{\psi}_{b}$.

If $\hat{b} \notin \text{pred}(b)$, then $\exists b', \tilde{b}$ such that $b' \in \text{pred}(b)$, $v(\hat{b}) < v(b')$ and $v(\tilde{b}) \sim v(\hat{b})$ s.t. $d \in (\mathcal{G}_{\tilde{b}} \cup \mathcal{M}_{\tilde{b}})$. Then **(1)** $\exists \tilde{b} \in \mathcal{V}(b', \hat{b})$ or **(2)** $\forall$ such $\tilde{b}$, $\tilde{b} \in \mathcal{V}(b, b')$. In other words, as $v(\tilde{b}) < v(b)$ but $\tilde{b} \notin \text{pred}(b)$, there must be an immediate predecessor b' of b such that $v(\hat{b}) < v(b')$. As $\tilde{b}$ is concurrent with $\hat{b}$ and both occur before b, then either **(1)** at least one such $\tilde{b}$ occurs strictly before b' or **(2)** there exists at least one such $\tilde{b}$ and all that exist are concurrent with b'.

(1): Since both $\hat{b}$ and $\tilde{b}$ occur before b', we can apply the inductive hypothesis for b' to conclude $d \in \text{IN}^{\psi}_{b'}$. Since $v(\hat{b}) < v(b')$, we know that $d \notin \mathcal{G}_{b'} \cup \mathcal{M}_{b'}$. Furthermore, no subblock strictly after $\hat{b}$ kills d, and we can apply transitivity to show that $d \notin \mathcal{K}_{b'}$. Therefore, $d \in \text{OUT}^0_{b'} = \mathcal{M}_{b'} \cup (\text{IN}^{\psi}_{b'} - (\mathcal{G}_{b'} \cup \mathcal{K}_{b'})$, and furthermore, any subblock after b' is necessarily after $\hat{b}$; by construction, nothing later generates or kills d; thus, $d \in \text{OUT}^{\psi}_{b'} - \bigcup_{b'' \in \mathcal{U}(b,b')}(\mathcal{G}_{b''} \cup \mathcal{K}_{b''}) \Rightarrow d \in \text{IN}^{\psi}_{b}$.

(2): We will begin by expanding the term $\left(\text{OUT}^0_{\hat{b}} - \bigcup_{b'' \in \mathcal{U}(b,\hat{b})}(\mathcal{G}_{b''} \cup \mathcal{M}_{b''})\right) \bigcap_{b'' \in \mathcal{V}(b,\hat{b})}(\mathcal{G}_{b''} \cup \mathcal{M}_{b''})$ in the definition of IN^{ψ}_{b}.

First, we apply the definition of $\text{OUT}^0_{b'}$:

$$\left(\text{OUT}^0_{b'} - \bigcup_{b'' \in \mathcal{U}(b,b')}(\mathcal{G}_{b''} \cup \mathcal{M}_{b''})\right) \bigcap_{b'' \in \mathcal{V}(b,b')} \left(\mathcal{G}_{b''} \cup \mathcal{M}_{b''}\right)$$
$$= \left[\left(\left(K_{b'} \cup (\text{IN}^0_{b'} - \mathcal{G}_{b'} \cup \mathcal{M}_{b'})\right) - \bigcup_{b'' \in \mathcal{U}(b,b')}(\mathcal{G}_{b''} \cup \mathcal{M}_{b''})\right)\right] \bigcap_{b'' \in \mathcal{V}(b,b')}(\mathcal{G}_{b''} \cup \mathcal{M}_{b''})$$

Then, we regroup terms:

$$= \left[\left(K_{b'} - \bigcup_{b'' \in \mathcal{U}(b,b')}(\mathcal{G}_{b''} \cup \mathcal{M}_{b''})\right) \cup \left(\left(\text{IN}^0_{b'} - (\mathcal{G}_{b'} \cup \mathcal{M}_{b'})\right) - \bigcup_{b'' \in \mathcal{U}(b,b')}(\mathcal{G}_{b''} \cup \mathcal{M}_{b''})\right)\right]$$
$$\bigcap_{b'' \in \mathcal{V}(b,b')}(\mathcal{G}_{b''} \cup \mathcal{M}_{b''})$$

We apply Lemma 44 to substitute $\textsc{Kill-In}_{b'}$ for $\textsc{In}^0_{b'}$:

$$= \left[\left(K_{b'} - \bigcup_{b'' \in \mathcal{U}(b,b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right) \cup \left((\textsc{Kill-In}_{b'} - (\mathcal{G}_{b'} \cup \mathcal{M}_{b'})) - \bigcup_{b'' \in \mathcal{U}(b,b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right) \right] \bigcap_{b'' \in \mathcal{V}(b,b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''})$$

$$= \Bigg[\left(K_{b'} - \bigcup_{b'' \in \mathcal{U}(b,b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right) \cup \left(\left(\bigcup_{\{\hat{b} | v(\hat{b}) < v(b')\}} \mathcal{K}_{\hat{b}} - \bigcup_{b'' \in \mathcal{X}(b',\hat{b})} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right) - \bigcup_{b'' \in \mathcal{U}(b,b') \cup \{b'\}} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right) \Bigg] \bigcap_{b'' \in \mathcal{V}(b,b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''})$$

We again regroup terms, and note that $b' \in \mathcal{X}(b', \hat{b})$ and $\mathcal{U}(b, b') \subseteq \mathcal{U}(b, \hat{b}) \subseteq \mathcal{X}(b, \hat{b})$:

$$= \left[\left(K_{b'} - \bigcup_{b'' \in \mathcal{U}(b,b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right) \cup \left(\bigcup_{\{\hat{b} | v(\hat{b}) < v(b')\}} \mathcal{K}_{\hat{b}} - \bigcup_{b'' \in \mathcal{X}(b',\hat{b}) \cup \{b'\} \cup \mathcal{U}(b,b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right) \right] \bigcap_{b'' \in \mathcal{V}(b,b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''})$$

It will suffice to show that:

$$d \in \left(\bigcup_{\{\hat{b} | v(\hat{b}) < v(b')\}} \mathcal{K}_{\hat{b}} - \bigcup_{b'' \in \mathcal{X}(b',\hat{b}) \cup \{b'\} \cup \mathcal{U}(b,b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right) \bigcap_{b'' \in \mathcal{V}(b,b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''})$$

Now by construction, $\tilde{b} \in \mathcal{V}(b, b')$, so $d \in \mathcal{G}_{b''} \cup \mathcal{M}_{b''}$ (right hand side).

We know that $\tilde{b} \neq b'$ as otherwise $v(\hat{b}) < v(\tilde{b})$; likewise, if $\tilde{b} \in \mathcal{V}(b, b')$, then $\tilde{b} \notin \mathcal{V}(b, b')$. We can decompose $\mathcal{X}(b', \hat{b}) = \mathcal{U}(b', \hat{b}) \cup \mathcal{V}(b', \hat{b})$. We know that $\tilde{b}$ does not occur strictly after $\hat{b}$, so $\tilde{b} \notin \mathcal{U}(b', \hat{b})$. Furthermore, $\forall$ such blocks $\tilde{b}$ (where $d \in$

$(\mathcal{G}_{\tilde{b}} \cup \mathcal{M}_{\tilde{b}}))$, $\tilde{b} \notin \mathcal{V}(b', \hat{b})$. Putting this together, $\tilde{b} \notin \mathcal{X}(b', \hat{b})$, which implies $d \in \mathcal{K}_{\hat{b}} - \bigcup_{b'' \in \mathcal{X}(b', \hat{b}) \cup \{b'\} \cup \mathcal{U}(b, b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''})$ (left hand side).

Therefore, $d \in \left(\bigcup_{\{\hat{b} | v(\hat{b}) < v(b')\}} \mathcal{K}_{\hat{b}} - \bigcup_{b'' \in \mathcal{X}(b', \hat{b}) \cup \{b'\} \cup \mathcal{U}(b, b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right) \bigcap_{b'' \in \mathcal{V}(b, b')} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''})$
$\Rightarrow d \in \left(\text{OUT}^0_{\hat{b}} - \bigcup_{b'' \in \mathcal{U}(b, \hat{b})} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \right) \bigcap_{b'' \in \mathcal{V}(b, \hat{b})} (\mathcal{G}_{b''} \cup \mathcal{M}_{b''}) \Rightarrow d \in \text{IN}^{\psi}_{b}$.

Case (3) By symmetry with Case 2.

□

LSOS equations

We begin by presenting the equations for $\text{LSOS}^{\mathcal{G}}_b$, $\text{LSOS}^{\mathcal{K}}_b$ and $\text{LSOS}^{\mathcal{M}}_b$.

$$\mathcal{K}^*_b = \begin{cases} K_b & b \in \text{MB}_{[l,l+1]} \\ K_b - \bigcup_{\left\{ \substack{\hat{b} | [v(\hat{b}) \sim v(b) \vee v(b) < v(\hat{b})], \\ b \in \text{MB}_{l-1}, \hat{b} \in \text{MB}_{l-2}} \right\}} (\mathcal{G}_{\hat{b}} \cup \mathcal{M}_{\hat{b}}) & b \in \text{MB}_{l-1} \end{cases}$$

$$\mathcal{G}^*_b = \begin{cases} G_b & b \in \text{MB}_{[l,l+1]} \\ G_b - \bigcup_{\left\{ \substack{\hat{b} | [v(\hat{b}) \sim v(b) \vee v(b) < v(\hat{b})], \\ b \in \text{MB}_{l-1}, \hat{b} \in \text{MB}_{l-2}} \right\}} (\mathcal{K}_{\hat{b}} \cup \mathcal{M}_{\hat{b}}) & b \in \text{MB}_{l-1} \end{cases}$$

$$\mathcal{M}^*_b = \begin{cases} M_b & b \in \text{MB}_{[l,l+1]} \\ M_b \cup \left(\mathcal{K}_b - \bigcap_{\left\{ \substack{\hat{b} | [v(\hat{b}) \sim v(b))], \\ b \in \text{MB}_{l-1}, \hat{b} \in \text{MB}_{l-2}} \right\}} (\mathcal{G}_{\hat{b}} \cup \mathcal{M}_{\hat{b}}) \right) \cup \left(\mathcal{G}_b - \bigcap_{\left\{ \substack{\hat{b} | [v(\hat{b}) \sim v(b)], \\ b \in \text{MB}_{l-1}, \hat{b} \in \text{MB}_{l-2}} \right\}} (\mathcal{K}_{\hat{b}} \cup \mathcal{M}_{\hat{b}}) \right) & b \in \text{MB}_{l-1} \end{cases}$$

$$\text{LSOS}^{\mathcal{G}}_b = \text{IN}^1_b \cup (\text{SOS}^{\mathcal{G}}_l - (\bigcup_{\left\{ \substack{b' | v(b') < v(b) \\ b' \in \text{MB}_{[l-1,l+1]}} \right\}} (\mathcal{K}^*_{b'} \cup \mathcal{M}^*_{b'})))$$

$$\text{LSOS}_b^{\mathcal{K}} = \text{IN}_b^0 \cup (\text{SOS}_l^{\mathcal{K}} - (\bigcup_{\left\{\substack{b'|v(b')<v(b)\\ b'\in \text{MB}_{[l-1,l+1]}}\right\}} (\mathcal{G}_{b'}^* \cup \mathcal{M}_{b'}^*)))$$

$$\text{LSOS}_b^{\mathcal{M}} = \text{IN}_b^{\psi} \cup (\text{SOS}_l^{\mathcal{M}} - (\bigcup_{\left\{\substack{b'|v(b')<v(b)\\ b'\in \text{MB}_{[l-1,l+1]}}\right\}} (\mathcal{G}_{b'}^* \cup \mathcal{K}_{b'}^*)))$$

Lemma 48. *If $d \in$ LSOS$_b^{\mathcal{K}}$ then $\forall$ valid vector orderings O of instructions in epochs $[0, l-2]$ and subblocks $\tilde{b} \in \text{MB}_{[l-1,l+1]}$ such that $v(\tilde{b}) < v(b)$, $d \in \mathcal{K}(O)$.*

Proof. If $d \in \text{LSOS}^{\mathcal{K}}$, then one of two conditions hold. Either $d \in \text{IN}_b^0$ or $d \in \text{SOS}_l^{\mathcal{K}} - \bigcup_{\left\{\substack{b'|v(b')<v(b)\\ b'\in \text{MB}_{[l-1,l+1]}}\right\}} (\mathcal{G}_{b'}^* \cup \mathcal{M}_{b'}^*)$.

Case (1): $d \in \text{IN}_b^0$: For any O of instructions in epochs $[0, l-2]$ and in subblocks $\tilde{b} \in \text{MB}_{[l-1,l+1]}$ such that $v(\tilde{b}) < v(b)$, let $O_{[l-2,l+1]}$ be the restriction of O to instructions in $l-2$ or later. By Lemma 45, $d \in \mathcal{K}(O_{[l-2,l+1]})$. Furthermore, there must be a kill beginning in epoch $l-1$ or later which is the last kill of d in $O_{[l-2,l+1]}$ (by definition of $d \in \text{IN}_b^0$, $\exists b' \in \text{pred}(b)$ such that $d \in \text{OUT}_{b'}^0$, and unwinding the recursion requires that the kill be within the $[l-1, l+1]$ window). Let (l', t', i) be the last instruction which kills d, and let O^S be the suffix of O beginning with instruction (l', t', i). By definition, $d \in K(O^S)$, as all instructions in O^S must come from epochs $[l-2, l+1]$ and thus O^S is a suffix of $O_{[l-2,l+1]}$. $d \in \mathcal{K}(O^S) \Rightarrow d \in \mathcal{K}(O)$.

Case (2): $d \in \text{SOS}_l^{\mathcal{K}} - (\bigcup_{\left\{\substack{b'|v(b')<v(b)\\ b'\in \text{MB}_{[l-1,l+1]}}\right\}} (\mathcal{G}_{b'}^* \cup \mathcal{M}_{b'}^*))$:

I will motivate the proof by assuming that $\mathcal{G}_{b'}^*$ is identically equal to $\mathcal{G}_{b'}$ (likewise, $\mathcal{M}_{b'}^*$ and $M_{b'}$), and then show that the proof still holds with the more complicated definitions given.

Consider any valid vector ordering O. If $d \in \text{SOS}_l^{\mathcal{K}}$ then for all VVO O' of instructions in epochs $[0, (l-2)^+]$, $d \in \mathcal{K}(O')$ – this follows by Lemma 41. Let O_s be the subsequence remaining when all instructions in $[0, l-2]$ are removed from O. Show-

ing $d \notin \mathcal{G}(O_s)$ and $d \notin \mathcal{M}(O_s)$ is sufficient to complete the proof, as the interleaving (maintaining order) of O' and O_s would then necessarily kill d.

Intuition: Suppose $d \in \mathcal{G}(O_s)$. Then there must exist some instruction $(\tilde{l}, \tilde{t}, \tilde{i})$ which generates d belonging to subblock $\tilde{b}$ such that it is also true that $d \in \mathcal{G}_{\tilde{b}}$ and $v(\tilde{b}) < v(b)$, $\tilde{b} \in \text{MB}_{[l-1,l+1]}$. But then $d \in \mathcal{G}^*_{\tilde{b}}$, contradicting $d \in \text{SOS}^{\mathcal{K}}_l - (\bigcup_{\substack{b'|v(b')<v(b) \\ b' \in \text{MB}_{[l-1,l+1]}}} (\mathcal{G}^*_{b'} \cup \mathcal{M}^*_{b'}))$. The same intuition holds for $d \in \mathcal{M}(O_s)$.

More formally: If $d \in \mathcal{G}_{\tilde{b}}$ and $v(\tilde{b}) < v(b)$, then there are two cases. If $\tilde{b} \in \text{MB}_{[l,l+1]}$, the intuition can be followed exactly. If, however, $d \in \text{MB}_{l-1}$, then interactions with epoch $l-2$ must be taken into account. The deviations are as follows:

(a) $\exists b' \in \text{MB}_{l-2}$ such that $v(b') \sim v(\tilde{b})$, $d \in \mathcal{G}_{\tilde{b}}$ and $d \in \mathcal{K}_{b'} \cup \mathcal{M}_{b'}$. In essence, we have a gen and a conflicting operation that are concurrent; this should be treated as an "uncertain", and indeed, the formula calls for $d \in \mathcal{M}^*_{\tilde{b}}$. We again reach the contradiction that $d \in \text{SOS}^{\mathcal{K}}_l - (\bigcup_{\substack{b'|v(b')<v(b) \\ b' \in \text{MB}_{[l-1,l+1]}}} (\mathcal{G}^*_{b'} \cup \mathcal{M}^*_{b'}))$.

(b) $\exists b' \in \text{MB}_{l-2}$ such that $v(\tilde{b}) < v(b')$, $d \in \mathcal{G}_{\tilde{b}}$ and $d \in \mathcal{K}_{b'} \cup \mathcal{M}_{b'}$. In this case, $\tilde{b}$ belongs to $(l-2)^+$, and its effects have been included when calculating the $\text{SOS}^{\mathcal{K}}_l$. Since $d \in \text{SOS}^{\mathcal{K}}_l$, we can apply Lemma 41, yielding that the gen must be guaranteed to always be followed by a kill in instructions $[0, (l-2)^+]$; we can safely ignore this gen.

As $\mathcal{M}_b \subseteq \mathcal{M}^*_b$, the intuition holds with minimal changes. This completes the proof.

□

Lemma 49. *If at least one of the following is true:*

(1) $\exists$ *VVO* O *of instructions in epochs* $[0, l-2]$ *and subblocks* $\tilde{b} \in \text{MB}_{[(l-1)^{\mp},l+1]}$ *where* $v(\tilde{b}) < v(b)$ *such that* $d \in \mathcal{M}(O)$*, with the last instruction to mark* d *uncertain occurring in epoch* $l-1$ *or later* ***OR***

(2) $\exists$ *VVO* O, O' *of instructions in epochs* $[0, l-2]$ *and subblocks* $\tilde{b} \in \text{MB}_{[(l-1)^{\mp}, l+1]}$ *where* $v(\tilde{b}) < v(b)$ *such that* $d \in \mathcal{K}(O)$ *and* $d \notin \mathcal{K}(O')$*, with the last instruction to kill* d *in* O *occurring in epoch* $l-1$ *or later* ***OR***

(3) $\exists$ *VVO* O, O' *of instructions in epochs* $[0, l-2]$ *and subblocks* $\tilde{b} \in \text{MB}_{[(l-1)^{\mp}, l+1]}$ *where* $v(\tilde{b}) < v(b)$ *such that* $d \in \mathcal{G}(O)$ *and* $d \notin \mathcal{G}(O')$*, with the last instruction to generate* d *in* O *occurring in epoch* $l-1$ *or later* ***OR***

(4) (Propagation) One of the conditions in Lemma 43 holds and instructions in subblocks $\tilde{b} \in \text{MB}_{[l-1, l+1]}$ *where* $v(\tilde{b}) < v(b)$ *do not generate/kill* d

then $d \in$ ***LSOS***$_b^{\mathcal{M}}$.

Proof. Case (1) Let b' be the last subblock to mark d uncertain in O. Then since O exists, it must be the case $\forall b''$ s.t. $v(b') < v(b''), d \notin \mathcal{G}_{b''} \cup \mathcal{K}_{b''}$. So $d \in \mathcal{M}_{b'} - \bigcup_{b'' \in \mathcal{U}(b,b')} \mathcal{G}_{b''} \cup \mathcal{K}_{b''}$ which implies $d \in \text{OUT}_{b'}^{\psi} - \bigcup_{b'' \in \mathcal{U}(b,b'')} \mathcal{G}_{b''} \cup \mathcal{K}_{b''}$. Note that if no block after b' and before b kills or generates d, d will be in $\text{IN}_{\hat{b}}^{\psi}$ and $\text{OUT}_{\hat{b}}^{\psi}$ for each intermediate block (follow by construction), so in particular, for some immediate predecessor $\tilde{b} \in \text{pred}(b), d \in \text{OUT}_{\tilde{b}} - \bigcup_{b'' \in \mathcal{U}(b,\tilde{b})} \mathcal{G}_{b''} \cup \mathcal{K}_{b''} \Rightarrow d \in \text{IN}_b^{\psi}$ and thus $d \in \text{LSOS}_b^{\mathcal{M}}$.

Case (2) Let $O_{[(l-1)^{\mp}, l^{+}]}$ be O restricted to instructions in epochs $[(l-1)^{\mp}, l^{+}]$ and belonging to $\tilde{b} \in \text{MB}_{[(l-1)^{\mp}, l+1]}$ where $v(\tilde{b}) < v(b)$. Note that by construction, $d \in \mathcal{K}(O_{[(l-1)^{\mp}, l^{+}]})$. Likewise, we can similarly construct $O'_{[(l-1)^{\mp}, l^{+}]}$ by restricting the instructions to the same window and note that $d \notin \mathcal{K}(O'_{[(l-1)^{\mp}, l^{+}]})$.). Furthermore, we know that the last kill of d in O occurs in epoch $l-1$ or later. Thus, we can apply Lemma 47 and conclude $d \in \text{IN}_b^{\psi} \Rightarrow d \in \text{LSOS}^{\mathcal{M}}$.

Case (3) Symmetric to case (2).

Case (4) As Lemma 43 holds, then $d \in \text{SOS}_l^{\mathcal{M}}$. Furthermore, subblocks b' such that $v(b') < v(b)$ do not generate or kill d, so $d \in \text{SOS}_l^{\mathcal{M}} - \left(\bigcup_{\{b' | v(b') < v(b), b' \in \text{MB}_{[l-1, l+1]}\}} (\mathcal{G}_{b'} \cup \mathcal{K}_{b'}) \right) \Rightarrow d \in \text{LSOS}_b^{\mathcal{M}}$.

□

5.7 TaintCheck with Uncertainty in Butterfly Analysis

We base our Chrysalis Analysis formulation of TAINTCHECK incorporating uncertainty on the Butterfly Analysis formulation, described in Section 5.5.

5.7.1 First Pass: Instruction-level Transfer Functions, Subblock Level Transfer Functions and Calculating Side-Out

As before, we define:

$$\mathcal{T}_{l,t,i} = \begin{cases} (x_{l,t,i} \leftarrow \bot) & \text{if } (l,t,i) \equiv \texttt{taint}(x) \\ (x_{l,t,i} \leftarrow \top) & \text{if } (l,t,i) \equiv \texttt{untaint}(x) \\ (x_{l,t,i} \leftarrow \{a\}) & \text{if } (l,t,i) \equiv x := \texttt{unop}(a) \\ (x_{l,t,i} \leftarrow \{a,b\}) & \text{if } (l,t,i) \equiv x := \texttt{binop}(a,b) \end{cases}$$

We use the set S:

$$S = \{\texttt{taint}, \texttt{untaint}, \texttt{uncertain}, \{a\}, \{a,b\} | a, b \text{ are memory locations}\}$$

to represent the set of all possible right-hand values in our mapping. We continue to utilize the function `loc()` that given (l,t,i) returns m, where m is the destination location for instruction (l,t,i).

At the end of the first pass, blocks in the wings will exchange the TRANSFER-SIDE-OUT (TSO) and create the TRANSFER-SIDE-IN (TSI). If $b = (l,t,(i,j))$, then the TRANSFER-SIDE-OUT of the instruction-level transfer functions are:

$$\text{TSO}_b = \bigcup_{i \leq k \leq j} \mathcal{T}_{l,t,k}$$

5.7.2 Between Passes: Calculating Side-In

Likewise, the TRANSFER-SIDE-IN for a maximal subblock b is the union of the TSO of maximal subblocks b' in the wings which are concurrent with b:

$$\text{TSI}_b = \bigcup_{\{b' | v(b') \sim v(b)\}} \text{TSO}_{b'}$$

Despite the notational difference, the TSO and TSI calculated in this section are identical to those in Section 4.2.1, there called GEN-SIDE-OUT$_b$ and GEN-SIDE-IN$_b$.

5.7.3 Resolving Transfer Functions to Metadata

As in Section 5.5.3, we will use a `resolve` algorithm to recursively evaluate instruction-level transfer functions in the wings to metadata values. The TRANSFER and MEET algorithms (Algorithms 3 and 4, respectively) introduced in Section 5.5.3 are unchanged. The `resolve` algorithm, shown in Algorithm 7, as well as the `do_resolve` algorithm shown in Algorithm 8, are modified to take into account the vector clock vc associated with the maximal subblock b which contains instruction (l, t, i).

There are two changes to the algorithm. The first change is to the definition of *proper predecessor* for $x_{l,t,i} \leftarrow s$, which is again any $y_{l',t',i'} \leftarrow s'$ such that $\texttt{loc}(l', t', i') \in s$ and $s \in S$; in addition, we now enforce that (l', t', i') executing before (l, t, i) does not violate any valid vector ordering rules (as compared to valid ordering) of prior instructions in H. We represent the new proper predecessor relation as $P(m, (l, t, i), vc, T, H)$. This necessitates the second change, which is the history H is now composed of pairs of $< (l, t, i), vc >$ of instructions (l, t, i) and their associated vector clock vc.

Algorithm 7 TAINTCHECK `resolve`$(s, (l,t,i), vc, T)$ Algorithm

Input: $s \in S$, (l,t,i): instruction (belonging to subblock b), vc: vector clock associated with subblock b, T: set of transfer functions in wings
if $s ==$ `taint` or $s ==$ `untaint` or $s ==$ `uncertain` **then**
  return s
else if $s == \{a\}$ for memory location a **then**
  a_state_{LSOS} = metadata state of m in LSOS
  a_state_{WING} = **do_resolve**$(m, t, (l,t,i), T, < (l,t,i), vc >)$
  $resolve\_state$ = **meet**$(a\_state_{\text{LSOS}}, a\_state_{\text{WING}})$
  return $resolve\_state$
else
  //$s == \{a, b\}$ for memory locations a, b
  //resolve metadata for a
  a_state_{LSOS} = metadata state of a in LSOS
  a_state_{WING} = **do_resolve**$(a, t, (l,t,i), T, < (l,t,i), vc >)$
  a_state_{MEET} = **meet**$(a\_state_{\text{LSOS}}, a\_state_{\text{WING}})$
  //resolve metadata for b
  b_state_{LSOS} = metadata state of b in LSOS
  b_state_{WING} = **do_resolve**$(b, t, (l,t,i), T, < (l,t,i), vc >)$
  b_state_{MEET} = **meet**$(b\_state_{\text{LSOS}}, m_2\_state_{\text{WING}})$
  //apply transfer function
  $resolve\_state$ = **transfer**$(a\_state_{\text{MEET}}, m_2\_state_{\text{MEET}})$
  return $resolve\_state$

5.7.4 Second Pass: Representing TaintCheck as an Extension of Reaching Definitions

As in Section 5.5.4, we can express TAINTCHECK as an extension of reaching definitions. The second pass of TAINTCHECK will again perform checks and resolve instruction-level transfer functions $\mathcal{T}_{l,t,i}$ to metadata value for destination address $m = \texttt{loc}(l,t,i)$. Our definitions are almost identical to those in Section 5.5.4, but with the updated signature for `resolve`:

$$\mathcal{G}_{l,t,i} = \begin{cases} m & \texttt{resolve}(m, (l,t,i), vc, \text{SIDE-IN}) \leftarrow \texttt{taint} \\ \varnothing & \text{otherwise} \end{cases}$$

$$\mathcal{K}_{l,t,i} = \begin{cases} m & \texttt{resolve}(m, (l,t,i), vc, \text{SIDE-IN}) \leftarrow \texttt{untaint} \\ \varnothing & \text{otherwise} \end{cases}$$

Algorithm 8 TAINTCHECK `do_resolve`$(m, orig\_tid, (l,t,i), vc, T, H)$ Algorithm

Input: m: current destination address, $orig\_tid$: original thread, (l,t,i): current instruction (belonging to subblock b), vc: vector clock associated with subblock b, T: set of transfer functions in wings, H: history <previously considered instruction, associated vector clock>
if $m ==$ `taint` or $m ==$ `untaint` or $m ==$ `uncertain` **then**
  return m
$num\_taint = num\_untaint = num\_uncertain = 0$
for all $(y_{(l',t',i')} \leftarrow s_j) \in P(m, (l,t,i), vc, T, H)$ **do**
  if $s_j ==$ `taint` or $s_j ==$ `untaint` or $s_j ==$ `uncertain` **then**
    return s_j
  else if $s_j == a$ for memory location a **then**
    a_state_{LSOS} = metadata state of a in LSOS
    $a\_state_{\text{WING}} =$ **do_resolve**$(a, orig\_tid, (l',t',i'), T, <(l,t,i), vc> :: H)$
    $resolve\_state =$ **meet**$(a\_state_{\text{LSOS}}, a\_state_{\text{WING}})$
    Increment counter of $\{num\_taint, num\_untaint, num\_uncertain\}$ that matches $resolve\_state$
  else
    //resolve metadata for a
    //$s_j = \{a, b\}$ for memory locations a, b
    a_state_{LSOS} = metadata state of a in LSOS
    $a\_state_{\text{WING}} =$ **do_resolve**$(a, orig\_tid, (l',t',i'), T, <(l,t,i), vc> :: H)$
    $a\_state_{\text{MEET}} =$ **meet**$(a\_state_{\text{LSOS}}, a\_state_{\text{WING}})$
    //resolve metadata for b
    b_state_{LSOS} = metadata state of b in LSOS
    $b\_state_{\text{WING}} =$ **do_resolve**$(b, orig\_tid, (l',t',i'), T, <(l,t,i), vc> :: H)$
    $b\_state_{\text{MEET}} =$ **meet**$(b\_state_{\text{LSOS}}, a\_state_{\text{WING}})$
    //apply transfer function
    $resolve\_state =$ **transfer**$(a\_state_{\text{MEET}}, b\_state_{\text{MEET}})$
    Increment counter of $\{num\_taint, num\_untaint, num\_uncertain\}$ that matches $resolve\_state$
//all proper predecessors have recursively been explored
if $num\_uncertain > 0$ or ($num\_taint > 0$ and $num\_untaint > 0$) **then**
  return `uncertain`
else if $num\_taint > 0$ **then**
  //$num\_untaint == 0$
  return `taint`
else
  //$num\_taint == 0$
  return `untaint`

$$\mathcal{M}_{l,t,i} = \begin{cases} m & \texttt{resolve}(m, (l,t,i), vc, \text{SIDE-IN}) \leftarrow \texttt{uncertain} \\ \varnothing & \text{otherwise} \end{cases}$$

The subblock equations ($\mathcal{G}_b, \mathcal{K}_b$ and $\mathcal{M}_b$), equations for SIDE-OUT ($\text{ALL}_b^{\mathcal{G}}$, $\text{ALL}_b^{\mathcal{K}}$, $\text{ALL}_b^{\mathcal{M}}$, GSO_b, KSO_b and MSO_b) and equations for SIDE-IN ($\text{WING}_b^{\mathcal{G}}$, $\text{WING}_b^{\mathcal{K}}$, $\text{WING}_b^{\mathcal{M}}$, GSI_b, KSI_b, MSI_b) are all identical to those in Section 5.6.1. Likewise, the strongly ordered state equations follow immediately from Section 5.6.2, and the LSOS equations from Section 5.6.3, as do their proofs and guarantees.

Lemma 50. *If* $\texttt{resolve}(s, (l,t,i), vc, \text{TSI}_{l,t})$ *returns* `untaint` *for location* $m = \texttt{loc}(l,t,i)$ *at instruction* (l,t,i)*, then under all valid vector orderings of the first* $l+1$ *epochs,* m *is* `untainted` *at instruction* (l,t,i).

Proof. Proof is identical to that of Lemma 35, with the exception that we are now enforcing valid vector orderings over valid orderings and our new definition of proper predecessor takes this into account. □

Lemma 51. *If* $\texttt{resolve}(s, (l,t,i), vc, \text{TSI}_{l,t})$ *returns* `taint` *for location* $m = \texttt{loc}(l,t,i)$ *at instruction* (l,t,i)*, then under all valid orderings of the first* $l+1$ *epochs,* m *is* `tainted` *at instruction* (l,t,i).

Proof. Proof is identical to that of Lemma 36, with the exception that we are now enforcing valid vector orderings over valid orderings and our new definition of proper predecessor takes this into account. □

Theorem 52. *Any error detected by the original* TAINTCHECK *on a valid execution ordering for a given machine (obeying intra-thread dependences and supporting cache coherence) will also be flagged by our butterfly analysis as either* `tainted` *or* `uncertain`*. Furthermore, any failed check of a* `tainted` *address is an error the original* TAINTCHECK *would discover under all valid execution orderings for a given machine. Thus, any potential false positives derive from failed checks of* `uncertain`.

Proof. As in Theorem 37, if there exists a valid execution with a failed check of `taint`, then there exists a valid vector ordering of the first $l+1$ epochs such that m is tainted at instruction

Benchmark	Inputs
BARNES	2048 bodies
FFT	$m = 20$ (2^{20} sized matrix)
OCEAN	Grid size: 258×258
LU	Matrix size: 1024×1024

Table 5.1: Benchmark Parameters

(l, t, i), and by the contrapositive of Lemma 50, `resolve` will not return `untaint` for m at (l, t, i). So, m will either be `tainted` or marked `uncertain`. The second statement follows directly from Lemma 51. If everything marked as `taint` is a true error, and nothing marked by `untaint` is ever an error, then all false positives must flow from a failed check of `uncertain`.

□

5.8 Experimental Setup

We now present the experimental evaluation of a prototype TAINTCHECK Butterfly Analysis tool which incorporates uncertainty. Like Butterfly and Chrysalis Analyses, the analysis is general purpose and can be implemented using a variety of dynamic analysis frameworks, including those based on binary instrumentation [20, 71, 82]. The results presented here extend the word-granularity Butterfly Analysis implementation of TAINTCHECK previous described in Chapter 3.3.2 and experimentally evaluated in Chapter 4.6, to incorporate uncertainty, as described in Section 5.5. In addition to implementing and testing uncertainty, we also executed experiments to measure the benefits of dynamic epoch resizing as described in section 5.3.1. The taint injection scheme described in Chapter 4.6 was modified to only taint 15% of the input data instead of 100% due to memory constraints.

All experiments were run on the Intel OpenCirrus[4] cluster. Each experiment was run inside an identically configured VM on an 8-core (2 Xeon E5440 quadcores) machine with 8GB of

[4]http://opencirrus.intel-research.net/

available RAM, a 32KB L1 and a 4MB L2 cache; to manage resource contention, each machine only ran one experiment at a time. For compatibility with LBA, a 32-bit OS was used, specifically 2.6.20-16-server. Any epoch elision was performed as a pre-processing step before the experiments were timed. The traces used in the experiments were copied from NFS to a local disk to remove NFS network effects from performance measurements. Table 5.1 describes the Splash-2.0 [113] benchmarks used in our experiments.

5.8.1 Gathering Fixed Traces

Unlike prior experiments for Butterfly Analysis and Chrysalis Analysis, presented in Chapters 3 and 4 respectively, our experimental evaluation involves an offline dynamic analysis of an execution trace. Using a fixed trace in our experiments allows us to gather performance and precision measurements for different effective epoch sizes while controlling for the underlying interleaving. We used LBA [23], which is implemented on top of Simics [108], to gather traces of thread execution which included heartbeats sized at 1K instructions/thread[5]. One additional benefit to using traces was that it enables the lifeguard itself to run natively, and larger input sizes than the simulator could otherwise tolerate could be used. Each trace was gathered with the benchmark running with four threads.

The traces were gathered using a "dummy" lifeguard which quickly returns from all handlers; by using a lifeguard which does minimal work and runs as fast as possible within the simulator, we minimize any slowdown or scheduling impact on the application whose trace is being gathered. Traces were stored in compressed form, with a single trace file for each lifeguard thread to consume.

[5] As before, for n threads, we inserted heartbeats after the LBA simulator observed $n \times 1024$ instructions executed globally.

5.8.2 Dynamic Epoch Resizing

Performance and precision measurements were taken for several configurations. First, SMALL numbers for performance and precision were gathered, assuming the 1K instruction/thread epoch size. Then, we tested LARGE, creating effective epoch sizes of $16K$ instructions per thread by only respecting the 16^{th} epoch boundary[6]. Any large epochs which experienced a failed check due to uncertainty were recorded. Finally, we tested three dynamic epoch resizing schemes to evaluate the limits of performance and precision that could be gained if a perfect oracle informed the lifeguard whether to skip or respect an epoch boundary, telling it to respect the underlying small epoch boundaries any time the corresponding larger epoch boundary had previously incurred a failed check due to uncertainty.

The three different dynamic adaptations tested are shown in Figure 5.4. In a large run, we assume epoch divisions that correspond with Figure 5.8.2(a). Suppose a thread observes a failed check of UNCERTAIN in epoch l. Under the first scheme, DYNAMIC$_{[l-1,l+1]}$will not elide the smaller epochs to form larger epochs when those smaller epochs would have belonged to larger epochs $l-1$, l or $l+1$; in other words, any large epoch in the sliding window is broken into its constituent smaller epochs. This is shown in Figure 5.8.2(b), which illustrates large epochs that correspond to three small epochs[7]. In a real system, DYNAMIC$_{[l-1,l+1]}$may be more difficult to implement, as it would require rollback of the second pass of epoch $l-1$ as well as undoing a commit to the SOS. Two more practical version of dynamic epoch resizing were also tested: DYNAMIC$_{[l,l+1]}$, which would not group small epochs that otherwise belonged to large epoch l and epoch $l+1$–in other words, it attempts to expand the sliding window except for large epoch $l-1$–shown in Figure 5.8.2(c). Another variation is to only expand the epoch which experience the failed check of `uncertain`, named DYNAMIC$_l$. This is shown in Figure 5.8.2(d).

[6]One exception were epochs at the termination of the program. To reduce complexity in handling edge cases in the prototype, the baseline/original epoch boundaries are always respected during periods where threads are exiting.

[7]In our experiments, one large epoch corresponds to 16 smaller epochs.

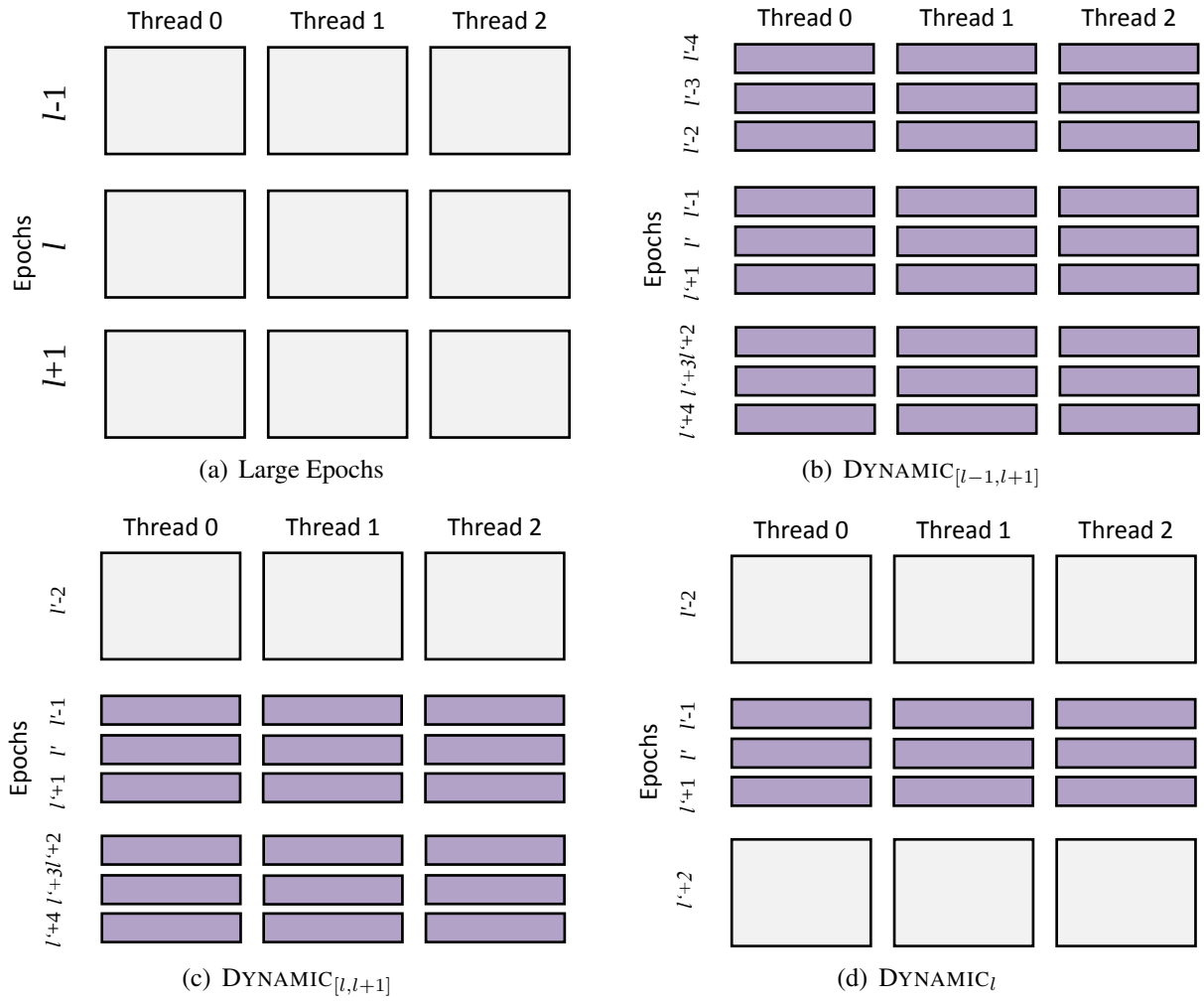


Figure 5.4: Dynamic epoch resizing. When a check of uncertain fails in large epoch l (with epoch divisions as in (a)), there are three possible dynamic adaptations. In (b), DYNAMIC$_{[l-1,l+1]}$is shown, where all three large epochs still in the sliding window revert to their underlying smaller epochs (in this example, there are three smaller epochs to every larger epoch). (c) and (d) illustrate DYNAMIC$_{[l,l+1]}$and DYNAMIC$_{l}$, which revert large epochs l and $l+1$, and only epoch l, respectively, to their underlying smaller epochs.

5.8.3 Types of Uncertainty

Our experiments will track two types of uncertainty: `heuristic` and `uncertain`. In the first case, `heuristic` is a specific type of uncertainty which occurs when the threshold for exploring potential parents in the wings is hit; in our experiments, that threshold is set at 512 parents. The second type of uncertainty, `uncertain`, is a catch-all that captures every other type of uncertainty in the system. It is possible for an address which is originally marked as

Benchmark	Epoch Size	Failed Taint	Failed Uncertain	Failed Heuristic
FFT-20	SMALL	0	0	0
	LARGE	0	3	3
	DYNAMIC$_l$	0	0	0
	DYNAMIC$_{[l,l+1]}$	0	0	0
	DYNAMIC$_{[l-1,l+1]}$	0	0	0
LU-1K	SMALL	0	0	0
	LARGE	0	3	3
	DYNAMIC$_l$	0	0	0
	DYNAMIC$_{[l,l+1]}$	0	0	0
	DYNAMIC$_{[l-1,l+1]}$	0	0	0
OCEAN-258	SMALL	0	2	0
	LARGE	0	38	6
	DYNAMIC$_l$	0	2	0
	DYNAMIC$_{[l,l+1]}$	0	2	0
	DYNAMIC$_{[l-1,l+1]}$	0	2	
BARNES-2K	SMALL	0	0	0
	LARGE	0	66	16
	DYNAMIC$_l$	0	12	0
	DYNAMIC$_{[l,l+1]}$	0	12	0
	DYNAMIC$_{[l-1,l+1]}$	0	12	0

Table 5.2: Precision Results: Comparing a SMALL effective epoch size with LARGE effective epoch size and three different varieties of dynamic epoch resizing.

`heuristic` to have its metadata transition to `uncertain`.

5.9 Evaluation

In this section, we will evaluate the precision gains and performance of the dynamic epoch resizing scheme, relative to a baseline SMALL epoch size of $1K$ instructions/thread and with LARGE sized at $16K$ instructions/thread.

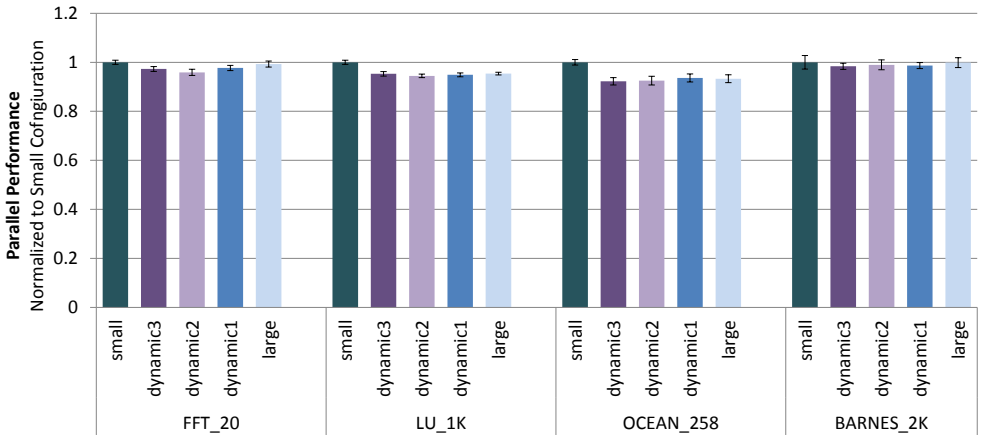
5.9.1 Precision

Table 5.2 displays the precision results for the five different configurations tested for each benchmark. We observe that for three benchmarks, FFT, LU and OCEAN, the SMALL configuration and the three DYNAMIC configurations experience the same precision. In fact, the DYNAMIC$_{[l-1,l+1]}$ configurations for FFT and LU experience the same identically zero potential errors as the SMALL configuration. For one benchmark, BARNES, the SMALL configuration has better precision than the three DYNAMIC configurations, but all are markedly better than the LARGE configuration. The LARGE configuration experiences failed checks of UNCERTAIN and HEURISTIC for all the benchmarks tested.

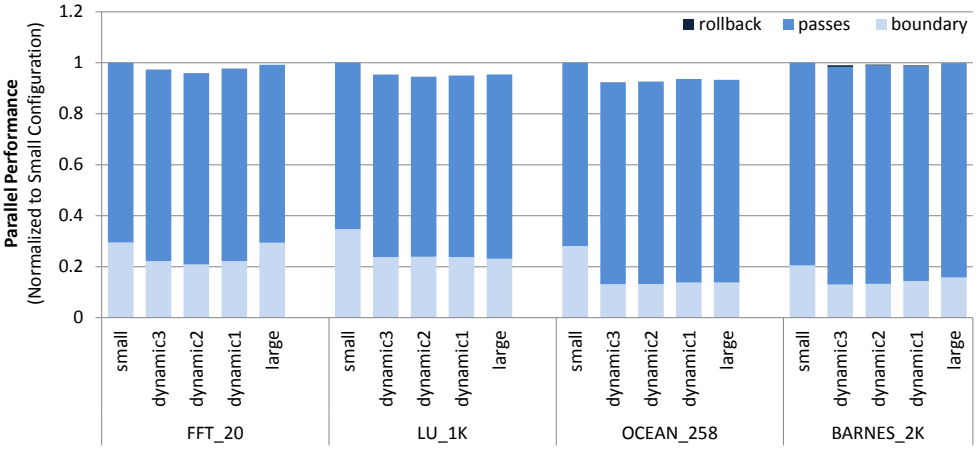
One of the goals of this work was to demonstrate the elimination of false positives; we see a 0 in the column titled *Failed Taint* for all benchmarks and configurations, indicating that no code checks failed due to a false check of TAINT. This is a marked improvement upon the results in Chapters 3 and 4, where even Chrysalis Analysis still observed potential errors which it could not disambiguate from true errors. While the number of failed checks of `uncertain` and `heuristic` is on par with standard Butterfly Analysis implementations of TAINTCHECK in the LARGE configuration, they now carry the additional information that they are not definitively true positives, something impossible to know in earlier versions of Butterfly Analysis. While the overall number of potential errors is similar to Chrysalis Analysis in the SMALL and DYNAMIC configurations, we can now disambiguate that the potential errors all are due to failed checks of `uncertain` and none are known true errors.

5.9.2 Performance

Figure 5.5(a) shows the performance of the parallel portion of execution, normalized to the SMALL configuration of each benchmark. Results shown are averaged over ten timing runs, with error bars indicating the 95% confidence interval. In most cases, the DYNAMIC and LARGE runs are outperforming the SMALL runs, even if the margin is small. In the best case, OCEAN,



(a) Parallel Performance. Averaged over six runs, with 95% confidence intervals



(b) Parallel Performance. Execution divided into into BOUNDARY, PASSES and ROLLBACK phases

Figure 5.5: (a) Parallel Performance, shown for SMALL, LARGE, and the three dynamic configurations: DYNAMIC$_{[l-1,l+1]}$, DYNAMIC$_{[l,l+1]}$and DYNAMIC$_l$labelled as dynamic3, dynamic2 and dynamic1, respectively. (b) Parallel Performance subdivided into BOUNDARY, PASSES and ROLLBACK. The sum of BOUNDARY and PASSES equals the average performance shown in (a). ROLLBACK is a cost only incurred by the dynamic schemes, estimated by dividing the average total running time of each configuration by the number of epochs which experience uncertainty. In the case of DYNAMIC$_{[l-1,l+1]}$ that estimate is multipled by two to cover the cost of rolling back not only the epoch which encounters uncertainty, but the prior epoch as well.

we see that DYNAMIC$_{[l-1,l+1]}$ DYNAMIC$_{[l,l+1]}$ and DYNAMIC$_l$ run at $0.91 - 0.92$X. No dynamic scheme consistently outperforms the others; rather, each has at least one benchmark where it performs the best.

To explore why larger epoch sizes were not having the expected speedup, we also measured

the time each thread spent doing BOUNDARY calculations during the parallel phase. Examples of boundary calculations include calculating the update to the global state, applying a pending global state update and calculating a thread-local LSOS. In general, any computation triggered by observing a heartbeat and which is not part of a linear pass (*e.g.*, the first pass or second pass) is considered a boundary calculation. Time spent in the parallel phase but not in the boundary is represented by PASSES. Finally, we estimate the cost of rolling back computation to perform dynamic epoch resizing in ROLLBACK. We calculate ROLLBACK by dividing the average total running time of each configuration by the number of epochs which experience uncertainty. In the case of DYNAMIC$_{[l-1,l+1]}$ that estimate is multiplied by two to cover the cost of rolling back not only the epoch which encounters uncertainty, but the prior epoch as well. Both DYNAMIC$_{[l,l+1]}$ and DYNAMIC$_l$ only need to rollback the exact large epoch which experiences the uncertainty. Figure 5.5(b) illustrates the breakdown of the parallel phase into BOUNDARY, PASSES and ROLLBACK. It is clear that LARGE as well as DYNAMIC configurations spend $29-54\%$ less in parallel time boundary calculations (compared to SMALL configurations). Our main expected expected source of performance gain from larger epoch sizes was a reduction of time spent in boundary calculations, so this goal was achieved as well.

While the overall boundary time is reduced for larger effective epoch sizes, the BOUNDARY time itself is a not a large fraction (from $13-30\%$) of parallel execution, which helps explain why the relatively large reduction in BOUNDARY calculations does not result in a larger improvement of parallel performance for large epoch sizes. Incorporating dynamic adaptations to uncertainty into Butterfly (respectively, Chrysalis) Analysis was not expected to reduce the time spent in passes; each instruction still needed to be analyzed. One likely explanation is that a larger epoch size corresponds to more elements in the wings, which may be slowing down the first and second passes in LARGE configurations relative to SMALL configurations.

5.9.3 Comparison of Dynamic Schemes

The competitive performance of DYNAMIC$_{[l,l+1]}$ and DYNAMIC$_l$ along with their equivalent precision to DYNAMIC$_{[l-1,l+1]}$ among all benchmarks, is both surprising and advantageous. Our original hypothesis had been that uncertainty could arise due to any event within the sliding window, and thus DYNAMIC$_{[l-1,l+1]}$ was hypothesized to best improve precision. Instead, both DYNAMIC$_{[l,l+1]}$ and DYNAMIC$_l$ had equivalent precision, and very similar performance. This is significant because the rollback requirements for DYNAMIC$_{[l,l+1]}$ and DYNAMIC$_l$ are only to the beginning of the second pass of the epoch which experiences a failed check of uncertainty; in contrast, the DYNAMIC$_{[l-1,l+1]}$ scheme requires rolling back not only epoch epoch l, which experiences the failed check of uncertain, but also the prior epoch $l-1$. Rolling back epoch $l-1$ would require a heavier cost; restarting epoch $l-1$ as its constituent smaller epochs requires not just a rollback, but also preservation of SOS$_{l-1}$ and LSOS$_{l-1,t}$. With both DYNAMIC$_{[l,l+1]}$ and DYNAMIC$_l$ performing as well as DYNAMIC$_{[l-1,l+1]}$, either is a competitive choice that reduces complexity of analysis overhead and which successfully leverages the incorporation of uncertainty into Butterfly Analysis to achieve performance similar to LARGE epochs with precision approaching, if not matching, SMALL epochs.

5.10 Chapter Summary

To enhance the ability of Butterfly Analysis and Chrysalis Analysis to disambiguate known errors from potential errors, we have proposed and evaluated adding an uncertain state to the metadata lattice. By explicitly tracking uncertainty, we have shown that it is possible to enable dynamic adaptations whenever a lifeguard experiences a failed check of an uncertain metadata location. Our implementation of TAINTCHECK incorporating uncertainty within Butterfly Analysis showed (1) that our proposal effectively isolates known errors from potential errors and (2) that dynamically adjusting the epoch size in the presence of potential error can recover the

precision of a smaller epoch size with performance similar to a much larger epoch size. While overall performance did not differ substantially between small and large effective epoch sizes, we observed a substantial decrease in time spent in boundary calculations for all configurations with a large overall effective epoch size, and precision close to or matching that of the underlying smaller epoch sizes for all DYNAMIC configurations.

Chapter 6

Conclusions

Despite programmer's best efforts, and researcher's attempts to help them, bugs persist in software. Compared with sequential software, it is even more difficult to avoid bugs in parallel software due to nonintuitive interleavings between threads. This increases the importance of tools and frameworks to help programmers detect bugs in parallel software, whether those tools. This thesis presented *dataflow analysis-based dynamic parallel monitoring*, a new software-based general purpose framework to enable analysis of parallel programs at runtime.

One major contribution of this work was enabling dynamic parallel monitoring without measuring inter-thread data dependences. This allowed us to avoid a reliance on hardware to measure inter-thread data dependences, and made it easier to (1) create a software-based framework that does not require specialized hardware and (2) support any shared-memory architecture regardless of its memory consistency model so long as it supported cache coherence. This was all made possible by the development of a thread execution model closely tied to modern parallel processor design; creating a low-level abstraction enabled development of Butterfly Analysis without making incorrect assumptions about the underlying hardware.

In developing Butterfly Analysis, we not only showed that it was possible to enable dynamic parallel monitoring of applications with access only to a partial order of application events, but we showed how such monitoring could proceed with provable guarantees to never miss an error

in the monitored application. Our implementation of ADDRCHECK, a real world memory lifeguard, within the Butterfly Analysis framework, demonstrated that for $\sim 2.1x$ slowdown over parallel unmonitored execution, you could enable parallel monitoring. To our knowledge, Butterfly Analysis is the first generic platform designed to enable porting sequential analyses to the parallel domain.

With the development of Chrysalis Analysis, we showed how to answer one of the shortcomings of Butterfly Analysis: where Butterfly Analysis occasionally incurred false positives when it was unaware of high-level synchronization preventing bad interleavings, Chrysalis Analysis recovered this precision at the cost of a more complex thread execution model and subsequently, more complicated analysis. We demonstrated that the provable guarantees provided by Butterfly Analysis could be generalized to apply to Chrysalis Analysis (itself a generalization of Butterfly Analysis). In a comparison of a TAINTCHECK implementation in both Butterfly and Chrysalis Analyses, we showed that the Chrysalis Analysis implementation incurred an average $1.9x$ slowdown but reduced false positivies by $17.9x$.

Finally, by explicitly modeling uncertainty within dataflow analysis-based dynamic parallel monitoring, we have shown that we can now provably isolate true errors from potential errors, while maintaining guarantees (both in Butterfly and Chrysalis Analysis). This has intrinsic value to programmers: knowing an error *actually* occurred during the monitored execution can increase its importance in debugging relative to potential errors where the analysis is uncertain. Furthermore, we've demonstrated the utility of uncertainty by illustrating how dynamic adaptations to the analyses in the presence of uncertainty can lead to improved precision, and deliver performance similar to large epoch sizes with precision similar to small epochs.

6.1 Future Directions

There are several directions for extending this work. Neither Butterfly Analysis or Chrysalis Analysis assume prior knowledge of the application being monitored; one of their strengths is

that both frameworks can deliver provably correct monitoring without requiring access to the source code of the application.

Incorporating Profiling Information

However, given access to profiling information or incorporating a JIT, one direction would be to optimize handlers whenever a memory location is provably thread-local; if a write operation only affects global state, and never can affect another thread, then it does not need to be considered in the SIDE-OUT or SIDE-IN, potentially minimizing the SIDE-IN and shrinking the search space for lifeguards like TAINTCHECK. This has a similar flavor to work done by Ruwase *et al.* [95], though there the optimizations could assume a sequential application and didn't need to consider how side effects of instruction reordering could affect other threads.

Paired with Static Analysis

Another interesting direction would be to couple dataflow analysis-based dynamic parallel monitoring with a static analysis phase when the source code is available. If the static analysis phase can prove that an operation is provably safe, then the dynamic analysis phase can elide checking that instruction; this has the potential to improve performance without sacrificing any provable guarantees.

Richer Uncertainty Analysis

Concentrating on the uncertainty analysis, another interesting direction would be to develop a richer set of uncertain states. For example, the result of the `meet(taint, untaint)` could be different than the `meet(untaint, heuristic)`, whereas now they both map to `uncertain`. Furthermore, the uncertainty analysis can itself become predictive and take into account how many parents of each metadata state it encounters, and try to assign a probability of

being a true error to each instance potential error reported.

More Analyses

Adapting more analyses to our frameworks would be yet another avenue to pursue. We have shown how to present Reaching Definitions and Available Expressions within our framework for both Butterfly Analysis and Chrysalis Analysis. In Chapter 5, we presented "Reaching Definitions" with uncertainty–but this version of Reaching Definitions was equivalent to Constant Propagation with only two precise states and equating `uncertain` with Not-A-Constant. Extending our abstraction to a true Constant Propagation is straightforward: each precise ("constant") state extends the symmetry of the $\mathcal{G}$ and $\mathcal{K}$ formulas.

It is also interesting to explore what other analyses are expressable within our framework, building on top of Reaching Definitions, Available Expressions, and Constant Propagation. Analyses like MEMCHECK [80, 81], which verify that memory is allocated along all paths and initialized before any access, can be expressed similar to the cross product of ADDRCHECK and TAINTCHECK. ADDRCHECK is suitable to ensure that memory is allocated along all paths; a similar construct to TAINTCHECK allows initialization status to flow when a value is copied from one address to another, and verify that there is not a path where an access is to unallocated data. An analysis which performs bounds-checking on all pointers, ensuring all pointers are valid and only used to access memory within their allocated region [9], is also expressible within our framework. Like ADDRCHECK, it must ensure that all accesses are to allocated data, whether stack or heap. Like TAINTCHECK, anytime a new pointer is derived from an old pointer, the new pointer must inherit bounds from the old pointer.

Dataflow analysis-based dynamic parallel monitoring's applications are not limited to correctness checking. An interesting performance-focused analysis that is easily expressible within dataflow analysis-based dynamic parallel monitoring would be a *false-sharing* detector. False-sharing arises when at least two processors are accessing disjoint memory locations on a shared

cache line, Processor A issues a write, and Processor B then incurs a miss due to the cache block being invalidated by Processor A. This occurs even though the data for that particular location was unchanged, because cache coherence operates at a coarser granularity than individual words [51]. Detecting this behavior is natural within dataflow analysis-based dynamic parallel monitoring. During the first pass each block computes a read and write set as its SIDE-OUT. The SIDE-IN likewise has two parts, the union of the concurrent reads and the union of the concurrent writes. In the second pass, the lifeguard looks for instances where a local read and a concurrent write or a local write and a concurrent write were to the same cache block but different memory words. This is a mere sampling of the analyses expressible within dataflow analysis-based dynamic parallel monitoring.

New Architectures

Dataflow analysis-based dynamic parallel monitoringwas designed for coherent, shared-memory multiprocessors. Interesting directions to extend the work would be to consider different architectures or settings: incorporating transactional memory as a first-class guarantor of atomicity, or porting dataflow analysis-based dynamic parallel monitoring to increasingly important architectures such as GPUs or heterogeneous multicores. One could even try to adapt Chrysalis Analysis to cloud- or supercomputer-based parallelism, where instead of synchronization the arcs would indicate communication between nodes. While the scale is vastly different than on-chip parallelism, there are also many similarities: relaxed consistency models and a need for help writing bug-free parallel programs.

This thesis presented dataflow analysis-based dynamic parallel monitoring, a new software-based framework for monitoring parallel programs at runtime to detect bugs and security exploits. We have demonstrated that it is possible to create an efficient software-based dynamic analysis framework based on windows of uncertainty which avoids the overhead of tracking detailed inter-thread data dependences. We have explored the tradeoffs between performance and

precision, from adjusting epoch size to including synchronization-based happens-before arcs and explicitly isolating uncertainty within the analysis to enable dynamic adaptations. We have presented both theoretical guarantees as well as experimental evaluations of our frameworks for both performance and precision. There are many future applications of this work, ranging from reducing dynamic work by statically analyzing the code or executable, to looking at supporting new architectures (non-homogeneous or potentially non-coherent), and extending to new areas entirely, such as enabling analysis for cloud-based parallelism. Bugs are a persistent problem for software developers, and dataflow analysis-based dynamic parallel monitoring is a new tool at their disposal.

Bibliography

[1] Valgrind User Manual. http://valgrind.org/docs/manual/manual-core.html.

[2] Intel 64 and IA-32 Architecture Software Developer's Manual, Volume 3: System Programming Guide. Order Number: 325384-051US, June 2014.

[3] Intel 64 and IA-32 Architectures Software Developers Manual Volume 3A. Order Number: 325462-050US, February 2014.

[4] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12), 1996.

[5] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[6] J. Arulraj, P.-C. Chang, G. Jin, and S. Lu. Production-Run Software Failure Diagnosis via Hardware Performance Counters. In *ASPLOS*, 2013.

[7] J. Arulraj, G. Jin, and S. Lu. Leveraging the Short-Term Memory of Hardware to Diagnose Production-Run Software Failures. In *ASPLOS*, 2014.

[8] J. Auerbach, D. F. Bacon, P. Cheng, D. Grove, B. Biron, C. Gracie, B. McCloskey, A. Micic, and R. Sciampacone. Tax-and-Spend: Democratic Scheduling for Real-time Garbage Collection. In *EMSOFT*, 2008.

[9] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *PLDI*, 1994.

[10] D. F. Bacon, C. R. Attanasio, H. B. Lee, V. Rajan, and S. Smith. Java without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector. In *PLDI*, 2001.

[11] R. Baldoni and M. Klusch. Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems. *IEEE Distributed Systems Online*, 3, February 2002.

[12] R. Beers. Pre-RTL Formal Verification: An Intel Experience. In *DAC*, 2008.

[13] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. In *ASPLOS*, 2010.

[14] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic Process Groups in dOS. In *OSDI*, 2010.

[15] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. In *OOPSLA*, 2009.

[16] J. G. Beu, J. A. Poovey, E. R. Hein, and T. M. Conte. High-Speed Formal Verification of Heterogeneous Coherence Hierarchies. In *HPCA*, 2013.

[17] C. Bienia and K. Li. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *MoBS*, June 2009.

[18] H.-J. Boehm. Reordering Constraints for Pthread-style Locks. In *PPoPP*, 2007.

[19] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional Detection of Data Races. In *PLDI*, 2010.

[20] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, 2004.

[21] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *ASPLOS*, 2010.

[22] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice & Experience*, 30(7), 2000.

[23] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible Hardware Acceleration for Instruction-grain Program Monitoring. In *ISCA*, 2008.

[24] M. Christiaens and K. De Bosschere. Accordion Clocks: Logical Clocks for Data Race Detection. In *Euro-Par 2001 Parallel Processing*, 2001.

[25] R. Chugh, J. W. Voung, R. Jhala, and S. Lerner. Dataflow Analysis for Concurrent Programs using Datarace Detection. In *PLDI*, 2008.

[26] J. Chung, M. Dalton, H. Kannan, and C. Kozyrakis. Thread-Safe Dynamic Binary Translation using Transactional Memory. In *HPCA*, 2008.

[27] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: A Programmable Macro Engine for Customizing Applications. In *ISCA*, 2003.

[28] D. E. Culler, J. P. Singh, and A. Gupta. *Shared Memory Multiprocessors*, chapter 5, pages 270–376. Morgan Kaufmann Publishers, Inc, 1999.

[29] N. Dave, M. C. Ng, and Arvind. Automatic Synthesis of Cache-Coherence Protocol Processors Using Bluespec. In *MEMOCODE*, 2005.

[30] D. Deng, W. Zhang, and S. Lu. Efficient Concurrency-Bug Detection Across Inputs. In *OOPSLA*, 2013.

[31] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *ASPLOS*, 2009.

[32] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: A Relaxed Consistency Deterministic Computer. In *ASPLOS*, 2011.

[33] J. Devietti, B. P. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer. RADISH: Always-on Sound and Complete Ra Detection in Software and Hardware. In *ISCA*, 2012.

[34] T. Elmas, J. Burnim, G. Necola, and Koushik. CONCURRIT: A Domain Specific Language for Reproducing Concurrency Bugs. In *PLDI*, 2013.

[35] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.

[36] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2), 2001.

[37] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, 2009.

[38] C. Flanagan and S. N. Freund. Adversarial Memory for Detecting Destructive Races. In *PLDI*, 2010.

[39] C. Flanagan and S. N. Freund. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *PASTE*, 2010.

[40] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, 2002.

[41] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *ISCA*, 1990.

[42] P. B. Gibbons, M. Merrit, and K. Gharachorloo. Proving Sequential Consistency of High-Performance Shared Memories. In *SPAA*, 1991.

[43] P. B. Gibbons and M. Merritt. Specifying Nonblocking Shared Memories. In *SPAA*, 1992.

[44] M. L. Goodstein, S. Chen, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Chrysalis Analysis: Incorporating Synchronization Arcs in Dataflow-Analysis-Based Parallel Monitoring. In *PACT*, 2012.

[45] M. L. Goodstein, E. Vlachos, S. Chen, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Butterfly Analysis: Adapting Dataflow Analysis To Dynamic Parallel Monitoring. In *ASPLOS*, 2010.

[46] R. Grisenthwaite. ARM Barrier Litmus Tests and Cookbook. PRD03-GENC-007826 1.0, November 2009.

[47] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *PPOPP*, 1993.

[48] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *OOPSLA*, 2009.

[49] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, fifth edition edition, 2012.

[50] J. L. Hennessy and D. A. Patterson. *Instruction-Level Parallelism and Its Exploitation*, chapter 3, pages 148–259. In [49], fifth edition edition, 2012.

[51] J. L. Hennessy and D. A. Patterson. *Thread-Level Parallelism*, chapter 5, pages 344–429. In [49], fifth edition edition, 2012.

[52] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, 6(3), 1998.

[53] D. R. Hower, P. Dudnik, M. D. Hill, and D. A. Wood. Calvin: Deterministic or Not? Free Will to Choose. In *HPCA*, 2011.

[54] J. Huang, C. Zhang, and J. Dolby. CLAP: Recording Local EExecution to Reproduce Concurrency Failures. In *PLDI*, 2013.

[55] L. Ivanov and R. Nunna. Modeling and Verification of Cache Coherence Protocols. In *ISCAS*, 2001.

[56] H. Jooybar, W. W. L. Fung, M. O'Connor, J. Devietti, and T. M. Aamodt. GPUDet: A Deterministic GPU Architecture. In *ASPLOS*, 2013.

[57] P. Joshi, M. Naik, K. Sen, and D. Gay. An Effective Dynamic Analysis for Detecting Generalized Deadlocks. In *FSE*, 2010.

[58] P. Joshi, C.-S. Park, Koushik-Sen, and M. Naik. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *PLDI*, 2009.

[59] J. Knoop. Partial dead code elimination for parallel programs. In *Euro-Par'96 Parallel Processing*, pages 441–450. 1996.

[60] J. Knoop. Parallel Constant Progagation. In *Euro-Par'98 Parallel Processing*. 1998.

[61] J. Knoop. Parallel Data-Flow Analysis of Explicitly Parallel Programs. In *Euro-Par'99 Parallel Processing*. 1999.

[62] J. Knoop, B. Steffan, and J. Vollmer. Parallelism for free: efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3), 1996.

[63] J. Knoop and B. Steffen. Code motion for explicitly parallel programs. In *PPoPP*, 1999.

[64] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Bitvector analyses ⇒ no state explosion! In *TACAS*. 1995.

[65] T. Liu, C. Curtsinger, and E. D. Berger. DTHREADS: Efficient Deterministic Multithreading. In *SOSP*, 2011.

[66] D. Long and L. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *ISSTA*, 1991.

[67] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *ASPLOS*, 2006.

[68] B. Lucia and L. Ceze. Finding concurrency bugs with Context-Aware Communication Graphs. In *MICRO*, 2009.

[69] B. Lucia and L. Ceze. Cooperative Empirical Failure Avoidance for Multithreaded Programs. In *ASPLOS*, 2013.

[70] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm. Conflict exceptions: simpli-

fying concurrent language semantics with precise hardware exceptions for data-races. In *ISCA*, 2010.

[71] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

[72] M. Lyons, B. Hay, and B. Frey. PowerPC storage model and AIX programming. http://www.ibm.com/developerworks/systems/articles/powerpc.html, November 2005.

[73] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFX: a simple and efficient memory model for concurrent programming languages. In *PLDI*, 2010.

[74] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*, 2008.

[75] A. Muzahid, S. Qi, and Jose. Vulcan: Hardware Support for Detecting Sequential Consistency Violations Dynamically. In *MICRO*, 2012.

[76] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. SigRace: Signature-Based Data Race Detection. In *ISCA*, 2009.

[77] S. Nagarakatte, S. Burckhardt, M. M. K. Martin, and M. Musuvathi. Multicore Acceleration of Priority-Based Schedulers for Concurrency Bug Detection. In *PLDI*, 2012.

[78] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *ISCA*, 2005.

[79] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, U. Cambridge, 2004. http://valgrind.org.

[80] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[81] N. Nethercote and J. Seward. How to Shadow Every Byte of Memory Used by a Program. In *VEE*, 2007.

[82] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *PLDI*, 2007.

[83] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[84] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS*, 2005.

[85] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS*, 2009.

[86] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *ASPLOS*, 2009.

[87] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *SOSP*, 2009.

[88] F. Pong and M. Dubois. Verification Techniques for Cache Coherence Protocols. *ACM Computing Surveys*, 29(1):82–126, March 1997.

[89] F. Pong and M. Dubois. Formal Verification of Complex Coherence Protocols Using Symoblic State Models. *Journal of the ACM*, 45(4):557–587, July 1998.

[90] F. Pong and M. Dubois. Formal Automatic Verification of Cache Coherence in Multiprocessors with Relaxed Memory Models. *IEEE Transactions on Parallel and Distributed Systems*, 11(9):989–1006, September 2000.

[91] N. Provos. Improving Host Security with System Call Policies. In *USENIX Security*, 2003.

[92] X. Qian, J. Torrellas, B. Sahelices, and D. Qian. Volition: Scalable and Precise Sequential Consistency Violation Detection. In *ASPLOS*, 2013.

[93] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies - a safe method to survive software failures. In *SOSP*, 2005.

[94] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Scalable and Precise Dynamic Datarace Detection for Structure Parallelism. In *PLDI*, 2012.

[95] O. Ruwase, S. Chen, P. B. Gibbons, and T. C. Mowry. Decoupled Lifeguards: Enabling Path Optimizations for Dynamic Correctness Checking Tools. In *PLDI*, 2010.

[96] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan. Parallelizing Dynamic Information Flow Tracking. In *SPAA*, 2008.

[97] V. Sarkar. Analysis and Optimization of Explicitly Parallel Programs Using the Parallel Program Graph Representation. *Lecture Notes in Computer Science*, 1366, 1998.

[98] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Race Detector for Multi-threaded Programs. *ACM Transactions on Computer Systems*, 15(4), 1997.

[99] K. Serebryany and T. Iskhodzhanov. ThreadSanitzer - Data Race Detection in Practice. In *WBIA*, 2009.

[100] J. Seward and N. Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *USENIX ATC*, 2007.

[101] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. X86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.

[102] M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Inf. Process. Lett.*, 1992.

[103] H. Srinivasan, J. Hook, and M. Wolfe. Static single assignment for explicitly parallel programs. In *PoPL*, 1993.

[104] H. Srinivasan and M. Wolfe. Analyzing programs with explicit parallelism. In *Languages and Compilers for Parallel Computing*. 1992.

[105] S. Srinivasan, P. S. Chhabra, P. K. Jaini, A. Aziz, and L. John. Formal Verification of

a Snoop-Based Cache Coherence Protocol Using Symbolic Model Checking. In *VLSID*, 1999.

[106] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *ASPLOS*, 2011.

[107] F. Verbeek and J. Schmaltz. Towards the Formal Verification of Cache Coherency at the Architectural Level. *ACM Transactions on Design Automation of Electronic Systems*, 17(3):20:1–20:16, June 2012.

[108] Virtutech Simics. http://www.virtutech.com/.

[109] E. Vlachos. *"Lightweight and Low-Cost Mechanisms to Enable Parallel Monitoring of Multithreaded Applications"*. PhD thesis, Carnegie Mellon University, 2013.

[110] E. Vlachos, M. L. Goodstein, M. A. Kozuch, S. Chen, B. Falsafi, P. B. Gibbons, and T. C. Mowry. ParaLog: Enabling and Accelerating Online Parallel Monitoring of Multithreaded Applications. In *ASPLOS*, 2010.

[111] C. Wang, Y. Wu, and Jaewoon. TSO-Atomicity: Efficient TSO Enforcement for Aggressive Program Optimization. In *ASPLOS*, 2013.

[112] D. L. Weaver and E. Tom Germond. The SPARC Architecture Manual, Version 9. SA-V09-R147-Jul2003.

[113] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.

[114] B. P. Wood, L. Ceze, and D. Grossman. Low-Level Detection of Language-Level Data Races with LARD. In *ASPLOS*, 2014.

[115] D. A. Wood, M. D. Hill, and D. J. Sorin. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, http://www.morganclaypool.com/doi/abs/10.2200/S00346ED1V01Y201104CAC016,

2011.

[116] J. Wu, Y. Tang, G. Ho, H. Cui, and J. Yang. Sound and Precise Analysis of Parallel Programs through Schedule Specialization. In *PLDI*, 2012.

[117] W. Xiong, S. Park, J. Zhang, Y. Zhou, Z. Ma, M. Frank, B. Kuhn, and P. Petersen. Ad Hoc Synchronization Considered Harmful. In *OSDI*, 2010.

[118] M. Xu, R. Bodik, and M. D. Hill. A 'Flight Data Recorder' for enabling full-system multiprocessor deterministic replay. In *ISCA*, 2003.

[119] M. Xu, R. Bodik, and M. D. Hill. A regulated transitive reduction (RTR) for longer memory race recording. In *ASPLOS*, 2006.

[120] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multiprocessor. In *ISCA*, 2009.

[121] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adapative Tracking. In *SOSP*, 2005.

[122] W. Zhang, M. d. Kruijf, A. Li, S. Lu, and K. Sankaralingam. ConAir: Featherweight Concurrency Bug Recovery Via Single-Threaded Idempotent Execution. In *ASPLOS*, 2013.

[123] W. Zhang, J. Lim, R. Olichandran, J. Sherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: Detecting Concurrency Bugs through Sequential Errors. In *ASPLOS*, 2011.

[124] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-assisted Lockset-based Race Detection. In *HPCA*, 2007.

[125] Y. Zhou, P. Zhou, F. Qin, W. Liu, and J. Torrellas. Efficient and flexible architectural support for dynamic monitoring. *ACM Transactions on Architecture and Code Optimization*, 2(1), 2005.