2015 International Conference on Parallel Architecture and Compilation

Tracking and Reducing Uncertainty in Dataflow Analysis-Based Dynamic Parallel Monitoring

Michelle L. Goodstein
Carnegie Mellon University
mgoodste@cs.cmu.edu

Phillip B. Gibbons
Carnegie Mellon University
gibbons@cs.cmu.edu

Michael A. Kozuch
Intel Labs Pittsburgh
michael.a.kozuch@intel.com

Todd C. Mowry
Carnegie Mellon University
tcm@cs.cmu.edu

***Abstract*—Dataflow analysis-based dynamic parallel monitoring (DADPM) is a recent approach for identifying bugs in parallel software as it executes, based on the key insight of explicitly modeling a sliding *window of uncertainty* across parallel threads. While this makes the approach practical and scalable, it also introduces the possibility of *false positives* in the analysis. In this paper, we improve upon the DADPM framework through two observations. First, by explicitly tracking new *"uncertain"* states in the metadata lattice, we can distinguish potential false positives from true positives. Second, as the analysis tool runs dynamically, it can use the existence (or absence) of observed *uncertain* states to adjust the tradeoff between precision and performance on-the-fly. For example, we demonstrate how the *epoch size* parameter can be adjusted dynamically in response to uncertainty in order to achieve better performance and precision than when the tool is statically configured. This paper shows how to adapt a canonical dataflow analysis problem (reaching definitions) and a popular security monitoring tool (TAINTCHECK) to our new uncertainty-tracking framework, and provides new provable guarantees that reported true errors are now precise.**

I. Introduction

Dynamic analysis tools [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] (aka "lifeguards") help programmers find bugs by performing sophisticated instruction-grain dynamic analysis of software as it executes. Lifeguards maintain *metadata*—shadow state about the application's memory and register state—to reason about whether the program is violating the lifeguard's model of correct execution. For example, TAINTCHECK [1] is a popular security lifeguard that detects stack-smashing attacks and other security exploits by tracking the flow of `tainted` data (i.e., data whose source is untrusted) throughout the application. TAINTCHECK ensures that operations such as pointer dereferences, indirect jumps and system calls always take trusted arguments.

While a variety of tools successfully monitor sequential programs, the monitoring of *parallel* programs is far more challenging due to *inter-thread data dependences* and *shared memory consistency models*. Consider the two application threads shown in Figure 1, each being monitored by its own TAINTCHECK thread, with metadata shared between them. If Thread 0 was the entire (sequential) program, it is trivial to conclude that `p` is `untainted` when it is dereferenced. When examining both parallel threads together, whether

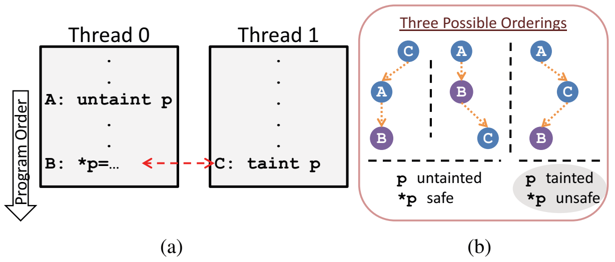


Figure 1: Impact of inter-thread data dependences on TAINTCHECK. Instruction `A` (`C`) untaints (respectively, taints) `p`. (a) The safety of `*p` in instruction `B` depends on (b) which of the interleavings occur.

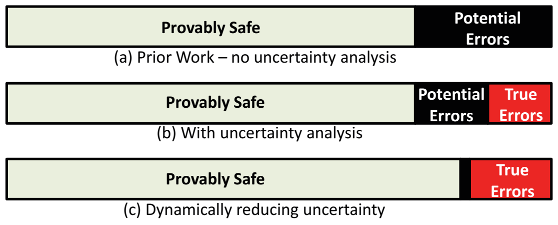

Provably Safe
Potential Errors
(a) Prior Work – no uncertainty analysis
Provably Safe
Potential Errors
True Errors
(b) With uncertainty analysis
Provably Safe
True Errors
(c) Dynamically reducing uncertainty

Figure 2: Dividing memory operations into provably safe (precise), true error (precise) and potential errors (conservative). Tracking uncertainty enables separation of potential errors and true errors; dynamic adaptations reduce analysis uncertainty.

or not the dereference of `p` is safe depends on which of the interleavings occurs in the application (shown at right). Moreover, the set of possible interleavings is dictated in part by the memory consistency model of the machine running the program.

Dataflow Analysis-Based Dynamic Parallel Monitoring. In prior work, we introduced *dataflow analysis-based dynamic parallel monitoring* (DADPM) [11, 12], an approach that enables parallel dynamic application monitoring and supports relaxed memory consistency models without requiring detailed inter-thread data dependence tracking or specialized hardware. DADPM (described in further detail in Section II) represents parallel execution using bounded *windows of uncertainty*, which model the finite upper bound of delay between when an instruction is issued and when its effects become globally visible. This platform-specific upper bound on the delay dictates the minimum size of a DADPM *epoch*; however, epochs may be larger than this minimum in order to reduce monitoring overheads, at the cost of a possible degradation in analysis precision [11]. Because epochs contain 1K–64K program instructions per thread, reasoning about all potential memory access interleavings within a window of uncertainty would be prohibitively expensive. Instead, DADPM employs a "closure" operation, inspired by *region-based dataflow analysis*, which quickly and conservatively accounts for the worst possible interleaving scenario.

Drawback: Errors Not Isolated From False Positives. While our prior works on DADPM (i.e., *Butterfly Analysis* [11] and *Chrysalis Analysis* [12]) have provided provable guarantees of zero missed errors, their conservative approach can lead to imprecision in cases where they *falsely classify a safe event as a potential error.* Consider Figure 2(a), illustrating a division of application events into provably safe operations and potential errors. Both Butterfly Analysis and Chrysalis Analysis reuse a single metadata state (e.g., `taint`) for both precise determinations (e.g., locations known to be `tainted`) and conservative judgments (e.g., locations where metadata state cannot be determined). As a result, *neither can distinguish true errors from potential errors.*

This Work: Explicitly Track and Dynamically Reduce Uncertainty. In this paper, we propose, prove the soundness of, and evaluate the effectiveness of extensions to dataflow analysis-based dynamic parallel monitoring that enable *explicit tracking of uncertainty within the analysis.* Unlike all previous work on *must* and *may* analysis in prior dataflow settings (e.g., [13, 14, 15, 16]), the uncertainty in DADPM arises not from control flow or memory aliasing, but instead due to the concurrent interleaving of threads. A primary goal of this work is to separate known errors from potential errors, as shown in Figure 2(b). We extend Butterfly Analysis in this paper; the corresponding extensions and proofs for Chrysalis Analysis are available in [17].

Besides improving the precision of state-of-the-art DADPM while maintaining correctness guarantees, tracking uncertainty also enables lifeguards to "drill down" on potential errors in order to remove uncertainty: with further analysis, many potential errors can be better classified as provably safe or as true errors (Figure 2(c)). As a case study, we explore the overhead vs. precision tradeoff noted above, by running the TAINTCHECK lifeguard with a larger epoch size (for lower overhead) but dynamically adjusting to a smaller epoch size precisely when uncertainty is detected (for higher precision). We show this achieves *better performance and precision than any statically configured epoch size.* Moreover, because the lifeguard analysis is happening on-the-fly while the program is executing, one might expect that periodic checkpoints of the shared metadata state, as well as per-thread trace logs of the program's execution since the last checkpoint, would be required to enable rollback for further analysis. Perhaps surprisingly, we show that our approach can remove significant uncertainty *without any additional checkpointing or logging!* Note that our drill-down techniques are also useful for improving precision in prior DADPM approaches—the main advantage of dynamic adaptations alongside tracking uncertainty is that these adaptations will be undertaken *precisely when there is a possibility to eliminate the uncertainty.*

Table I: Comparison of Parallel Program Monitoring Frameworks

	General purpose	Software only	Loosely coupled	Handles weak consistency
FlexiTaint [6]	**X**	**X**	**X**	✓
Thread-safe DBT [5]	✓	uses STM	**X**	✓
HAPM [8, 9, 10]	✓	**X**	✓	TSO only
DADPM [11, 12]	✓	✓	✓	✓

Contributions. The main contributions of this paper are:

- We present the first *dataflow analysis-based dynamic parallel monitoring* (DADPM) framework that can explicitly track uncertainty, thereby isolating known true errors from potential errors, with the guarantee that any false positive derives from a lifeguard check on metadata marked as "uncertain".
- We present (sound) formalizations for Butterfly Analysis with uncertainty extensions. To ease comparison to prior work, our extensions are based on reaching definitions, though our formalization is more similar to constant propagation. Expanding the metadata lattice from 2 to 3 states introduced a number of challenges into the formulas and proofs, e.g., dataflow *transfer* functions that are different from *meet* functions.
- We show how to use our uncertainty extensions for TAINTCHECK, a security lifeguard that is challenging to handle because taint status propagates from instruction sources to destinations.
- We present an approach for dynamically adapting the epoch size when uncertainty is detected. Our approach maintains soundness guarantees without requiring additional checkpointing. We show experimentally that for a variety of parallel applications being monitored by TAINTCHECK, our dynamic adaptations to uncertainty can deliver the precision comparable to small epochs with the performance comparable to large epochs.

II. BACKGROUND AND RELATED WORK

We begin with general lifeguard background before discussing DADPM in depth, and conclude with related work.

A. Lifeguard Frameworks

Lifeguards have been written to find security [1], memory [2] and concurrency [3, 4] bugs, among others. While lifeguard analysis always occurs dynamically, it can be either *synchronous* or *asynchronous*. In *synchronous* frameworks, lifeguard metadata is always updated immediately so that the input state of each instruction in the monitored application can be checked before it executes. This tight coupling—where metadata is updated atomically with the corresponding application event—delays the application whenever a slow metadata operation arises, and requires either transactional memory [5] or special hardware support [6]. In contrast, *asynchronous* (a.k.a., "decoupled" or "loosely coupled") frameworks allow metadata updates (and evaluations for correctness) to lag behind the events that triggered them,[1] which typically results in much higher performance (because it is more tolerant of bursts in metadata computation) [7].

Table I compares monitoring frameworks where both the program and the lifeguard are parallel programs. FlexiTaint [6] is specific to TAINTCHECK and hence not "general purpose". Thread-safe DBT [5] uses (software) transactional memory. Hardware-assisted parallel monitoring (HAPM) frameworks such as Paralog [8], PTAT/PTRT [9] and FADE [10] are general purpose and asynchronous, but they require special hardware support to track inter-thread data dependences and do not support memory consistency models weaker than TSO. DADPM, in contrast, is a general purpose, software-only, asynchronous approach that supports weak memory consistency memory models (specifically, any consistency model that provides at least cache-coherence). Because of these advantages, it is important to find low-overhead techniques for improving the precision of DADPM frameworks.

B. Butterfly Analysis

We first present the thread execution model common to DADPM frameworks before discussing Butterfly Analysis.

Thread Execution Model. Butterfly Analysis builds its thread execution model on the insight that modern multicore systems have only a finite amount of buffering. Taking into account the store buffer, reorder buffer, and maximum memory access latency, the authors observed that if two instructions on distinct threads are separated by "enough" instructions (on the order of 1000-10,000s of instructions or more), the earlier instruction must have retired, and any related store must have drained, before the later instruction was ever issued.

Butterfly Analysis leverages this insight by dividing execution into *epochs*. Epoch boundaries can be implemented in software via a token ring and using the processor's (memory) fence operation. Epochs are sized to account for the reorder buffer, store buffer, maximum memory access latency, and the time to circulate the token among all the threads. By construction, instructions in *non-adjacent* epochs, i.e., epochs that do not share a boundary, are ordered; this follows from how epoch boundaries are defined. Because epoch sizing takes into account all sources of buffering in the pipeline and memory system, every instruction in the earlier epoch must have been issued, retired, and all related stores must have drained before any instruction in a later, non-adjacent epoch could be issued. In Figure 3, epochs $l-1$ and $l+1$ are non-adjacent.

[1]Program safety is ensured by syncing up the program and lifeguard (only) at safety critical points, e.g., at system calls for TAINTCHECK.

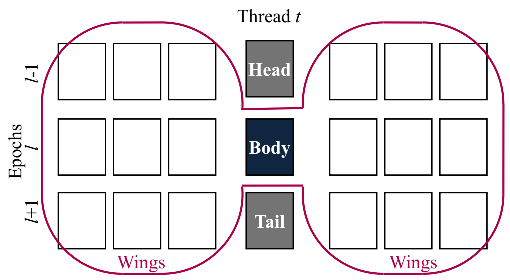


Figure 3: Butterfly Analysis' thread execution model.

However, *adjacent* epochs, i.e., epochs that share an epoch boundary, are not guaranteed to be ordered and in fact instructions can execute concurrently. This leads to a bounded three epoch sliding window of concurrency, shown in Figure 3. In Butterfly Analysis, an epoch-thread pair (l, t) defines a *block*. With respect to thread t in Figure 3 (each column is a thread), if instructions in the *body* (block (l, t)) are currently executing, then because a given thread can assume sequential semantics for its own instructions, we can assume that instructions in the *head* (block $(l-1, t)$) have already executed and instructions in the *tail* (block $(l+1, t)$) have not yet executed. Other threads' blocks within the sliding window are concurrent with the body and labeled the *wings*.

Incorporating Concurrency in Dataflow Analysis. In addition to the standard dataflow primitives of IN, OUT, GEN and KILL, Butterfly Analysis introduces two new dataflow primitives: SIDE-OUT and SIDE-IN. SIDE-OUT captures the effects of concurrency a block exposes to other concurrent threads (i.e., the effect of the butterfly's body on its wings). In contrast, SIDE-IN captures the effects of concurrency other threads expose to a concurrent block (i.e., the effect of the butterfly's wings on its body).

Lifeguards as two pass algorithms. Lifeguards are implemented as two pass algorithms for processing an epoch. An invariant is maintained that when processing epoch l, the *strongly ordered state* (*SOS*) has been computed, which summarizes all instructions up through epoch $l-2$. The *local strongly ordered state* (*LSOS*) at a thread t represents the SOS augmented with the effects of the head (block $(l-1, t)$). Lifeguard metadata and checks can be influenced by local state and/or events in the wings. In the first pass, dataflow analysis is performed independently in parallel at each thread using locally available state (i.e., ignoring the wings for that thread). Next, the lifeguard threads compute the *meet* of all the summaries produced in the wings. In a second pass, the dataflow analysis is repeated incorporating state from the wings, performing the lifeguard-specific checks. Finally, the threads collectively compute a summary of the entire epoch's activity, and update the SOS. The lifeguard writer specifies (only) the events the analysis will track, the meet operation, the metadata format, and the checking algorithm; these are plugged into the two pass framework and applied for every epoch during execution of the monitored program.

C. Chrysalis Analysis

Chrysalis Analysis [12] is a generalization of Butterfly Analysis designed to improve Butterfly Analysis' precision. While Butterfly Analysis avoids the overhead of tracking detailed inter-thread data-dependence traffic, it also fails to track semantically meaningful high-level synchronization-based happens-before arcs. We showed that incorporating high-level happens-before arcs based on synchronization (e.g., from an `unlock` to the next subsequent `lock` of the same lock variable) can vastly improve the precision of Butterfly Analysis, but at the cost of a more complicated thread execution model. Lifeguards are again implemented as two pass algorithms, as in Butterfly Analysis.

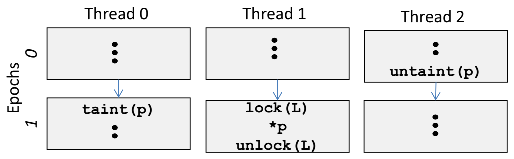
While Chrysalis Analysis greatly improves on the precision of Butterfly Analysis, both suffer from one key drawback: Of the potential errors both analyses report, neither can distinguish a true error that *must* have occurred on the monitored execution from a potential error that *may* have occurred.

D. Related Work

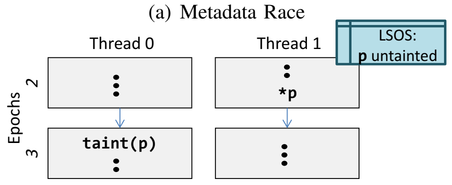
Traditional *static* analysis is also impacted by forms of uncertainty, but for entirely different reasons than in DADPM. For static analysis, *run-time data values* (e.g., the values of pointers) and *control flow* are often unknown at compile time. For this reason, previous work on pointer alias analysis [18, 19] and shape analysis [13, 14, 15] has explicitly distinguished *must* vs. *may* states, and abstract interpretation techniques [13, 14, 15, 16] have explicitly tracked uncertain or *unknown* states. In contrast, since DADPM is *dynamic* analysis, there is no uncertainty about run-time values or control flow: that information is explicitly captured in the execution logs that feed DADPM's analysis. The uncertainty in DADPM arises instead from the fact that the *interleaving* of events across parallel threads is unknown within DADPM's *window of uncertainty*, and this directly impacts how the lifeguard's *metadata* is to be updated and checked. As we will see later in Sections V and VI, because the uncertainty in DADPM arises from a fundamentally different source than in static analysis, it also affects the analysis differently.

III. Overview of Uncertainty

Incorporating uncertainty into dataflow analysis-based dynamic parallel monitoring requires two key changes: (i) a new state must be introduced to capture uncertainty, and (ii) all the dataflow equations must be updated: we are combining *must* analysis for precise states with *may* analysis for the uncertain state.



(a) Metadata Race



(b) Wings, local state conflict

Figure 4: Examples of uncertainty within Butterfly Analysis.

We motivate the incorporation of uncertainty using TaintCheck, as it is one of the most difficult lifeguards to adapt to DADPM, and the work necessary to adapt TaintCheck easily generalizes to other lifeguards that require both dataflow and propagation such as MemCheck [2, 20]. Tracking uncertainty within TaintCheck requires that the equations for when an address is `tainted` must now separate true taint from potentially tainted; potentially tainted addresses will be considered `uncertain`. We motivate our framework by examining different scenarios where uncertainty arises.

A. Uncertainty Examples

Consider Figure 4(a), depicting Threads 0 and 2 racing on pointer `p`: Thread 0 executes an instruction that taints `p` whereas Thread 2 executes an instruction that untaints `p`. 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 states. We will call this a *metadata race*.

Figure 4(b) illustrates how uncertainty can arise even without a metadata race. In the figure, the LSOS at Thread 1 indicates that `p` is `untainted`, whereas Thread 0 in epoch 3 taints `p`. Based on the thread execution model, we know that instructions that 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 its dereference of `p`, and hence whether (if before) or not (if after) `p` is tainted. Thus, despite the `untaint` and `taint` events being ordered, it is still the case that Thread 1 is `uncertain` of the metadata value for `p`.

B. Impact of Uncertainty on the DADPM Framework

While Sections V and VI describe how uncertainty is incorporated into DADPM in detail, we now present a qualitative discussion of the major changes. One fundamental difference is that while prior DADPM frameworks (i.e., without uncertainty tracking) model a *single* precise state (i.e., the "Provably Safe" case in Figure 2(a)), our enhanced DADPM framework (that tracks uncertainty) models *multiple* precise states (i.e., the "Provably Safe" and "True Errors" cases in Figure 2(b)). This difference causes fundamental changes in both the *analysis equations* and the *correctness proofs* (shown later in Sections V and VI). For example, the equations corresponding to the "Provably Safe" vs. "Potential Errors" cases in prior DADPM frameworks [11, 12] were *asymmetric*, because only the former was precisely tracked (and all other possibilities fell into the latter bucket). In addition, proofs of correctness in prior DADPM frameworks [11, 12] needed to show only that the "Provably Safe" case never contained false negatives.

In contrast, the enhanced DADPM framework has two major challenges. First, with three metadata states (including two precise "must" states), ensuring mutual exclusivity requires greater care and increased metadata versioning. Second, the inclusion of a more general `meet` function, which can `meet` contradictory precise states (between predecessors or the wings) into an uncertain state, complicates the new formulas for SIDE-IN, SIDE-OUT, SOS and LSOS. Combined, our formulas, invariants and correctness proofs are more complicated in order to deliver higher precision.

IV. LEVERAGING UNCERTAINTY

The benefits of explicitly tracking uncertainty are not limited to distinguishing between true and potential errors. This approach also provides opportunities for lifeguard innovation; we describe one such application here.

By enabling the lifeguard to dynamically distinguish between true errors and cases where insufficient information exists to make a precise judgment, it can attempt to dynamically increase the information available and improve precision. For example, if a fast but conservative analysis pass leads to an uncertain result, that pass can be rerun using a slower but more precise approach. Depending on their frequency, these dynamic adaptations may provide an opportunity to improve both precision and performance. The dynamic adaptations are an independent contribution to uncertainty and could also be implemented on top of any DADPM framework; pairing with uncertainty allows us to enable the adaptations only when precision improvements are possible.

Dynamically Resizing Epochs. Recall from Section II that epoch size is bounded from below but not from above; epochs need 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. Our prior work [11] 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.

A potential barrier to dynamically resizing epochs is the lag between when the epoch boundaries are emitted into the application's instruction stream and when the lifeguard encounters uncertainty; by the time uncertainty is discovered and desires smaller epochs, it is too late to change the epoch size. However, inserting epoch boundaries is relatively inexpensive, so we propose to "oversample" the epoch boundaries, and ignore the ones that are not needed. Correctness of the Butterfly model is maintained as long as all lifeguard threads skip the same boundaries. Dynamic epoch resizing may be enabled, then, by generating small epochs on the application side and coalescing those small epochs into larger ones on the lifeguard side. When a lifeguard thread encounters a potential error that requires more information to resolve, it must coordinate a rollback to the smaller epochs (whose boundaries are already correctly in the log) and restart analysis from there. As our experiments in Section VII show, only a small rollback is typically required–in fact, for most tested configurations, we can support rollback with minimal overhead (no new state checkpointing).

V. REACHING DEFINITIONS

Butterfly Analysis [11] is an DADPM framework: it is formulated as an adaptation of (traditionally static) dataflow analysis to a dynamic parallel monitoring setting. As such, the addition of an `uncertain` state to Butterfly Analysis involves modifying the dataflow equations. We will begin with calculating reaching definitions, a canonical dataflow analysis, and show how to add an `uncertain` state to the metadata lattice. It would be more natural to use constant propagation as our model rather than reaching definitions; uncertainty would be equivalent to an instruction yielding Not-A-Constant (NAC) in constant propagation. We instead use reaching definitions to ease comparison to the original.

A. First Pass Equations

We represent the (fully precise) dataflow primitives GEN and KILL as $\mathcal{G}$ and $\mathcal{K}$, respectively, and uncertainty ("maybe") as $\mathcal{M}$. We assume, for notational simplicity, an instruction exists with uncertain effects on definition d; d may or may not be generated.

Instruction-Level. If thread t in epoch l executes instruction i which {generates, kills, marks as uncertain} d, then d is in in $\mathcal{G}_{l,t,i}$, $\mathcal{K}_{l,t,i}$ or $\mathcal{M}_{l,t,i}$ (respectively).

Block-level. Let $\mathcal{G}_{l,t,(i,j)}$ be the set of definitions generated which are not subsequently killed or marked uncertain, restricted to consecutive instructions (l,t,i) through (l,t,j).

$$\mathcal{G}_{l,t,(i,i)} = \mathcal{G}_{l,t,i} \qquad \mathcal{K}_{l,t,(i,i)} = \mathcal{K}_{l,t,i} \qquad \mathcal{M}_{l,t,(i,i)} = \mathcal{M}_{l,t,i}$$

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

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

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

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:

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

B. Between Passes: Side-Out and Side-In

Computing SIDE-OUT is more complex than in prior work, beyond just needing sets for $\mathcal{G}$ (GSO), $\mathcal{K}$ (KSO), and $\mathcal{M}$ (MSO). For example, a block (l,t) which both generates and kills definition d creates uncertainty for other blocks that have (l,t) in their wings, thus $d \in \text{MSO}_{l,t}$. For clarity, we calculate intermediate sets $\text{ALL}^{\mathcal{M}}$, $\text{ALL}^{\mathcal{G}}$ and $\text{ALL}^{\mathcal{K}}$ first—these resemble GSO from original Butterfly Analysis—before calculating MSO, GSO and KSO as refinements.

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

Similarly, the three SIDE-IN sets (GSI, KSI, MSI) add complexity. For example, if a block (l,t) sees $d \in \text{GSO}_{l,t'}$ and $d \in \text{KSO}_{l,t''}$ (for distinct threads t, t', t''), we must take the `meet` and place $d \in \text{MSI}_{l,t}$. We use $\text{WING}^{\mathcal{M}}$, $\text{WING}^{\mathcal{G}}$ and $\text{WING}^{\mathcal{K}}$ as intermediate steps in the SIDE-IN calculations:

$$\text{WING}^{\mathcal{G}}_{l,t} = \bigcup_{\{t' \neq t\}} \bigcup_{\{l' | l-1 \leq l' \leq l+1\}} \text{GSO}_{l',t'}$$
$$\text{WING}^{\mathcal{K}}_{l,t} = \bigcup_{\{t' \neq t\}} \bigcup_{\{l' | l-1 \leq l' \leq l+1\}} \text{KSO}_{l',t'}$$
$$\text{WING}^{\mathcal{M}}_{l,t} = \bigcup_{\{t' \neq t\}} \bigcup_{\{l' | l-1 \leq l' \leq l+1\}} \text{MSO}_{l',t'}$$
$$\text{GSI}_{l,t} = \text{WING}^{\mathcal{G}}_{l,t} - (\text{WING}^{\mathcal{K}}_{l,t} \cup \text{WING}^{\mathcal{M}}_{l,t})$$
$$\text{KSI}_{l,t} = \text{WING}^{\mathcal{K}}_{l,t} - (\text{WING}^{\mathcal{G}}_{l,t} \cup \text{WING}^{\mathcal{M}}_{l,t})$$
$$\text{MSI}_{l,t} = \text{WING}^{\mathcal{M}}_{l,t} \cup (\text{WING}^{\mathcal{G}}_{l,t} \cap \text{WING}^{\mathcal{K}}_{l,t})$$

C. Incorporating Uncertainty Into State

Summarizing an epoch. We begin with the must-kill and must-generate equations to summarize an epoch, as both generate and kill are now precise states.[2] The prior epoch is included to account for potential interference between instructions in adjacent epochs. 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 $\mathcal{K}_{(l-1,l),t}$):

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

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

[2]If instead we wanted to represent may-kill and may-generate, we would use $\bigcup_t \mathcal{K}_{l,t}$ and $\bigcup_t \mathcal{G}_{l,t}$, respectively. Butterfly Analysis previously used may-generate and must-kill in epoch summaries.

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.

To accurately capture the uncertainty within an epoch, we need to account for separate sources of uncertainty. First, the difference between may-kill and must-kill (equivalently, may-generate and must-generate) captures the fact that orderings may exist in which two different outcomes are possible when examining all instructions in an epoch. We also capture any "organic" uncertainty (e.g., the last check of d in block (l,t) returns `uncertain`).

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

Invariants for epoch summaries. As in Butterfly Analysis, define a *valid ordering* of a set of instructions as any total ordering of the instructions that respects the partial order defined by the Butterfly Analysis thread execution model. The following invariants and proofs are made more challenging by the uncertainty extensions.

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

Proof: If $d \in \mathcal{G}_l$, then there exists thread t such that $d \in \mathcal{G}_{l,t} - (\bigcup_{t' \neq t} [\mathcal{K}_{(l-1,l),t'} \cup M_{(l-1,l),t'}])$. Let (l,t,i) be the last instruction in block (l,t) to generate 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 $d \notin (\bigcup_{t' \neq t} [\mathcal{K}_{(l-1,l),t'} \cup M_{(l-1,l),t'}])\ \forall t' \neq t$, then any kill or uncertain by a thread t' must be followed in the same thread by a subsequent gen that is the final "operation" on d, as $d \notin \mathcal{K}_{(l-1,l),t'}$ and $d \notin \mathcal{M}_{(l-1,l),t'}$. Thus, any kill or uncertain after instruction (l,t,i) is itself followed by a gen, so $d \in \mathcal{G}(O')$ and $d \in \mathcal{G}(O)$. ■

Lemma 2. *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: By symmetry to Lemma 1. ■

Lemma 3. *If any of the following 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:

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

Case 3: Follows by symmetry with case 2. ■

SOS Equations. There will now be three types of SOS. The equations that follow hold for $l \geq 2$; for $l = 0$ or $l = 1$, they are all identically empty.

$$\begin{aligned}
\mathrm{SOS}_l^{\mathcal{G}} &= \mathcal{G}_{l-2} \cup (\mathrm{SOS}_{l-1}^{\mathcal{G}} - (\mathcal{K}_{l-2} \cup \mathcal{M}_{l-2})) \\
\mathrm{SOS}_l^{\mathcal{K}} &= \mathcal{K}_{l-2} \cup (\mathrm{SOS}_{l-1}^{\mathcal{K}} - (\mathcal{G}_{l-2} \cup \mathcal{M}_{l-2})) \\
\mathrm{SOS}_l^{\mathcal{M}} &= \mathcal{M}_{l-2} \cup (\mathrm{SOS}_{l-1}^{\mathcal{M}} - (\mathcal{G}_{l-2} \cup \mathcal{K}_{l-2}))
\end{aligned}$$

SOS Invariants. We sketch the proofs for each SOS invariant (full proofs appear in [17]).

Lemma 4. *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 sketch (by induction): We sketch the inductive step. There are two cases following from $d \in \mathrm{SOS}_l^{\mathcal{G}}$, either (1) $d \in \mathcal{G}_{l-2}$ or (2) $d \in \mathrm{SOS}_{l-1}^{\mathcal{G}} - (\mathcal{K}_{l-2} \cup \mathcal{M}_{l-2})$.

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 instruction in $(l-2, t)$ kills d or marks d as uncertain. 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 $d \in \mathrm{SOS}_{l-1}^{\mathcal{G}} - (\mathcal{K}_{l-2} \cup \mathcal{M}_{l-2})$ then in particular $d \in \mathrm{SOS}_{l-1}^{\mathcal{G}}$ 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 $d \in \mathcal{G}(O)$. ■

Lemma 5. *If $d \in SOS_l^{\mathcal{K}}$ then $\forall$ valid orderings O of instructions in epochs $[0, l-2]$, $d \in \mathcal{K}(O)$.*

Proof: By symmetry to Lemma 4. ■

Lemma 6. *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 sketch (by cases, using induction): We consider each case in turn.

Case 1: We sketch the inductive step. Let (l', t, i) be the last instruction in O such that $d \in \mathcal{M}_{l',t,i}$. There are two cases, where (a) $l' = l-2$ or (b) $l' \leq l-3$.

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

(b): $l' \leq l-3$: 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, $d \in \mathrm{SOS}_{l-1}^{\mathcal{M}}$. We first consider if $l' < l-3$: then, no instruction in epoch $l-2$ can generate or kill d, else it would come after (l', t, i) in O implying $d \notin M(O)$ (a contradiction), so $d \notin \mathcal{K}_{l-2} \cup \mathcal{G}_{l-2}$ and thus $d \in \mathrm{SOS}_l^{\mathcal{M}}$. We next consider $l' = l-3$: Let $O_{[l-3,l-2]}$ be O restricted to epochs $[l-3, l-2]$, which includes (l', t, i)–still 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 1 and 2, $d \notin \mathcal{K}_{l-2} \wedge d \notin \mathcal{G}_{l-2}$, therefore $d \in \text{SOS}_l^{\mathcal{M}}$.

Case 2: We sketch the inductive step. The proof is a straightforward extension of the proof of Lemma 3, 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 3, 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 $d \in \text{SOS}_l^{\mathcal{M}}$. ■

Calculating local state. In calculating the LSOS, we consider the must-kill and must-gen versus may-kill and may-gen summaries for the head. 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).[3]

The must-{kill,gen} formulas for block (l, t)'s head are:

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

Lemma 7. *If $d \in \mathcal{G}^*_{l-1,t}$ then $\forall$ valid orderings O of instructions in epoch $l-2$ and block $(l-1, t)$, $d \in \mathcal{G}(O)$.*

Proof: If $d \in \mathcal{G}^*_{l-1,t}$ then $d \in \mathcal{G}_{l-1,t}$, so $\exists$ instruction $(l-1, t, i)$ in block $(l-1, t)$ which generates d and is not followed by a kill or an operation that marks d as uncertain. Consider the instructions in epoch $l-2$. 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 $\forall t' \neq t$, $d \notin K_{l-2,t'} \cup \mathcal{M}_{l-2,t'}$. So for any subblock $(l-2, t')$, it either also generates 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 $d \in \mathcal{G}(O)$. ■

Lemma 8. *If $d \in \mathcal{K}^*_{l-1,t}$ then $\forall$ valid orderings O of instructions in epoch $l-2$ and block $(l-1, t)$, $d \in \mathcal{K}(O)$.*

Proof: By symmetry with Lemma 7. ■

In representing what the head $(l-1, t)$ marks as uncertain, we include everything $(l-1, t)$ marked as uncertain as well as anything the head may-but-not-must have {gen, kill}.

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

The proof of Lemma 9 is omitted but strongly resembles Lemma 3.

Lemma 9. *If any of the following 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 $\mathcal{M}^*_{l,t} \supseteq \mathcal{M}_{l,t}$. Anything from the head $(l-1, t)$ that 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. We can express the LSOS equations, which strongly resemble SOS equations:

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

LSOS Invariants. We present invariants for the LSOS together with their proofs.

Lemma 10. *If $d \in LSOS^{\mathcal{G}}_{l,t}$ then $\forall$ valid orderings O of instructions in epochs $[0, l-2]$ and 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 d nor mark d uncertain, 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 d is either in $\mathcal{K}^*_{l-1,t}$ or in $\mathcal{M}^*_{l-1,t}$ (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 11. *If $d \in LSOS^{\mathcal{K}}_{l,t}$ then $\forall$ valid orderings O of instructions in epochs $[0, l-2]$ and block $(l-1, t)$, $d \in \mathcal{K}(O)$.*

Proof: By symmetry to Lemma 10. ■

Lemma 12. *If one of the following is true:*

[3] $\mathcal{K}_{l-1,t}$ ($\mathcal{G}_{l-1,t}$) is the may-kill (may-gen) set for block (l, t)'s head.

(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 6 holds and $d \notin (\mathcal{K}^*_{l-1,t} \cup \mathcal{G}^*_{l-1,t})$.

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

Proof (by cases): We will proceed by cases.

Case 1: Proceed as in Case 1 of Lemma 6. 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 $d \in \mathcal{M}^*_{l-1,t} \Rightarrow d \in \mathrm{LSOS}^{\mathcal{M}}_{l,t}$. Otherwise, $l' \leq l-2$; we can apply Case 1 of Lemma 6 to show that $d \in \mathrm{SOS}^{\mathcal{M}}_{l}$. 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 8 and 7 to yield that $d \notin \mathcal{K}^*_{l-1,t} \cup \mathcal{G}^*_{l-1,t}$, showing that $d \in (\mathrm{SOS}^{\mathcal{M}}_{l} - (\mathcal{K}^*_{l-1,t} \cup \mathcal{G}^*_{l-1,t}))$ and therefore $d \in \mathrm{LSOS}^{\mathcal{M}}_{l,t}$.

Case 2: By the contrapositive of Lemma 7, $d \notin \mathcal{G}^*_{l-1,t}$ (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 $d \in \mathrm{LSOS}^{\mathcal{M}}_{l,t}$.

Case 3: By symmetry with Case 2.

Case 4: As Lemma 6 holds, then $d \in \mathrm{SOS}^{\mathcal{M}}_{l}$. Furthermore, $d \notin (\mathcal{K}^*_{l-1,t} \cup \mathcal{G}^*_{l-1,t})$. If $d \in \mathrm{SOS}^{\mathcal{M}}_{l} - (\mathcal{K}^*_{l-1,t} \cup \mathcal{G}^*_{l-1,t})$ then $d \in \mathrm{LSOS}^{\mathcal{M}}_{l,t}$. ∎

VI. TaintCheck

In this section we present TaintCheck with uncertainty, building on Section V. TaintCheck is a "worst case" lifeguard for our framework, requiring not only dataflow but also *inheritance*. Further, our extensions are straightforward to adapt to other lifeguards requiring inheritance such as MemCheck [2, 20].

A. First Pass: Instruction-level Transfer Functions

Let $T_{l,t,i}$ represent the instruction-level transfer function that TaintCheck generates during its first pass over instruction (l, t, i):[4]

Algorithm 1 TaintCheck transfer(s_1, s_2)

```
Input: $s_1, s_2 \in$ {taint, untaint, 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
    return untaint
```

Algorithm 2 TaintCheck meet(s_1, s_2)

```
Input: $s_1, s_2 \in$ {taint, untaint, 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
    return uncertain
```

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

For a unary instruction $x :=$ unop(a), x *inherits (metadata) from* a, and likewise for a binary instruction $x :=$ binop(a, b), x *inherits (metadata) from* a *and* b. We use the set $S = \{$taint,untaint, uncertain,$\{a\}$,$\{a,b\}$ $\mid a, b$ mem locations$\}$ to represent the set of all possible right-hand values in our mapping. We will also utilize the function loc(l, t, i) that returns an instruction's destination x.

Calculating Side-Out and Side-In. 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 equations for a block (l, t) are:[5] $\texttt{tso}_{l,t} = \bigcup_i \mathcal{T}_{l,t,i}$ and $\texttt{tsi}_{l,t} = \bigcup_{l-1 \leq l' \leq l+1} \bigcup_{t' \neq t} \texttt{tso}_{l',t'}$.

B. Resolving Transfer Functions to Metadata

After the first pass in TaintCheck[6], the $\texttt{tso}_{l,t}$ and $\texttt{tsi}_{l,t}$ are available, but they are in the form of instruction-level transfer functions instead of metadata values. In order to perform checks (the purpose of the second pass), we require the metadata values associated with each destination address. To convert between transfer functions and metadata values in TaintCheck, we utilize a resolve algorithm (Algorithm 3). resolve considers possible paths of inheritance where metadata can flow between source and destination, and returns one of three possible return values: taint, untaint or uncertain. To maintain our guarantees that both taint and untaint are fully precise states, we will show that resolve returns taint (respectively, untaint) only when it

[4]For ease of comparison with prior work, we maintain $\bot$ as the symbol for taint, even though uncertain(?) is the new bottom in our metadata lattice. We do, however, deviate from the instruction-level transfer function notation used by the original Butterfly Analysis work, which reused $\mathcal{G}_{l,t,i}$.

[5]tso and tsi for TaintCheck are identical to GSO and GSI in prior work [11] when uncertainty is removed.

[6]Specifically, after the first pass over epoch $l+1$, the $\texttt{tso}_{l+1,t}$ are available to compute $\texttt{tsi}_{l,t}$.

Algorithm 3 TAINTCHECK $\texttt{resolve}(s,(l,t,i),T)$

```
Input: $s \in S$, $(l,t,i)$:instruction, $T$:wing transfer functions
if $s ==$ taint, untaint or uncertain then
    return $s$
else if $s == \{a\}$ for memory location $a$ then
    $a\_state_{\text{LSOS}}$ = metadata state of $a$ in LSOS
    $a\_state_{\text{WING}} = \texttt{do\_resolve}(a,t,(l,t,i),T,(l,t,i))$
    return meet$(a\_state_{\text{LSOS}}, a\_state_{\text{WING}})$
else if $s == \{a,b\}$ for memory locations $a,b$ then
    Compute $\{a,b\}\_state_{\text{WING}}$, $\{a,b\}\_state_{\text{LSOS}}$ as above
    $a\_state_{\texttt{meet}} = \texttt{meet}(a\_state_{\text{LSOS}}, a\_state_{\text{WING}})$
    $b\_state_{\texttt{meet}} = \texttt{meet}(b\_state_{\text{LSOS}}, b\_state_{\text{WING}})$
    return $\texttt{transfer}(a\_state_{\texttt{meet}}, b\_state_{\texttt{meet}})$
```

is the only possible metadata value the destination could inherit.

Challenge: Meet Function $\neq$ Transfer Function. `resolve` uses two subroutines: `transfer` (Algorithm 1) and `meet` (Algorithm 2). The `transfer` function is applied when evaluating a statement of the form $x = m_1 + m_2$. TAINTCHECK defines a destination address x to be tainted if either of its sources m_1 or m_2 is `tainted`, and `untainted` only if both sources m_1 and m_2 are `untainted`. 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` function conservatively calculates the "worst" of what the thread sees. Thus, unlike in the original Butterfly Analysis, the two functions are not identical: `transfer` returns `taint` if at least one of its inputs is `tainted`, while `meet` returns `taint` only if both its inputs are `tainted`.

Resolve Algorithm Incorporates Uncertainty. `resolve` takes as 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 $\mathcal{T}_{l,t,i} = (m \leftarrow s)$. It recursively evaluates transfer functions in the wings, subject to termination conditions, in a depth-first search fashion. $\texttt{resolve}((s,(l,t,i),T)$ is called initially, and $\texttt{do\_resolve}(s,tid,(l,t,i),T,H)$ is called recursively until exhaustion.[7] Algorithm 4 contains a pseudocode implemention of `do_resolve`.

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.[8] 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.

[7]In practice, 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`.

[8]Valid ordering rules preclude an instruction repeating itself.

Algorithm 4 TAINTCHECK $\texttt{do\_resolve}(m,tid,(l,t,i),T,H)$

```
Input: $m$: current destination, $tid$: original thread, $(l,t,i)$: current instruction, $T$: wing transfer functions, $H$: history of already explored instructions
if $m ==$taint, untaint or uncertain then
    return $m$
$num\_taint = num\_untaint = num\_uncertain = 0$
//recursively evaluate proper predecessors of $m$
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
        $a\_state_{\text{LSOS}}$ = metadata state of $a$ in LSOS
        $a\_state_{\text{WING}} = \texttt{do\_resolve}(a,tid,(l',t',i'),T,(l,t,i) :: H)$
        $resolve\_state = \texttt{meet}(a\_state_{\text{LSOS}}, a\_state_{\text{WING}})$
        (counter of $\{num\_taint, num\_untaint, num\_uncertain\}$ that matches $resolve\_state$)++
    else if $s_j = \{a,b\}$ for memory locations $a,b$ then
        Compute $\{a,b\}\_state_{\text{WING}}$, $\{a,b\}\_state_{\text{LSOS}}$ as above
        $a\_state_{\texttt{meet}} = \texttt{meet}(a\_state_{\text{LSOS}}, a\_state_{\text{WING}})$
        $b\_state_{\texttt{meet}} = \texttt{meet}(b\_state_{\text{LSOS}}, a\_state_{\text{WING}})$
        $resolve\_state = \texttt{transfer}(a\_state_{\texttt{meet}}, b\_state_{\texttt{meet}})$
        (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
    return taint
else
    return untaint
```

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.

C. Second Pass: Performing Checks

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 resolves the transfer function $\mathcal{T}_{l,t,i} = m \leftarrow s$ to a metadata value for destination address m. Let:

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

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

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

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`.

To convert GSI, KSI or MSI to state, we first define, for $s \in \{\bot, \top, ?\}$:

$$\text{ALL}^s_{l,t} = \{m | \exists i \text{ s.t. } \mathcal{T}_{l,t,i} = (m \leftarrow p) \wedge \texttt{resolve}(p,(l,t,i),\texttt{tsi}_{l,t}) \leftarrow s\}$$
$$\text{ALL}^{\mathcal{G}}_{l,t} = \text{ALL}^{\bot}_{l,t} \quad \text{ALL}^{\mathcal{K}}_{l,t} = \text{ALL}^{\top}_{l,t} \quad \text{ALL}^{\mathcal{M}}_{l,t} = \text{ALL}^{?}_{l,t}$$

Then the equations for $\text{GSO}_{l,t}$, $\text{KSO}_{l,t}$ and $\text{MSO}_{l,t}$ immediately follow, as do those for $\text{WING}^{\mathcal{G}}_{l,t}$, $\text{WING}^{\mathcal{K}}_{l,t}$ 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 13. *If* `resolve`$(s,(l,t,i),\texttt{tsi}_{l,t})$ *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 sketch: We show the full proof for $s = \{a\}$. (All cases appear in [17].)

`resolve` 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 11 so that every valid ordering of instructions in epochs $[0, l-2]$ and block $(l-1, t)$ must have a `untainted`.

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 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, and m can only inherit `untaint` from a. ■

Lemma 14. *If* `resolve`$(s,(l,t,i),\texttt{tsi}_{l,t})$ *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 sketch: As in Lemma 13, we show the full proof for $s = \{a\}$. (All cases appear in [17].) Lemma 10 guarantees that every valid ordering of instructions in epochs $[0, l-2]$ and block $(l-1, t)$ must have m `tainted`. As in Lemma 13, `resolve` explores all proper predecessors, and will not miss an ordering of instructions in the wings that could potentially lead to a being `untainted` or `uncertain`. 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 13. ■

Theorem 15. *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 butterfly analysis with uncertainty extensions 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 13, `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 14. 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`. ■

VII. 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 [21, 22, 23]. We present results for a word-granularity Butterfly Analysis implementation of TAINTCHECK that incorporates uncertainty, as described in Section VI. In addition to implementing and testing uncertainty, we also executed experiments to measure the benefits of dynamic epoch resizing as described in Section IV.

We synthetically tainted 15% (expected) of the benchmarks' input data at random (using a fixed seed to ensure reproducibility); successive runs for the same benchmark with different epoch size parameters experienced the same input data being tainted. Taint injection occurred once, at the beginning of the parallel phase; taint propagation continued until program termination.

All experiments were run on the Intel OpenCirrus cluster (opencirrus.intel-research.net/). Each experiment was run inside an identically configured virtual machine hosted on

Table II: Benchmark Parameters

Benchmark	Inputs
BARNES	2048 bodies
FFT	$m = 20$ (2^{20} sized matrix)
OCEAN	Grid size: 258×258
LU	Matrix size: 1024×1024

an 8-core (2 quadcore Xeon E5440 processors) machine with 8GB of available RAM; to manage resource contention, each machine only ran one experiment at a time. We used a trace-based approach to gather performance and precision measurements for different effective epoch sizes while controlling for the underlying interleaving; the lifeguards under test consumed the traces natively. Using LBA [7], which is implemented on top of Simics [24], we gathered traces of thread execution which included heartbeats sized at 1K instructions/thread.[9] Each trace was gathered with the benchmark running with four threads. For compatibility with LBA, a 32-bit Linux OS was used with kernel 2.6.20-16-server. Any epoch elision was performed as a pre-processing step before the experiments were timed. Table II describes the Splash-2.0 [25] benchmarks used.

A. Dynamic Epoch Resizing

Performance and precision measurements were taken for several configurations. First, performance and precision results were gathered using an epoch size of 1K instructions/thread (labeled SMALL). Using the same traces, we then gathered results for LARGE ($16K$ instructions/thread) effective epoch sizes by only respecting every 16th epoch boundary.[10] Any large epochs that 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. In a large run, we assume epoch divisions that correspond with Figure 5(a). Suppose a thread observes a failed check of UNCERTAIN in epoch l. Under the first scheme, DYNAMIC$_{[l-1,l+1]}$emits the smaller epochs corresponding to larger epochs $[l-1, l+1]$, as shown in Figure 5(b). In a real system, DYNAMIC$_{[l-1,l+1]}$ would incur the cost of rollback of the second pass of epoch $l-1$ as well as undoing an SOS update. Two more practical versions of dynamic epoch resizing were

[9]For n threads, we inserted heartbeats after the LBA simulator observed $n \times 1024$ instructions executed globally.

[10]To reduce complexity in handling edge cases in the prototype, small epoch boundaries were always used when threads are exiting.

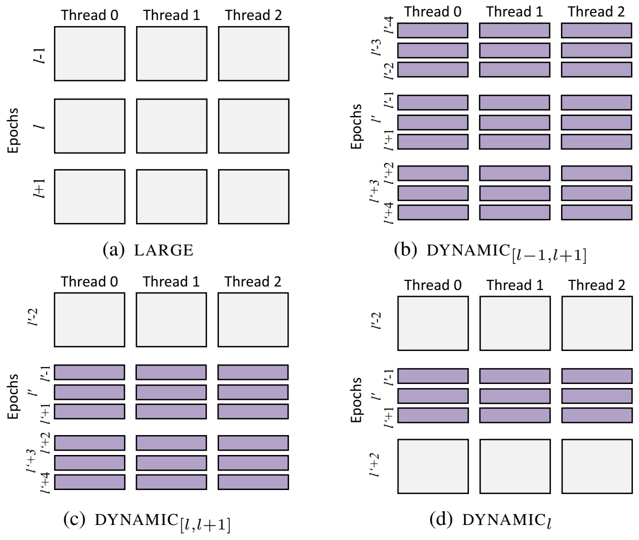


Figure 5: Dynamic epoch resizing

also tested: DYNAMIC$_{[l,l+1]}$(Figure 5(c)), which emits the smaller epochs corresponding to larger epochs $[l, l+1]$, and DYNAMIC$_l$ (Figure 5(d)), which emits the smaller epochs corresponding only to large epoch l.

B. Types of Uncertainty

Our experiments will track two types of uncertainty: `heuristic` and `uncertain`. `heuristic` is a specific type of uncertainty that occurs because we intentionally cut off the recursive exploration of `resolve` when reaching either a threshold for exploring potential parents in the wings (in our experiments, we set the threshold at 512 parents) or a threshold for the maximum number of parents tracked via registers. `uncertain` is a catch-all that captures every other type of uncertainty in the system.

VIII. Evaluation

We now evaluate the precision and performance of the dynamic epoch resizing scheme.

A. Precision

Table III 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 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 eliminate false positives; we see a 0 in the column titled *Failed Taint* for

Table III: Precision results for all tested configurations.

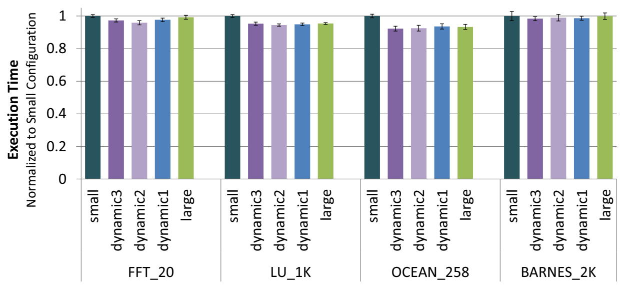
Benchmark	Epoch Size	Failed Taint	Failed Uncertain	Failed Heuristic
FFT-20	SMALL	0	0	0
	DYNAMIC$_{[l-1,l+1]}$	0	0	0
	DYNAMIC$_{[l,l+1]}$	0	0	0
	DYNAMIC$_{l}$	0	0	0
	LARGE	0	3	3
LU-1K	SMALL	0	0	0
	DYNAMIC$_{[l-1,l+1]}$	0	0	0
	DYNAMIC$_{[l,l+1]}$	0	0	0
	DYNAMIC$_{l}$	0	0	0
	LARGE	0	3	3
OCEAN-258	SMALL	0	2	0
	DYNAMIC$_{[l-1,l+1]}$	0	2	0
	DYNAMIC$_{[l,l+1]}$	0	2	0
	DYNAMIC$_{l}$	0	2	0
	LARGE	0	38	6
BARNES-2K	SMALL	0	0	0
	DYNAMIC$_{[l-1,l+1]}$	0	12	0
	DYNAMIC$_{[l,l+1]}$	0	12	0
	DYNAMIC$_{l}$	0	12	0
	LARGE	0	66	16

all experiments, indicating that no code checks failed due to a false check of TAINT. This improves upon previous work [11, 12] which observed potential errors that it could not disambiguate from true errors; for each row in Table III, the value of *Failed Taint* in Butterfly Analysis is the sum of the *Failed Uncertain*, *Failed Heuristic* and *Failed Taint* values.

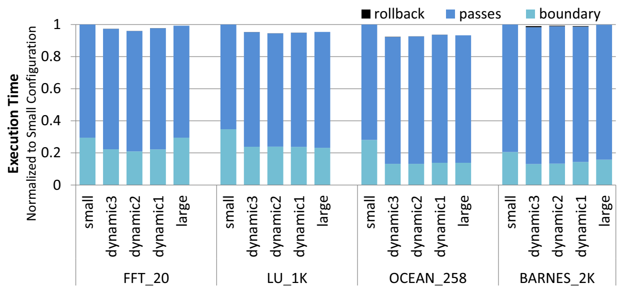
B. *Performance*

Figure 6(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, we see that DYNAMIC$_{[l-1,l+1]}$ DYNAMIC$_{[l,l+1]}$ and DYNAMIC$_{l}$ run 8–9% faster than SMALL. No dynamic scheme consistently outperforms the others; rather, each has at least one benchmark where it performs the best. Thus, considering both precision and performance, either DYNAMIC$_{[l,l+1]}$ or DYNAMIC$_{l}$ is a competitive choice that achieves good results with minimal rollback cost.

While these experiments show that the DYNAMIC schemes achieve roughly the precision of the SMALL epochs for only the cost of the LARGE epochs, the overall performance savings was much less than expected, primarily because LARGE epochs were only modestly faster than SMALL epochs. Because prior work (using a different lifeguard) had reported a significant performance gain from using large epochs instead of small epochs [11], we were surprised by this result.



(a) Average execution time, with 95% confidence intervals



(b) Execution subdivided into `rollback`, `passes` and `boundary` phases

Figure 6: Average execution time (restricted to parallel execution), (a) with 95% confidence intervals and (b) subdivided into phases.

To explore why larger epoch sizes were not having the expected speedup for the TAINTCHECK lifeguard we studied, we 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 that 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 that 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 6(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 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.

IX. Conclusion

Dataflow Analysis-based Dynamic Parallel Monitoring (DADPM) is the only current approach to monitoring parallel programs that offers all of the following advantages: loosely-coupled parallel lifeguards, weak memory consistency support, software-only, and general purpose. In this paper, we improved upon DADPM by adding support for explicitly tracking and dynamically reducing uncertainty, in order to greatly reduce its false positives. These extensions and their proofs of soundness required significant and subtle changes to prior DADPM approaches. We showed how to enhance the popular TAINTCHECK lifeguard, a particularly challenging lifeguard, with our uncertainty extensions, and used it to evaluate the effectiveness of dynamically adapting the epoch size in response to uncertainty. Our experiments showed that, without adding any checkpointing overhead, we could obtain the best of small and large epoch sizes: the high precision of small epochs and the low cost of large epochs.

Acknowledgments

This research was funded in part by the Intel Science and Technology Center for Cloud Computing and by a grant from the National Science Foundation. The second author contributed to this work while with Intel Labs Pittsburgh.

References

[1] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," in *NDSS*, 2005.

[2] N. Nethercote and J. Seward, "Valgrind: A Program Supervision Framework," *Elec. Notes in Theor. Comp. Sci.*, vol. 89, no. 2, 2003.

[3] 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*, vol. 15, no. 4, 1997.

[4] C. Flanagan and S. N. Freund, "FastTrack: Efficient and Precise Dynamic Race Detection," in *PLDI*, 2009.

[5] J. Chung, M. Dalton, H. Kannan, and C. Kozyrakis, "Thread-Safe Dynamic Binary Translation using Transactional Memory," in *HPCA*, 2008.

[6] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "FlexiTaint: A Programmable Accelerator for Dynamic Taint Propagation," in *HPCA*, 2008.

[7] 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.

[8] 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.

[9] H. Kannan, "Ordering Decoupled Metadata Accesses in Multiprocessors," in *MICRO*, 2009.

[10] S. Fytraki, E. Vlachos, Y. O. Kocberber, B. Falsafi, and B. Grot, "FADE: A programmable filtering accelerator for instruction-grain monitoring," in *HPCA*, 2014.

[11] 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.

[12] 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.

[13] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv, "Precise and compact modular procedure summaries for heap manipulating programs," in *PLDI*, 2011.

[14] G. Yorsh, T. Reps, M. Sagiv, and R. Wilhelm, "Logical characterizations of heap abstractions," *ACM Trans. Comput. Logic*, vol. 8, no. 1, Jan. 2007.

[15] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," in *POPL*, 1999.

[16] T. Reps, M. Sagiv, and R. Wilhelm, "Static Program Analysis via 3-Valued Logic," in *Computer Aided Verification*, ser. LNCS, R. Alur and D. Peled, Eds. Springer, 2004, vol. 3114.

[17] M. L. Goodstein, "Dataflow Analysis-Based Dynamic Parallel Monitoring," PhD Thesis, Carnegie Mellon University, August 2014, CMU-CS-14-132.

[18] M. Emami, R. Ghiya, and L. J. Hendren, "Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers," in *PLDI*, 1994.

[19] M. Hind, M. Burke, P. Carini, and J.-D. Choi, "Interprocedural Pointer Alias Analysis," *ACM TOPLAS*, vol. 21, no. 4, 1999.

[20] N. Nethercote and J. Seward, "How to Shadow Every Byte of Memory Used by a Program," in *VEE*, 2007.

[21] ——, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *PLDI*, 2007.

[22] 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.

[23] D. Bruening, "Efficient, Transparent, and Comprehensive Runtime Code Manipulation," Ph.D. dissertation, MIT, 2004.

[24] Virtutech Simics, http://www.virtutech.com/.

[25] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *ISCA*, 1995.