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

Michelle L. Goodstein[1], Shimin Chen[2], Phillip B. Gibbons[3], Michael A. Kozuch[3], Todd C. Mowry[1]
[1]Carnegie Mellon University [2]HP Labs China [3]Intel Labs Pittsburgh
mgoodste@cs.cmu.edu, shimin.chen@hp.com,
{phillip.b.gibbons, michael.a.kozuch}@intel.com, tcm@cs.cmu.edu

ABSTRACT

Software *lifeguards*, or tools that monitor applications at runtime, are an effective way of identifying program errors and security exploits. 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*. In prior work, we introduced a new software framework for online parallel program monitoring inspired by dataflow analysis, called Butterfly Analysis. Butterfly Analysis uses bounded windows of uncertainty to model the finite upper bound on delay between when an instruction is issued and when all its effects are visible throughout the system. While Butterfly Analysis offers many advantages, it ignored one key source of ordering information which affected its false positive rate: explicit software synchronization, and the corresponding high-level happens-before arcs.

In this work we introduce *Chrysalis Analysis*, which extends 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—parallel monitoring, no detailed inter-thread data dependence tracking, no strong memory consistency requirements, and no missed errors—while significantly reducing the number of false positives.

Categories and Subject Descriptors

F.3.2 [**Logics & Meanings of Programs**]: Semantics of Programming Languages—*Program Analysis*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Monitors*

Keywords

Data Flow Analysis, Parallel Programming, Dynamic Program Monitoring, Vector Clocks, High-Level Synchronization

PACT'12, September 19–23, 2012, Minneapolis, Minnesota, USA.

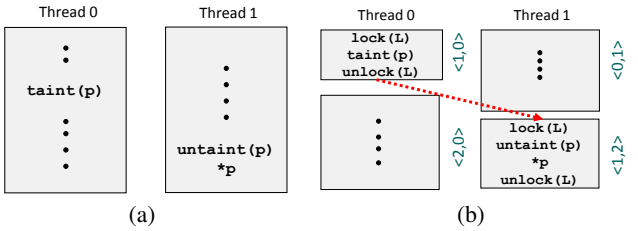


Figure 1: (a) Butterfly Analysis ignores synchronization arcs (such as from locks), and hence views the `taint(p)` and `*p` as racing if they are close in time, even if the source code resembles (b). (b) Chrysalis Analysis eliminates such false positives by dynamically capturing explicit synchronization arcs. (Note that `[un]taint(p)` indicates an application operation that would cause the lifeguard to [un]set the "tainted" metadata value for `p`.)

1. INTRODUCTION

To help programmers identify software bugs, a number of powerful tools have been developed for recognizing incorrect behaviors (e.g., memory [15], security [16], and concurrency [17] bugs) by performing sophisticated analysis of the dynamic execution path at run-time. These dynamic tools (aka *"lifeguards"*) are typically implemented using either dynamic binary instrumentation (e.g., Valgrind [14], Pin [11], or DynamoRio [3]) or with hardware-assisted logging [4].

Lifeguards often analyze program execution at the instruction level by maintaining shadow state (*metadata*) for program memory and register locations [14]. Examples of lifeguards discussed throughout this paper include ADDRCHECK [13] and TAINTCHECK [16]. ADDRCHECK ensures that every memory access in the monitored application touches a properly allocated region of memory by intercepting calls to library routines such as `malloc` and `free` to maintain metadata state, and checking the state on every memory access. TAINTCHECK detects security exploits due to memory overwrites by tagging as *tainted*, in its metadata, any application memory locations corresponding to unverified input data, tracking the propagation of tainted data through the application, and checking that tainted data is not used in indirect jump targets, system call arguments, and other critical operations.

Parallel Monitoring via Butterfly Analysis. A key challenge in extending lifeguards such that they can find bugs in *parallel* software is dealing with *inter-thread data dependences* [9, 24]. In contrast to proposals for hardware-assisted solutions [24], our recent work introduced "Butterfly Analysis" [9] as a *software-only* solution. Butterfly Analysis avoids depending upon capturing a precise interleaving of events across threads (which is impractical to capture on modern machines) and instead, uses a new form of

Table 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 [24]	Yes	Yes	TSO only	high	high
Chrysalis Analysis	*No*	*No*	*Yes*	*good*	*high*
Butterfly Analysis [9]	No	No	Yes	fair	fair

dataflow-style analysis (inspired by *interval analysis* [21]) that automatically reasons about *bounded windows of uncertainty*. Such windows capture the fact that on modern machines there is an upper bound on the delay between when an instruction is issued and when all its effects are reflected system-wide. Thus, uncertainty concerning the relative ordering among events on different threads is limited to instructions issued within a bounded window of time of each other. In order to avoid considering a combinatorial explosion of potential orderings, Butterfly Analysis instead performs a new form of "closure" operation across sliding windows of uncertainty.

False Positives in Butterfly Analysis. 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 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.

Chrysalis Analysis: Adding Happens-Before Arcs to Butterfly Analysis. In this paper, 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 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 3. 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.

Related Work. This work significantly extends our previous work on Butterfly Analysis [9], which we will review in more detail later in Section 2, making use of vector clocks to track synchronization events. Vector clocks [1, 5, 19] have been used in a number of data race detectors [2, 7, 8, 12, 18, 26]. For example, Flanagan and Freund proposed *FastTrack* [8], 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 [17]. Muzahid *et al.* [12] 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 implementing a broad range of sophisticated lifeguards.

Prior work in adapting lifeguards to parallel applications falls into two categories: *synchronous* frameworks, where lifeguard metadata values are updated immediately when triggering application operations are encountered, and *non-synchronous* frameworks, where the metadata values are updated more lazily. The primary challenge with synchronous approaches (e.g., binary instrumentation [3, 11, 14]) is ensuring that the lifeguard metadata is updated atomically with the application data. Synchronous examples include Chung *et al.* [6], which relies on transactional memory and exhibits significant performance slowdowns, and FlexiTaint [22], which requires extensive hardware support. Non-synchronous approaches, such as Kannan [10] and ParaLog [24], focus on enabling a parallel lifeguard to reproduce the order of application events. While these designs provide good monitoring fidelity, they require hardware to monitor cache coherency and are unable to handle memory-ordering models weaker than TSO [20].

Table 1 compares Chrysalis Analysis with Butterfly Analysis and ParaLog [24]. 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 (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.

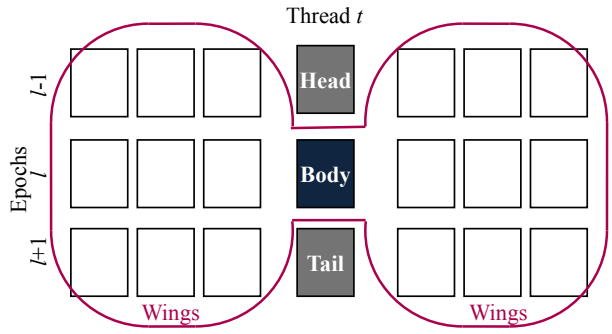
Contributions. This paper makes the following contributions:

- 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 our prior work [9], 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.
- 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.

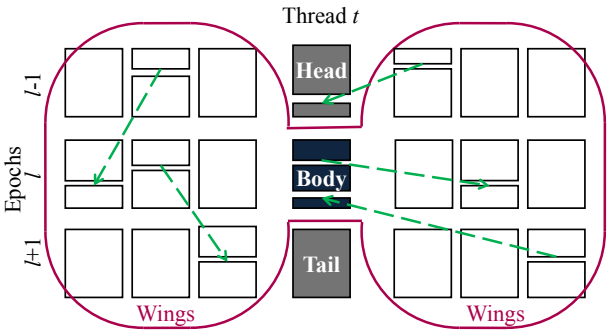
2. BACKGROUND: BUTTERFLY ANALYSIS

Butterfly Analysis [9] is a parallel dataflow analysis framework supporting lifeguard analysis of parallel applications. Unlike traditional dataflow analysis, which performs static analysis on control flow graphs, Butterfly Analysis analyzes *dynamic* execution traces of instructions on different threads.

[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 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.

Insights and Structure of Butterfly Analysis. Butterfly Analysis is based on the observation that modern systems have only a finite amount of buffering. Even taking into account memory access latency, and store reorder buffer sizes, if two instructions are issued sufficiently "far apart" (meaning many instructions issued between them) then they could not have both been buffered within the system simultaneously. Butterfly Analysis leverages this observation *to extract a partial order of a parallel application's events without monitoring detailed inter-thread data dependence traffic.*

Butterfly Analysis achieves this goal by dividing execution time into epochs. The interval between issuing epoch boundaries is constructed to account for the reorder buffer, store buffer, memory latency and skew in delivering notice of an epoch change. Epoch boundaries are communicated to cores, using either a software-only token ring combined with a memory fence or a simple piece of hardware. Note that epoch boundaries need not be established instantaneously—the sole requirement is a guaranteed maximum skew among the times a given epoch boundary is established at every core.

Note that by construction, two instructions in *non-adjacent epochs* (i.e., epochs which do not share an epoch boundary) cannot interleave. This follows from how epochs are defined: instructions in non-adjacent epochs were already implicitly ordered by the system, as the earlier instruction must have committed, with any related store draining from the store buffer, before the later instruction was even issued. In Figure 2(a), instructions in epoch $l-1$ must all have committed with any related store draining before any instruction in epoch $l+1$ is issued. This motivates the use of a 3-epoch sliding window in Figure 2(a); instructions separated by an epoch or more are automatically ordered.

On the other hand, instructions in *adjacent* epochs, i.e., epochs that share an epoch boundary, are assumed to be potentially concurrent when they are not in the same thread. For example, in Figure 2(a), epochs $l-1$ and l are adjacent, as are epochs l and $l+1$. Figure 2(a) depicts *blocks*[2] of instructions for three epochs across seven threads, summarizing the potential concurrency with instructions by thread t in epoch l (labeled the **body** of the "butterfly"). Instructions in the body of the butterfly are concurrent with instructions in the **wings**, but ordered with respect to the **head** and **tail**, as intra-thread data dependences are still respected. In reality, epoch boundaries will be staggered due to skew in delivery latency, and blocks will be of somewhat varying sizes.

Two-Pass Lifeguard Analysis. Analysis proceeds one epoch at a time, using a sliding window of three epochs. Lifeguard threads, one per application thread, execute in parallel and lag at least two epochs behind the application threads. Unlike traditional dataflow analysis, Butterfly Analysis maintains global state to summarize earlier epochs that are no longer within the window. The *Strongly Ordered State (SOS)* represents the global metadata resulting from all instructions *executed at least two epochs prior* to epoch l. Each lifeguard thread operating on a butterfly also knows that the *head* has already executed relative to the *body* for its application thread (Figure 2(a)), and augment the SOS with this additional, locally known information to form their thread's *Local Strongly Ordered State (LSOS)*.

Analysis for a given epoch proceeds in two passes. In the first pass, each lifeguard thread performs dataflow analysis using only local events, and produces a summary or SIDE-OUT of lifeguard-relevant events. In the next phase, lifeguard threads compute the *meet* of these summaries to create the SIDE-IN. In the second pass, dataflow analysis is repeated, this time including the SIDE-IN, and lifeguard checks are performed. Finally, a summary of the entire epoch's execution is created, and used to update the SOS.

The events the dataflow analysis will track, the meet operation, the metadata format, and the checking algorithm are all specified by the lifeguard writer. Tolerating uncertainty introduces imprecision into the dataflow analysis. Our prior paper [9] proved that Butterfly Analysis does not miss any errors (zero *false negatives*), but it sacrifices precision in the form of *false positives* arising from the lack of a relative ordering among events within a sliding window.

3. 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.

3.1 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 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 3(b). Here, we see Thread 1 issuing `untaint(p); unlock(L)`. After Thread 1 unlocks `L`, Thread 0 is the next thread to

[2]A block is specified by an epoch-thread tuple (l,t).

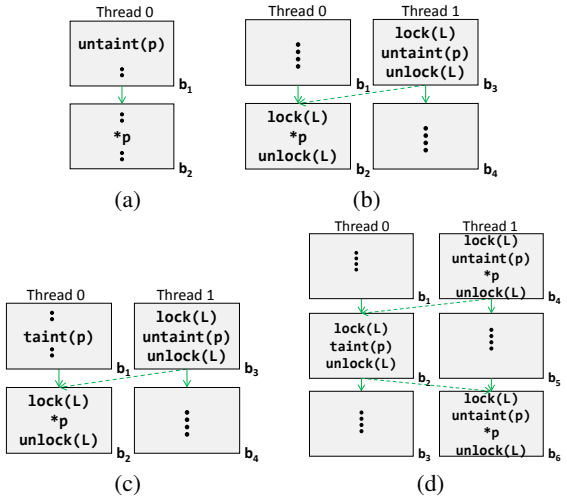


Figure 3: TAINTCHECK examples for dereferencing a pointer p. (a) In single-threaded execution, respect of intra-thread dependences implies p is untainted. (b) The synchronization between Thread 1 and Thread 0 means block b_3 executes before block b_2. Since b_1 does not assign to p, p is untainted. This 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.

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 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 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 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 2(b)). Including additional partial ordering information, while making the analysis more precise, also makes the analysis much more difficult!

3.2 Maximal Subblocks

Consider again Figures 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.[3] 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 1(b) and 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, (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'.

3.3 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 [1, 2, 5, 8, 19]; they are a natural distributed clock primitive. We are using them here to label individual subblocks based on synchronization events and epoch boundaries.[4] The addition of vector clocks and subblocks transforms the Butterfly Analysis diagram, shown in Figure 1(a), into Figure 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

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

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

[3] 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.

[4] 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.

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.

3.4 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 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 3.1. *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, 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.

3.5 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 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 possible 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 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.

3.6 Challenge: Updating Local State

Once again, consider Figure 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 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. 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). Later in Section 5 we will give an example lifeguard, TAINTCHECK, based on reaching definitions. Extensions of Chrysalis Analysis to Available Expressions, another classical dataflow analysis problem, and ADDRCHECK, a memory lifeguard, appear in Appendices A and B, respectively.

4.1 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. For maximal subblocks b and b', where $b=(l,t,(j,k))$, 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 NBEFORE, 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:

$$\text{MB}_l = \{b | b \text{ is a maximal subblock in epoch } l\}.$$
$$\text{AFTER}_b = \{b' | b' \in \text{MB}_{l^+} \text{ and } v(b) < v(b')\}$$
$$\text{NBEFORE}_b = \{b' | b' \in (\text{MB}_{l-1} \cup \text{MB}_{l^+}) \wedge (v(b) \sim v(b') \vee v(b) < v(b'))\}$$
$$\mathcal{G}_l = \bigcup_{\{b | b \in \text{MB}_{l^+}\}} \left(\mathcal{G}_b - \bigcup_{b' \in \text{AFTER}_b} \mathcal{K}_{b'} \right)$$
$$\mathcal{K}_l = \bigcup_{\{b | b \in \text{MB}_{l^+}\}} \left(\mathcal{K}_b - \bigcup_{\{b' | b' \in \text{NBEFORE}_b\}} \mathcal{G}_{b'} \right)$$

LEMMA 1. *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 (l,t,i)[5] in the total order O_l such that (l,t,i) generates d and no subsequent instruction kills d. Let b be the maximal subblock containing (l,t,i). By the definition of VVO, there is no instruction (l',t',i') that kills d such that either $v(l,t,i) < v(l',t',i')$ or (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 2. *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, $d \notin \mathcal{G}(O)$. □

[5]Actually, the instruction must exist in l^+, not just in l. For simplicity of exposition, we chose to use l.

4.2 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 $\mathcal{K}_l$.

$$\text{SOS}_0 = \text{SOS}_1 = \varnothing$$
$$\text{SOS}_l = \mathcal{G}_{l-2} \cup (\text{SOS}_{l-1} - \mathcal{K}_{l-2}) \; \forall l \ge 2$$

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 3. *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 1. 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 1, 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:

$\tilde{l} \ge l$: 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 $d \in \text{SOS}_{l+2}$.

□

4.3 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 3.6, 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 $(l-1,t)$):

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

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 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 OUT = GEN ∪ (IN − 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 3(c), where b_1 and b_3 both occur before b_2 but the taint status of 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 4. *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 5. *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 $d \notin \text{GEN-SIDE-OUT}_{b''}$ for all $b'' \in \text{DEL-INK}_{b'}$, 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, $\text{LSOS}_{l,t,i}$. Then $\text{LSOS}_b = \text{IN}_b^{\mathcal{G}} \cup (\text{SOS}_l - \text{IN}_b^{\mathcal{K}})$.

THEOREM 6. *If $\exists$ a valid vector ordering O of the instructions from epochs $[0, (l-2)^+]$ and $\text{HB}(b)$ such that $d \in \mathcal{G}(O)$, then $d \in \text{LSOS}_b$.*

PROOF. The proof follows from a straightforward extension of Lemma 4. Instead of limiting ourselves to an ordering of instructions in $\text{HB}(b)$, we consider all instructions from epochs $[0, (l-2)^+]$ and $\text{HB}(b)$. Then if the instruction (l', t', i') generating d has $l' > l-2$, Lemma 4 dominates. Otherwise, it is still the case the d is not in a later kill set (represented by $\text{IN}_b^{\mathcal{K}}$). If we restrict the ordering to the first $[0, (l-2)^+]$ epochs, this is the same as the proof that $d \in \text{SOS}_l$ (Theorem 4.2), so $d \in \text{SOS}_l - \text{IN}^{\mathcal{K}}{}_b$. □

4.4 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 $b = (l, t, (j, j'))$:

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

4.5 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.

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 $S = \{\top, \bot, \{a\}, \{a,b\} | a, b \text{ are memory locations}\}$ 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). 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. Our previous work [9] 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 1. 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 $v(l,t,i) \not< v(l',t',i')$.

The sequential consistency and relaxed memory consistency models termination conditions[6] are maintained. 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.

[6]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 [9].

Algorithm 1 TAINTCHECK `resolve`(m,l,t,i)

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 $(x_{l,t,i} \leftarrow \top)$.

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.

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 $\bot$. More formally:

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

If the `resolve` function for a location m ever returns $\bot$, $m \in$ 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 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 4.3, 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:

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

$$\text{IN}_b^{\mathcal{K}} = \bigsqcap_{b' \in \text{pred}(b)} \left(\text{OUT}_{b'}^{\mathcal{K}} - \left(\bigcup_{b'' \in \text{DEL-IN}\mathcal{K}_{b'}} \text{GEN-SIDE-OUT}_{b''} \right) \right)$$

At first, it seems paradoxical. However, the situation is salvageable after making a key observation. If $b'' \in$ DEL-IN$\mathcal{K}_{b'}$ and $v(b'') \not< v(b)$, 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-

IN$^{\mathcal{K}}$, $v(b) \not< v(b'')$, a contradiction.) In other words, when we first need to calculate IN$_b^{\mathcal{K}}$ before the second pass over block b, if not all of the actual 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 IN$_b^{\mathcal{K}}$ and OUT$_b^{\mathcal{K}}$, namely, seeding the next epoch's initial 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 IN$^{\mathcal{K}}$ for the first subblock of epoch l in each thread, we can recompute IN$^{\mathcal{K}}$ and 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 IN$_b^{\mathcal{K}}$, OUT$_b^{\mathcal{K}}$, IN$_b^{\mathcal{G}}$ and OUT$_b^{\mathcal{G}}$, we can compute the LSOS.

THEOREM 7. *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* $\perp$ *at instruction* (l, t, i).

PROOF SKETCH. Suppose there were a VVO such that $x_{l,t,i} \leftarrow \perp$ 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 $x_{l,t,i} \leftarrow \perp$ and returned $\perp$, 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.

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.

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 [3, 11, 14]. We built Chrysalis Analysis on top of the *Log-Based Architectures* (LBA) framework [4]. 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 [23] 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 [25], shown in Table 2, where we synthetically tainted the benchmarks' input data. We tested two different configurations for both Butterfly and Chrysalis Analysis, shown in Table 3. To capture happens-before arcs, we leveraged the wrapper for shared library calls used by LBA [4] and generated vector clocks on the producer side, which were communicated to the lifeguard cores via the log.

Table 2: Splash-2 [25] 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 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)[7]

Table 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 [9], focusing on ADDRCHECK[8] as 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.

6.2 Results

The primary motivation for Chrysalis Analysis was improved precision relative to Butterfly Analysis. As shown in Table 4, precision in TAINTCHECK improved significantly compared to Butterfly Analysis for all benchmarks and both configurations (4- and

[7] 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.

[8] The extension of ADDRCHECK to Chrysalis Analysis appears in Appendix B.

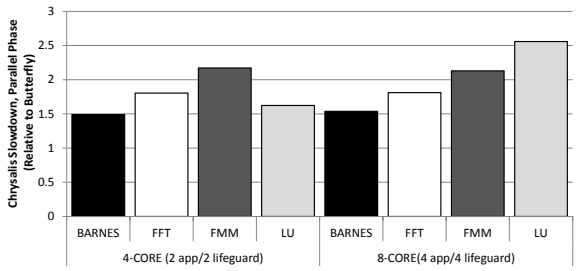


Figure 4: Chrysalis Analysis, normalized to Butterfly Performance.

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 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 errrors 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 primitives 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 are conservative.

7. CONCLUSION

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.

Future work will focus on isolating actual errors from possible errors (where not enough information is known), further reducing false positives, and optimizing the implementation to reduce its overheads.

8. ACKNOWLEDGMENTS

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

References

[1] R. Baldoni and M. Klusch. Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems. *IEEE Distributed Systems Online*, 3, February 2002.

[2] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional Detection of Data Races. In *PLDI*, 2010.

[3] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, 2004.

[4] 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.

[5] M. Christiaens and K. De Bosschere. Accordion Clocks: Logical Clocks for Data Race Detection. In *Euro-Par 2001 Parallel Processing*, 2001.

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

[7] C. Flanagan and S. N. Freund. Adversarial Memory for Detecting Destructive Races. In *PLDI*, 2010.

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

[9] 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.

[10] H. Kannan. Ordering Decoupled Metadata Accesses in Multiprocessors. In *MICRO*, 2009.

[11] 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.

[12] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. SigRace: Signature-Based Data Race Detection. In *ISCA*, 2009.

[13] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, U. Cambridge, 2004. http://valgrind.org.

[14] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *PLDI*, 2007.

[15] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 89 (2), 2003.

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

[17] 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.

[18] K. Serebryany and T. Iskhodzhanov. ThreadSanitzer - Data Race Detection in Practice. In *WBIA*, 2009.

[19] M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Inf. Process. Lett.*, 1992.

[20] SUN Microsystems, Inc. The SPARC Architecture Manual, 1991. version 8.

[21] R. E. Tarjan. Fast Algorithms for Solving Path Problems. *Journal of the ACM*, 28(3), 1981.

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

[23] Virtutech Simics. http://www.virtutech.com/.

[24] 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.

[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.

[26] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adapative Tracking. In *SOSP*, 2005.

APPENDIX

A. 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.

A.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 $b = (l,t,(j,k))$:

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

Epoch-Level. In defining the epoch-level sets necessary for Available Expressions, we reuse the sets MB_l, AFTER_b, and NBEFORE_b defined in Section 4.1. For an epoch l and maximal subblocks $b \in \text{MB}_{l^+}$, the epoch-level summaries are:

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

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 8 and 9 are quite similar to those of Lemmas 2 and 1, respectively, with the roles of $\mathcal{G}$ and $\mathcal{K}$ reversed. They are provided for completeness.

LEMMA 8. *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 9. *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 $e \in \mathcal{K}_l$. □

A.2 Strongly Ordered State

The equation for computing SOS_l is unchanged from Reaching Definitions (Section 4.2); 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 10. *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 8 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 8. 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 8. 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 9, 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 $e \in \mathcal{G}(O_l)$.

□

A.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.

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

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

In our correctness result, we reuse the notation $\text{HB}(b)$ as defined in Section 4.3.

LEMMA 11. *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 11 is essentially identical to Lemma 4, 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.

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

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

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

LEMMA 12. *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 12 is essentially identical to the proof of Lemma 5, 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 13. *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 12.

$e \in \text{SOS}_l - \text{IN}_b^{\mathcal{K}}$: In this case, $e \in \text{SOS}_l$ and $e \notin \text{IN}_b^{\mathcal{K}}$. Applying Theorem 10, we know that for every VVO O' of instructions in epochs $[0, (l-2)^+]$, $e \in \mathcal{G}(O')$. The contrapositive of Lemma 11 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.

B. 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 14. *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 SKETCH. The correctness proof follows the lines of the corresponding proof in the Butterfly Analysis paper [9]. 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. □