Butterfly Analysis: Adapting Dataflow Analysis to Dynamic Parallel Monitoring

Michelle L. Goodstein[1], Evangelos Vlachos[1], Shimin Chen[2],
Phillip B. Gibbons[2], Michael A. Kozuch[2], Todd C. Mowry[1]

[1]Carnegie Mellon University [2]Intel Labs Pittsburgh

mgoodste@cs.cmu.edu, evlachos@ece.cmu.edu,
{shimin.chen, phillip.b.gibbons,michael.a.kozuch}@intel.com, tcm@cs.cmu.edu

Abstract

Online program monitoring is an effective technique for detecting bugs and security attacks in running applications. Extending these tools to monitor parallel programs is challenging because the tools must account for inter-thread dependences and relaxed memory consistency models. Existing tools assume sequential consistency and often slow down the monitored program by orders of magnitude. In this paper, we present a novel approach that avoids these pitfalls by not relying on strong consistency models or detailed inter-thread dependence tracking. Instead, we only assume that events in the distant past on all threads have become visible; we make no assumptions on (and avoid the overheads of tracking) the relative ordering of more recent events on other threads. To overcome the potential state explosion of considering all the possible orderings among recent events, we adapt two techniques from static dataflow analysis, reaching definitions and reaching expressions, to this new domain of dynamic parallel monitoring. Significant modifications to these techniques are proposed to ensure the correctness and efficiency of our approach. We show how our adapted analysis can be used in two popular memory and security tools. We prove that our approach does not miss errors, and sacrifices precision only due to the lack of a relative ordering among recent events. Moreover, our simulation study on a collection of Splash-2 and Parsec 2.0 benchmarks running a memory-checking tool on a hardware-assisted logging platform demonstrates the potential benefits in trading off a very low false positive rate for (i) reduced overhead and (ii) the ability to run on relaxed consistency models.

Categories and Subject Descriptors F.3.2 [*Logics and 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

General Terms Algorithms, Design, Experimentation, Measurement, Performance, Reliability

Keywords Data Flow Analysis, Static Analysis, Parallel Programming, Dynamic Program Monitoring

1. Introduction

Despite the best efforts of programmers and programming systems researchers, software bugs continue to be problematic. To help address this problem, a number of tools have been developed over the years that perform *static* [7, 13, 15], *dynamic* [6, 14, 22, 27, 34], or *post-mortem* [24, 39] analysis to diagnose bugs. While these different classes of tools are generally complementary, our focus in this paper is on *dynamic* (online) tools, which we refer to as *"lifeguards"* (because they watch over a program as it executes to make sure that it is safe). To avoid the need for source code access, lifeguards are typically implemented using either a dynamic binary instrumentation framework (e.g., Valgrind [27], Pin [22], DynamoRio [6]) or with hardware-assisted logging [8]. Lifeguards maintain shadow state to track a particular aspect of correctness as a program executes, such as its memory [28], security [29], or concurrency [34] behaviors.

Existing lifeguards have focused on monitoring sequential programs. As difficult as it is to write a bug-free sequential program, however, it is even more challenging to avoid bugs in parallel software, given the many opportunities for non-intuitive interactions between threads. Hence we would expect bug-finding tools such as lifeguards to become increasingly valuable as more programmers wrestle with parallel programming. Unfortunately, the way that lifeguards have been written to date does not extend naturally to parallel software due to a key stumbling block: *inter-thread data dependences*.

1.1 Key Challenge: Inter-Thread Data Dependences

As we will discuss in greater detail later in Section 2, lifeguards typically operate on shadow state that they associate with every active memory location in the program (including the heap, registers, stack, etc.). As the monitored application executes, the lifeguard follows along, instruction-by-instruction, performing an analogous operation to update the corresponding shadow state. For example, when a lifeguard that is tracking the flow of data that has been "tainted" by external program inputs [29] encounters an instruction such as "`A = B + C`", the lifeguard will look up the boolean tainted status for locations `B` and `C`, `OR` these values together, and store the result in the shadow state for `A`.

When monitoring a single-threaded application, it is straightforward to think of the lifeguard as a finite state machine that is driven by the dynamic sequence of instructions from the monitored application. The order of events in this input stream is important. For single-threaded applications, it is simply the dynamic order in which the thread executes, since this will preserve all intra-thread data dependences in the lifeguard analysis. For parallel applications with a shared address space, however, the potential for data dependences across threads complicates the ordering.

ASPLOS'10 March 13–17, 2010, Pittsburgh, Pennsylvania, USA.

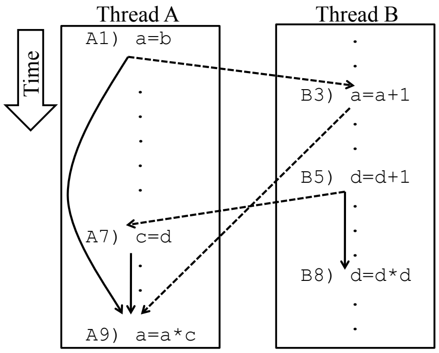


Figure 1. Example illustrating *intra-thread* (solid arrows) and *inter-thread* (dashed arrows) data dependences.

How do we deal with inter-thread data dependences? One approach that might sound appealing would be to capture a single serialized ordering that corresponded to the interleaving of events in the application, and feed the instructions to the lifeguard in that order. In Figure 1, for example, any topological sort of the data dependences graph would suffice. Unfortunately, this approach has two problems. First, it is impractical to capture such an ordering on most machines: a serialized interleaving across threads is only guaranteed to exist if the machine's memory consistency model [1] is sequentially consistency [12], which is not the case for the vast majority of commercial machines. Even if a machine is sequentially consistent, the serializable order is merely a hypothetical order: the actual memory system processes requests out-of-order, and it simply provides the illusion of serializability. Hence reconstructing a serialized ordering by observing the actual machine behavior not only assumes sequential consistency but also requires non-trivial modifications to the memory system hardware [24, 39]. (Proposals for relaxed consistency such as TSO only capture values read [39], not an actual ordering certain lifeguards require.) The second problem is that even if a serialized ordering could be captured, we would not want the lifeguard to process this merged stream of instructions sequentially for performance reasons; in order to keep up with the parallel application, the lifeguard also needs to run in parallel.

On the other hand, if there does not exist or we do not capture a serialized ordering and therefore have only partial information regarding inter-thread data dependences and their implications, this implies that multiple event orderings are possible, and the lifeguard will need to reason about this set of possibilities. For example, if the dashed dependence arcs in Figure 1 cannot be captured, then the lifeguard would need to consider the possibilities that "`a = a+1`" in THREAD B occurred before, after, or concurrent with "`a = b`" in THREAD A. While this approach more faithfully captures the behavior of non-sequentially-consistent machines, it unfortunately leads to a potential state space explosion, which may cause the lifeguard to run prohibitively slowly.

1.2 Our Approach: Tolerate Windows of Uncertainty Through a Modified Form of Dataflow Analysis

To tolerate the lack of total ordering information across threads that occurs in today's machines while avoiding the state space explosion problem, we have developed a new framework for performing lifeguard analysis that automatically reasons about bounded windows of uncertainty using an approach inspired by *interval analysis* [35]. Unlike traditional dataflow analysis, which performs static analysis on control flow graphs, our approach analyzes *dynamic* traces of instructions on different threads. Given the finite buffering of instructions and memory accesses in modern pipelines, we know that instructions that executed in the distant past on other threads must have committed by now, but the relative ordering between an instruction on a given thread and instructions from either the near past or near future on other threads is unknown. For example, in Figure 1, we do not know the relative ordering of events between these portions of the traces from the two threads (i.e. the dashed arcs are missing) because they occur close together in time. We model these windows of uncertainty using *uncertainty epochs* across the dynamic traces from concurrently-executing threads.

Tolerating uncertainty potentially introduces error into our dataflow analysis. However, by always behaving conservatively, we will be able to make two guarantees. One, that we will only have one-way error; and secondly, any mistake we do make will be a *false positive*, where we falsely classify a safe event in the analyzed program as an error. We will provably have zero *false negatives*; there will never be an occasion where we miss an error in the monitored application. Our dataflow analysis efficiently summarizes the net effects of these uncertainty epochs, requiring significant modifications over the standard analysis.

1.3 Related Work

Several researchers have proposed adaptations of dataflow analysis techniques (e.g., reaching definitions) to parallel architectures and programming languages [16, 19, 20, 33]. These adaptations often involve adapting a control flow graph to reflect explicit programmer annotated parallel functions and can be limited in the memory models they support [19]; some assume no shared variables [20], while others support only restricted classes of programs or memory models, such as deterministic or data-race-free programs, otherwise requiring a sequentially consistent memory model or a copy-in/copy-out semantics [16, 33]. Knoop *et al.* introduce a framework that generalizes sequential static unidirectional bit-vector analyses to work with explicitly annotated parallel regions [18]. Chugh *et al.* [9] propose a framework that first generates a static non-null analysis and later uses data race detection to kill facts that parallelism no longer guarantees to be true.

Most of these proposals assume the dataflow analysis is being conducted on a static compile-time representation of the application. Chugh *et al.* [9] demonstrate the effectiveness of conservative analysis; we will make similar determinations in our work.

There has also been work on probabilistic replay using only partial orderings of instruction execution [30] and using communication graphs to find bugs [21].

Others have used epochs to partition execution into manageable chunks. For example, in [25] the authors propose *strata*, a mechanism ordering data accesses in the context of deterministic replay. In [23], the authors use epochs, combined with signatures, to detect data races. In the realm of garbage collection, [4] use epochs as a way of dividing up time, without assuming that boundaries are simultaneous across threads. In [3], epochs are combined with a concept of local and global state that enables a thread to know when it is safe to update global state based on an epoch counter.

Finally, concurrent with this work, Vlachos *et al.* [37] explored an alternative approach to parallel monitoring that involves more substantial hardware support for the sake of reducing the number of false positives.

1.4 Contributions

This paper makes the following research contributions:

- We propose a model of bounded regions of uncertainty as a framework for performing dynamic program monitoring.
- We develop a generic framework for performing forward dataflow analysis problems, called "butterfly analysis", with

provably zero false negatives, as illustrated by reaching definitions and reaching expressions.

- We support any relaxed memory model that respects its own intra-thread dependences and provides cache coherence.
- We apply this framework to two popular memory and security lifeguards to show the applicability of our approach.
- We implement an initial prototype of butterfly analysis with our adapted memory lifeguard, and conduct performance and sensitivity analyses that demonstrate the potential benefits in trading off a very low false positive rate for (i) reduced overhead and (ii) the ability to run on relaxed consistency models.

2. Background

Program monitoring performs on-the-fly checking during the execution of applications, and is an important technique for improving software reliability and security. Program monitoring tools (a.k.a. lifeguards) can be categorized according to the granularity of application events that they care about, from system-call-level [17, 31] to instruction-level [27–29, 34]. Compared to the former, the latter can obtain highly detailed dynamic information, such as memory references, for more accurate and timely bug detection. However, such fine-grained monitoring presents great challenges for system support. This paper focuses on instruction-level lifeguards, although the results readily extend to coarser-grained settings as well.

Lifeguards. Although different instruction-level lifeguards perform different checking, they share three common characteristics [8, 27]: (i) maintaining (separate) fine-grained state information (called *metadata*) for every memory location in the application's address space; (ii) updating the metadata as a result of certain events; and (iii) checking invariants on the metadata in response to certain events. We describe two representative lifeguards in the following:

ADDRCHECK [26] is a memory-checking lifeguard. By monitoring memory allocation calls such as `malloc` and `free`, it maintains the allocation information for each byte in the application's address space. Then, ADDRCHECK verifies whether every memory reference visits an allocated region of memory by reading the corresponding allocation information.

TAINTCHECK [29] is a security-checking lifeguard for detecting overwrite-based security exploits (e.g., buffer overflows or printf format string vulnerabilities). It maintains metadata for every location in the application's address space, indicating whether the location is *tainted*. After a system call that receives data from network or from an untrusted disk file, the memory locations storing the untrusted data are all marked as tainted. TAINTCHECK monitors the inheritance of the tainted state: For every executed application instruction, it computes a logical `OR` of the tainted information of all the sources to obtain the tainted information of the destination of the instruction. TAINTCHECK raises an error if tainted data is used in jump target addresses (to change the control flow), format strings, or other critical ways.

General-Purpose Lifeguard Infrastructure. Existing general-purpose support for running lifeguards can be divided into two types depending on whether lifeguards share the same processing cores as the monitored application or lifeguards run on separate cores. In the first design, lifeguard code is inserted in between application instructions using dynamic binary instrumentation in software [6, 22, 27] or micro-code editing in hardware [11]. Lifeguard functionality is performed as the modified application code executes. In contrast, the second design offloads lifeguard functionality to separate cores. An execution trace of the application is captured at the core running the application through hardware, and shipped (via the last-level on-chip cache) on-the-fly to the core running the lifeguard for monitoring purposes [8].

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

Most previous works studied sequential application monitoring. (A notable exception is [10], which assumes transactional memory support.) However, in the multicore era, applications increasingly involve parallel execution; therefore, monitoring support for multithreaded applications is desirable. Unfortunately, adapting existing sequential designs to handle parallel applications is non-trivial, as discussed in Section 1. This paper proposes a solution that does not require extensive hardware dependence-tracking mechanisms or a strong consistency model.

To begin, we consider a model of monitoring support with multiple event sequences: one per application thread. Each sequence is processed by its own lifeguard thread, which may be the same thread as the one generating the sequence. As in the separate-core design, the lifeguard analysis may lag behind the application execution somewhat, relying on existing techniques [8] to ensure that no real damage occurs during this (short) window.[1] As discussed in Section 1, event sequences do not contain detailed inter-thread dependences information.

3. Challenges in Adapting Dataflow Analysis to Dynamic Parallel Monitoring

In the absence of detailed inter-thread dependence information, there are many possible interleavings of the event sequences lifeguards see when monitoring parallel programs.[2] Our approach is to adapt dataflow analysis, traditionally run statically at compile-time, as a dynamic run-time tool that enables us to reason about possible interleavings of different threads' executed instructions.

In this section, we will motivate our design decisions, showing how simpler constructions are either too inefficient, too imprecise, or both. Throughout this section and through Section 4.3, we will assume a sequentially consistent machine, for ease of exposition. This will be relaxed in Section 4.4.

Our first attempt at modeling a lack of fine-grain interthread dependence information was to assume no ordering information whatsoever between threads, even at a coarse granularity. The most natural abstraction was a control flow graph (CFG). A control flow graph expresses relationships between basic blocks within a program, but does not necessarily guarantee a particular ordering between blocks; it also is the data structure dataflow analysis requires.

A dynamic trace of events is similar to a program; instead of a language, we have assembly. Unlike programs, these sequences of events are linear and have no aliasing issues. However, we can use directed arcs from an instruction i to an instruction j to indicate that j is a potential immediate successor of i.

[1] A lifeguard thread raising an error may interrupt the application to take corrective action [32]. Some delay between application error and application interrupt is unavoidable, due to the lag in interrupting all the application threads.

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

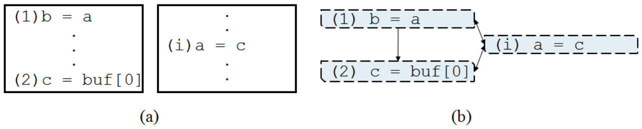


Figure 2. Two threads modify three shared memory locations, shown **(a)** as traces and **(b)** in a CFG. Throughout this paper, solid rectangles contain blocks of instructions, dashed hexagons contain single instructions, and "empty" blocks contain instructions that are not relevant to the current analysis.

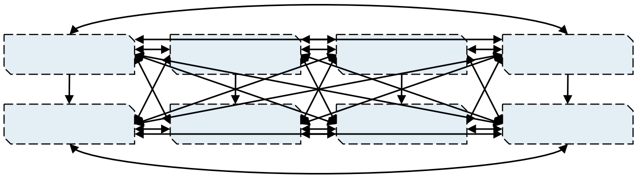


Figure 3. CFG of 4 threads with 2 instructions each.

Because there is arbitrary interleaving among instructions executed by different threads, we must make nodes out of individual instructions rather than basic blocks. We place directed arcs in both directions between any two instructions that could execute in parallel, and a directed arc between instructions i and $i+1$ in the same thread, indicating that the trace for a thread is followed sequentially. This yields a graph that at first glance resembled a control flow graph; it seemed at first that enough of the structure would be similar to apply dataflow analysis. However, this approach suffers from three major problems.

Problem 1: Too many edges. Figure 2(a) shows a very simple code example of two threads modifying three variables. Even with only three total instructions, we still require several arcs to reflect all the possible concurrency, shown in Figure 2(b). This may look manageable; unfortunately, adding arcs over an entire dynamic run leads to an explosion in arcs and the space necessary to keep this graph in memory. Figure 3 shows how quickly the number of arcs increases with only four threads, each executing two instructions. For T threads with N instructions each executing concurrently, there are $O(NT)$ edges due to the sequential nature of execution within a thread and $O((NT)^2)$ edges due to potential concurrency: each of the NT nodes has edges to all the nodes in all the other threads.

Problem 2: Arbitrarily delayed analysis. Unlike a control flow graph, whose size is bounded by the actual program, the dynamic runlength of a program is unbounded and potentially infinite in size if the program never halts. Since the halting problem is undecidable, analysis could not be completed until the program actually ended, because only then would the actual graph be known. This model of parallel computation quickly becomes intractable.

Problem 3: Conclusions based on impossible paths. The third problem with this approach is that it can lead to conclusions based on impossible paths. Recall the TAINTCHECK lifeguard described in Section 2. Suppose we were interested in running the TAINTCHECK lifeguard on the code in Figure 2(b), where `buf` has been tainted from a prior system call. Instruction 2 in Thread 1 taints `c`. Instructions (1) and (i) propagate taint from the source to their destination. According to the graph, it is valid for instruction (i) to be the immediate successor of instruction (2), implying there is a way for `a` to be tainted by inheriting taint from `c` at instruction (2). Likewise, it is valid for instruction (1) to be the immediate successor of instruction (i), implying `b` is tainted due to `a` being tainted. However, for all three memory locations to be tainted, we must have (2) execute before (i), and (i) before (1)–contradicting the sequential consistency assumption.

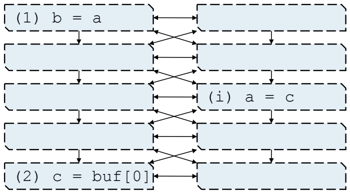


Figure 4. Two threads concurrently update `a,b` and `c`.

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

While this intuition proved useful, it did not solve all the aforementioned problems. Even after modifying our CFG-like approach to include edges only between individual instructions that are potentially concurrent, we could still conclude that an instruction at the end of the program taints the destination of the first instruction of the first thread, by zig-zagging up from the bottom of the graph to the top. This is possible even if each instruction only has edges to three other instructions in the other thread, as depicted in Figure 4. Because there are still paths from the end of a thread's execution to its beginning, we can potentially conclude that every address is tainted for almost the entire execution based on a single taint occurring at the very end.

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

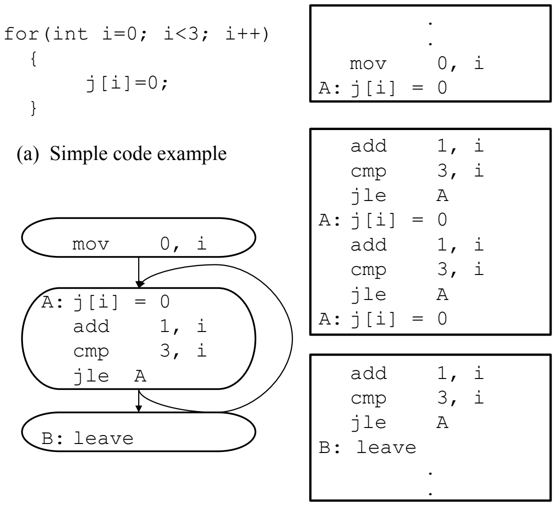
4. Butterfly Analysis: An Overview

In this section, we introduce a new model of parallel program execution, which formalizes what it means for one instruction to become globally visible *strictly before* another instruction, and shows how to group instructions into meaningful sliding windows to avoid the problems described in Section 3. Finally, we provide a formalization that is well-suited for adapting dataflow analysis to dynamic parallel monitoring, called **Butterfly Analysis**.

4.1 Mechanics

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

We do not require instantaneous heartbeat delivery, but do assume a maximum skew time for heartbeats to be delivered. By mak-



```
for(int i=0; i<3; i++)
  {
       j[i]=0;
  }
```

(a) Simple code example

```
   mov    0, i

A: j[i]  = 0
   add    1, i
   cmp    3, i
   jle  A

B: leave
```

(b) Control flow graph

```
          .
          .
   mov    0, i
A: j[i] = 0
```

```
   add    1, i
   cmp    3, i
   jle    A
A: j[i] = 0
   add    1, i
   cmp    3, i
   jle    A
A: j[i] = 0
```

```
   add    1, i
   cmp    3, i
   jle    A
B: leave
          .
          .
```

(c) Butterfly blocks

Figure 5. Unlike basic blocks (which are static), *butterfly blocks* contain *dynamic* instruction sequences, demarcated by heartbeats.

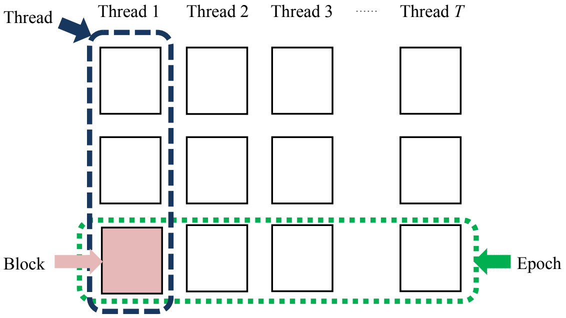
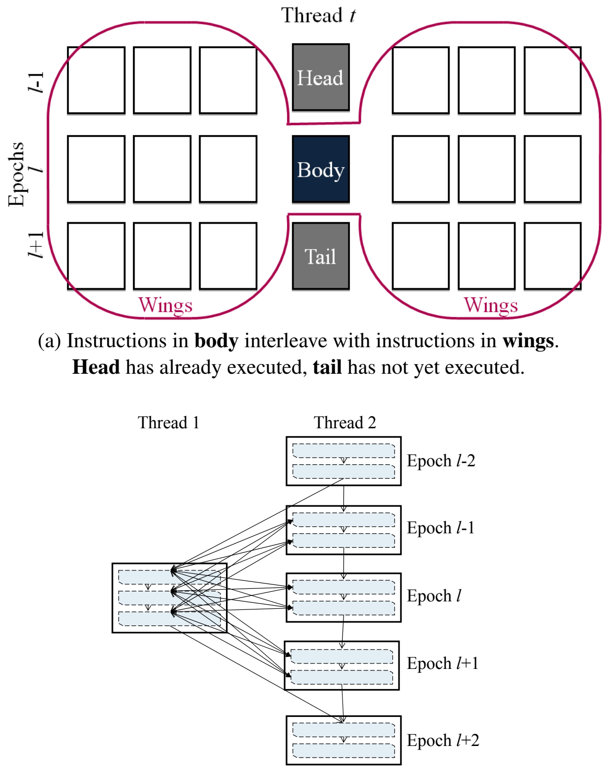


Figure 6. A particular block is specified by an epoch id l and thread id t. In reality, epoch boundaries will be staggered and blocks will be of differing sizes.

ing sure that the time between heartbeats accounts for (i) memory latency for instructions involving reads or writes, (ii) time for all instructions in the reorder and store buffers to become globally visible, and (iii) reception of the heartbeat including the maximum skew in heartbeat delivery time, we can guarantee *non-adjacent* epochs (i.e., epochs that do not share a heartbeat boundary) have strict happens-before relationships.[3] On the other hand, we will consider instructions in *adjacent* epochs, i.e., epochs that share a heartbeat boundary, to be *potentially concurrent* when they are not in the same thread.

An epoch contains a **block** in each thread, where a block is a series of consecutive instructions, and each block represents approximately the same number of cycles. Note that a block in our model is not equivalent to a standard basic block. As an example, the code in Figure 5(a) transforms into a few basic blocks, illustrated as a CFG in Figure 5(b), whereas Figure 5(c) shows blocks in our model. The epoch boundaries across threads are not precisely synchronized, and correspond to reception of heartbeats. Our model, illustrated in Figure 6, incorporates possible delays in receiving the heartbeat into its design. Formally, given an epoch ID l and a thread ID t, a block is uniquely defined by the tuple (l, t). A particular instruction can be specified by (l, t, i), where i is an offset from the start of block (l, t).



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

(b) Arcs between instructions indicate possible interleavings.

Figure 7. Potential concurrency modeled in butterfly analysis, shown at the **(a)** *block* and **(b)** *instruction* levels.

Our model has three main assumptions. Our first assumption will be that instructions within a thread are sequentially ordered, continuing our sequential consistency assumption from Section 2; we will later relax this assumption.

Our second assumption is that all instructions in epoch l execute (their effects are globally visible) before any instructions in epoch $l + 2$, implying that any instructions in epoch l executes strictly after all instructions in epoch $l - 2$. Finally, instructions in block (l, t) can interleave arbitrarily with instructions in blocks of the form $(l - 1, t'), (l, t')$, and $(l + 1, t')$ where $t' \neq t$. The final two assumptions of this model handle the various possible delays (in receiving a heartbeat, in memory accesses, due to the reorder buffers, etc.). If an instantaneous heartbeat would have placed an instantaneous instruction j in epoch l, our model will require that instruction j instead will always be in either epoch $l - 1$, l or $l + 1$.

Butterfly analysis formalizes the intuition that it may be difficult to observe orderings of nearby operations but easier to observe orderings of far apart instructions. We now motivate the term **butterfly**, which takes as parameter a block (l, t); see Figure 7(a). We

[3] This guarantee is by construction. Time between epochs is always large enough to account for the reorder buffer, store buffer, memory latency, and skew in heartbeat delivery. Instructions more than one epoch apart were already implicitly ordered, since the earlier instruction has committed, with any related store draining from the store buffer, before the later instruction is even issued. We do assume cache coherency for ordering writes to the same address.

call block (l, t) the **body** of the butterfly, $(l-1, t)$ the **head** of the butterfly and $(l+1, t)$ the **tail** of the butterfly. The head always executes before the body, which executes before the tail. For all threads $t' \neq t$, blocks $(l-1, t'), (l, t')$ and $(l+1, t')$ are in the **wings** of block (l, t)'s butterfly.

4.2 Butterfly Framework

As described, our framework resembles a graph of parallel execution, where directed edges indicate that instruction i can be the direct predecessor of instruction j. Figure 7(b) illustrates this from the perspective of a block in Thread 1, epoch l.

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

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

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

4.3 Lifeguards As Two Pass Algorithms

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

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

The lifeguard writer specifies the events the dataflow analysis will track, the meet operation, the metadata format, and the checking algorithm. Examples will be given in Section 6.

4.4 Relaxed Memory Models

Our butterfly framework is well-suited to relaxed memory models. There are relaxed assumptions on when a memory access becomes globally visible (two epochs later). There are relaxed assumptions on memory access interleavings within a sliding window. In fact, the analysis accounts for different threads possibly observing different orderings of the same accesses, e.g., two writes A and B such that thread 1 may observe A before B while thread 2 may observe B before A. We make only the weak assumptions that (i) for a given thread's view, its own intra-thread dependences are respected, and (ii) cache coherency orders writes to the same address. As discussed in Sections 5 and 6, our analysis for what happens at other threads is based on set operations; set union and intersection are both commutative operations, and set difference only becomes a problem if we change metadata before an instruction was able to read it, which will not happen given our above weak assumptions.

While this suffices for our reaching definitions and reaching expressions analyses in Section 5, and hence any lifeguards based on them, for lifeguards such as TAINTCHECK, there may be more false positives with relaxed models than when assuming sequential consistency (as discussed in Section 6.2).

5. Butterfly Analysis: Canonical Examples

This section presents our adaptation of dataflow analysis to dynamic program monitoring, called butterfly analysis. Specifically, we adapt reaching definitions and reaching expressions [2], two simple forward dataflow analysis problems that exhibit a generate/propagate structure common to many other dataflow analysis problems. Previous studies [8, 40, 41] have shown that this structure is a common structure for lifeguards, including lifeguards that check for security exploits, memory bugs, and data races.

We show how reaching definitions and reaching expressions can be formulated as generic lifeguards using butterfly analysis. In standard dataflow analysis, there are equations for calculating IN, OUT, GEN and KILL; our approach extends beyond these four, as discussed below. In our setting, the lifeguard's stored metadata tracks definitions or expressions, respectively, that are known to have reached epoch l. While the generic lifeguards do not define specific checks, their IN and OUT calculations provide the information useful for a variety of checks. (Section 6 shows how our generic lifeguards can be instantiated as ADDRCHECK and TAINTCHECK lifeguards.)

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

- We perform our analysis over a sliding window of 3 epochs rather than the entire execution trace. This not only enables our analysis to proceed as the application executes, it also bounds the complexity of our analysis.
- We require only two passes over each epoch. The time per pass is proportional to the complexity of the checking algorithm provided by the lifeguard writer.
- We introduce state (SOS) that summarizes the effects of instructions in the distant past (i.e., all instructions prior to the current sliding window). This enables using a sliding window model without missing any errors.
- The symmetric treatment of the instructions/blocks in the wings means we can efficiently capture the effects of all the instructions in the wings. To do so, we add four new primitives: GEN-SIDE-IN, GEN-SIDE-OUT, KILL-SIDE-IN and KILL-SIDE-OUT, as defined below.

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

Processing a Level. To motivate our work, we examine Figure 8, a reaching expressions example. First, we note that for any sliding window of size 3, the strongly ordered states SOS_{l-1}, SOS_l and SOS_{l+1}, which summarize execution through epochs $l - 3$, $l - 2$, and $l - 1$, respectively, are available after the lifeguard has consumed events through $l - 1$. This can be shown by induction, as follows. The very first butterfly uses only epochs 0 and 1, and has $\mathrm{SOS}_0 = \mathrm{SOS}_1 = \varnothing$. After concluding all butterflies with bodies in epoch 0, we have SOS_2 as well. From then on, we inductively have the correct SOS for each epoch in the butterfly.

Now consider Figure 8(a). The LSOS is available for block $(l, 2)$ because the head (not shown) is available and so is SOS_l. In our first pass we discover that block $(l, 2)$ kills expression `a-b` through a redefinition of `b`. The epoch's summary need only be generated once (in reaching expressions, the summary contains the killed expressions), so we do not need to regenerate the summary as the block changes position in the sliding window. After the first butterfly, we are performing a first pass only on the blocks in the newest epoch under consideration; the summaries for older blocks have already been completed.

Using that information, we now examine Figure 8(b). This shows the entire butterfly for block $(l, 2)$. As part of step 2, summaries from all blocks in the wings (computed earlier, as argued above) are first collected and combined (represented by the circle labeled "meet"), producing one summary for the entire wings (in reaching expressions, this is KILL-SIDE-IN$_{l,2}$). (Note: The meet function for KILL-SIDE-OUT is not the standard reaching expressions intersection, but rather computed as the union. This is explained in Section 5.2.) Finally, $(l, 2)$ repeats its analysis and performs checks (step 3), noting that it is not the only block to kill `a-b`.

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

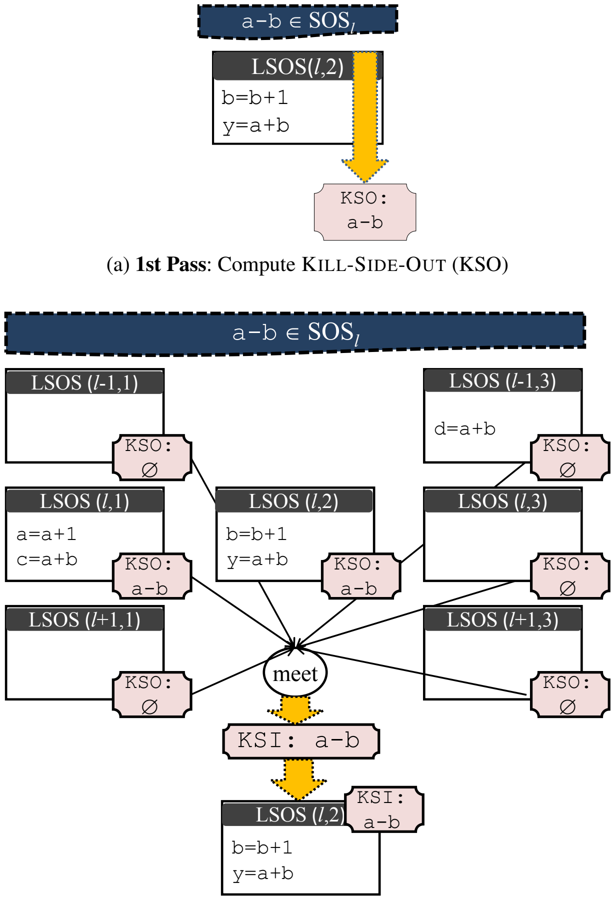
Note that there is a single writer for each of the data structures (one of the threads can be nominated to act as master for global objects such as the SOS), and objects are not modified after being released for reading. Hence, synchronizing accesses to the lifeguard metadata is unnecessary.

In Sections 5.1 and 5.2, we will formalize the well-known problems of reaching definitions and reaching expressions [2] using butterfly analysis.

Valid Ordering. We introduce the concept of a **valid ordering** O_k, which is a total sequential ordering of all the instructions in the first k epochs, where the ordering respects the assumptions of butterfly analysis. A **path** to an instruction (block) is the prefix of a valid ordering that ends just before the instruction (the first instruction in the block, respectively).

We observe that the set of valid orderings is a superset of the possible application orderings: Nearly all machines support at least cache coherency, which creates a globally consistent total order among conflicting accesses to the same location. Because our analysis considers each definition event independently, our approach has no false negatives (as argued in Section 4.4), even for relaxed memory models.

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



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

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

Figure 8. Computing KILL-SIDE-OUT and KILL-SIDE-IN in reaching expressions, from the perspective of the body, block $(l, 2)$, of the butterfly. Boxes with beveled edges are summaries.

5.1 Dynamic Parallel Reaching Definitions

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

5.1.1 Generating and Killing Across An Epoch

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

$$\textsc{Gen}_l = \bigcup_t \textsc{Gen}_{l,t}$$

$$\textsc{Kill}_l = \bigcup_t (\textsc{Kill}_{l,t} \cap (\bigcup_{t' \neq t} \textsc{Kill}_{(l-1,l),t'} \cup \textsc{Not-Gen}_{(l-1,l),t'}))$$

$$\text{where } \textsc{Kill}_{(l-1,l),t} = (\textsc{Kill}_{l-1,t} - \textsc{Gen}_{l,t}) \cup \textsc{Kill}_{l,t}$$
$$\text{and } \textsc{Not-Gen}_{(l-1,l),t} = \{d_k | d_k \notin \textsc{Gen}_{l-1,t} \wedge d_k \notin \textsc{Gen}_{l,t}\}$$

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

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

Lemma 5.1. *If $d_k \in \textsc{Gen}_l$ then there exists a valid ordering O_l such $d_k \in \textsc{Gen}(O_l)$. If $d_k \in \textsc{Kill}_l$ then under all valid orderings O_l, $d_k \notin \textsc{Gen}(O_l)$.*

5.1.2 Updating State

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

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

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

$$\text{SOS}_l := \textsc{Gen}_{l-2} \cup (\text{SOS}_{l-1} - \textsc{Kill}_{l-2}) \quad \forall l \geq 2$$
$$\text{SOS}_0 = \text{SOS}_1 = \varnothing$$

Lemma 5.2. $\text{SOS}_l := \textsc{Gen}_{l-2} \cup (\text{SOS}_{l-1} - \textsc{Kill}_{l-2})$ *achieves the invariant.*

The proof is in the appendix.

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

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

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

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

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

$$\text{LSOS}_{l,t,k} = \begin{cases} \text{LSOS}_{l,t} \text{ if } k = 0 \\ \textsc{Gen}_{l,t,k-1} \cup (\text{LSOS}_{l,t,k-1} - \textsc{Kill}_{l,t,k-1}) \text{ ow} \end{cases}$$

This is the standard $\textsc{Out} = \textsc{Gen} \cup (\textsc{In} - \textsc{Kill})$ formula, with $\text{LSOS}_{l,t,k-1}$ acting as $\textsc{In}$ and $\text{LSOS}_{l,t,k}$ as $\textsc{Out}$.

5.1.3 Calculating In and Out

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

$$\textsc{In}_{l,t} = \textsc{Gen-Side-In}_{l,t} \bigcup \text{LSOS}_{l,t}$$
$$\textsc{In}_{l,t,i} = \textsc{Gen-Side-In}_{l,t} \cup \text{LSOS}_{l,t,i}$$

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

$$\textsc{Out}_{l,t,i} = \textsc{Gen}_{l,t,i} \bigcup (\textsc{In}_{l,t,i} - \textsc{Kill}_{l,t,i})$$
$$\textsc{Out}_{l,t} = \textsc{Gen}_{l,t} \bigcup (\textsc{In}_{l,t} - \textsc{Kill}_{l,t})$$

Using reaching definitions as a lifeguard, we have now shown how to compute the checks, the $\textsc{Out}$ computation.

5.1.4 Applying the Two-Pass Algorithm

We can now set our parameters for the two-pass algorithm proposed in Section 4.3. For step 1, our local computations are $\textsc{Gen}_{l,t}$, $\textsc{Kill}_{l,t}$ and $\text{LSOS}_{l,t}$. These are used for our checking algorithm. The summary information is $\textsc{Gen-Side-Out}_{l,t}$. For step 2, the meet function is $\cup$, calculated over the $\textsc{Gen-Side-Out}$ from the wings, to get $\textsc{Gen-Side-In}_{l,t}$. We then use $\textsc{Gen-Side-In}_{l,t}$ to perform our second pass of checks (step 3). Finally, we use $\textsc{Gen}_l$ and $\textsc{Kill}_l$ to update the SOS (step 4).

5.2 Dynamic Parallel Reaching Expressions

An expression e reaches a block (l, t) only if there is no path to the block that kills e, i.e., no valid ordering in which e is killed before the first instruction of the block. In such cases, there is no need to recompute the expression. However, if any path to the block kills e, then there is no guarantee that e is precomputed and we must recompute it in block (l, t). With reaching definitions, d_k reaches a particular point p if in at least one valid ordering d_k reaches p; in reaching expressions, e_k only reaches p if in all valid orderings e_k reaches p. This gives some intuition that $\textsc{Kill}$ in reaching expressions behaves like $\textsc{Gen}$ in reaching definitions, and likewise $\textsc{Gen}$ in reaching expressions behaves like $\textsc{Kill}$ in reaching definitions.

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

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

$$\textsc{Kill-Side-Out}_{l,t} = \bigcup_i \textsc{Kill}_{l,t,i}$$
$$\textsc{Kill-Side-In}_{l,t} = \bigcup_{l-1 \leq l' \leq l+1} \bigcup_{t' \neq t} \textsc{Kill-Side-Out}_{l',t'}$$

In reaching expressions, $\textsc{Gen-Side-In} = \textsc{Gen-Side-Out} = \varnothing$ for the same reason that $\textsc{Kill-Side-In} = \textsc{Kill-Side-Out} = \varnothing$ in reaching definitions; no block has enough information to know that every path to a particular instruction has generated a particular expression.

The properties we desire for $\textsc{Gen}_l$ and $\textsc{Kill}_l$ are roughly the opposite of those from reaching definitions.

$$\textsc{Kill}_l = \bigcup_t \textsc{Kill}_{l,t}$$

$$\textsc{Gen}_l = \bigcup_t (\textsc{Gen}_{l,t} \cap (\bigcup_{t' \neq t} \textsc{Gen}_{(l-1,l),t'} \cup \textsc{Not-Kill}_{(l-1,l),t'}))$$

$$\text{where } \textsc{Gen}_{(l-1,l),t} = (\textsc{Gen}_{l-1,t} - \textsc{Kill}_{l,t}) \cup \textsc{Gen}_{l,t}$$
$$\text{and } \textsc{Not-Kill}_{(l-1,l),t} = \{e_k | e_k \notin \textsc{Kill}_{l-1,t} \wedge e_k \notin \textsc{Kill}_{l,t}\}$$

The correctness of $\textsc{Kill}_l$ and $\textsc{Gen}_l$ follows along the lines of the proof of Lemma 5.1, with the roles of $\textsc{Gen}$ and $\textsc{Kill}$ reversed.

5.2.1 Updating State

The SOS has the same equation and update rule as described in Section 5.1.2. The LSOS has a slightly different form, reflecting the different roles $\textsc{Gen}$ and $\textsc{Kill}$ play. In reaching expressions, an expression only reaches an instruction (l, t, i) if it has been defined along all paths to the instruction. So, if $e_k \in \text{SOS}_l \wedge e_k \notin \textsc{Kill}_{l-1,t}$, then $e_k \in \text{LSOS}_{l,t}$ because e_k is calculated along all paths. However, if $e_k \in \textsc{Kill}_{l-1,t}$ then at least one path exists where the expression is not defined. If $e_k \notin \text{SOS}_l$, the only way that $e_k \in \text{LSOS}_{l,t}$ is if it is defined by the head ($e_k \in \textsc{Gen}_{l-1,t}$) and no other thread t' ever kills e_k in epoch $l - 2$; otherwise, because the head can interleave with epoch $l - 2$, there is a possible path where e_k is killed before the body executes. This leads to:

$$\text{LSOS}_{l,t} = \Big(\textsc{Gen}_{l-1,t} - \textstyle\bigcup_{t' \neq t}\{e_k | e_k \in \textsc{Kill}_{l-2,t'}\}\Big) \cup \\ (\text{SOS}_l - \textsc{Kill}_{l-1,t})$$

$\text{LSOS}_{l,t,k}$ has the same update rule as stated in Section 5.1.2.

5.2.2 Calculating In and Out

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

$$\textsc{In}_{l,t} = \text{LSOS}_{l,t} - \textsc{Kill-Side-In}_{l,t}$$
$$\textsc{In}_{l,t,i} = \text{LSOS}_{l,t,i} - \textsc{Kill-Side-In}_{l,t}$$

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

$$\textsc{Out}_{l,t,i} = \textsc{Gen}_{l,t,i} \bigcup (\textsc{In}_{l,t,i} - \textsc{Kill}_{l,t,i})$$
$$\textsc{Out}_{l,t} = \textsc{Gen}_{l,t} \bigcup (\textsc{In}_{l,t} - \textsc{Kill}_{l,t})$$

5.2.3 Applying the Two-Pass Algorithm

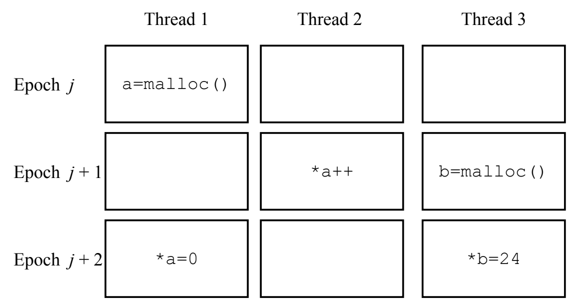
The parameters for the two-pass algorithm are similar to those in Section 5.1.4. In step 1, we again calculate $\textsc{Gen}_{l,t}$, $\textsc{Kill}_{l,t}$ and $\text{LSOS}_{l,t}$, but now the summary information is $\textsc{Kill-Side-Out}_{l,t}$. Step 2 uses $\cup$ for the meet function but calculates over all the $\textsc{Kill-Side-Out}_{l',t'}$ in the wings, to get $\textsc{Kill-Side-In}_{l,t}$, which is then used to perform our second pass of checks (step 3). Finally, we use $\textsc{Gen}_l$ and $\textsc{Kill}_l$ to update SOS_l (step 4).

6. Implementing Lifeguards With Butterfly Analysis

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

6.1 AddrCheck

ADDRCHECK [26], as described in Section 2, checks accesses, allocations and deallocations as a program runs to make sure they are safe. In the sequential version, this is straightforward; writing to unallocated memory is an error. In butterfly analysis, one thread can allocate memory before another writes, but if these operations are in adjacent epochs, the operations are potentially concurrent. In ADDRCHECK, a false positive occurs when the program behaves safely but the lifeguard believes it has seen an invalid sequence of malloc, free, and memory access events.



	Thread 1	Thread 2	Thread 3
Epoch j	a=malloc()		
Epoch $j+1$		*a++	b=malloc()
Epoch $j+2$	*a=0		*b=24

Figure 9. ADDRCHECK examples of interleavings between allocations and accesses. There is a potentially concurrent access to a by Thread 2 during its allocation by Thread 1, but the allocation of b by Thread 3 is *isolated* from other threads.

We describe ADDRCHECK as an adaptation of reaching expressions, associating allocations with GEN and deallocations with KILL. We chose reaching expressions because we want to guarantee zero false negatives; for all valid orderings, we want to know whether an access to memory location m is an access to allocated memory, and always detect accesses to unallocated memory regions. Let $\textsc{Gen}_{l,t,i} = \{m\}$ if and only if instruction (l, t, i) allocates memory location m and otherwise $\varnothing$. Likewise, $\textsc{Kill}_{l,t,i} = \{m\}$ if and only if instruction (l, t, i) deallocates memory location m and otherwise $\varnothing$. $\textsc{Gen}_{l,t}$, $\textsc{Kill}_{l,t}$, $\textsc{Gen}_l$, $\textsc{Kill}_l$, SOS_l and $\text{LSOS}_{l,t}$ use the equations and update rules from Section 5.2.

Checking Algorithm

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

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

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

Consider Figure 9. When thread 2 accesses a in epoch $j + 1$, a does not appear allocated yet, because its allocation will not be reflected in the SOS for another epoch. However, when thread 3 allocates b in epoch $j + 1$, it appears deallocated within the thread and no other thread is accessing it. The subsequent access to b in epoch $j + 2$ is also safe because it is within the same thread, even though the allocation is not yet reflected in the SOS.

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

Next, we want to ensure that allocations and deallocations were *isolated* from any other concurrent thread. This occurs during the second pass, using the summaries created in the first pass. For ADDRCHECK, the summary is $s_{l,t} = (\mathrm{GEN}_{l,t}, \mathrm{KILL}_{l,t}, \mathrm{ACCESS}_{l,t})$, where $\mathrm{ACCESS}_{l,t}$ contains all addresses that block (l, t) accessed. Combining the wing summaries yields:

$$S_{l,t} = (\bigcup_{\text{wings}} \mathrm{GEN}_{l',t'}, \bigcup_{\text{wings}} \mathrm{KILL}_{l',t'}, \bigcup_{\text{wings}} \mathrm{ACCESS}_{l',t'}).$$

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

$$\begin{gathered}((s.\mathrm{GEN}_{l,t} \cup s.\mathrm{KILL}_{l,t}) \bigcap (S.\mathrm{GEN}_{l,t} \cup S.\mathrm{KILL}_{l,t})) \bigcup \\ (s.\mathrm{ACCESS}_{l,t} \bigcap (S.\mathrm{GEN}_{l,t} \cup S.\mathrm{KILL}_{l,t})) \bigcup \\ (S.\mathrm{ACCESS}_{l,t} \bigcap (s.\mathrm{GEN}_{l,t} \cup s.\mathrm{KILL}_{l,t}))\end{gathered}$$

and otherwise flag an error.

Theorem 6.1. *Any error detected by the original* ADDRCHECK *on a valid execution ordering for a given machine (obeying intra-thread dependences and supporting cache coherence) will also be flagged by our butterfly analysis.*

A proof sketch appears in the appendix. This shows that the butterfly implementation of ADDRCHECK has zero false negatives.

6.2 TaintCheck

TAINTCHECK [29] tracks the propagation of taint through a program's execution; if at least one of the two sources is tainted, then the destination is considered tainted. When extending TAINTCHECK to work using butterfly analysis, we extend this conservative assumption as follows. If some valid ordering O causes an address x to appear tainted at instruction (l, t, i), we conclude that (l, t, i) taints x even if it does not taint x under any other valid ordering. We modify reaching definitions to accommodate TAINTCHECK. In this setting, a false negative refers to concluding that data is untainted when it is actually tainted, whereas a false positive refers to believing data to be tainted when it is actually untainted.

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

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

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

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

Let $S = \{\top, \perp, \{a\}, \{a, b\} | \exists \text{memory locations } a, b\}$. In other words, S represents the set of all possible right-hand values in our mapping. We define the set $\mathrm{KILL}_{l,t,i} = \{(x_{l,t,j} \leftarrow s) | s \in S, j < i, \text{ and } \texttt{loc}(l,t,j) = \texttt{loc}(l,t,i)\}$. In TAINTCHECK, $\mathrm{GEN\text{-}SIDE\text{-}OUT}_{l,t}$, $\mathrm{KILL\text{-}SIDE\text{-}OUT}_{l,t}$, $\mathrm{GEN\text{-}SIDE\text{-}IN}_{l,t}$, $\mathrm{KILL\text{-}SIDE\text{-}IN}_{l,t}$, $\mathrm{GEN}_{l,t}$ and $\mathrm{KILL}_{l,t}$ all function identically as defined in Section 5.1 for reaching definitions.

Checking Algorithm

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

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

Algorithm 1 TAINTCHECK Check Algorithm

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

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

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

Theorem 6.2. *If Check returns $(x_{l,t,i} \leftarrow \top)$, then there is no valid ordering of the first $l+1$ epochs such that x is $\bot$ at instruction (l,t,i). Therefore, any error detected by the original* TaintCheck *on a valid execution ordering for a given machine (obeying intra-thread dependences and supporting cache coherence) will also be flagged by our butterfly analysis.*

A proof sketch appears in the appendix. In other words, the butterfly version of TaintCheck experiences zero false negatives.

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

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

Lemma 6.3. *If there exists a valid ordering O among 3 consecutive epochs such that x is tainted then*
(1) x is tainted via an interleaving of the first 2 epochs;
(2) x is tainted via an interleaving of the last 2 epochs; or
(3) there exist a predecessor y of x such that y is tainted in the first two epochs and there exists a path from x to y in the last two epochs using only transfer functions from the last two epochs.

SOS and LSOS

Instead of transfer functions the SOS and LSOS will track locations believed to be tainted. Once again, TaintCheck is slightly more complicated than reaching definitions. We can conclude that a variable is tainted in epoch l based on an interleaving with epoch $l+1$. Consider Figure 10. If we do not commit a to the SOS before beginning a butterfly for block $(j+2, 2)$ we may conclude that d is untainted, even though there is a path where d is tainted. If we consider a to be tainted before beginning epoch $j+2$, though, there is no guarantee the instruction that taints a has actually already executed. However, considering an address to be tainted early is merely imprecise, while considering an address to be tainted too late violates our guarantees.

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

$$
\begin{gathered}
\textsc{Gen}_l = \textstyle\bigcup_t \{x \mid \textsc{LastCheck}(x,l,t) = \bot\} \\
\textsc{Kill}_l = \textstyle\bigcup_t \{x \mid \textsc{LastCheck}(x,l,t) = \top \wedge \\
(\forall t' \neq t, \textsc{LastCheck}(x,(l-1,l),t) = \top \vee \\
\textsc{LastCheck}(x,(l-1,l),t) = \varnothing)\}
\end{gathered}
$$

	Thread 1	Thread 2	Thread 3
Epoch j	a=b		
Epoch $j+1$		b=buf[0]	
Epoch $j+2$		d=c	c=a

Figure 10. Updating the SOS is nontrivial for TaintCheck. By the end of epoch $j+1$, a has been tainted, but the SOS may need to be updated before blocks in epoch $j+2$ begin butterfly analysis.

Table 1. Simulator and Benchmark Parameters

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

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

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

Condition 1. *If there exists a valid ordering O_s of the first $l-2$ epochs such that x is tainted in O_s then $x \in$ SOS$_{l-2}$.*

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

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

7. Evaluation of A Butterfly Analysis Prototype

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

7.1 Experimental Setup

While the generality of butterfly analysis makes it applicable to a wide variety of dynamic analysis frameworks (including software-only frameworks based upon binary instrumentation [6, 22, 27]), we chose to build upon the *Log-Based Architectures* (LBA) [8]

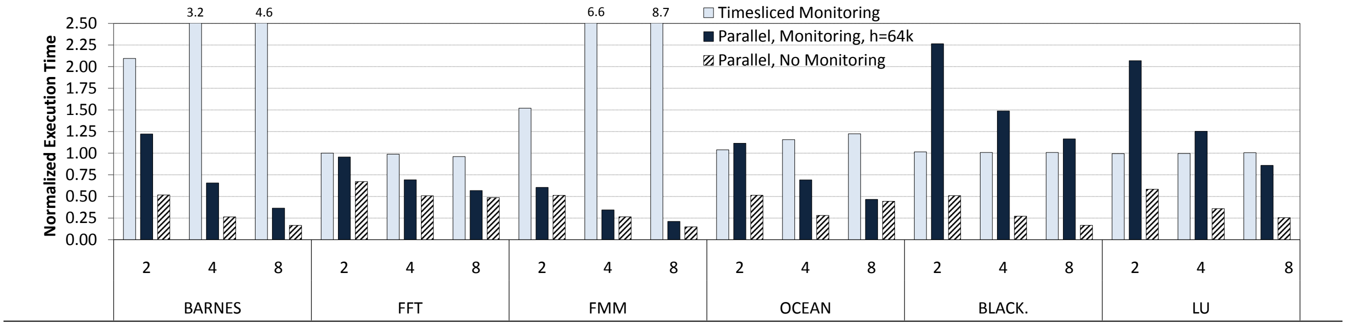


Figure 11. Relative performance, normalized to sequential, unmonitored execution time. X-axis: number of application threads.

framework in our experiments due to its relatively low run-time overheads. With LBA, each application thread is monitored by a dedicated lifeguard thread running concurrently on a separate processor on the same CMP. The LBA hardware captures a dynamic instruction log per application thread and passes it (via the L2 cache) to the corresponding lifeguard thread. When lifeguard processing is slower than the monitored application (as is the case in our experiments), the monitored application stalls whenever the log buffer is full; hence our performance results show lifeguard processing time, which is equivalent to application execution time including such log buffer stalls.

Because the LBA [8] hardware support is not available on existing machines, we simulated the LBA hardware functionality (including log capture and event dispatch) on a shared-memory CMP system using the Simics [36] full-system simulator. Although the LBA hardware is simulated, the full software stack for butterfly analysis is executed faithfully in our experiments. Table 1 shows the parameters for our machine model as well as the benchmarks that we monitored (taken from Splash-2 [38] and Parsec 2.0 [5]).

For our lifeguard, we implemented a parallel, heap-only version of ADDRCHECK–based upon [26]—using butterfly analysis as described in Section 6.1. The LBA logging mechanism makes it easy to generate and communicate heartbeats: we simply insert heartbeat markers into the log after h instructions have occurred per thread,[4] where h equals 8K or 64K instructions in our experiments. We use the *metadata-TLB* and *idempotent filtering*[5] accelerators from LBA [8], and we filter out stack accesses.

7.2 Experimental Results

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

Performance Analysis. Because lifeguards involve additional processing and no direct performance benefit for the monitored application, the performance question is how much they slow down performance relative to unmonitored execution. Figure 11 shows the performance impact of butterfly analysis, where the y-axis is execution time normalized to the given application running sequentially on a single thread without monitoring (hence shorter bars are faster). We show performance with 2, 4, and 8 application threads,[6] as labeled on the x-axis. Within each set of bars, we show three cases: (i) *"Timesliced Monitoring"*, the current state of the art where all application threads are interleaved on one core, and are monitored by a sequential lifeguard (running on a separate core); (ii) *"Parallel, Monitoring,"* which is with our butterfly analysis; and (iii) *"Parallel, No Monitoring,"* which is the application running in parallel without any monitoring (as a point of comparison).

As we observe in Figure 11, when monitoring only *two* application threads, the performance of butterfly analysis relative to the state-of-the-art timesliced approach is mixed: it is significantly better for BARNES and FMM, comparable for FFT and OCEAN, and significantly worse for BLACKSCHOLES and LU. A key advantage of butterfly analysis relative to timesliced analysis, however, is that the analysis itself can enjoy parallel speedup with additional threads. Hence as the scale increases to *eight* application (and lifeguard) threads, butterfly analysis outperforms timesliced analysis in five of six cases, and in four of those cases by a wide margin. In the one case where timesliced outperforms butterfly analysis with eight threads (i.e., BLACKSCHOLES), one can observe in Figure 11 that butterfly analysis is speeding up well with additional threads, but it has not quite reached the crossover point with eight threads.

While our butterfly approach offers the advantage of exploiting parallel threads to accelerate lifeguard analysis, the implementation of our current prototype has the disadvantage that it performs more work per monitored instruction than the timesliced approach. For example, in the first pass, our current implementation executes roughly 7-10 instructions for each monitored load and store instruction simply to record it for the second pass, above and beyond performing the same first-pass checks as traditional ADDRCHECK. We believe that this overhead is not fundamental to butterfly analysis, and that it could be significantly reduced by caching parts of our first-pass analysis and reusing it when the same monitored code is revisited. (We plan to explore this possibility in future work.) Despite the inefficiencies of our initial prototype, we observe in Figure 11 that it offers compelling performance advantages relative to the state-of-the-art due to its ability to exploit parallelism.

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

Figure 12 shows the impact of epoch size on performance. We show two epoch sizes (h): 8K and 64K instructions. (Note that the epoch size in Figure 11 was h=64K.) As we see in Figure 12, in nearly all cases (i.e., everything except the two and four thread

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

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

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

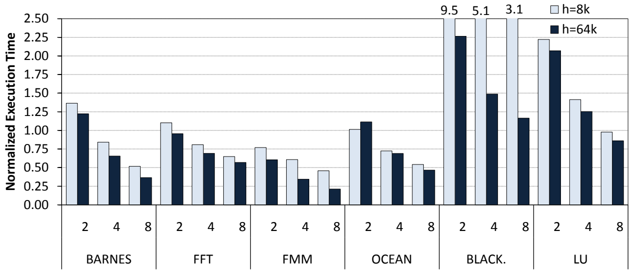


Figure 12. Performance sensitivity analysis with respect to epoch size. Results shown for h=8K and 64K.

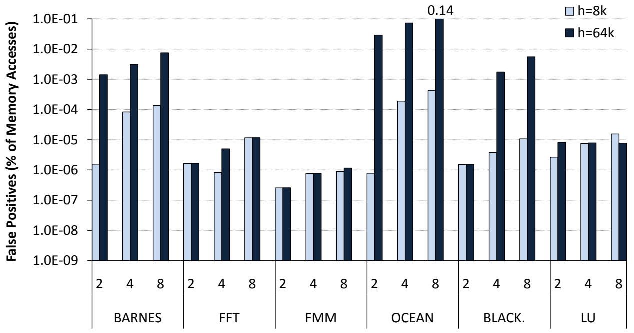


Figure 13. Precision sensitivity to epoch size. Results shown for h=8K and 64K. Y-axis: false positives as percentage of memory accesses, shown on a logscale.

cases for OCEAN), the performance improves with a larger epoch size. Intuitively, this makes sense because the fixed costs of analysis per epoch—including barrier stalls after each pass—are amortized over a larger number of instructions. To understand what happened in OCEAN, let us first consider the impact of epoch size on *accuracy*.

While the advantage of a larger epoch size is better performance, Figure 13 shows that the disadvantage is an increase in the false positive rate[7] of the analysis. (Recall that false negatives are impossible with butterfly analysis.) In a number of cases (e.g., FFT, FMM, and LU), the false positive rate did not increase significantly when the epoch size increased from 8K to 64K instructions, but in other cases it did increase by orders of magnitude. In fact, the increase in the false positive rate for OCEAN helps explain why its performance degraded with a larger epoch size: false positives are expensive to process in ADDRCHECK, and in OCEAN they increased enough to offset the savings in amortized overhead. Aside from OCEAN, the false positive rates remain below 0.01% of memory accesses even with the larger epoch size. With the smaller epoch size, all programs have false positive rates well below 0.001% of memory accesses. Overall, we observe that the epoch size is a knob that can be tuned to trade off performance versus accuracy (subject to a minimum size, as discussed in Section 4.1), and that there are reasonable epoch sizes that offer both high performance and high accuracy.

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

8. Conclusion

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

Acknowledgments

This work is supported in part by grants from NSF and Google.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12), 1996.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[3] J. Auerbach, D. Bacon, P. Cheng, D. Grove, B. Biron, C. Gracie, B. McCloskey, A. Micic, and R. Sciampacone. Tax-and-spend: Democratic scheduling for real-time garbage collection. In *EMSOFT*, 2008.

[4] D. F. Bacon, C. R. Attanasio, H. B. Lee, V. Rajan, and S. Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *PLDI*, 2001.

[5] C. Bienia and K. Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *MoBS*, June 2009.

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

[7] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice & Experience*, 30 (7), 2000.

[8] 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.

[9] R. Chugh, J. Voung, R. Jhala, and S. Lerner. Dataflow analysis for concurrent programs using datarace detection. In *PLDI*, 2008.

[10] J. Chung, M. Dalton, H. Kannan, and C. Kozyrakis. Thread-safe dynamic binary translation using transactional memory. In *HPCA*, 2008.

[11] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: A programmable macro engine for customizing applications. In *ISCA*, 2003.

[12] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *ISCA*, 1986.

[13] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.

[14] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2), 2001.

[15] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, 2002.

[16] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *PPOPP*, 1993.

[17] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3), 1998.

[18] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3), 1996.

[19] J. Krinke. Static slicing of threaded programs. *ACM SIGPLAN Notices*, 33(7), 1998.

[20] D. Long and L. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *ISSTA*, 1991.

[21] B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. In *MICRO*, 2009.

[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] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. SigRace: Signature-based data race detection. In *ISCA*, 2009.

[24] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *ISCA*, 2005.

[25] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. In *ASPLOS*, 2006.

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

[27] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[28] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[29] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[30] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. Lee, and S. Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *SOSP*, 2009.

[31] N. Provos. Improving host security with system call policies. In *USENIX Security*, 2003.

[32] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies - a safe method to survive software failures. In *SOSP*, 2005.

[33] V. Sarkar. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. *Lecture Notes in Computer Science*, 1366, 1998.

[34] 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.

[35] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3), 1981.

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

[37] 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.

[38] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.

[39] M. Xu, R. Bodik, and M. D. Hill. A regulated transitive reduction (RTR) for longer memory race recording. In *ASPLOS*, 2006.

[40] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-assisted lockset-based race detection. In *HPCA*, 2007.

[41] Y. Zhou, P. Zhou, F. Qin, W. Liu, and J. Torrellas. Efficient and flexible architectural support for dynamic monitoring. *ACM Transactions on Architecture and Code Optimization*, 2(1), 2005.

Appendix

Proof of Lemma 5.1. Each statement is proved independently.
If $d_k \in \textsc{Gen}_l$ *then there exists a valid ordering* O_l *such that* $d_k \in \textsc{Gen}(O_l)$.

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

If $d_k \in \textsc{Kill}_l$ *then under all valid orderings* O_l*,* $d_k \notin \textsc{Gen}(O_l)$.

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

Proof of Lemma 5.2. By Induction.
Base Case. $\mathrm{SOS}_0 = \mathrm{SOS}_1 = \varnothing$. According to the invariant, we wish to show $\mathrm{SOS}_2 = \textsc{Gen}(O_0) = \textsc{Gen}_0$.

Any definition $d_k \in \mathrm{SOS}_2$ must be generated by some instruction $(0, t, i)$ in block $(0, t)$, implying it is in $\textsc{Gen}_{(0,t)}$ and $\textsc{Gen}_0$. We can construct a valid ordering O_0 with all instructions in block $(0, t)$ last, so the invariant is satisfied.
Inductive hypothesis: If $s \leq l$, $\mathrm{SOS}_s := \textsc{Gen}_{s-2} \cup (\mathrm{SOS}_{s-1} - \textsc{Kill}_{s-2})$ achieves the invariant.
Inductive step: Consider the SOS for epoch $l+1$. It must include everything generated by epoch $l-1$, which is $\textsc{Gen}_{l-1}$. Now, we must consider how many definitions $d_k \in \mathrm{SOS}_l \wedge d_k \notin \mathrm{SOS}_{l+1}$, which are precisely those definitions d_k such that for all valid orderings, epoch $l-1$ kills d_k. This is exactly what $\textsc{Kill}_{l-1}$ calculates; the elements of $\textsc{Kill}_{l-1}$ are precisely those that should be removed from the SOS_l when creating SOS_{l+1}. This yields the equation: $\mathrm{SOS}_{l+1} = \textsc{Gen}_{l-1} \bigcup (\mathrm{SOS}_l - \textsc{Kill}_{l-1})$. □

Proof Sketch of Theorem 6.1. Observe that the original ADDRCHECK detects errors that occur pairwise between operations (i.e., allocations, accesses, and deallocations) on the same address. It is therefore sufficient to restrict our analysis to pairs of instructions involving the same address.

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

Proof Sketch of Theorem 6.2. We first sketch the proof for the sequentially consistent termination condition. Suppose there was a valid ordering of the first $l+1$ epochs such that $x \leftarrow \perp$ at instruction (l, t, i). That implies there exists a sequence of $k+1$ transfer functions $\hat{f}$ such that the associated instructions in order would taint x.

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

The proof for the relaxed memory model termination condition proceeds similarly, instead arguing that there exists a finite sequence of transfer functions which taints x. □

Proof of Lemma 6.3. A valid ordering of 3 epochs that taints x might taint x when restricted only to (1) the first two epochs, or (2) restricted only to the last 2 epochs. The final case is when all three epochs are used to taint x. In this case, there can be no interleaving between the first and third epochs, because all instructions in the first epoch must commit before any instructions in the third epoch begin. As the first epoch cannot taint x directly and neither can the third epoch (this would put us into cases 1 or 2) then it must be the case that some predecessor of x is tainted by an interleaving of the first epoch with some of the second epoch, and then that there is a valid interleaving between the remaining instructions in the second epoch with the third epoch such that x inherits from y. This is conservatively handled by (3). □