The Butterfly Model: Theoretical Foundations

Michelle Goodstein **Evangelos Vlachos**
Shimin Chen **Phillip Gibbons** **Michael Kozuch**
Todd Mowry

February 12, 2009
CMU-CS-08-170

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research is supported by grants from the National Science Foundation and by Intel Research Pittsburgh. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Intel, the NSF or the US government.

Keywords: dynamic progam monitoring, dataflow analysis

Abstract

Dynamic program monitoring is an effective technique for detecting bugs and security attacks in running applications. Because of the industry-wide shift to multicore chips, program monitoring tools must be extended to monitor parallel programs. Parallel programs introduce a new challenge for monitoring tools: inter-thread dependences. 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 detailed inter-thread dependences. Instead, we assume only that events in the distant past on other threads have become visible; we make no assumptions on 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 prove that our approach is accurate, and sacrifices precision only due to the lack of a a relative ordering among recent events. Finally, we show how our adapted analysis can be used in two popular memory and security tools.

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* [4, 10, 12], *dynamic* [3, 11, 18, 22, 27], or *post-mortem* [19, 29] analysis to diagnose bugs. While these different classes of tools are generally complementary, our focus in this paper is on *dynamic* tools, which we refer to as *"lifeguards"* (since 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 [22], Pin [18], DynamoRio [3], etc.) or with hardware-assisted logging [5]. Lifeguards maintain shadow state to track a particular aspect of correctness as a program executes, such as its memory [21], security [23], or concurrency [27] behaviors.

As we look to the future, the industry-wide shift from increasing processor clock rates to increasing the number of processors integrated onto "multicore" chips suggests that *parallel programming* will become far more commonplace and important. As difficult as it is to write a bug-free sequential program, 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*.

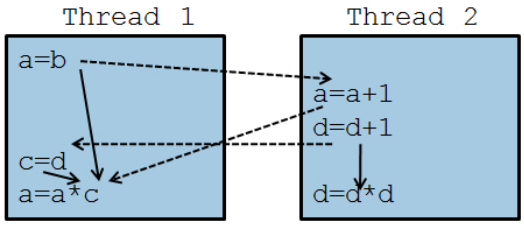


Figure 1: Intra-thread dependences (solid arrows) and inter-thread dependences (dashed arrows). We study the setting in which lifeguards operate without knowing the inter-thread dependences.

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 which 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 [23] 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.

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 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 sequential consistency [9], 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 requires non-trivial modifications to the memory system hardware [19, 29]. 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 we do not capture a serialized ordering and therefore have only partial ordering information regarding inter-thread data dependences, 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 2 occurred before, after, or concurrent with "`a = b`" in THREAD 1. 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 which automatically reasons about bounded windows of uncertainty using an approach inspired by *interval analysis* [28]. 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 efficiently summarize the net effects of these windows of uncertainty across the dynamic traces from concurrently-executing threads using a modified Kleene closure operation.

1.3 Related Work

Several researchers have proposed adaptations of reaching definitions and other dataflow analysis techniques to parallel architectures and programming languages [17, 26, 13, 16]. 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 [16]; some assume no shared variables [17], 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 [26, 13]. Knoop *et al.* introduce a framework that generalizes sequential static unidirectional bit-vector analyses to work with explicitly annotated parallel regions [15]. Chugh *et al.* [6] propose a framework which 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.* [6] demonstrate that while their analysis may be conservative, it remains correct; we will make similar determinations in our work.

1.4 Contributions

This paper makes the following research contributions:

- We propose the "butterfly model" with its bounded regions of uncertainty as a framework for performing dynamic program monitoring.
- We develop a generic framework for performing forward dataflow analysis problems within the butterfly model, as illustrated by reaching definitions and reaching expressions.
- We apply this framework to two lifeguards (ADDRCHECK and TAINTCHECK) to show the applicability of our approach.

2 Background: Dynamic Parallel Monitoring

An important technique for improving software reliability and security, program monitoring performs on-the-fly checking during the execution of applications. 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 [14, 24] to instruction-level [23, 21, 22, 27]. Compared to the former, the latter can obtain highly detailed dynamic information, such as memory references, for detecting bugs more accurately and timely. 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 [5]: (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 [20]: ADDRCHECK 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 [23]: TAINTCHECK 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 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 [3, 18, 22] or micro-code editting in hardware [8]. Lifeguard functionality is performed as the modified application code executes. In contrast, the second design offloads lifeguard functionality to separate cores. An execution trace (or log) of the application is captured at the core running the application through hardware, and shipped (via the last-level on-chip cache) to the core running the lifeguard for monitoring purposes [5].

We observe that lifeguards see a simple sequence of application events[1] 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 (log) 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 [7], which assumes transactional memory support.) However, in the multicore era, applications increasingly involve parallel execution; therefore, monitoring support for multithreaded applications

[1] We will only monitor user level instructions; system level instructions are beyond our scope.

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.

To begin, we extend the model of monitoring support to include 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, we will permit the lifeguard analysis to lag behind the application execution somewhat, and rely on existing techniques [5] to ensure that no real damage occurs during this (short) window.[2] Note that, as discussed in Section 1, the event sequences do not contain detailed fine-grain inter-thread dependences information.

3 Challenges in Adapting Dataflow Analysis to Parallel Monitoring

In the absence of detailed fine-grain inter-thread dependence information, there are many possible interleavings of the event sequences lifeguards see.[3] Our approach is to adapt dataflow analysis, traditionally run statically at compile-time, as a dynamic run-time tool that enables us to reason about possible interleavings of different threads' executed instructions.

In this section, we will motivate our design decisions, showing how simpler constructions are either too inefficient, too imprecise, or both. Throughout this section and through Section 4.3, we will assume a sequentially consistent machine. 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.

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 is followed sequentially. This yields a graph that at first glance resembled a control flow graph; it seemed that enough of the structure would be similar to apply dataflow analysis.

Figure 2(a) shows a very simple code example of two threads modifying three variables. Even

[2] A lifeguard thread raising an error may interrupt the application to take corrective action [25]. Some delay between application error and application interrupt is unavoidable, due to the lag in interrupting all the application threads.

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

with only three total instructions, we still require several arcs to reflect all the possible concurrency, shown in Figure 2(b). This may look like manageable ; unfortunately, adding arcs over an entire dynamic runlength leads to an explosion in arcs and 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 $O(tn)$ edges due to the sequential nature of execution within a thread and $O(n^2t)$ edges due to potential concurrency: each node in thread "j has edges to all nodes in all other threads, which is $O(nt)$ and there are n instructions per thread.

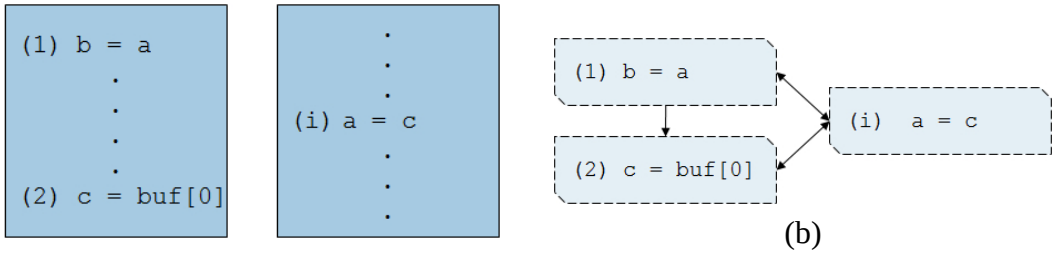


Figure 2: Two threads modifying three shared memory locations, shown (a) as traces and (b) in a CFG. Throughout this paper, concurrent blocks of instructions appear in rectangles with solid borders and single instructions appear as hexagons with dashed borders.

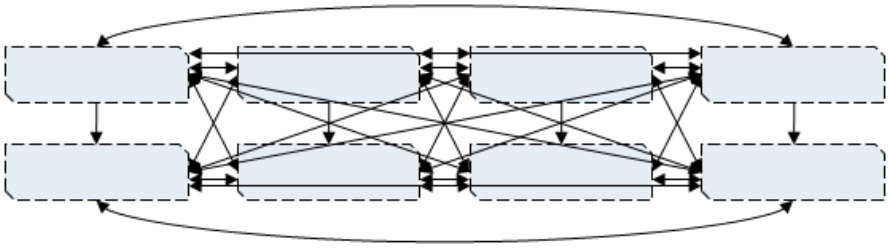


Figure 3: CFG of 4 threads with 2 instructions each.

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.

Another 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

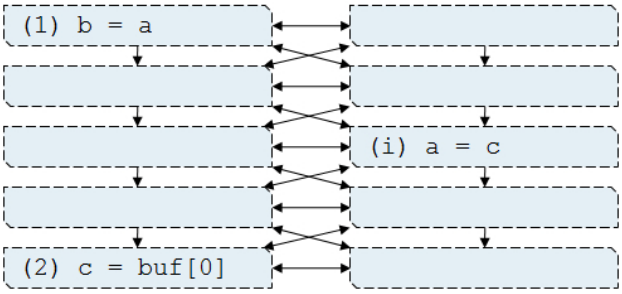


Figure 4: Two threads concurrently updating variables `a`, `b` and `c`

be tainted, we must have (2) execute before (i), and (i) before (1)–contradicting the sequential consistency assumption.

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.

Using this intuition, we could modify our CFG-like approach to only draw edges between individual instructions which are potentially concurrent. Unfortunately, we can still conclude that an instruction at the end of the program taints the destination of the first instruction in Thread 1, by zig-zagging up between 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 called the **Butterfly Model**.

4 The Butterfly Model

In this section, we introduce a new model of parallel program execution, termed the butterfly model. The butterfly model 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 graphical formalization which is suitable for adapting dataflow analysis to perform dynamic parallel application monitoring.

4.1 Mechanics

We rely on a regular signal, or **heartbeat**, to be reliably delivered to all cores. For lifeguards using 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 logs into **epochs**.

By making sure that the frequency of heartbeat accounts for reception of the heartbeat, memory latency for instructions involving reads or writes, and time for all instructions in the reorder and store buffers to become globally visible, we can guarantee *non-adjacent* epochs, or epochs that do not share a heartbeat boundary, have strict happens-before relationships. On the other hand, due to the latency in receiving a heartbeat, we will consider instructions in *adjacent* epochs, or epochs which 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.

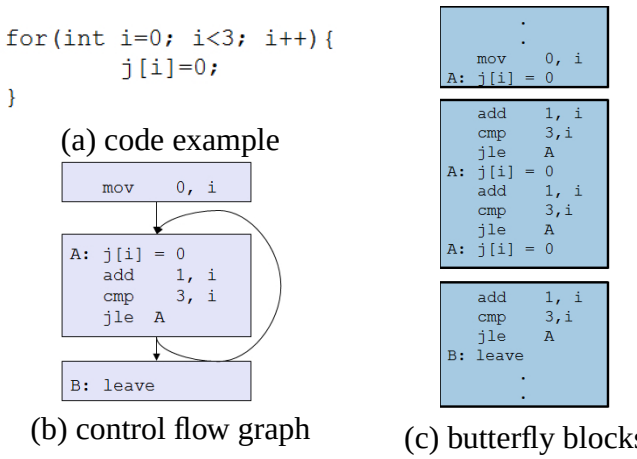


Figure 5: Unlike a basic block, a butterfly block is a sequence of dynamic application events.

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).

The butterfly 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 before any instructions in

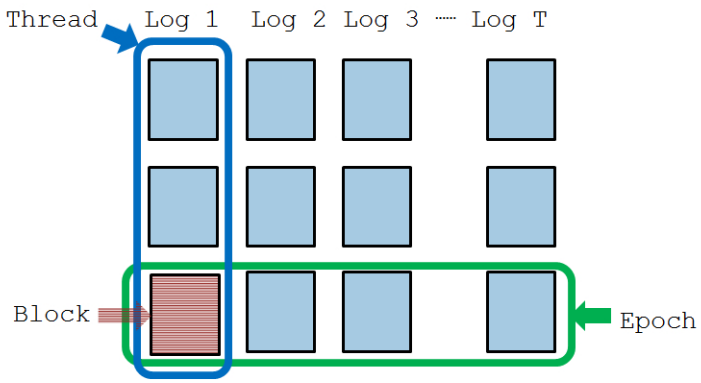


Figure 6: A particular block is specified by an epoch id l and thread id t. In reality, epoch boundaries will be staggered and blocks will be of different sizes.

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 delays in receiving a heartbeat. If an instantaneous heartbeat would have placed instruction j in epoch l, our model will require that instruction j instead will always be in either epoch $l - 1$, l or $l + 1$.

The butterfly model 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). We call block (l, t) the **body** of the butterfly, $(l - 1, t)$ the **head** of the butterfly and $(l + 1, t)$ the **tail** of the butterfly. The head always executes before the body, which executes before the tail. Relative to the body, for all threads $t' \neq t$, blocks $(l - 1, t'), (l, t')$ and $(l + 1, t')$ are in the **wings** of the block (l, t)'s butterfly. Figure 7(a) illustrates these definitions.

4.2 Butterfly Framework

As described, the butterfly model 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 $(2, l - 2)$, and the last instruction of $(l, 1)$ can be followed immediately by the first instruction of $(2, l + 2)$. This graph contains many edges, even though they are bounded. While we still wish to adapt dataflow analysis techniques, we will make the final observation that behaving conservatively can retain accuracy, allow us to retain the flavor of dataflow analysis, but not require multiple iterations. In fact, we will show that with only two passes over each block, we can reach an accurate conclusion about the metadata state.

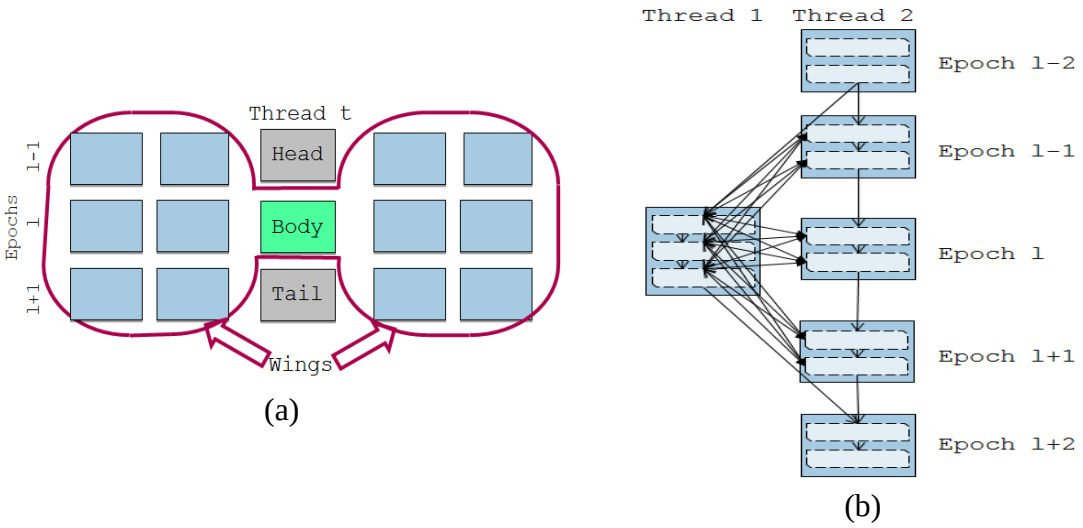


Figure 7: Potential concurrency in the butterfly model, shown at the (a) block and (b) instruction levels

In the butterfly model, there is only a bounded degree of arbitrary interleaving; dataflow analysis makes sense on subgraphs of three contiguous epochs only. Since the framework we propose only considers 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** or LSOS which is the SOS modified to take into account that all instructions in the head of the butterfly have already executed as well.

4.3 A Two Pass Algorithm

We develop a two pass algorithm inspired by dataflow analysis. In the first stage, we perform dataflow analysis using locally available state, eagerly perform any lifeguard required checks, and produce a summary of lifeguard-relevant events. Checks performed at this stage are preliminary; any errors detected are real, but not all errors will be detected in this stage. This is considered the first pass. In the second stage of the algorithm, the threads compute the meet of all summaries produced in the wings.[4]

In the third stage, we repeat our dataflow analysis, this time incorporating state from the wings. This is considered the second pass; in this pass, we repeat the checking algorithm. In the fourth and final stage, the threads agree on an accurate summarization of the entire epoch's activity, and an update to the SOS is computed.

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]As a performance optimization, this can be done using a combining tree and utilizing $O(\log t)$ meet operations over summaries

4.4 Relaxed Memory Models

Relaxed memory models can actually be sufficient for our purposes as long as sequences of assignments causing x to inherit state from y are respected. Completely independent instructions can be reordered and tolerated, because our operations are on sets; 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 if we do not reorder dependent instructions. The butterfly model preserves intra-thread dependences, but does not have access to inter-thread dependences. An example was shown in Figure 1; arcs that serialize reads and writes across multiple threads are not preserved in the butterfly model.

However, for lifeguards like TAINTCHECK, 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]. For these specific cases, we will have to choose between requiring sequential consistency or further relaxing the checking algorithm. If we do not have sequential consistency, then we must assume the processor is arbitrarily reordering instructions, only honoring intra-thread dependences. This is discussed in Section 6.2.

5 Dynamic Parallel Dataflow Analysis

This section presents our adaptation of dataflow analysis to dynamic program monitoring in the butterfly model. 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 [31, 30, 5] 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 in the butterfly model. 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. Later in Section 6 we will show 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—motivated by the fact that events in epoch l can only interleave with events from $l-1$, l, or $l+1$. This not only enables our analysis to proceed as the application executes, it also bounds the complexity of our analysis.

- We deviate from the normal $O(n^2)$ iterative convergence of the standard analysis, requiring only two passes over each epoch. The time spent 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 compensates for the loss of iterative convergence, and enables using a sliding window model without sacrificing accuracy.

- 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 block (l,t) generated which 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 which 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 definitions no later than epoch $l-2$ that could reach epoch l.

Processing a Level. To motivate our work, we examine Figure 8. 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 is due to initializiation; the very first butterfly uses only epochs 0 and 1. 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 is available and so is SOS_l. We can eagerly perform checks, and 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), we do not need to regenerate the summary as the block changes position in the sliding window. So after the first butterfly, we are only performing a first pass on the newest epoch under consideration; the summaries for older blocks have already been completed.

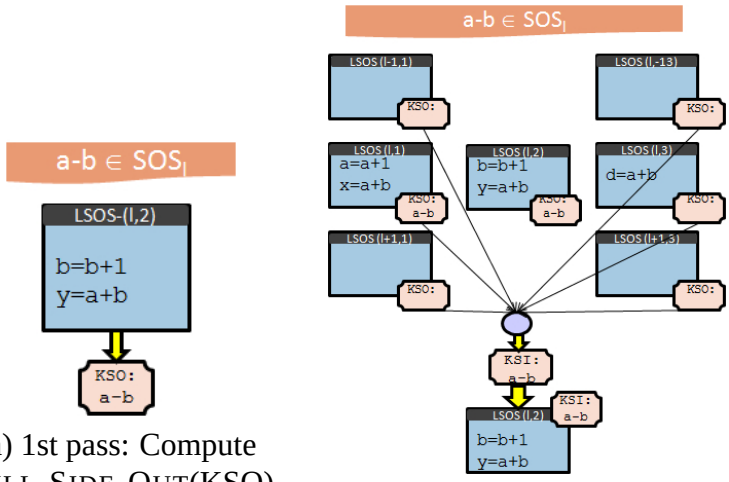
Using that information, we now examine Figure 8(b). This shows the entire butterfly centered around block $(l,2)$. As part of the second stage, summaries from all blocks in the wings are first collected and combined (represented by the central circle), producing one summary for the entire wings (in reaching expressions, this is $\text{KILL-SIDE-IN}_{l,2}$). Finally, $(l,2)$ can repeat its checking algorithm, noting that it is not the only block to kill `a-b`. Once the second pass is over, the final stage creates an epoch summary.

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) yields $a+b$ defined at the end. Hence, `a+b` is added to SOS_{l+2}, and `a-b` is removed.

Every definition within the block is first inserted into a common summary. These summaries are then combined into a wing summary–a different summary for each block. The summaries immediately below the first pass blocks represent GEN-SIDE-OUT, whereas the wing summaries are actually the GEN-SIDE-IN. In general with many threads, there would be many more GEN-SIDE-OUTs combining to form the GEN-SIDE-IN. Finally, the second pass is repeated; in

both passes, a copy of the LSOS is available, but in the second pass, the blocks also have access to the wing summary. Figure 8 does not illustrate the steps of summarizing an entire epoch, and finally updating the SOS.

Note that a single writer corresponds to 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.



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

(b) 2nd pass: Aggregate KSOs from the wings, combine into KILL-SIDE-IN(KSI) and rerun checks.

Figure 8: Computing KILL-SIDE-OUT and KILL-SIDE-IN in reaching expressions. Boxes with beveled edges are summaries.

5.1 Dynamic Parallel Reaching Definitions

Generating a definition in the butterfly model 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 the butterfly model 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 interleaving or even a later redefinition in the same block. For this reason, we conservatively set KILL-SIDE-OUT$_{l,t}$ = KILL-SIDE-IN$_{l,t}$ = $\varnothing$, 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 allow us to define reaching an entire epoch l to mean that there is

some interleaving of all instructions in the first l epochs such that running a sequential reaching definitions analysis will conclude that d_k reaches. We calculate:

$$\text{GEN}_l = \bigcup_t \text{GEN}_{l,t}$$

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

$$\text{where } \text{KILL}_{(l-1,l),t} = (\text{KILL}_{l-1,t} - \text{GEN}_{l,t}) \cup \text{KILL}_{l,t}$$
$$\text{NOT-GEN}_{(l-1,l),t} = \{d_k | d_k \notin \text{GEN}_{l-1,t} \wedge d_k \notin \text{GEN}_{l,t}\}$$

To prove that these formulations are correct, 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 the butterfly model. 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 is accurate (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 instructions 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. We will define the set $\text{GEN}(O_k)$ 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.

Theorem 1. *If $d_k \in \text{GEN}_l$ then there exists a valid ordering O_l such $d_k \in \text{GEN}(O_l)$.*

Proof. If $d_k \in \text{GEN}_l$, then there is some block (l,t) such that $d_k \in \text{GEN}_{l,t}$, implying there exists some index i such that $d_k \in \text{GEN}_{l,t,i} \wedge \forall j > i, d_k \notin \text{KILL}_{l,t,j}$. Then O_l is any valid ordering where all instructions in block (l,t) execute last. □

Theorem 2. *If $d_k \in \text{KILL}_l$ then under all valid orderings O_l, $d_k \notin \text{GEN}(O_l)$.*

Proof By Contradiction. Suppose $d_k \in \text{KILL}_l$ but there exists some ordering O_l such that $d_k \in \text{GEN}(O_l)$.There are two cases:

Case 1: *There is some instruction in epoch l or $l-1$ in thread t which generates d_k and no later instruction in thread t kills d_k.* This violates the assumption that every thread either kills d_k in epoch l or $l-1$ without subsequently regenerating, or else refrains from generating d_k in those epochs. $\Rightarrow\Leftarrow$

Case 2: *No instruction in epoch l or epoch $l-1$ generates d_k.*

If no instruction in epoch l or $l-1$ generates d_k, then d_k must be killed in O_l, since there exists at least one block such that $d_k \in \text{KILL}_{l,t}$ and the instruction which kills d_k must occur strictly after any instruction that generates d_k. $\Rightarrow\Leftarrow$

Both cases violate the assumption that $d_k \in \text{KILL}_l$. □

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 say that $d_k \in \text{GEN}(O_l)$ if a sequential analysis on O_l concludes that d_k reaches the end of O_l. We will require the following invariant:

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

The SOS update rule becomes:

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

Theorem 3. $\text{SOS}_l := \text{GEN}_{l-2} \cup (\text{SOS}_{l-1} - \text{KILL}_{l-2})$ *achieves the invariant.*

Proof. Base cases: $\text{SOS}_0 = \text{SOS}_1 = \varnothing$. According to the invariant, $\text{SOS}_2 = \text{GEN}(O_0) = \text{GEN}_0$. Any definition $d_k \in \text{SOS}_2$ must be generated by some instruction $(0, t, i)$ in block $(0, t)$, implying it is in GEN_0. For any definition $d_k \in \text{GEN}_0$, we could find one block $(0, t)$ such that $d_k \in \text{GEN}_{0,t}$ and execute all instructions in this block last in O_0 and the ordering would still be valid, so the invariant is satisfied.

Then, $d_k \in \text{GEN}(O_0)$, so we satisfy the invariant. Since we can construct a valid ordering for all $d_k \in \text{GEN}_0$, $\text{SOS}_2 = \text{GEN}_0$. Since $SOS_1 = \varnothing, (\text{SOS}_1 - \text{KILL}_0) = \varnothing \Rightarrow \text{GEN}_0 \cup \varnothing = \text{GEN}_0$.
Inductive hypothesis: If $s \leq l$, $\text{SOS}_s := \text{GEN}_{s-2} \cup (\text{SOS}_{s-1} - \text{KILL}_{s-2})$ achieves the invariant.
Inductive step: Consider the SOS for epoch $l + 1$. It must include everything generated by epoch $l - 1$, which is GEN_{l-1}. Now, we must consider how many definitions $d_k \in \text{SOS}_l \wedge d_k \notin \text{SOS}_{l+1}$, which are precisely those definitions d_k such that for all valid orderings, epoch $l - 1$ kills d_k. This is exactly what KILL_{l-1} calculates; the elements of KILL_{l-1} are precisely those that should be removed from the SOS_l when creating SOS_{l+1}. This yields the equation: $\text{SOS}_{l+1} = \text{GEN}_{l-1} \bigcup (\text{SOS}_l - \text{KILL}_{l-1})$. □

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. We have:

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

The invariant required for the LSOS is:

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

A similar proof to Theorem 3 shows this update rule achieves its invariant. Let $\text{LSOS}_{l,t,k}$ denote the updated version of the LSOS after k instructions have executed.

$$\text{LSOS}_{l,t,k} = \begin{cases} \text{LSOS}_{l,t} \text{ if } k = 0 \\ \text{GEN}_{l,t,k-1} \cup (\text{LSOS}_{l,t,k-1} - \text{KILL}_{l,t,k-1}) \text{ otherwise} \end{cases}$$

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

5.1.3 Calculating In and Out

Let $\text{IN}_{l,t,0} = \text{IN}_{l,t}$ represent the set of definitions that could possibly reach the beginning of block (l,t). $\text{IN}_{l,t}$ should be the union of the set of valid definitions of all possible interleavings that could execute before instruction $(l,t,0)$ executes. Let $\text{IN}_{l,t,i}$ be the set of inputs that reach instruction (l,t,i). $\text{OUT}_{l,t,i}$ and $\text{OUT}_{l,t}$ are the sets of definitions that are still defined after executing instruction (l,t,i) or block (l,t), respectively. Then:

$$\text{IN}_{l,t} = \text{GEN-SIDE-IN}_{l,t} \bigcup \text{LSOS}_{l,t} \; \text{IN}_{l,t,i} = \text{GEN-SIDE-IN}_{l,t} \cup \text{LSOS}_{l,t,i}$$
$$\text{OUT}_{l,t,i} = \text{GEN}_{l,t,i} \bigcup (\text{IN}_{l,t,i} - \text{KILL}_{l,t,i})$$
$$\text{OUT}_{l,t} = \text{GEN}_{l,t} \bigcup (\text{IN}_{l,t} - \text{KILL}_{l,t})$$

Using reaching definitions as a lifeguard, we have now shown how to compute the checks–essentially, the 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 part one, our local computations are $\text{GEN}_{l,t}$, $\text{KILL}_{l,t}$ and $\text{LSOS}_{l,t}$. These are used for our checking algorithm. The summary information is $\text{GEN-SIDE-OUT}_{l,t}$. For the second stage, $\sqcap$ is $\cup$, calculated over the GEN-SIDE-OUT from the wings, to get $\text{GEN-SIDE-IN}_{l,t}$. We then use $\text{GEN-SIDE-IN}_{l,t}$ to perform our second round of checks. Finally, we use GEN_l and KILL_l to update the SOS.

5.2 Dynamic Parallel Reaching Expressions

Expressions only reach a block if no path to a block (l,t) kills expression e. If every path to a block has the expression survive, then there is no need to recompute the expression. However, if any path 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 along all valid orderings e_k reaches p. This gives some intuition that KILL in reaching expressions will behave like GEN in reaching definitions, and likewise GEN in reaching expressions behaves like KILL in reaching definitions.

We will again let $\text{GEN}_{l,t,i}$ represent the set of expressions generated by instruction (l,t,i) and $\text{KILL}_{l,t,i}$ represent the set of definitions killed by instruction (l,t,i). $\text{GEN}_{l,t,i} = \{e_k\}$ if and only if instruction (l,t,i) generates expression e_k, and is empty otherwise. Let $\text{KILL}_{l,t,i}$ be the set of expressions killed by instruction i in thread t and epoch l. We calculate $\text{GEN}_{l,t}$ and $\text{KILL}_{l,t}$ as usual.

Let $\text{KILL-SIDE-OUT}_{l,t}$ 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 $\text{KILL}_{l,t,i}$. Let $\text{KILL-SIDE-IN}_{l,t}$ represent the set of expressions visible to block (l,t) that are killed by the wings.

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

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

The properties we desire for GEN_l and KILL_l are roughly the opposite of those from reaching definitions.

$$\begin{gathered}
\text{KILL}_l = \bigcup_t \text{KILL}_{l,t} \\
\text{GEN}_l = \bigcup_t (\text{GEN}_{l,t} \cap (\bigcup_{t' \neq t} \text{GEN}_{(l-1,l),t'} \cup \text{NOT-KILL}_{(l-1,l),t'})) \\
\text{where } \text{GEN}_{(l-1,l),t} = (\text{GEN}_{l-1,t} - \text{KILL}_{l,t}) \cup \text{GEN}_{l,t} \\
\text{NOT-KILL}_{(l-1,l),t} = \{e_k | e_k \notin \text{KILL}_{l-1,t} \wedge e_k \notin \text{KILL}_{l,t}\}
\end{gathered}$$

The proof that KILL_l is correct is analogous to the proof that GEN_l is correct in reaching definitions; and likewise the GEN_l proof corresponds to the KILL_l proof in reaching definitions.

5.2.1 Updating State

The SOS has the same equation and update rule. The LSOS has a slightly different form, reflecting the different roles GEN and KILL play. In the reaching expressions environment, an expression only reaches an instruction (l, t, i) if it has been defined along all paths. So, if $e_k \in \text{SOS} \wedge e_k \notin \text{KILL}_{l-1,t}$, then $e_k \in \text{LSOS}_{l,t}$ since e_k is calculated along all paths. However, if $e_k \in \text{KILL}_{l-1,t}$ then at least one path exists where the expression is not defined. If $e_k \notin \text{SOS}$, the only way that $e_k \in \text{LSOS}_{l,t}$ is if it is defined by the head ($e_k \in \text{GEN}_{l-1,t}$) and no other thread t' ever kills e_k in epoch $l-1$; otherwise, since 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:

$$\begin{gathered}
\text{LSOS}_{l,t} = \left(\text{GEN}_{l-1,t} - \bigcup_{t' \neq t} \text{KILL}_{l-2,t'}\right) \cup \\
(\text{SOS} - \text{KILL}_{l-1,t})
\end{gathered}$$

$\text{LSOS}_{l,t,k}$ has the same update rule as in reaching definitions.

5.2.2 Calculating In and Out

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

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

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

$$\begin{aligned}
\text{OUT}_{l,t,i} &= \text{GEN}_{l,t,i} \bigcup (\text{IN}_{l,t,i} - \text{KILL}_{l,t,i}) \\
\text{OUT}_{l,t} &= \text{GEN}_{l,t} \bigcup (\text{IN}_{l,t} - \text{KILL}_{l,t})
\end{aligned}$$

5.2.3 Applying the Two-Pass Algorithm

Parameters for the Two-Pass Algorithm are similar to those in Section 5.1.4. We again calculate $\text{GEN}_{l,t}$, $\text{KILL}_{l,t}$ and $\text{LSOS}_{l,t}$, but now the summary information is $\text{KILL-SIDE-OUT}_{l,t}$. The second stage uses $\cup$ for $\sqcap$ but calculates over all the $\text{KILL-SIDE-OUT}_{l',t'}$ in the wings, to get $\text{KILL-SIDE-IN}_{l,t}$, which is then used to perform our second round of checks. Finally, we use GEN_l and KILL_l to update the SOS.

6 Lifeguard Analysis in the Butterfly Model

Now that we have shown how the butterfly model can be applied to basic dataflow analysis, we extend it to two lifeguards. For each of these lifeguards, we will show that we lose some precision but no accuracy; compared against any valid ordering, we will catch all errors present in the valid ordering but potentially flag some false positives. Our main contribution will be accurate, parallel adaptations of ADDRCHECK and TAINTCHECK which do not need access to inter-thread dependences. For each lifeguard, we also show how to update metadata using dataflow analysis.

6.1 AddrCheck

ADDRCHECK[20], 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 the butterfly model, one thread can allocate memory before another writes, but if these operations are in adjacent epochs, the operations are potentially concurrent.

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 accuracy; for all valid orderings, we want to know whether an access to memory location m is always an access to allocated memory. We modify $\text{GEN}_{l,t,i} = \{m\}$ if and only if instruction (l, t, i) allocates memory location m and otherwise $\varnothing$. Likewise, $\text{KILL}_{l,t,i} = \{m\}$ if and only if instruction (l, t, i) deallocates memory location m and otherwise $\varnothing$. $\text{GEN}_{l,t}$, $\text{KILL}_{l,t}$, GEN_l, KILL_l, SOS and LSOS all retain the same equations and update rules.

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. In the first pass, we check that any address being deallocated or accessed is in our LSOS. We check an address being malloced is not in our LSOS. If we violate either of these conditions, we report an error.

We produce a summary $s_{l,t} = (\text{GEN}_{l,t}, \text{KILL}_{l,t}, \text{ACCESS}_{l,t})$, where $\text{ACCESS}_{l,t}$ contains all addresses that block (l, t) accessed. When we combine the SIDE-IN, we produce

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

In the second pass, we want to ensure that allocations and deallocations were *isolated* from any other concurrent thread. To do so, we perform the following operation, where $s_{l,t}$ is abbreviate s and $S_{l,t}$ is abbreviated S:

$$(s.\text{GEN}_{l,t} \cup s.\text{KILL}_{l,t}) \bigcap (S.\text{GEN}_{l,t} \cup S.\text{KILL}_{l,t} \cup) \bigcup$$
$$(s.\text{ACCESS}_{l,t} \bigcap (S.\text{GEN}_{l,t} \cup S.\text{KILL}_{l,t})) \bigcup$$
$$(S.\text{ACCESS}_{l,t} \bigcap (s.\text{GEN}_{l,t} \cup s.\text{KILL}_{l,t}))$$

Theorem 4. *Given a particular division of instructions into epochs, any error detected by* ADDRCHECK *on a valid ordering will also be flagged in the butterfly model.*

Proof. We consider memory location x, and show that if a valid ordering could lead to an error in a sequentially consistent model, it will also be reported as an error in the butterfly model.

If instruction i on thread t is executed before instruction j on thread t' in a valid ordering, then either i will occur at least one epoch before j and will be ordered in the model, or i and j will occur in adjacent epochs and appear possibly concurrent. Whenever i and j are separated by at least one epoch, then the original ADDRCHECKand our modified version come to the same conclusion. However, when i and j are within the same butterfly, then we cannot tell whether the error actually manifested, and instead report a *potentially concurrent error*. By treating all potentially concurrent errors as true errors, we catch every error that would be reported for any valid ordering. □

6.2 TaintCheck

As described in Section 2 TAINTCHECK[23]tracks the propagation of taint through a program's execution; if one of the two sources is tainted, then the destination is considered tainted. When extending TAINTCHECK to work in the butterfly model, we extend this conservative assumption; if a valid ordering O that causes some address m, to appear tainted at instruction (l, t, i), we conclude that (l, t, i) taints m even if it does not taint m under any other valid ordering. We modify reaching definitions to accomodate TAINTCHECK.

Unfortunately, adapting TAINTCHECK to the butterfly model is not as simple as modifying ADDRCHECK in Section 6.1. 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 tuple (l, t, i) instead of integers. We also define a function `loc()` which given a SSA numbering (l, t, i) returns x, where x is the location being written by instruction (l, t, i). Our metadata are transfer functions between SSA-numbered variables and their taint status, with $\bot$ as taint and $\top$ as untaint. The SOS will only contain addresses believed to be tainted. Then:

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

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

Checking Algorithm

The main difference between TAINTCHECK and reaching definitions is the checking algorithm. Like ADDRCHECK , TAINTCHECK requires more than IN or OUT can provide.

Given a function $(x \leftarrow s)$, a location $y_{l,t,i}$ is a **parent** of x if $\exists z_{l',t',i'} \in s$ s.t. $\texttt{loc}(l,t,i) = \texttt{loc}(l',t',i')$. 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 $(l,t,i) < (l',t',i')$.

We define a function Check, in Algorithm 1 which takes a particular transfer function of the form $(x_{l,t,i} \leftarrow s)$ and a set of transfer functions T.

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

As discussed in Section 4.4, TAINTCHECK requires a guarantee of either sequential consistency, or a conservative, relaxed checking termination condition. If we have sequential consistency, then it makes sense to enforce sequential execution within each thread. To do so, we associate a 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 maintaining accuracy. By disallowing a parent to eventually be replaced by itself we will prevent infinite loops, since there are only a bounded number of potential parents; it will not guarantee that the ordering which taints memory location m actually is valid. This resembles iteration as performed in dataflow analysis to resolve loops.

Theorem 5. *If check returns $\top$, then there is no valid ordering of the first $l+1$ epochs such that x is $\bot$ at instruction (l,t,i).*

Proof. We will restrict our analysis to the sequentially consistent termination condition. Suppose there was a valid ordering of the first $l+1$ epochs such that $x \leftarrow \bot$ at instruction (l,t,i). That implies there exists a sequence of $k+1$ transfer functions $\hat{f}$ such 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. This would be a legitimate sequence of parents to follow, so we would conclude $(x \leftarrow \bot)$. □

False Dependencies:. 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 all 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.

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.

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

Proof. A valid ordering of 3 epochs that taints x might taint x when restricted only to (1) the first two epochs, or (2) restricted only to the last 2 epochs. The final case is when it needs all three epochs to taint x. In this case, there can be no interleaving between the first and third epochs, since 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 precisely (3). □

6.2.1 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 9. 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 which taints a has actually already executed. However, considering an address to be tainted early is accurate yet imprecise, and so we choose accuracy over precision.

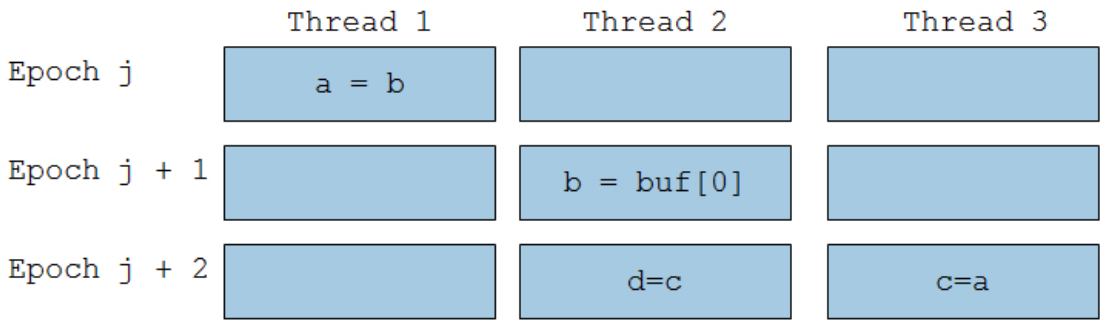


Figure 9: Updating SOS is nontrivial for Taintcheck. By the end of epoch $j+1$, a has been tainted, but the SOS might need to be updated before blocks in epoch $j+2$ begin butterfly analysis.

Define the function $\text{LASTCHECK}(x, l, t)$ 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 $\text{LASTCHECK}(x, l, t)$ will return $\top$ or $\bot$; otherwise, it returns $\varnothing$. We can extend this definition to $\text{LASTCHECK}(x, (l-1, l), t)$ 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

$$\text{GEN}_l = \bigcup_t \{x | \text{LASTCHECK}(x, l, t) = \bot\}$$
$$\text{KILL}_l = \bigcup_t \{x | \text{LASTCHECK}(x, l, t) = \top \wedge (\forall t' \neq t, \text{LASTCHECK}(x, (l-1, l), t) = \top \vee \text{LASTCHECK}(x, (l-1, l), t) = \varnothing)\}$$

This is an almost identical formulation to reaching definitions; the difference is that we use LASTCHECK to change our metadata format from transfer functions to tainted addresses.

$$\text{SOS}_l := \text{GEN}_{l-2} \cup (\text{SOS}_{l-1} - \text{KILL}_{l-2}) \quad \forall l \geq 2$$
$$\text{SOS}_0 = \text{SOS}_1 = \varnothing$$
$$\text{LSOS}_{l,t} = \text{GEN}_{l-1,t} \cup (\text{SOS}_l - \text{KILL}_{l-1,t}) \cup \{d_k | \exists t' \neq t \text{ s.t. } d_k \in \text{GEN}_{l-2,t'} \wedge d_k \in \text{KILL}_{l-1,t} \wedge d_k \in \text{SOS}_l\}$$

We will 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 $x \in \text{SOS}_{l-2}$.*

Condition 2. *If $x \in \text{SOS}_{l-2}$, then there exists at least one thread t s.t. 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 Conclusions

In this paper, we presented a new approach for dynamic, parallel program monitoring without relying on inter-thread dependences, called the butterfly model. Requiring only a simple heartbeat mechanism, this model supports sophisticated analysis of multithreaded programs. We demonstrated how to adapt dataflow analysis techniques, traditionally used statically at compile time, to dynamically compute reaching definitions and reaching expressions over an execution trace using the butterfly model. Finally, we showed how to express two lifeguards in terms of reaching expressions and reaching definitions, providing new methods of dynamic parallel program monitoring.

References

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

[2] A.V. Aho, R. Sethi, and J.D. Ullman. Compilers: Principles, Techniques, and Tools. *Reading, Mass*, 1986.

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

[4] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *SPE*, 30(7), 2000.

[5] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. Flexible Hardware Acceleration for Instruction-grain Program Monitoring. In *ISCA*, 2008.

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

[7] Jaewoong Chung, Michael Dalton, Hari Kannan, and Christos Kozyrakis. Thread-safe dynamic binary translation using transactional memory. In *HPCS*, Feb 2008.

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

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

[10] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.

[11] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2), 2001.

[12] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, 2002.

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

[14] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[15] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: efficient and optimal bitvector analyses for parallel programs. *TOPLAS*, 18(3):268–299, 1996.

[16] Jens Krinke. Static slicing of threaded programs. In *ACM SIGPLAN Notices*, 1998.

[17] D. Long and L.A. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, 1991.

[18] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[19] Satish Narayanasamy, Gilles Pokam, and Brad Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *ISCA*, 2005.

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

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

[22] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

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

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

[25] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies - a safe method to survive software failures. In *SOSP*, 2005.

[26] V. Sarkar. Analysis and Optimization of Explicitly Parallel Programs Using the Parallel Program Graph Representation. *Lecture Notes in Computer Science*, pages 94–113, 1998.

[27] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic race detector for multi-threaded programs. *ACM TOCS*, 15(4), 1997.

[28] Robert Endre Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, 1981.

[29] Min Xu, Rastislav Bodik, and Mark D. Hill. A regulated transitive reduction (rtr) for longer memory race recording. In *ASPLOS*, 2006.

[30] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-assisted lockset-based race detection. In *HPCA-13*, 2007.

[31] Yuanyuan Zhou, Pin Zhou, Feng Qin, Wei Liu, and Josep Torrellas. Efficient and flexible architectural support for dynamic monitoring. *ACM TACO*, 2(1), 2005.