Parallelizing Dynamic Information Flow Tracking

Olatunji Ruwase† Phillip B. Gibbons‡ Todd C. Mowry† Vijaya Ramachandran§
Shimin Chen‡ Michael Kozuch‡ Michael Ryan‡

†Carnegie Mellon University, Pittsburgh, PA, USA
‡Intel Research Pittsburgh, Pittsburgh, PA, USA
§The University of Texas at Austin, Austin, TX, USA

oor@cs.cmu.edu, phillip.b.gibbons@intel.com, tcm@cs.cmu.edu, vlr@cs.utexas.edu,
shimin.chen@intel.com, michael.a.kozuch@intel.com, michael.p.ryan@intel.com

ABSTRACT

Dynamic information flow tracking (DIFT) is an important tool for detecting common security attacks and memory bugs. A DIFT tool tracks the flow of information through a monitored program's registers and memory locations as the program executes, detecting and containing/fixing problems on-the-fly. Unfortunately, sequential DIFT tools are quite slow, and DIFT is quite challenging to parallelize. In this paper, we present a new approach to parallelizing DIFT-like functionality. Extending our recent work on accelerating sequential DIFT, we consider a variant of DIFT that tracks the information flow only through unary operations (*relaxed DIFT*), and yet makes sense for detecting security attacks and memory bugs. We present a parallel algorithm for relaxed DIFT, based on *symbolic inheritance tracking*, which achieves linear speed-up asymptotically. Moreover, we describe techniques for reducing the constant factors, so that speed-ups can be obtained even with just a few processors. We implemented the algorithm in the context of a Log-Based Architectures (LBA) system, which provides hardware support for logging a program trace and delivering it to other (monitoring) processors. Our simulation results on SPEC benchmarks and a video player show that our parallel relaxed DIFT reduces the overhead to as low as 1.2X using 9 monitoring cores on a 16-core chip multiprocessor.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*

General Terms

Algorithms, Reliability, Security

Keywords

dynamic information flow tracking (DIFT), program monitoring, log-based monitoring, parallel algorithm, taint analysis

SPAA'08, June 14–16, 2008, Munich, Germany.

1. INTRODUCTION

Programmers and users of software care about three metrics: correctness, performance, and power consumption. Of the three, correctness is paramount. However, imperfect code consistently finds its way to market despite advancements in software development tools. This is partly due to the challenge of writing bug-free code and partly due to software life cycle pressures that encourage software vendors to continuously rush new functionality to market. Making matters worse, even the most obscure bug can represent a security vulnerability that potential attackers may exploit.

A variety of tools have been developed to help find and even fix bugs [1–12, 14–27, 29–36]. One class of such tools—which we call *lifeguards*—monitors an application as it executes. Complementing compile-time or other pre-execution tools, lifeguards have the advantages of (i) observing the actual dynamic state (e.g., program inputs, memory aliasing, etc.), and (ii) hopefully limiting the damage of a problem through either containment or on-the-fly repair. Unfortunately, many of the most useful lifeguards monitor nearly every program instruction, and as a result, slow down the monitored program significantly (e.g., by 3–50X [19]). Examples include lifeguards that check whether each memory reference is to allocated memory (AddrCheck [17]), each read is to initialized memory (MemCheck [19]), and each memory location is accessed with a consistent locking policy—i.e., there is no data-race (LockSet [24]).

Given the high overheads for lifeguards that monitor nearly every program instruction, it is natural to consider whether these overheads can be significantly reduced by parallelizing the monitoring functionality. For certain lifeguards such as AddrCheck, the parallelization is straightforward because the monitoring work can be partitioned on a per-address basis and divided among the lifeguard processors. Program instructions that allocate or free blocks of memory may need to be broadcast to all the lifeguard processors, but other than these rare events, each lifeguard processor can independently check memory accesses that fall within its assigned partition. Similarly, LockSet, although slightly more complex, can be readily partitioned such that each lifeguard processor independently checks accesses that fall within its partition.

Far more challenging to parallelize, however, are lifeguards that track the flow of information through a monitored program's registers and memory locations. Such lifeguards, which are called *dynamic information flow tracking (DIFT)* lifeguards [8, 9, 20, 27, 30], are used to detect security exploits (e.g., TaintCheck [20]) as well as certain memory

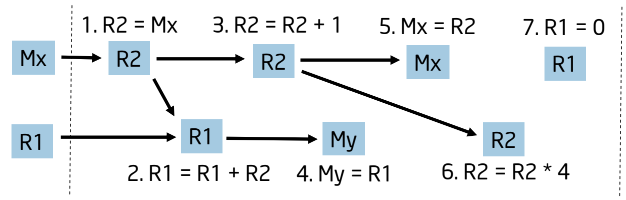

Figure 1: Example code segment, showing the information flow dependencies.

bugs (e.g., MemCheck, as explained in Section 2). In DIFT, certain types of instructions initiate a "taint" status on their destination (register or memory location) and other types of instructions are errors if they have a "tainted" source. Most instructions simply propagate the status: if the instruction has one or more tainted sources, then its destination becomes tainted. Because information flows between addresses, simple address-based partitioning (as described for AddrCheck) is impractical. To further understand the challenge in parallelizing DIFT lifeguards, consider the following example.

1.1 Challenge: Serial Dependencies in DIFT

Figure 1 depicts an example code segment (between the dotted lines) from a monitored program. There are seven instructions, shown as square nodes that are labeled by their destination. The edges between the nodes show the information flow (serial dependencies) between instructions: an edge from i to j exists if instruction j has a source that instruction i defines. Note that information flows to instructions 1 and 2 from two instructions occurring prior to this segment, one that writes `Mx` and one that writes `R1`, respectively. A sequential DIFT lifeguard would process this segment only after processing these prior instructions, and hence would know the status of `Mx` and `R1` before processing the segment. Thus, it could readily process these seven instructions according to the propagation rule. For example if `Mx` were tainted then `R2` would become tainted by instruction 1, and so on. In a parallel DIFT lifeguard, on the other hand, a lifeguard processor P assigned to this segment would seek to process this segment in parallel with preceding segments. In other words, P needs to process the segment without knowing the relevant incoming status.

Recently and independently, Nightingale et al. [21] proposed an approach to parallelizing DIFT on commodity chip multiprocessors (CMPs). Their approach is to have each lifeguard "worker" processor scan through its assigned segment and compress it in a manner akin to mark-and-sweep garbage collection, preserving only instructions that either affect the taint status of locations (addresses/registers) that are inputs to error checks or that determine the end-of-segment taint status of any location. Then a master processor processes the compressed logs sequentially. On their platform, running sequential taintcheck slows down a media player benchmark by 18X. By parallelizing TaintCheck across nine processor cores (a master and 8 workers), they were able to reduce this slowdown to 9X.

1.2 Solution: Symbolic Inheritance Tracking

In this paper, we present a new approach to parallelizing DIFT-like functionality. The key idea is that rather than always computing taint values explicitly after each instruction (as is done in sequential DIFT algorithms), parallel DIFT workers track the taint status *symbolically* whenever they are unable to determine the value explicitly.

The first step is to temporally segment the execution trace (as suggested by Figure 1) and to assign each segment to a separate "worker" processor. Next, to overcome the challenge of processing a segment without knowing the relevant incoming status, the lifeguard processor assigned to a segment computes the status *symbolically* by tracking the inheritance of the status: i.e. lifeguard processor P processes instruction `1` in Figure 1 by recording that `R2` *inherits* the status that `Mx` holds at the start of the segment, etc. The end result is an *inheritance table* for the segment, which summarizes the net propagation effect of the segment, as shown in Figure 2(a). At the end of the segment, `R1` is untainted (instruction `7` clears its taint value), `R2` inherits its taint value from the start-of-segment status of `Mx`, and so on. Thus, given the taint status at the start of the segment, we can compute the taint status at the end of the segment by plugging in the appropriate starting taints; we call this *resolving* an inheritance table (see Figure 2(b)). Resolving is done by a distinct lifeguard ("master") processor, concurrently with the "worker" processors who process subsequent segments. Because the typical program segment reuses destinations many times over (locality of reference), the inheritance table is often considerably smaller than the segment size, resulting in potentially large speed-ups.

Unfortunately, our experiments with up to six worker processors on SPEC benchmarks show that the parallel DIFT lifeguard is slower than the sequential DIFT lifeguard! A key bottleneck, not surprisingly, is the processing of binary nodes. Because each binary node depends on the taint status of two other nodes, which may in turn depend on the status of other binary nodes, a single destination may inherit its taint from (the `OR` of) a large number of nodes. Processing such nodes slows down parallel DIFT considerably.

This bottleneck arising from binary nodes led us to consider versions of DIFT that propagate only through unary nodes. In our recent work on accelerating *sequential* DIFT [4], we addressed the slowdowns incurred by binary nodes in sequential DIFT by proposing a hardware mechanism for *relaxed DIFT*, a variant of DIFT that tracks the information flow only through unary operations. As argued in [4] and in Section 3, such relaxed versions make sense from a lifeguard perspective (for both TaintCheck and MemCheck). In this paper, we show how removing the binary node bottleneck can be exploited for parallel DIFT. First, with relaxed DIFT and the inheritance approach, all destinations inherit from at most one source (as shown in Figure 2(c)), making the master processing fast. Second, any pair of inheritance tables for consecutive segments can be combined into a single inheritance table. Finally, all entries of the first unresolved inheritance table can be resolved in parallel. (See Section 3 for details.) As a result, our parallel relaxed DIFT achieves linear speed-up asymptotically.

While the parallel DIFT algorithm is asymptotically optimal, our implementation revealed that constant factors limited the performance improvement for modest numbers of processors. In particular, a small number of workers are unable to process the execution segments fast enough to keep the master busy. Consequently, we have the master lifeguard processor run sequential DIFT (with its small constant factors) and use parallel DIFT as an accelerator. In

destination	status
R1	untainted
R2	from Mx
Mx	from Mx
My	from Mx & R1

(a)

destination	status
R1	untainted
R2	tainted
Mx	tainted
My	tainted

(b)

destination	status
R1	untainted
R2	from Mx
Mx	from Mx
My	untainted

(c)

Figure 2: For the code segment in Figure 1, (a) the inheritance table for DIFT, (b) the resolved inheritance table when given that Mx is tainted and R1 is untainted at the start of the segment, and (c) the inheritance table for relaxed DIFT.

other words, segments of the monitored program's instructions processed using sequential DIFT are alternated with segments processed using parallel DIFT. By balancing the work rates, we achieve our best speed-ups for small numbers of processors.

1.3 Contributions

The main contributions of this paper are as follows. First, we present a new approach to parallelizing DIFT-like functionality, based on symbolic inheritance tracking. We show that while standard DIFT is challenging to parallelize, we can obtain linear speed-ups (asymptotically) for relaxed DIFT. Second, we present techniques for reducing the overheads in practice, both in cases with just a few processors and cases with more processors. Finally, we provide an extensive study of an implementation of the algorithm in the context of a Log-Based Architectures (LBA) system [3, 4]. LBA provides hardware support for logging a program trace and delivering it to other (monitoring) processors. Our simulation results on SPEC benchmarks and a video player show that, on a 16-core CMP, our parallel relaxed DIFT reduces the overhead to as low as 1.2X using 9 monitoring cores.

In the remainder of the paper, Section 2 provides background information and discusses related work. Section 3 presents our algorithms. Section 4 describes our implementation in LBA. Section 5 presents our experimental study, and Section 6 presents conclusions.

2. BACKGROUND AND RELATED WORK

2.1 Example DIFT Lifeguards

A Security Checking Lifeguard. TaintCheck [20] protects a vulnerable application from a common class of security exploits that corrupt the application's memory with malicious input data (e.g., via buffer overflows). By overwriting branch target addresses (e.g., return addresses on the stack, or the global offset table for calling shared libraries), a successful stack/heap overflow attack can gain control of the application by supplying a branch or jump instruction with a corrupted address. By overwriting a format string interpreted by printf-family functions, a format string attack can write arbitrary data to an arbitrary memory location. TaintCheck prevents such attacks by monitoring the propagation of input data in the application's memory. Input data to the application (e.g via `socket recv` or `read` system calls), are noted as suspect or *tainted* in a bitmap maintained by the TaintCheck lifeguard.

Subsequently, the propagation of tainted data through the application is carefully tracked using this bitmap, which contains one bit per byte of the application's address space as well as one bit per architectural register. When the application copies data from one memory location to another, the lifeguard copies the associated taint bits. For binary operations, the result destination is tainted if at least one of the sources is tainted. TaintCheck reports an error if tainted data are used in a critical way, such as in jump target addresses, format strings, or system call arguments; such uses are often good indicators of security exploits.

A Memory Checking Lifeguard. MemCheck [17, 18] is designed to detect memory violations, including accesses to unallocated memory regions, uses of uninitialized values, double free's, invalid free's, and memory leaks. To do so, the MemCheck lifeguard maintains two bits of metadata for every byte in the application's address space: one bit which indicates if the corresponding memory location has been allocated and one indicating if it has been initialized.

While maintenance of the allocated bits does not require information flow tracking (accesses between `malloc/free` events may be considered independently), maintenance of the initialized bits does—but for a subtle reason. Because programs frequently copy data structures that may have one or more uninitialized elements, reporting each read/copy of uninitialized values yields an unacceptably high number of error messages that the programmer would consider to be false positives. Consequently, MemCheck tracks the propagation of uninitialized values in much the same way that TaintCheck tracks taint values and defers the reporting of uninitialized memory accesses until a critical use, such as pointer dereferencing, conditional test determination, or system call argument evaluation. Therefore, MemCheck falls squarely into the category of DIFT lifeguards.

2.2 Decoupling Application Execution and Lifeguard Monitoring

The emergence of chip multiprocessors (CMPs) has inspired efforts to improve software monitoring performance by decoupling application execution from lifeguard execution [4, 21, 22, 25]. In these techniques, the application and lifeguard concurrently execute on separate cores. While the application runs, a log of its interesting execution events is captured and is subsequently consumed by the lifeguard to perform its checking functionality. We refer to these techniques as *log-based monitoring*. Because the storage space available for buffering the log is finite, if the lifeguard's rate of log consumption does not keep pace with the application's rate of production, the application must be repeatedly stalled waiting for the lifeguard. These stalls translate to significant slowdowns in the application execution time for the DIFT lifeguards considered in this paper.

One consequence of the decoupled nature of execution and checking is that the lifeguard is often performing checks on instructions the application executed thousands of cycles earlier, and hence will detect a violation long after it has occurred. Therefore log-based monitoring schemes rely on

OS level support for fault containment, from stalling the application at system calls until the lifeguard catches up [4] to checkpoint-based execution rollback [21].

2.3 Related Work

Newsome and Song [20] reported that the TaintCheck lifeguard slowed down monitored applications by over 30X. They implemented sequential TaintCheck using each of two state-of-the-art dynamic binary instrumentation tools, Valgrind [19] and DynamoRio [1]. In their experiments, the application and lifeguard run on commodity single core hardware. Qin et al. [23] proposed dynamic compilation techniques that reduce the slowdown of sequential TaintCheck to 3.6X for SPEC benchmarks. In light of the high overhead of software-only solutions, several recent studies [8, 9, 27, 30] proposed hardware support for DIFT by modifying the processor pipeline to automatically propagate taint status and/or by enhancing registers/caches/memories with taint status tags. None of these works considered parallel DIFT or relaxed DIFT.

Our previous work [3, 4] proposed Log-Based Architectures (LBA) that provide a set of hardware extensions to CMPs in order to support general-purpose log-based monitoring for a wide range of lifeguards, including TaintCheck, MemCheck, and a number of non-DIFT lifeguards. Our baseline LBA system achieves a factor of 3.4X slowdown for TaintCheck and a factor of 7.8X slowdown for MemCheck on SPEC benchmarks. We also proposed three hardware acceleration techniques beyond the baseline LBA, including a technique for relaxed DIFT that reduces the overheard of monitoring information flow through registers. (The other two techniques introduce redundant event filtering and new instructions for speeding up lifeguard metadata address translations.) Together, the three techniques reduce the slowdown to 1.4X for TaintCheck and 3.3X for MemCheck. While our previous work focused on sequential lifeguards, this paper focuses on parallelization as the means to improve lifeguard performance. Therefore, we use the baseline LBA (without the three acceleration techniques) as the baseline system for our performance study. The baseline LBA will be described in further detail in Section 4.1.

As mentioned in Section 1, Nightingale et al. [21] proposed a parallel log-based monitoring approach in which the log is partitioned into disjoint segments that are processed in parallel on commodity CMPs. Our approach differs by (i) its use of symbolic inheritance tables that collapse unary chains, (ii) its use of sequential DIFT when there are few lifeguard processors and a parallelized master when there are more lifeguard processors, and (iii) its consideration of relaxed DIFT and its study under a different baseline architecture, resulting in substantially different performance (2.1X vs. 9X on a media player benchmark). Costa et al. [6] consider a variant of relaxed DIFT that monitors only copy operations, but do not report any slowdown numbers. To our knowledge, there are no other papers that study parallelizing DIFT or relaxed DIFT.

3. PARALLEL INHERITANCE TRACKING

In this section, we present our parallel symbolic inheritance tracking algorithms for accelerating DIFT lifeguards. We call our first parallel algorithm `Original-DIFT` because it provides full information-propagation tracking. Then we discuss the rationale of relaxing full propagation tracking and present our second algorithm, called `Relaxed-DIFT`. We also show how to improve the performance when there are only a few workers. Finally, we show how we can obtain linear speedup (asymptotically) by parallelizing the master functionality.

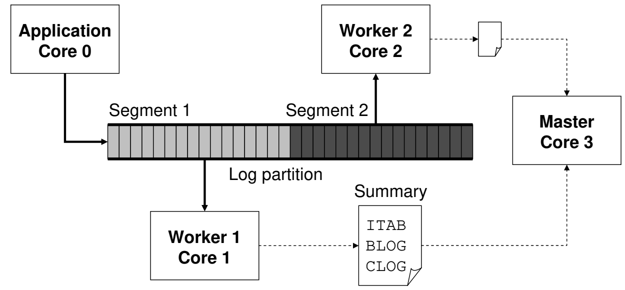

Figure 3: A parallel DIFT lifeguard with a master thread and two worker threads is monitoring an application on a quad-core log-based monitoring system. Every worker thread computes a summary from its assigned log segment. The master thread consumes the summaries and performs the actual checking.

3.1 Algorithm General Structure

Figure 3 depicts an overview picture of our parallel DIFT algorithms. Here, we assume an underlying log-based support (such as LBA) that extracts the execution trace of the monitored application into a log, which can then be consumed by lifeguard threads for monitoring purposes. We further assume that the lifeguard thread can selectively read portions of the log.[1] We consider a single application thread. For applications with multiple threads, each thread generates a log. We rely on the underlying log-based support to provide the needed coordination between logs (this support is beyond the scope of the paper).

The lifeguard is composed of a master thread and a number of worker threads (e.g., two workers in Figure 3), each running on a separate core. The log is conceptually divided into disjoint and equal-sized partitions. Each partition is further divided into k disjoint segments if there are k workers, where the ith worker is assigned the ith segment. All workers read their assigned segments in parallel, and generate a data structure summarizing the segment. After processing a log segment, a worker passes the generated summary to the master thread, waits until the next assigned log segment in the next log partition is available, and starts working on the next segment. The master thread combines the summaries according to the log order. It updates lifeguard metadata and performs the actual checking for each log segment.

3.2 Original-DIFT for General Propagation

For tracking general propagation, a lifeguard worker thread generates three data structures in the summary: (i) an inheritance table (`ITAB`); (ii) a log of binary operations (`BLOG`);

[1]If log records are compressed, they may not be individually accessible. In this paper, we will assume that the log is decomposed into chunks of reasonable sizes such that each chunk is compressed independently. This way, the log can be partitioned among the lifeguard threads at chunk granularity.

```
Initialize ITAB, BLOG, CLOG to empty;
Foreach record R in the log segment {
  if (R is checking event) {
    Foreach param of R {
     CLOG.append (ITAB.get (param));
    }
  }
  if (R is metadata setting event) {
    set affected locations in ITAB;
  }
  if (R propagates information) {
    retrieve dest, src[] from R;
    switch (number of sources of R) {
    case 1: /* unary operation */
      ITAB.set(dest, ITAB.get(src[0]));
      break;
    case 2: /* binary operation */
     if (Relaxed-DIFT) {
       ITAB.set(dest, CLEAR);
     } else { /* Original-DIFT */
        if (ITAB.isknown(src[0]) OR
           ITAB.isknown(src[1])) {
           either propagate the other source
        or the propagation result is known
        } else {
         binop_ID = BLOG.append(
         ITAB.get(src[0]),ITAB.get(src[1]));
         ITAB.set(dest, binop_ID);
        }
      }
      break;
    } /* switch */
  } /* if */
} /* for */
```

Figure 4: The DIFT worker algorithm.

and (iii) a log of checking operations (CLOG). Figure 4 describes the Original-DIFT worker algorithm.

At the start of the algorithm, all three structures are set to empty. A worker processes a log segment by consuming the log records of that segment in order. As log records are consumed, the worker updates the ITAB so that it reflects the state of all distinct destinations (register names or memory addresses) encountered up to that point in the log. Each element in the ITAB associates a unique destination with one of three types of state information: (i) a known value (e.g. tainted or untainted for TaintCheck, initialized or uninitialized for MemCheck); (ii) an address or register name, y, indicating that the destination's state is inherited from the state of y at the beginning of this log segment; or (iii) an ID identifying a BLOG entry, which means that the state is derived from the corresponding binary operation. ITAB supports three operations. `set(x,v)` sets the state of `x` to `v`. `get(x)` gets the state of `x`. If `x` is an immediate, then `get(x)` returns untainted for TaintCheck and initialized for MemCheck. If `x` is a register or address that does not exist in ITAB, `get(x)` returns `x` indicating that `x` has not been overwritten, and therefore, it retains its original state. A call to `isknown(x)` returns true if the state of `x` is one of the known values. The worker records binary operations in BLOG, and lifeguard-specific checking events (e.g., indirect jumps, printf-like calls, and system calls for TaintCheck) into CLOG. BLOG and CLOG are append-only structures. A call to `append(y)` adds `y` to the end of the structure and returns a unique identifier of the entry.

The worker is interested in three kinds of log record events: information flow, checking, and events that explicitly set

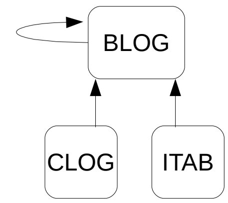

Figure 5: Dependency graph of BLOG, CLOG and ITAB.

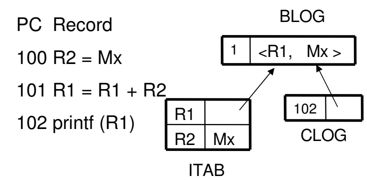

Figure 6: Simple code snippet and its summary.

state (such as `socket recv` system call in TaintCheck). To track information flow, the worker obtains the destination and sources from the log records. If there is one source, then the state of the source is propagated to the destination. If there are two sources, a BLOG entry is created and its ID is recorded as the state of the destination. However, if at least one of the two sources operands is in a known state, then a BLOG entry is not created, instead the propagation rule is used to derive the state of the destination. Although quite rare, it is possible to have more than two sources in a log record. Such cases are handled by combining the sources into a binary tree where a BLOG entry is created for each inner node. The ID of the entry corresponding to the root is assigned to the destination. The CLOG records the states of the parameters of checking events.

After processing a log segment, a worker sends the summary to the master. Because all three data structures may include references to particular entries in the BLOG, the master must be careful to respect the dependencies introduced by these references. That is, the result of a BLOG entry must be computed before any other entries which depend on it may be processed. The dependencies are depicted in Figures 5 and 6. Note that, although the BLOG structure introduces some dependencies, the ITAB and CLOG structures may be processed in parallel. In fact, the individual elements within those structures may also be processed in parallel.

Therefore given a summary, the master first processes BLOG in FIFO order[2] to compute the outcome state of every binary operation. Note that a BLOG entry contains states of the two sources of a binary operation, which are obtained using `ITAB.get()` in the worker algorithm. In other words, the states may be known values, may refer to starting states of other addresses, or may refer to *earlier* binary operations. In all three cases, the master can obtain the input states and compute the outcome state.

Next, the master processes the CLOG and ITAB. There are no dependencies among the entries in CLOG, therefore each entry can be processed in parallel. The master processes the ITAB structure in two logical steps. First, the final state value for each entry in the ITAB is computed using the segment starting states and the resolved BLOG entries. This step can be done in parallel for each entry since all the writes are to distinct locations in the ITAB. Second, the main logical

[2] FIFO order is not strictly required. Independent chains of BLOG entries could be processed in parallel.

bitmap (for the full address space) is updated with the final states for the destinations recorded in the ITAB. This step can also be done in parallel because all the writes are to distinct locations in the logical bitmap. At the end of this step, the logical bitmap reflects the correct DIFT state associated with the processing of all the records of the current segment.

3.3 Relaxed-DIFT for Unary Propagation

From the above discussion, we see that the BLOG processing presents a special challenge to fully parallelizing DIFT lifeguards. This challenge could be eliminated if, say, the outcome of binary operations could be determined at the time the workers handle binary operations. Such determinations could be made if either (A) the lifeguard considers binary operations to be checking operations or (B) binary operations are considered to be non-propagating operations. In both cases, the taint status of the destination is cleared as a result of the operation (as shown in the Relaxed-DIFT worker algorithm of Figure 4). The difference is that in case (A), the worker also inserts two checking operations into the CLOG (one for each source). We consider both cases to be relaxations of the lifeguard; applying case (B) weakens the lifeguard's checks, and applying case (A) strengthens the lifeguard's checks. The benefit of both relaxations is that the BLOG may be completely eliminated.

We suggest that case (A) is a reasonable relaxation for MemCheck. This relaxation suggests that, if the input to a binary operation is uninitialized, an error should be brought to the programmer's attention immediately rather than deferring the message until the result propagates to a checking operation. Consequently, this relaxation is more conservative in the sense that it catches a superset of the errors identified by the original and it alerts the programmer at least as early during execution as the original.

We also suggest that case (B) is a reasonable relaxation for TaintCheck despite the fact that it provides slightly weaker security guarantees than the original. First, how is it weaker? Relaxed TaintCheck could potentially fail to identify an attack if the malicious, triggering input is subjected to non-unary computation before being exercised by the critical operation that corresponds to the check that would have caught the attack in the original version. Note, however, that the attacker is unable to supply this non-unary computation; no attacker-supplied code executes until *after* the critical operation. Hence, such code must already exist in the monitored application (in addition to the overflow vulnerability).

Further, note that detecting such situations is challenging for potential attackers. Third-party analysts [28] often identify overwrite-based security vulnerabilities in proprietary software by causing a software crash through the introduction of a long input composed of a known pattern (e.g. repeating 0x55). A vulnerability is identified if the pattern is observed in expected locations in the core dump. This technique relies on a direct (unary) propagation of the input. To empirically evaluate these claims, we analyzed the first six months of CVE security alert entries in 2007 [28]. For the entries involving open source software, we studied the source code patches and found that every memory overwrite vulnerability was due to unary propagation. The security literature [7, 32] similarly reports that overwrite attacks (e.g. buffer overflow) rely almost exclusively on direct copying. Costa et al. [6], in fact, reported successful detection of the infamous Slammer, Blaster, and CodeRed Internet worms using a further-relaxed propagation model, *copy-only*, which tracks propagation solely through copy operations.

The bottom line, of course, is that relaxation presents a performance-security trade-off. As we shall see, the performance benefit may be significant, so the trade-off may be worthwhile—particularly if the overhead of running the original algorithm is so great that the user chooses to operate with no lifeguard. However, certain applications may be so sensitive that even small security compromises are unacceptable. Consequently, we present results for both versions of the algorithm in Section 5.2.

3.4 Achieving Speedups with Few Workers

When the number of workers is small, e.g., 1-4, our experimental study in Section 5 reveals that constant factors limited the speed-ups. In a nutshell, the common case for sequential DIFT takes 3 instructions per log entry (after careful tuning), whereas the common case for a worker in parallel DIFT takes 6 instructions (after careful tuning). To help overcome this, we divide each log partition into an initial portion that is processed using *sequential* DIFT and the remainder portion that is processed using parallel DIFT. In other words, the parallel DIFT is used as an accelerator for the sequential algorithm. The size of the initial portion is chosen to minimize the overall time. For example, with 1 worker, we allocate 2/3 of the partition to sequential DIFT and 1/3 to parallel DIFT. This variant makes sense only for small numbers of workers.

3.5 Parallelizing the Lifeguard Master for Linear Speedup

We have described the processing of the workers' inheritance tables as being performed by one "master" thread as shown in Figure 3. When there are more than a few workers, this processing by the master becomes the bottleneck. Fortunately, with relaxed DIFT, each entry in an inheritance table is a pair (destination, status), where the status is "tainted", "untainted" or "inherited" from a prior segment (as in Figure 2(c)). Thus, the entries of this table can be processed in parallel, segment by segment. A given segment will be ready for processing once all prior segments have been processed, because the metadata will be current up to the start of the segment. An entry in the current segment is processed by updating its status by a look-up to the metadata table. After all segments in the table have been resolved in this way, one final parallel step can revisit each entry in parallel and update the status in the metadata table. See Figure 7.

Theorem 3.1. *Consider a log partition of n instructions (RAM operations). Then the algorithm in Figure 7 runs in* $O(\frac{n}{p}+p)$ *time on a CREW PRAM.*

Proof. Both the metadata and inheritance tables can readily be implemented to support look-ups and updates in constant time (i.e., with a constant number of operations)–Section 4 describes one such implementation. Because each processor reads n/p log entries and updates its own inheritance table n/p times, step 3 takes $O(n/p)$ time. Each inheritance table T_j has $m_j \leq n/p$ entries (typically, $m_j \ll n/p$). In step 6, each processor resolves at most m_j/p entries, updating its own part of T_j. In step 7, each processor writes m_j/p entries. Because a given destination occurs at most

```
Parallel Relaxed Taint Analysis w/ Linear Speedup:

1. Foreach log partition L in turn {
2.   divide L into p equal-sized segments
       S_1,S_2,...,S_p
3.   each processor i (in parallel)
       processes log entries in S_i in order,
       creating an inheritance table T_i
4.   For j=1 to p {
5.      partition the entries in T_j into p
          equal-sized parts
6.      each processor i (in parallel)
          resolves the status of each entry
          (destination, from X) in its part
          by looking up the status of X in
          the metadata table
7.      each processor i (in parallel)
          updates the metadata table with the
          resolved status
8.   } /* for at step 4 */
9. } /* foreach at step 1 */
```

Figure 7: Parallel relaxed taint analysis with linear speed-up.

once in T_j, these updates are to exclusive metadata table entries. Considering all p iterations of the loop, the overall time is $O(\sum_{i=1}^{p} m_j/p) = O(n/p + p)$ on a CREW PRAM. $\square$

Thus, as long as n is $\Omega(p^2)$, the algorithm in Figure 7 achieves linear speed-up. Because we consider $p \leq 16$, it is easy to select log partitions such that $n \geq p^2$.

The above proof ignores a subtle issue that arises with how we represent metadata. Namely, as discussed in Section 4.4, each 32-bit word of the application's address space is represented with one 8-bit byte of metadata. Thus, technically the algorithm is exclusive write with respect to a PRAM with byte-sized memory cells. In practice, two processors could possibly race to write two bytes of the same word concurrently in step 7, but existing CMPs will handle this correctly, albeit with some performance hit.

Another potential source of parallelism is that any pair of inheritance tables T_j, T_{j+1} for consecutive segments can be combined into a single inheritance table: For each unresolved entry "(X, from Y)" in T_{j+1}, see if Y is a destination in T_j, and if so, copy Y's status into X's status. In addition, add to T_{j+1} any entry with a destination NOT in T_j, and then delete T_j. Such a step doubles the segment size for the inheritance table, and can be repeated $\log p$ times to obtain a single inheritance table for the entire table, where p is the number of initial segments (and hence the number of processors). With each such doubling step we also reassign the processors so that each new pair of segments (of double the previous size) is assigned all of the processors assigned initially to the component segments.

The above method is fast, but incurs work $O(n \log p)$. If we have the further assumption that the number of distinct destinations in a segment of m instructions is at most m^α for some constant $\alpha < 1$ (because of locality of reference), then this method enables linear speed-up for any $n \geq p \log p$, because the parallel time per step becomes geometrically decreasing.

Finally, we note that similar bounds are not likely for the original DIFT for the following reason. Once we have the entire log for a partition, we have a directed graph with its input values known (i.e., tainted or not tainted), and DIFT can be viewed as computing DAG reachability from the tainted inputs. Thus, the problem of constructing an efficient NC algorithm (i.e., a polylog time algorithm that is within a polylog factor of linear work) suffers from the well-known transitive closure bottleneck (see, e.g., [13]). With relaxed DIFT, in contrast, the graph is a forest of trees with the inputs at the roots. In theory, parallel tree contraction and related techniques [13] could be used for the propagation. However, in practice, such techniques suffer from the high costs of performing extensive pointer manipulations and contending for cache lines and locks.

4. IMPLEMENTATION OF A PARALLEL TAINT ANALYSIS LIFEGUARD

In this section, we apply our parallel DIFT algorithms to taint analysis. We describe the parallelization of the TaintCheck lifeguard on Log Based Architectures (LBA) [4], a platform that supports log-based monitoring.

4.1 Log Based Architectures (LBA): A Log-Based Monitoring Platform

We choose the baseline LBA [4] as our underlying log-based monitoring platform. LBA augments every processor core of a chip multiprocessor (CMP) with a log producer component and a log consumer component. To monitor an application, we enable the log producer component on the core running the application, and enable the log consumer component on all the cores running the lifeguard threads.

The log producer component captures an execution trace of the application and writes instruction records into a log buffer in the last level on-chip cache (e.g., L2 cache if the CMP has two levels of caches). (The log buffer takes 1/8 of the cache in our experiments in Section 5.) The log consumer component reads instruction records from the log buffer and delivers it to the lifeguard thread. The application (lifeguard) core stalls if the log buffer is full (empty, respectively). To support parallel DIFT lifeguards, we enhanced LBA to enable multiple log consumer components to read different portions of the log buffer.

LBA makes an important optimization at the consumer component: It supports hardware-based event-driven lifeguard execution. The goal is to remove the software loop that retrieves and interprets each log record (such a loop is depicted in Figure 4). Because the real work performed for a log record can be as little as a few instructions, the overhead of such a loop can be very significant in or even dominate the lifeguard execution time. Given this optimization, a lifeguard is implemented as a set of event handlers, which are registered with the LBA consumer component. The consumer component repeatedly retrieves the next log record, determines its event type, and calls the handler registered for this event type.

4.2 Synchronization Between Lifeguard Master and Workers

As described in Section 3.1, workers generate summaries of log segments, which are processed by the master. At lifeguard initialization, the master forks a number of workers as child processes. For each worker, the master allocates a

Table 1: Simulation Setup.

Simulator description		Simulation parameters	
Simulator	Virtutech Simics 3.0.22	Private L1I	16KB, 64B line, 2-way assoc, 1-cycle access lat.
Extensions	Log capture and dispatch	Private L1D	16KB, 64B line, 2-way assoc, 1-cycle access lat.
Processor core	in-order scalar	Shared L2	64B line, 8-way, 10-cycle access lat., 4 banks
Number of cores	8 or 16	L2 Size	2MB (for 8 cores) / 4MB (for 16 cores)
Cache simulation	g-cache module	Main Memory	200-cycle latency
Target OS	Fedora Core 5 for x86	Log buffer	1/8 of L2 size, assuming 1B per compressed record [4]

shared memory region called the *Summary Page Pool* (SPP) for the worker to generate its summaries. The SPP is divided into equal-sized pages. The pages are accessed (for writing by the worker, and for reading by the master) in a sequential order starting with the first page. Each worker process repeatedly processes a log segment and writes the summary into the next SPP page. Semaphores are used for synchronizing SPP page accesses.

4.3 Parallel Taint Analysis: Worker

Workers perform the algorithm in Figure 4, except that the software event processing loop is replaced by a set of event handlers. For example, each `if` test in the loop corresponds to several LBA-defined events. The different `switch` cases also correspond to separate LBA-defined events. Therefore, the event handlers roughly correspond to the `if/case` code segments. In the following, we mainly focus on the description of the three data structures that summarize a log segment.

ITAB. The inheritance table `ITAB` is conceptually an index structure that maps distinct addresses/registers to state information. The tracking is done at application byte granularity. The state records for registers are represented using a fixed-sized array. To reduce its memory footprint, the index for addresses is implemented as a two-level page-table-like structure, where the second level structure is allocated on demand if there is a destination address in the corresponding range. This structure maps every application word address to a 4-byte state record pointer, which is initialized to null. When an application word (or part of the word) is used as the destination of an operation for the first time, the worker allocates four contiguous state records, each per application byte, and sets the word's corresponding pointer to the starting address of the first state record. The two-level structure is accessed privately by every worker, while the state records are allocated in pages in the SPP. A state record is a tuple of `<DestAddr, Type, InheritFrom>`. `DestAddr` is the destination address. `Type` is one of TAINTED, UNTAINTED, ADDR (inherits from a memory address), REG (inherits from a register), and BINOP (inheritance information derived from a binary operation). For ADDR and REG, `InheritFrom` holds an address or a register name, respectively. For BINOP, `InheritFrom` holds a pointer to the corresponding `BLOG` entry.

BLOG. The binary log `BLOG` is allocated in the SPP. Each `BLOG` entry is a tuple of `<Src0-Type, Src0-InheritFrom, Src1-Type, Src1-InheritFrom>`, which represents the inheritance information of the two sources of the binary operation. This inheritance information has the same semantics as that of the `ITAB` state record.

CLOG. The checking log `CLOG` is also allocated in the SPP. `CLOG` records the state information ($<$`Type`, `InheritFrom`$>$) of parameters of checking operations (e.g., indirect jumps, system calls, printf-like calls). The program counter (PC) is also included in the entry so that the master, if it detects a security violation, can report the PC in its error messages.

4.4 Parallel Taint Analysis: Master

The master process follows the algorithm described in Section 3.2 for processing the summaries of log segments according to the log order. The taint tracking metadata structure in the master is implemented as a two level table that holds taint values of the memory bytes in the application's address space. For space efficiency, the second level structure is allocated only if the corresponding application addresses are used. Because 4 byte accesses are the most frequent, we optimize the metadata access for 4-byte application memory accesses as follows. We use 2 bits to represent the metadata for each byte, so that 4 bytes of application memory is represented by a meta data byte. Taint information for the architectural registers are stored in a small fixed-sized array.

Although the `CLOG` and `ITAB` could be processed in parallel, our current implementation does not exploit this opportunity.

4.5 Parallel Relaxed Taint Analysis

The implementation of the master and worker algorithms are similar to that of the original parallel taint analysis described above except for the handling of binary operations. The result of a binary operation is set to be UNTAINTED, as described in Figure 4. Moreover, there is no need for the `BLOG`.

5. PERFORMANCE EVALUATION

In this section, we evaluate our parallel DIFT lifeguard algorithms through simulations. We begin by describing the experimental setup.

5.1 Experimental Setup

Simulation Methodology. We evaluate our algorithms on the baseline Log-Based Architecture (LBA) systems, as described in [4]. We simulate LBA by extending the Virtutech Simics full-system simulator with log capture and dispatch components. The simulation parameters are shown in Table 1. We model chip-multiprocessor systems with private L1 caches and a shared L2 cache. Each processor core is an in-order single-issue scalar core. We use an 8-core configuration with 2MB L2 cache and a 16-core configuration with 4MB L2 cache. The log buffer for application instruction records is configured to be 1/8 of the entire L2 cache. LBA's compression mechanism can achieve about 0.8 bytes per instruction record. We conservatively assume 1 byte per log record in the simulation. The application and the lifeguard are running as two processes on a 32-bit Fedora Core operating system modified to recognize the association of lifeguards and applications. The application core stalls if

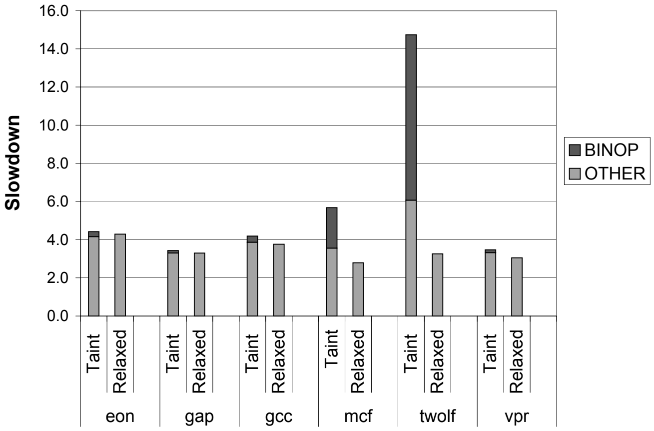

Figure 8: Application slowdown with `Relaxed-Hybrid` and `Original-Hybrid` monitoring (1 worker, 8-core configuration).

the log buffer is full, while the lifeguard cores stall if the log buffer is empty. The simulator models the detailed cache contention effects between lifeguard cores and application cores.

Lifeguard Designs. The lifeguard design has two orthogonal options: (i) original taint tracking (`Original`) vs. relaxed taint tracking (`Relaxed`); (ii) the lifeguard master only resolves inheritance tables produced by the workers (`Pure`) vs. it also processes part of the log (`Hybrid`). We implemented all four varieties of the lifeguard. We examine the first option in Section 5.2, and the second option in Section 5.3. We did not implement the techniques in Section 3.5 for parallelizing the master functionality.

Benchmarks. We choose CPU-intensive benchmarks, thus making it more difficult for lifeguards to keep up than with I/O-intensive applications. Our benchmarks include 10 of the SPEC2000 integer programs running the `test` input. We also studied the Mplayer version 1.0rc2 video player, with the trailer of the movie National Treasure as input. We simulated all benchmark runs to completion.

5.2 Relaxed vs. Original: Performance Benefits of Relaxing Taint Propagation

We begin our performance analysis by quantifying the benefits of a key feature of our algorithm: relaxing DIFT such that we need to track dependencies only through unary operations rather than binary operations. Figure 8 compares the performance of the `Relaxed-Hybrid` and `Original-Hybrid` algorithms, where the Y-axis shows the slowdown compared to the execution time of the benchmark without lifeguard monitoring. To err on the conservative side, we run our parallel algorithm with only a single worker thread, since this maximizes the partition size (thereby minimizing the expected number of unknown partition inputs per instruction). In Figure 8, the `Original` bars are each broken down into two components: (i) time spent processing the binary-operation log in the lifeguard master (`BINOP`), and (ii) the remainder of the time (`OTHER`). As we see in Figure 8, the `Relaxed` algorithm offers significant performance gains over the `Original` algorithm in several cases, and this improvement is due to eliminating the work associated with binary operations (`BINOP`). For the benchmarks where the `BINOP` time is small, we observe that most of the binary operations had at least one input in a known (`TAINTED/UNTAINTED`) state. For larger numbers of workers, we observe a similar (and sometimes even larger) performance gain from the `Relaxed` algorithm. Hence we focus on the `Relaxed` algorithm in the remainder of our evaluation.

5.3 Pure vs. Hybrid: Performance Benefits of Reducing Master Idle Time

Recall that in the `Pure` model, the lifeguard master thread only resolves inheritance tables produced by the workers, and in the `Hybrid` model, it also processes part of the log for the sake of not remaining idle. Intuitively, we expect the master thread to waste more of its time in an idle state when there are fewer worker threads.

Figure 9 compares the performance of `Relaxed-Pure` vs. `Relaxed-Hybrid` using eight of the SPEC benchmarks. The Y-axis shows the execution time of `Relaxed-Pure` normalized to that of `Relaxed-Hybrid` as we increase the number of worker threads from one to six. As we see in Figure 9, the `Relaxed-Hybrid` algorithm does indeed offer significant performance gains (up to 2.9X for gcc) relative to `Relaxed-Pure` when there is only a single worker thread. As more worker threads are added, the performance gap decreases; with six workers, the two schemes perform nearly the same because the master thread is saturated with processing inheritance tables from the workers. In summary, when the number of available cores is limited such that there are few worker threads, the `Relaxed-Hybrid` approach offers significant advantages; with six or more workers in our experiments, however, the `Relaxed-Pure` approach offers similar performance and it enjoys the advantage of being simpler to implement. Hence, we focus on `Relaxed-Pure` in the remainder of our evaluation.

5.4 Bottom Line Performance

Our final set of experiments studies the bottom line performance of our parallel lifeguard for relaxed taint propagation, measured two ways.

First, in Figure 10, we show the speed-up of our parallel lifeguard (`Relaxed-Pure`) over the sequential lifeguard (`Relaxed`) for relaxed taint propagation. We consider all ten benchmarks in our study. We used the 16-core configuration and evaluated `Relaxed-Pure` with 4, 6, and 8 workers. As shown in Figure 10, we achieve 1.2X–3.4X speedup with 8 workers.

Second, in Figure 11, we show the application slowdown resulting from running our parallel lifeguard (`Relaxed-Pure`). Recall that the application is stalled whenever the log buffer fills up, so the application's running time is dominated by the time for the lifeguard (even though the lifeguard is running in parallel with the application). We again consider all ten benchmarks under the 16-core configuration. For each benchmark, we report the lifeguard overhead, i.e., the running time with the lifeguard executing normalized to the running time with no lifeguard. As a point of comparison, the overhead for sequential relaxed taint propagation (`Relaxed`), averaged over our benchmarks, is 3.4X, the same as for sequential *full* taint propagation.

As shown in Figure 11, the lifeguard overhead of `Relaxed-Pure` using 8 workers is as low as 1.2X (vpr, crafty). However, some benchmarks experience significant slowdowns (up to 3.1X for parser). We observed that for these benchmarks with high overheads, the workers are unable to reduce signif-

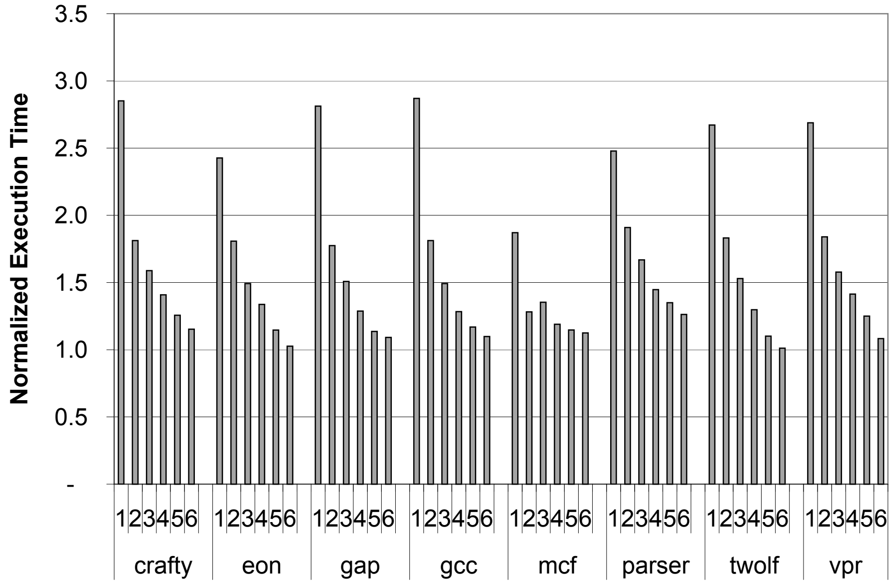

Figure 9: Execution time of `Relaxed-Pure` normalized to that of `Relaxed-Hybrid` (1 to 6 workers, 8-core configuration).

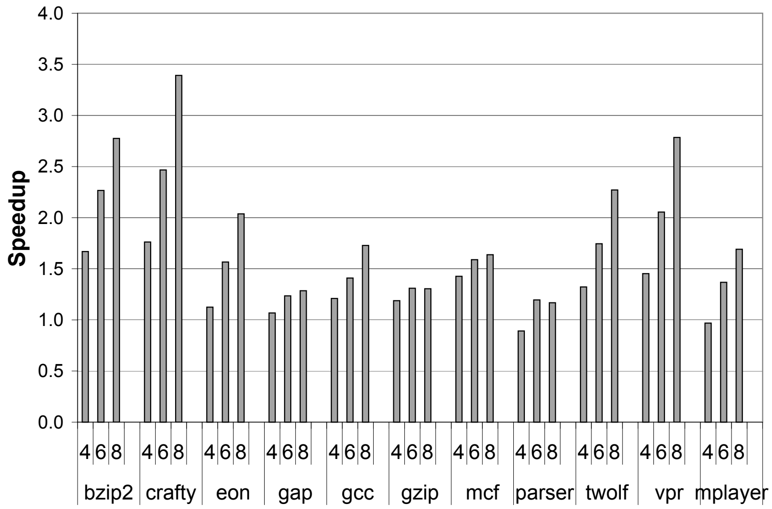

Figure 10: Speedup of `Relaxed-Pure` over sequential `Relaxed` on SPEC benchmarks while varying number of workers.

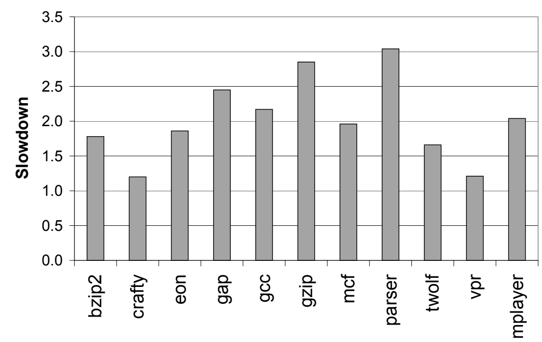

Figure 11: Application slowdown with `Relaxed-Pure` using 8 workers.

icantly the amount of work done by the master. For example the average number of ITAB entries per log segment for parser is 3X that of crafty. Therefore the master, which executes sequentially in our implementation, becomes the overall bottleneck. Parallelizing the master, as outlined in Section 3.5, would help mitigate this bottleneck. The slowdown for our non-SPEC benchmark, Mplayer, was 2.1X, in the middle of these two extremes.

While a direct comparison of our approach with Speck [21] cannot be made because of the differences in platforms, our study does include the one benchmark (Mplayer) used in the Speck study of propagation tracking. For this benchmark, we reduced the slowdown for Mplayer from 3.6X to 2.1X with 8 workers, while Speck reduced the slowdown from 18X to 9X with 8 workers.

6. CONCLUSIONS

In this paper, we have proposed and evaluated a novel form of parallel dynamic information flow tracking (DIFT) that (1) uses *inheritance tables* to track symbolically the net effects of segments that are processed in parallel by worker threads, and (2) *relaxes* DIFT such that information flow needs to be tracked only through unary operations. Our experimental results demonstrate speedups of 1.5X to 3.4X with 8 worker threads for seven of the ten SPEC benchmarks that we study, which is a positive result considering how difficult it is to speed up this "embarrassingly sequential" application. We observe that our relaxed DIFT offers significant performance advantages over the original DIFT in several cases (e.g., a factor 3X improvement for `twolf`) by reducing the work associated with tracking information flow through binary operations. We also observe that our hybrid algorithm offers significant performance advantages when the number of cores is limited, because it reduces idle time on the master thread. Perhaps most importantly, we have reduced the overall slowdown of DIFT to as low as 1.2X for some benchmarks, approaching our goal of making it practical to run DIFT continuously on deployed code.

Acknowledgements. This work was supported in part by a grant from the National Science Foundation. Ramachandran's research was supported in part by NSF grant CCF-0514876. We thank Jim Newsome for his TaintCheck code and discussions related to this work.

7. REFERENCES

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

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

[3] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *ASID Workshop at ASPLOS*, 2006.

[4] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *ISCA*, 2008.

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

[6] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of Internet worms. In *SOSP*, 2005.

[7] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security*, 1998.

[8] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *MICRO-37*, 2004.

[9] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *ISCA*, 2007.

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

[11] 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.

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

[13] R. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science*. Elsevier, 1990.

[14] 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.

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

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

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

[18] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *VEE*, 2007.

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

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

[21] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *ASPLOS*, 2008.

[22] H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in c programs. *Softw. Pract. Exper.*, 27(1):87–110, 1997.

[23] F. Qin, C.Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO-39*, 2006.

[24] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic race detector for multi-threaded programs. *ACM TOCS*, 15(4), 1997.

[25] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. Heapmon: A helper-thread approach to programmable, automatic, and low-overhead memory bug detection. *IBM J. on Research and Development*, 50(2/3), 2006.

[26] W. Shi, H.-H. S. Lee, L. Falk, and M. Ghosh. An integrated framework for dependable and revivable architectures using multicore processors. In *ISCA*, 2006.

[27] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ACM ASPLOS-XI*, 2004.

[28] The MITRE Corporation. Common vulnerabilities and exposures (cve). http://cve.mitre.org/.

[29] G.-R. Uh, R. Cohn, B. Yadavalli, R. Peri, and R. Ayyagari. Analyzing dynamic binary instrumentation overhead. In *WBIA Workshop at ASPLOS*, 2006.

[30] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *HPCA*, 2008.

[31] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. MemTracker: Efficient and programmable support for memory access monitoring and debugging. In *HPCA-13*, 2007.

[32] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *NDSS*, 2003.

[33] M. Xu, R. Bodik, and M. D. Hill. A 'Flight Data Recorder' for enabling full-system multiprocessor deterministic replay. In *ISCA*, 2003.

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

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

[36] Y. Zhou, P. Zhou, F. Qin, W. Liu, and J. Torrellas. Efficient and flexible architectural support for dynamic monitoring. *ACM TACO*, 2(1), 2005.