Decoupled Lifeguards: Enabling Path Optimizations for Dynamic Correctness Checking Tools

Olatunji Ruwase[1], Shimin Chen[2], Phillip B. Gibbons[2], Todd C. Mowry[1]

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

oor@cs.cmu.edu, {shimin.chen,phillip.b.gibbons}@intel.com, tcm@cs.cmu.edu

Abstract

Dynamic correctness checking tools (a.k.a. lifeguards) can detect a wide array of correctness issues, such as memory, security, and concurrency misbehavior, in unmodified executables at run time. However, lifeguards that are implemented using dynamic binary instrumentation (DBI) often slow down the monitored application by 10–50X, while proposals that replace DBI with hardware still see 3–8X slowdowns. The remaining overhead is the cost of performing the lifeguard analysis itself. In this paper, we explore compiler optimization techniques to reduce this overhead.

The lifeguard software is typically structured as a set of event-driven handlers, where the events are individual instructions in the monitored application's dynamic instruction stream. We propose to *decouple* the lifeguard checking code from the application that it is monitoring so that the lifeguard analysis can be invoked at the granularity of *hot paths* in the monitored application. In this way, we are able to find many more opportunities for eliminating redundant work in the lifeguard analysis, even starting with well-optimized applications and hand-tuned lifeguard handlers. Experimental results with two lifeguard frameworks—one DBI-based and one hardware-assisted—show significant reduction in monitoring overhead.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis; D.3.4 [*Programming Languages*]: Processors—Code generation, optimization

General Terms Design, Performance, Reliability, Security

Keywords Dynamic code optimizations, Dynamic correctness checking, Dynamic program analysis

1. Introduction

Dynamic correctness checking tools have become quite popular thanks to the availability of powerful dynamic binary instrumentation (DBI) frameworks such as Valgrind [21], Pin [17], and DynamoRio [3]. These dynamic tools (a.k.a. *lifeguards*) have the advantages that they do not require source code (because they start with binary executables as input) and they can observe the full dynamic behavior of the application as it executes. Lifeguards are complementary to tools that perform either static analysis [4, 10, 11] or post-mortem crash analysis [18, 36]. Particularly valuable are sophisticated lifeguards that invoke a lifeguard handler after nearly every instruction in the monitored application's dynamic instruction stream [21]. Such instruction-grain lifeguards are used to check a diverse set of correctness issues, including memory [22], security [24], and concurrency [31] misbehavior.

While instruction-grain lifeguards offer many compelling advantages, their major disadvantage is runtime overhead: lifeguards such as MEMCHECK [22] or TAINTCHECK slow down CPU-intensive benchmarks by 10–50X [21, 24]. Why are the overheads so large? One reason is that DBI itself imposes a significant overhead when it is performed at an instruction-by-instruction granularity. For example, the NULLGRIND "no instrumentation" lifeguard has a slowdown of roughly 4X on SPEC benchmarks [21], even though it performs no real work. To eliminate the binary instrumentation overhead, recent proposals such as DISE [8] and LBA [5, 6] propose hardware-assisted mechanisms for extracting the instruction-level information of the monitored application and feeding it to the lifeguard software as a stream of events. While these approaches significantly reduce the runtime overhead, there is still a slowdown of roughly 3–8X [5] for instruction-grain lifeguards, due to the cost of performing the lifeguard analysis itself. In a recent paper, Chen *et al.* [5] observe that redundancy often exists dynamically across lifeguard handlers (e.g., when accessing lifeguard state (called *metadata*), when performing redundant checks, and when performing unnecessary copying), and they propose adding hardware accelerators to help reduce these unnecessary overheads. In this paper, we explore an alternative approach, which is to recognize and eliminate this redundancy through *software*.

1.1 Key Optimization Stumbling Block: Performing Lifeguard Checks Synchronously

For decades, optimizing compilers have successfully improved software performance by recognizing and eliminating redundant computations along execution paths [1]. For traditional static analysis, a *control flow graph* is typically used to summarize the set of all possible execution paths; for more recent JIT-style dynamic code analysis, the optimizations are typically applied to an observed set of *hot paths*.

Our goal is to apply these types of redundancy-elimination optimizations across the lifeguard checking code. Unfortunately, a key structural property of most existing lifeguard frameworks makes it difficult to do this: their lifeguards perform correctness checks *synchronously* such that the checking for a given instruction (which is typically implemented as a call to an event-driven handler that handles certain classes of instructions) is completed before that instruction executes. On the one hand, it makes intuitive sense to check an application instruction before it executes, because this

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI '10 June 5–10, 2010, Toronto, Canada.

Copyright © 2010 ACM 978-1-4503-0019/10/06…$10.00

makes it straightforward to prevent bad things from happening. On the other hand, one implication of this synchronous approach is that the lifeguards behave much like interpreted code, where the dynamic instruction stream from the monitored application serves as the input. With this interpreter-like structure, there is little or no opportunity to optimize across lifeguard handlers to exploit redundancy caused by the structure and repeated patterns of the monitored application.

1.2 Our Approach: Decoupling Lifeguard Checks to Enable Path Optimizations

To enable more aggressive redundancy elimination of lifeguard checking code, we propose to *decouple* the lifeguard checking code from the application that it is monitoring. In contrast to traditional synchronous frameworks that invoke lifeguard analysis at the granularity of individual instructions in the monitored application (*instruction handlers*), our decoupled framework makes it possible to invoke the equivalent lifeguard analysis at the granularity of *hot paths* (potentially spanning large numbers of basic blocks) in the monitored application (*path handlers*). By exposing the lifeguard analysis associated with an entire hot path to our optimizer, we can find *many more opportunities for eliminating redundant work* in the lifeguard analysis.

In addition to exploiting the usual forms of redundancy elimination that are utilized by modern optimizing compilers, we also use *domain-specific knowledge about the lifeguard behaviors* to perform more aggressive optimizations. We incorporate our decoupled approach into two diverse lifeguard frameworks: one that uses DBI [21] and one that uses hardware-assisted logging [5]. Perhaps surprisingly, we show that, even starting with well-optimized applications and hand-tuned lifeguard handlers, our optimizations can find and eliminate significant redundancy in both frameworks, greatly reducing monitoring overhead.

1.3 How Decoupled Lifeguards Deal with Correctness Issues that Have Already Happened

While the benefit of our decoupled approach is that it exposes more opportunities for optimizing the lifeguard code, it also creates a potential complication. Namely, because the lifeguard checks are lagging behind the monitored application, by the time the lifeguard observes a correctness issue in the running application, the application has already continued executing beyond that point. To address this problem, a decoupled lifeguard framework must be able to (i) contain the damage to the application caused by such correctness issues, and (ii) protect the lifeguard state from corruption caused by such issues.

For requirement (i), we make sure that all remaining lifeguard checking codes are executed before certain critical events. For example, we ensure that all checks are complete before executing system calls. We also complete all checks before application indirect jumps to ensure that executed application code is well defined. Thus, the paths selected for path handlers do not cross application system calls or indirect jumps.

Requirement (ii) is trivially satisfied when a lifeguard and the monitored application are in separate address spaces, such as in LBA [5] and Speck [25], where corruption is not possible. However, when the lifeguard and the application share the same address space, we must protect three key memory components of this address space: the lifeguard code, any dynamically instrumented application code, and any metadata maintained by the lifeguard. The first two can be protected using page protection mechanisms, while the third is slightly more involved—see Section 2.2.

1.4 Related Work

Our approach builds upon a large body of previous work on optimizing interpreters [15, 28], partial evaluation [14], and dynamic code optimization [2, 7, 16, 35]. In contrast to this earlier work, our context is unusual because we are using the runtime behavior of one program (the monitored application) to optimize something else (the lifeguard).

Perhaps a more closely related topic is work on optimizing machine simulators [7, 35] by compiling sequences of simulator code to match hot paths in the simulated application. In both cases, an application path is the input to the optimization process; in our case, however, the lifeguard analysis is more closely connected with the structure of the monitored application than the work done in a machine simulator. Hence the lifeguard domain-specific optimizations that we explore are different from this earlier work.

Finally, there have been a number of proposals for accelerating lifeguard functionality [5, 9, 25, 29, 30, 32]. Raksha [9] and Hardgrind [32] accelerate monitoring by implementing most of the checking and propagation functionality of a particular lifeguard in hardware and handling exceptional cases in software. Our system eliminates redundant checks and propagation in software and is applicable to all lifeguards. Chen *et al.* [5] propose hardware mechanisms for accelerating the most frequent operations commonly performed by lifeguards. These include mechanisms for fast metadata lookup, redundant checks elimination and propagation inheritance tracking. Our optimizer similarly eliminates redundant lookups, checks and propagations, but without the need for such hardware accelerators. LIFT [29] observed that TAINTCHECK often performs redundant propagations because the sources and destination were typically untainted. It therefore skips propagation entirely within a program path if the live-in and live-out register/memory data of the path are untainted. In contrast, our optimizations do not rely on the runtime values of metadata for redundancy elimination and are therefore complementary to LIFT. While our redundant checks elimination within program paths is similar to LIFT's, our optimizations go further to eliminate checks across loop path iterations. Recent works accelerate lifeguards by parallelizing their monitoring task, either for sequential [25, 30] or parallel [13, 34] programs; our work is complementary to these efforts.

1.5 Contributions

This paper makes the following main contributions:

- To our knowledge, this is the first study to explore dynamic code optimization techniques for lifeguards.
- We propose to *decouple* the lifeguard checking code from the monitored application so that the lifeguard analysis can be invoked at the granularity of *hot paths* in the monitored application for more aggressive redundancy elimination.
- Beyond the usual redundancy elimination optimizations, we propose and evaluate lifeguard domain-specific optimizations that improve performance further.
- We evaluate our approach on a diverse set of instruction-grain lifeguards on two lifeguard platforms: a popular DBI platform, Valgrind, and a simulated hardware-assisted platform, LBA. For CPU-intensive benchmarks, we observe reductions in monitoring overhead of up to 31% on Valgrind and 53% on LBA.

2. Understanding Dynamic Correctness Checking

In this section, we first discuss several representative lifeguards. We then discuss frameworks for supporting lifeguards, and how decoupled lifeguards would fit in. Finally, we analyze lifeguards' common characteristics to point out optimization opportunities.

```
path1:
--------------------
mov %eax, [%ebx]
mov %edx, 0x8[%ebx]
add %edx, %ecx
mov 0x8[%ebx], %edx
jmp %eax
--------------------
```

(a)

```
----------------------------
mem_to_reg (eax, [%ebx])
mem_to_reg (edx, 0x8[%ebx])
add_reg_to_reg (edx, ecx)
reg_to_mem (0x8[%ebx], edx)
check_reg_indirect_jmp (eax)
----------------------------
```

(b)

Figure 1. (a) x86 code path of a monitored program (denoted path1), and (b) the corresponding invoked TAINTCHECK instruction handlers.

2.1 Representative Lifeguards

In our study, we focus on the following four instruction-grain lifeguards that represent a wide range of functionality:

ADDRCHECK [20] checks whether every application memory access is to an allocated memory area. In particular, for every application byte, it maintains a 1-bit "allocated" state as its metadata. The metadata are updated when ADDRCHECK observes memory allocation calls such as `malloc` and `free`.

TAINTCHECK [24] detects security exploits by monitoring suspect data in the application's address space. It maintains for every application byte a 1-bit "tainted" metadata, which is initialized to untainted. Unverified input data, such as those from network or from untrusted disk files, are marked as tainted. TAINTCHECK tracks the propagation of tainted data: For each executed application instruction, TAINTCHECK computes and updates the tainted state of the destination of the instruction by performing a logical OR operation on the tainted states of all the source operands. If tainted data are used in critical ways, such as in jump target addresses or printf-like calls' format strings, then TAINTCHECK flags a violation.

MEMCHECK [22, 23] enhances ADDRCHECK with protection against uninitialized values. Such protection is non-trivial because it is not an error to read an uninitialized value, e.g., when copying a partially initialized data structure. Instead, errors are raised only when uninitialized values are actually used improperly: e.g., dereferenced as pointers or passed into system calls. MEMCHECK maintains a 1-bit allocated state and a 1-bit initialized state for every application byte. The allocated state is updated and checked as in ADDRCHECK, while the initialized state is propagated like the tainted state in TAINTCHECK.

LOCKSET [31] monitors each application memory access to detect data races in parallel programs. For each shared memory location of the application, LOCKSET maintains the set of common locks held by different application threads when accessing the location. If the common lock set becomes empty, LOCKSET reports a potential data race. Since the total number of possible lock sets is typically much smaller than the number of memory locations, an optimization is to store the lock sets in a separate data structure and keep a pointer to the data structure as the per-location metadata.

2.2 Lifeguard Frameworks and Decoupled Lifeguards

As the monitored application executes, a sequence of application instruction events occur (conceptually). A lifeguard registers an event handler for every application event type that it cares about. For example, ADDRCHECK registers for memory read and write event types, while TAINTCHECK cares about almost every type of instruction. On x86, an instruction that performs multiple types of operations, such as memory access and computation, will be mapped to multiple event types. For each observed instruction event, the lifeguard framework invokes the registered lifeguard event handler with the dynamic event values (e.g., the effective address for a memory access) as handler arguments. Figure 1 shows an example application event sequence and the corresponding

```
void mem_to_reg(r,m) {
 taint(r) = taint(m);
}
```

(a)

```
void check_reg_indirect_jmp(r){
 if(taint(r)==tainted){
  error ("....");
 }
}
```

(b)

```
UChar taint (UINT32 addr) {
 map *mp = level1_index[addr >> 16];
  //  mov %ecx, %eax
  //  shr %ecx, $16
  //  mov %ecx, level1_index[,%ecx,4]
 int idx = (addr & 0xffff) >> 2;
  // and  %eax, 0xffff
  // shr  %eax,  $2
  return mp[idx];
  // movzbl  %eax, [%ecx, %eax, 1]
}
```

(c)

Figure 2. (a) TAINTCHECK propagation handler, (b) TAINTCHECK checking handler, and (c) an implementation of the taint() function for retrieving the taint status of a memory location.

TAINTCHECK handler calls. (In this paper, destination operands appear to the left of source operands.) Implementations of a propagation handler and a checking handler are shown in Figure 2. There are also special handlers for high-level events such as `malloc` and `free`; these are typically invoked via an instrumentation of the corresponding library call.

This event-driven model can be supported both in software and in hardware. The software-only approach is typically based on Dynamic Binary Instrumentation (DBI) [3, 17, 21], where executing application code is modified (instrumented) to insert lifeguard event handlers in between application instructions. Log Based Architectures (LBA) [5] is a state-of-the-art general-purpose hardware-assisted design, which runs a monitored application and a lifeguard on two separate cores in a multi-core system. Instruction records are extracted at the core running the application, transferred through a log buffer to the core running the lifeguard, and delivered in the event-driven fashion. In this design, lifeguard checking can lag the monitored application (by tens of thousands of instructions). This is a conscious design choice to enable hardware optimizations such as compressing sequences of log records prior to transfer. For correctness, LBA contains detected errors within the application's process boundary by stalling the application at system calls and waiting for the lifeguard to catch up and complete necessary checking.

This paper proposes to *decouple* the lifeguard checking code from the monitored application so that the lifeguard analysis can be invoked at the granularity of *hot paths* in the monitored application for more aggressive optimizations. Note that this differs from LBA because LBA does not provide hot-path capabilities and because we propose to use decoupled lifeguards even within DBI frameworks.

As discussed in Section 1.3, frameworks in which the lifeguard and the application share the same address space (e.g., DBI, DISE) raise additional challenges for correct execution of decoupled lifeguards. Namely, we must protect the lifeguard code, the instrumented code, and the lifeguard metadata from spurious application writes that may arise *before* the lagging lifeguard analysis detects the problem in the application. The lifeguard code can be easily protected using page protection mechanisms. Similarly, instrumented code pages can be protected after generation. Note that containment (requirement (i) of Section 1.3) ensures that corrupted code can never use system calls to remove page protections.

It is more challenging to protect the lifeguard metadata, which for instruction-grain lifeguards is primarily a one-to-one mapping from every memory location/byte in the application's address space to a lifeguard-specific *shadow value* [21] (examples above). This is

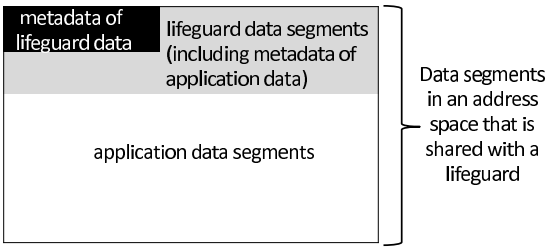

Figure 3. Metadata maintained by the lifeguard shadow all the data segments in the shared address space. Metadata either (i) shadow application data and are accessed by the lifeguard for correctness checking or (ii) shadow lifeguard data but should never be accessed because the lifeguard does not self-check.

because we must ensure that the lifeguard can update this metadata, but the application cannot. A simple, but costly, approach would be to insert a range check before every application write to ensure that it does not fall in the metadata range. A better-performing approach (see Figure 3) is to observe that, because a lifeguard maps the entire address space including the monitored application and the lifeguard to the metadata, the metadata must itself be mapped to a sub-range of the metadata. The lifeguard would normally never need to access this self-referenced range, so we can protect the range (e.g., via `mprotect`) with little overhead. If the application either directly accesses the self-referenced range or accesses anywhere else in the metadata causing the lifeguard to access the corresponding self-referenced range, an exception will be triggered. We can register a signal handler to detect and report such application misbehavior.

2.3 Lifeguard Optimization Opportunities

In this section we discuss lifeguard optimization opportunities when using decoupled lifeguards. The concern is that even decoupled lifeguards provide little opportunity for optimization because (i) lifeguard event handlers are already well optimized by existing compilers and are sometimes even hand-tuned; and (ii) the sequence of event handler calls corresponds to the sequence of application instructions in a well-optimized application, so that any redundant calls would seem to map back to redundant application instructions that would have been optimized away. Fortunately, despite these realities, there remains plenty of optimization opportunities, because of the following common properties of our representative lifeguards.

First, a lifeguard's behaviors are much simpler than the monitored application because an event type often corresponds to many different instructions. For example, ADDRCHECK and LOCKSET only care about memory accesses; they do not distinguish the computation operations (e.g., addition or multiplication using a memory location as a source operand are both regarded as a memory read). In TAINTCHECK and MEMCHECK, any computation in the application is converted into a logical OR of the source(s) metadata. Because of the many-to-one mapping of operations, even well-optimized application code sequences can result in sub-optimal lifeguard code sequences.

Second, metadata accesses are the most important operations in any lifeguard. Metadata in lifeguards are often constructed as a two-level data structure [23]. The first level is a pointer array, pointing to metadata chunks in the second level. The higher part of an application effective address is used to index the first level, while the lower part indexes the second level chunk. This organization saves space and is more flexible than allocating a monolithic metadata block for the entire application's virtual address space: metadata are allocated only when the corresponding virtual memory space is actually used by the application. Moreover, the monolithic approach may not be feasible for large metadata such as those in LOCKSET. As a result, any metadata access has to perform an indirect memory access with several shift and mask operations. For example, as shown in Figure 2(c) retrieving the taint status of memory locations in TAINTCHECK requires up to six x86 instructions. Therefore, reducing metadata accesses may significantly improve lifeguard performance.

Third, spatial locality of application data accesses results in spatial locality of metadata accesses. The common metadata design among the four lifeguards we study is that each second level metadata chunk shadows a 2^{16} byte range in the application address space, thus subsequent accesses to a metadata chunk can be done cheaply (avoiding the five instruction sequence in Figure 2(c)), by expressing the new location as an offset of a previously accessed location within the chunk.

Fourth, temporal locality of data accesses in the application code is mapped to temporal locality of metadata accesses in the lifeguard code. Given the many-to-one mapping of operations, it is possible that different or even dependent operations on the same memory location in the application are mapped to redundant lifeguard operations. For example, for a sequence of application instructions with multiple loads/stores to a given location without intervening memory (de)allocation calls, ADDRCHECK will perform a check per load/store. However, because the checks all read the same metadata, an optimization is to use only a single check for that location, removing all the other redundant checks. Performing such optimizations requires knowledge of both the monitored program's control flow and the lifeguard's checking and/or propagation rules. Unfortunately, existing optimizers are not aware of either as they see only the lifeguard code.

Finally, most lifeguards (including the four in our study) care only about the data flow pattern of the application in terms of the source and destination addresses. They do not care about the actual data values. Because handlers are insensitive to data values, they are simple (recall Figure 2(a)) and abundantly reused, making redundancies more likely.

The key to taking advantage of these opportunities is to bundle multiple handler calls together and optimize them as a unit, as enabled by our decoupled lifeguard approach, and described next.

3. Effective Optimization of Lifeguard Code

In this section, we present our solution for effectively optimizing lifeguard code by exploiting the observations in the preceding section. One constraint that we want to satisfy is to keep our solution generic so that it can be applied to a wide range of lifeguards. In the following discussions, we present our solution step by step.

Our optimizations rely on the following assumptions on lifeguard instruction event handlers. First, a lifeguard maintains unique metadata (i.e., shadow values) for each application register/memory location for the program properties that the lifeguard is monitoring. Second, the metadata are often organized using a two-level data structure as discussed in Section 2.3. Third, event handlers that correspond to application instruction events can access only the metadata associated with the handler arguments. These instruction event handlers are frequently executed and are the focus of our study. Fourth, the checks performed by these instruction event handlers are deterministic functions of the handler arguments and the metadata states. Two checks with the same handler arguments and the same associated metadata values are idempotent, giving the same outcome. Finally, the updates performed by the instruction event handlers are also deterministic given the handler arguments and the metadata states. A handler may read source metadata locations and write to destination metadata locations. If the handler arguments and the source metadata values are the same, then the destination metadata values will be the same. (An example of an

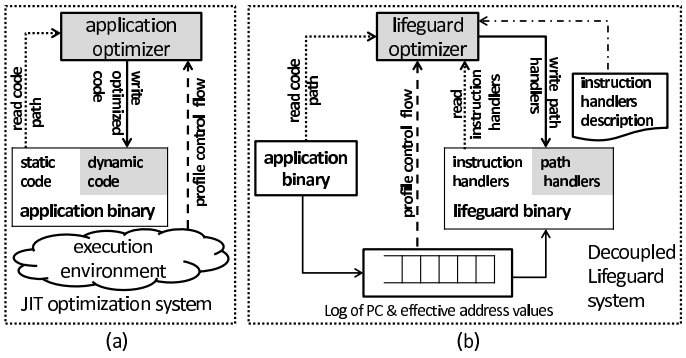

Figure 4. A high level view of how (a) a traditional JIT optimizer is used on application code, and (b) a JIT lifeguard optimizer would fit into a decoupled lifeguard system.

event handler *not* satisfying this assumption is a "profiling" handler that increments a counter each time it is called.) The above assumptions typically hold for the implementations of instruction-grain lifeguards [21], including the four in our study as described in Section 2.1.

3.1 JIT Optimization of Decoupled Lifeguards

We propose dynamic optimization for decoupled lifeguards. In Figure 4, we show how a JIT "lifeguard" optimizer fits into a decoupled lifeguard system in comparison to how traditional JIT optimizers are used. As shown in Figure 4(a), a traditional JIT optimizer profiles the execution environment (e.g., using performance counters, interpreters) to discover the frequently executed code paths of the application. The optimizer then reads a hot code path from the application binary, optimizes the hot code path, and dynamically *patches the application binary* with the optimized code for future execution. In contrast, as shown in Figure 4(b), the decoupled lifeguard optimizer obtains hot control flow profiles from a "log" that streams program counter values and effective addresses from the application to the lifeguard. It then reads the hot code path from the application binary, reads the relevant instruction event handlers from the lifeguard binary, and composes the appropriate sequence of event handlers into a *path handler*. (Example path handlers are shown in Figure 5.) The optimizer then applies the optimizations described later in this section to the path handler and *patches the lifeguard binary* with the optimized path handler, which is used for analyzing future executions of the hot code path. Note that the optimizer is designed to execute off the critical path of application-to-lifeguard communication and hence should have minimal adverse impact on application and lifeguard performance.

In this paper, a *path* is an acyclic sequence of dynamic instructions that can contain up to a predefined number (e.g., 8 in our experiments) of forward branches or terminates at the first backward branch, indirect jump, or system call. This not only simplifies path identification (each path is identified by its starting address and taken/not taken branch pattern), but also satisfies the containment requirement as discussed in Section 1.3.

Extending Lifeguard Frameworks to Support Path Handlers. With the decoupled lifeguard approach, there is an opportunity to look ahead in the log to see application events that have not yet been delivered to the lifeguard. Thus, a lifeguard framework can identify when an application path matches a previously determined hot path, and invoke the corresponding path handler. If a match is not found, the framework falls back to invoking an instruction handler for each event, until the next match.

For correctness, it is required that the lifeguard operations (i.e., metadata updates and checks) performed by a path handler must

```
void taintcheck_path1_hdlr (){
 mem_to_reg(eax, [%ebx])
 mem_to_reg(edx, 0x8[%ebx])
 add_reg_to_reg(edx, ecx)
 reg_to_mem(0x8[%ebx], edx)
 check_reg_indirect_jmp(eax)
}
```

(a)

```
void addrcheck_path1_hdlr (){
 check_allocated([%ebx])
 check_allocated(0x8[%ebx])

 check_allocated(0x8[%ebx])

}
```

(b)

Figure 5. TAINTCHECK and ADDRCHECK handlers for path1 in Figure 1(a), before optimizations.

be equivalent to the operations performed by the individual event handlers comprising the path handler. Given the way that the path handler is constructed, it is sufficient to satisfy that the same event arguments are supplied to the event handler calls inside the path handler as before. To achieve this, the lifeguard framework is extended to record in an array all event arguments since the start of the path. Because the number of event arguments is fixed for any event, the argument location of a particular event in a path will be found at a fixed offset from the array start. Therefore, we can supply an array reference with a constant index for any non-register event argument in a path handler. Register event arguments can be represented as small integer constants (i.e., register IDs). Communicating arguments to event handlers in this way enables the composing and inlining of non-trivial handlers, in contrast to copy-and-annotate [21] approaches such as in Pin, which work only for simple handlers.

Traditional Compiler Optimizations on Path Handlers are Suboptimal. After constructing a path handler, we could simply use a traditional compiler to optimize the path handler by inlining the event handler calls of the path handler. As shown in our experiments in Section 5 (the *path(stdopts)* bars), this approach indeed reduces lifeguard overhead in many cases, albeit only modestly. However, examining the generated lifeguard path handler code, we find that a lot of redundancy still exists in the code. But why?

3.2 Removing Redundant Lifeguard Operations within Individual Path Handlers

To understand why traditional compilers fail to remove many redundant lifeguard operations, let us look at the example path handler shown in Figure 5(b). We can see that the third check is clearly a duplicate of the second check. However, the two-level metadata structure hinders traditional compilers from recognizing this fact. Disambiguating the metadata manipulated by lifeguard event handlers is quite difficult because metadata are accessed using indirect memory references (as discussed in Section 2.3; see Figures 2(c) and 8(a) for examples), for which existing alias analysis techniques are quite ineffective. However, without such disambiguations, it would be impossible to verify that checks or propagations are redundant because potentially any metadata could be read or updated. Thus, traditional compilers have to conservatively retain the redundant lifeguard operations.

But why do we *intuitively* know that the third check is a duplicate? This is because we understand the mapping from application address to metadata address, and we understand the high-level semantics of the event handlers. `check_allocated` performs a metadata read, and the same application address is mapped to the same lifeguard metadata address, whose value may change only upon application memory (de)allocation events. However, this analysis may not apply to another lifeguard, e.g., TAINTCHECK. If two propagation handler calls share a common address argument, the determination of whether the second is a duplicate must distinguish between cases where the common address is used in a source or destination operand in each respective call. In other words, detecting redundant checks and propagations requires reasoning about the runtime effects of the event handler calls on meta-

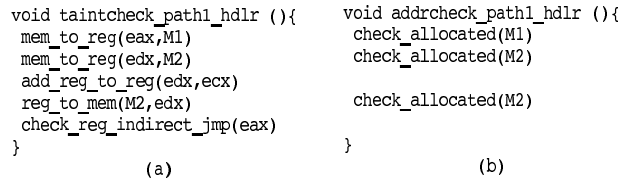

Figure 6. Applying our alias analysis to path1 handlers in Figure 5. Symbolic address expressions expose potential redundancies.

data. To do this analysis at compile time, there are two challenges: (i) understanding the effects of each lifeguard event handler, e.g., whether it performs propagations or checks; and (ii) disambiguating which metadata are manipulated.

We tackle the first challenge by using an "instruction handlers description" configuration file written by the lifeguard writer that describes how each handler manipulates (reads/writes) the metadata of its arguments, and whether the handler obeys the assumptions (idempotency, determinism) on common handlers that were listed at the beginning of Section 3. The configuration file also indicates the size of metadata values, e.g., ADDRCHECK maintains 1 bit for each application byte.

For the second challenge, we exploit the 1-1 mapping from application addresses to lifeguard metadata addresses. Rather than disambiguating metadata references in a path handler, we disambiguate the corresponding memory and register references in the corresponding hot code path in the monitored application. We expect reasonable success with this approach because (i) registers are trivial to disambiguate and (ii) memory references in the application code are often dominated by direct memory references, for which alias analysis is more effective. In this way, we convert the difficult task of disambiguating indirect memory references in the lifeguard code into a much easier task of disambiguating registers and direct memory references in the monitored application code.

We analyze the hot path in the monitored application code. By keeping track of expressions used for forming addresses (i.e., base register, index register, offset field, and scale field), we determine effective address arguments in the path handler that always-alias or may-alias with others. An address argument always-aliases with another address argument that is formed using the same expression, and it may-aliases with other address arguments. Note that, as in traditional JIT optimizers, our JIT lifeguard optimizer is performing alias analysis based on a static analysis of the hot path (and not the effective addresses in a dynamic instance of the path), so that the dynamically compiled path handler can be applied to any instance of the path. Figure 6 shows the outcome of applying our alias analysis to the path handlers in Figure 5; effective address arguments that are always-aliased are replaced with the same symbol. For example, the two address arguments formed using `0x8[%ebx]` are replaced with `M2` since they always resolve to the same effective address and are used to access the same metadata location at runtime. The analysis also notes that `M1` and `M2` are eight bytes apart and hence the corresponding metadata accesses can be optimized using the technique described in Section 3.4.

Detecting redundant checks and propagations is much easier in this representation. For example, given the above configuration file, it is easy to determine that the check performed by the third handler call in Figure 6(b) is redundant to that performed by the second call.

In addition to removing redundant events, we can further leverage the descriptions in the configuration file. For propagation-style lifeguards, such as TAINTCHECK and MEMCHECK, a propagation event handler performs a logical OR of the source operands' metadata. There are frequent opportunities to short circuit this operation to improve lifeguard performance. For example, in TAINTCHECK, a destination is tainted if at least one of the sources is tainted regardless of the status of the other source(s). In contrast, short circuiting opportunities are far less common in more general programs, where operands (e.g., integers) have significantly larger value ranges and participate in the full set of arithmetic/logical operations.

3.3 Exploiting Knowledge Beyond Individual Paths for Further Optimizations

The optimizations described above are fundamentally limited by the path boundaries. Here, we extend our optimizations to consider the context of individual paths within enclosing loops.

We observe that paths inside an application loop translate into event sequences that get repeated each loop iteration and consequently into repeated invocations of the same lifeguard path handler. Although the original application loop code is generated by traditional compilers that already perform loop optimizations (e.g., loop-invariant code motion), the resulting lifeguard operations often still have many redundancies across loop iterations because of the many-to-one mapping of application operations to lifeguard operations. To exploit this observation, we require the underlying lifeguard framework to remember the path delivered prior to the current path, as well as to support looking ahead in the application event sequence for one more path beyond the current path. This is a reasonable requirement. For example, LBA uses a log buffer that can contain tens of thousands of instruction events. In this way, a given instance of a loop path can be identified as the first, the last, or some middle iteration of the loop path.

We optimize loop path handlers by eliminating loop redundancies in a manner similar to that in traditional compilers. We analyze a loop path handler to detect lifeguard operations that perform loop-invariant checks/propagations and loop-dead propagations. It is important to note that because we are dealing with paths, the propagation/check only has to be invariant on iterations of that particular loop path, and might be variant on other paths in the loop [12]. Similarly, loop-dead propagations are propagations that are only live on the exit of the loop path. During the monitoring run, the path handler invokes loop-invariant handlers only on the first iteration and loop-dead handlers only on the last iteration.

Figure 7 shows an example of eliminating redundancies in loop path handlers. The path in Figure 7(a) is a hot loop path from 181.mcf, a SPEC2000 benchmark. As shown in Figure 7(b), the TAINTCHECK path handler contains 12 instruction handlers before metadata disambiguation. We focus on Figure 7(c), which shows the handler after all the optimizations in Section 3.2. The first event handler call, which merges the taint status of memory `M1` into that of register `esi`, is loop-invariant, because both of their taint metadata are read-only otherwise in the path. Handler call (7), which propagates the status of `M2` into `ebx`, is loop-dead, because the status of `ebx` is live only at the loop exit. Figure 7(d) shows the actual generated path handlers. In this case, our optimizations successfully detect that all the handlers are either loop-invariant or loop-dead, thus eliminating the need for a loop body handler!

3.4 Exploiting Spatial Locality for Cheap Metadata Access

After eliminating redundant event handler calls, the remaining event handler calls are inlined to enable a cheap metadata access optimization described next. As discussed in Section 2.3, a common metadata design for lifeguards is to shadow each 2^{16} byte aligned region in the application address space with a second metadata level chunk. This implies that repeated data accesses in this byte range result in accesses to different locations of this metadata chunk. If the optimizer could identify accesses to the same metadata chunk, it could avoid the five instructions required for computing a metadata address for all but the first access and perform the remaining accesses as offsets of the first one. However, because this is as difficult as directly disambiguating metadata accesses, our

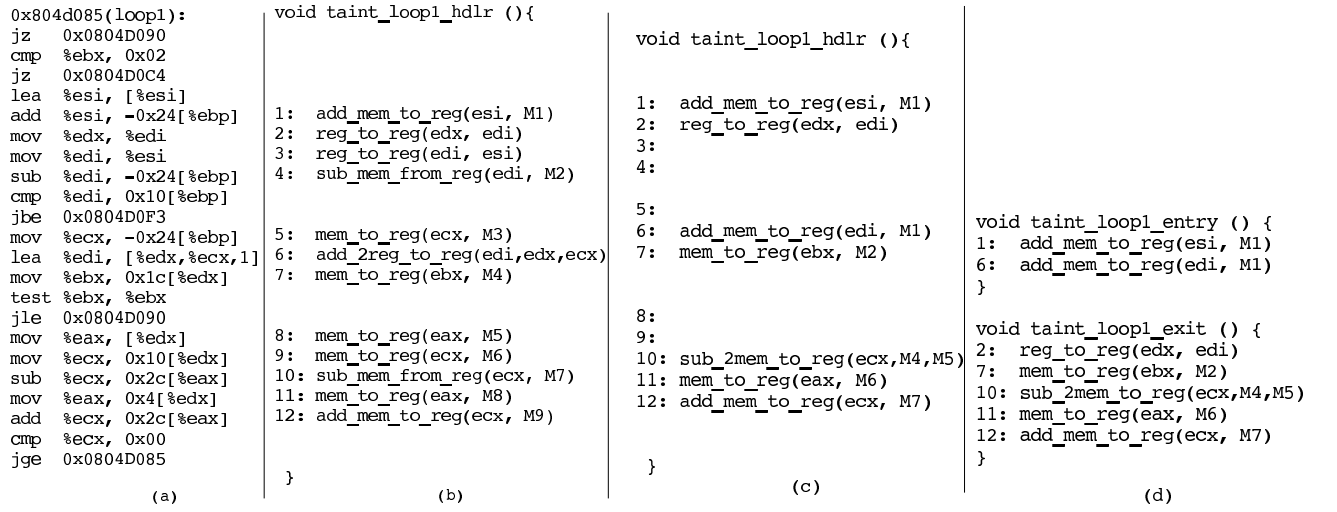

Figure 7. (a) A hot loop path (denoted loop1) from the 181.mcf benchmark, (b) the TAINTCHECK path handler after alias analysis but before metadata disambiguation, which translates to 81 x86 instructions after inlining, (c) the path handler after metadata disambiguation and intra-path redundancy elimination, which results in 54 x86 instructions after inlining, and (d) the entry and exit path handlers containing the loop invariant and loop-dead handler calls, which results in 12 and 43 x86 instructions, respectively, after inlining. Because all the handler calls in (c) are in either the entry or exit path handlers, the path handler for the body of the loop is empty.

optimizer instead identifies application memory references in the code path that are likely to be in the same 2^{16} byte address range. It employs the heuristic that memory references that are formed using a "base register + offset" addressing mode and that differ only in the offset are often shadowed by the same metadata chunk. Our evaluations confirm this to be a highly accurate heuristic.

Having identified a set of memory references that are likely to be shadowed by the same metadata chunk, the optimizer derives the relative distance of the corresponding metadata in the chunk using the size of metadata values and optimizes the path handler as shown in Figure 8. Figure 8(a) shows an excerpt of the TAINTCHECK path handler from Figure 6(a) after inlining of event handlers, showing the accesses to taint values of two memory locations `M1` and `M2`. Because both memory references are formed in the application hot path using addressing modes targeted by our heuristic, the second taint read is performed using the address of the first one, as shown in Figure 8(b). Here, the second taint value is a 1 byte offset from the first because `M1` and `M2` are 8 bytes apart and a metadata value is 1 bit for every application byte. This optimized sequence is executed only after a three instruction runtime check is used to determine when the metadata accesses are indeed to the same metadata chunk and hence the optimization is safe. If the check fails, the unoptimized sequence in Figure 8(a) is executed as a fall back. For applications with good spatial locality, the overheads of this optimization are amortized across multiple metadata accesses that fall within the same chunk, as shown by our experiments.

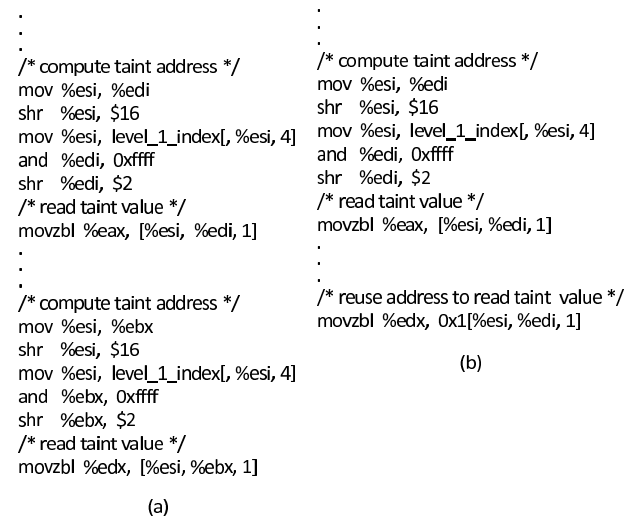

Figure 8. (a) An excerpt from the TAINTCHECK path handler from Figure 6(a) after inlining of event handlers, showing the accesses to taint values of two memory locations `M1` (passed to the path handler in `edi`) and `M2` (passed to the path handler in `ebx`) that are 8 bytes apart. (b) The path handler after our metadata address computation optimization: the second taint access is performed as an offset of the first one, because the memory locations are shadowed by the same metadata chunk.

3.5 Summary of Optimizations

In summary, we optimize decoupled lifeguard code by (i) automatically constructing path handlers from lifeguard event handlers, (ii) performing alias analysis on every hot path of the monitored application code for disambiguating the metadata manipulated by event handlers in the corresponding path handler, (iii) eliminating redundant event handler calls in the context of individual paths within enclosing loops, including loop-invariant and loop-dead handler optimizations, and finally (iv) eliminating expensive metadata address computations by exploiting the spatial locality of metadata accesses. Experimental evaluation in Section 5 shows that it is suboptimal to employ only traditional compiler optimizations (including inlining) after step (i), and that our new, lifeguard domain-specific optimizations, i.e., steps (ii)–(iv), lead to significant benefits beyond traditional optimizations.

4. Implementation

We implemented our proposed lifeguard path optimizer in two frameworks: Valgrind and LBA. We describe the two implementations in Section 4.1 and 4.2, respectively.

4.1 Extending Valgrind for Lifeguard Path Optimizations

Valgrind [21] is a state-of-the-art dynamic binary instrumentation framework. Given an application executable, Valgrind disassembles up to three branches from the application x86 code at a time into the Valgrind intermediate representation (IR). Then, it inserts the relevant lifeguard code before the associated application instructions in the IR. After that, it optimizes and converts the IR back to x86 code, caches the code in a hash-indexed code cache, then executes the instrumented code. This instrumentation process is performed only when code to execute is not found in the code cache. The overhead is further reduced by recording the starting addresses of the most frequently used codes in a small array for fast lookup and dispatch. Valgrind directly manages shadow registers. The IR optimizations eliminate redundant checks and propagation among shadow registers. Therefore, we mainly focus on reducing redundant memory events for Valgrind lifeguards.

Starting from Valgrind-3.4.0, we implemented decoupled lifeguards and path optimizations as follows. First, we extend Valgrind to disassemble up to eight branches from the application code at a time in order to form a path. Second, unlike the original Valgrind, we insert a lifeguard path handler only at the end of the IR of a path. Third, we instrument the application code path to generate a log of the effective addresses of the memory operations. This log is consumed by the lifeguard path handler. We do not log program counter values because Valgrind already keeps track of them. Fourth, we reduce the logging overhead by (i) logging one address for each set of aliasing memory references, and (ii) logging loop-invariant memory addresses only in the first loop iteration. Finally, we perform path-based lifeguard handler optimizations and replace the original Valgrind instrumentation only when the estimated benefit of the optimizations (i.e., the number of eliminated handler calls) outweighs the logging overhead.

Our current Valgrind extension is limited in three ways. First, for every path starting address, it can optimize only a single hot path, which reduces the coverage of the optimizations because multiple hot paths (such as in a hot loop) may share the same starting address. Second, there is no mechanism for detecting the last iteration of a self-loop. This prevents loop-dead handler elimination as described in Section 3.3. Third, memory event handlers are not inlined, thus our metadata address optimization cannot be performed.

4.2 Extending LBA for Lifeguard Path Optimizations

As described in Section 2.2, LBA is a state-of-the-art design for a hardware-assisted lifeguard framework. It exploits multi-core processors to run a monitored program and its lifeguard on separate cores. A log buffer is maintained in the last level on-chip cache for transferring event records from the application core to the core running the lifeguard. At the lifeguard core, a hardware dispatch mechanism efficiently supports event-driven lifeguard execution.

We extended the baseline LBA simulator with hardware mechanisms for (i) detecting paths in the log record sequence that match registered path handlers; (ii) logging effective address arguments from the instruction records into a dedicated hardware table for communicating to the path handlers; (iii) dispatching path handlers; and (iv) remembering the previous path and looking ahead in the log to identify the next path for supporting our optimizations for loop path handlers. If there is no matching path or the log buffer is not full enough to form a path, we fall back to the baseline LBA approach of using instruction handlers to consume the log record sequence. However, note that the log buffer is usually full because the application is typically faster than the lifeguard. With these mechanisms, we implemented all the path optimizations described in Section 3.

Table 1. Multithreaded Benchmarks for LOCKSET.

Benchmark	Description and Input
blast v2.2.16 [19]	Searching a nucleotide and protein database of 134K sequences
pbzip2 v1.0.1 [26]	Parallel data compressor, compress half of CPU2000's ref input.source
pbunzip2 v1.0.1 [26]	Decompress pbzip2's output in parallel
zchaff 2002.7.15 [27]	SAT (Boolean Satisfiability Problem) solver, circuit fault analysis

5. Performance Evaluation

We begin by presenting the experimental methodology in Section 5.1. We study the effectiveness of our solution in reducing redundant events in Section 5.2. Then, we study the impact of our techniques on lifeguard performance for both our Valgrind (Section 5.3) and LBA (Section 5.4) implementations.

5.1 Experimental Setup

Lifeguards and Benchmarks. Our evaluation uses the four instruction-grain lifeguards presented in Section 2.1: ADDRCHECK, TAINTCHECK, MEMCHECK, and LOCKSET. We implemented three versions of our lifeguard optimizer by gradually applying our proposed optimizations, for the purpose of quantifying the incremental benefits of our techniques:

(i) *path(stdopts)*: applying standard compiler optimizations on path handlers in the decoupled lifeguards (Section 3.1);

(ii) *path+lgopts*: in addition to (i), applying domain knowledge to reduce redundant lifeguard handler calls (Sections 3.2, 3.3);

(iii) *path+lgopts+maddropts*: in addition to (i) and (ii), optimizing metadata accesses (Section 3.4).

Our evaluation focuses on CPU-intensive applications, which are known to incur the largest lifeguard overheads, as opposed to I/O-intensive applications. ADDRCHECK, MEMCHECK and TAINTCHECK all monitor single-threaded applications. We use ten SPEC2000 integer benchmarks for evaluating them. The benchmarks use reference inputs for our augmented Valgrind framework running on a real machine, and use test inputs for our augmented LBA running on a simulator. LOCKSET is a data race detector. Therefore, we evaluate it using four multi-threaded applications, as shown in Table 1, running on a single core. All the experiments are run to completion. We report execution time normalized to a "baseline" execution that runs on our augmented frameworks but without any path handlers. We observe that this baseline execution performs comparably to the original Valgrind and LBA, incurring 4–38X slowdowns for Valgrind and 1.3–13.3X slowdowns for LBA, when compared to benchmark execution without any lifeguard monitoring.

Decoupled-Valgrind on a Real Machine. We extend Valgrind as described in Section 4.1, and evaluate our optimization techniques using two lifeguards, ADDRCHECK and MEMCHECK, that are available on Valgrind. We run the experiments on an x86-64 machine with dual 2.33GHz quad-core Intel Xeon E5345 CPUs, 8MB L2 cache, and 16GB RAM, running the unmodified 64-bit Fedora Core 5 with Linux 2.6.19 kernel. gcc-3.2.3 is used to compile Valgrind and the lifeguards. The default compilation settings in Valgrind are used. Denote this set-up as *Decoupled-Valgrind*.

Decoupled-LBA Simulation Platform. We extend LBA [5] as described in Section 4.2. The LBA hardware is simulated on the Virtutech Simics [33] full-system simulator. We use the same simulation parameters as in [5], as shown in Table 2. The monitored application and the lifeguard are running as two processes on two separate cores. The simulated 32-bit Fedora Core 5 operating sys-

Table 2. Simulation Setup for LBA.

Simulator description	
Simulator	Virtutech Simics 3.0.22
Extensions	Log capture and dispatch
Processor cores	Two in-order scalar cores
Cache model	g-cache module
Target OS	Fedora Core 5 for x86
Simulation parameters	
Private L1I	16KB, 64B line, 2-way, 1-cycle access lat.
Private L1D	16KB, 64B line, 2-way, 1-cycle access lat.
Shared L2	512KB, 64B line, 8-way, 10-cycle access lat. 4 banks
Main Memory	200-cycle latency
Log buffer	1/8 of L2 size, assuming 1B per compressed record [5]

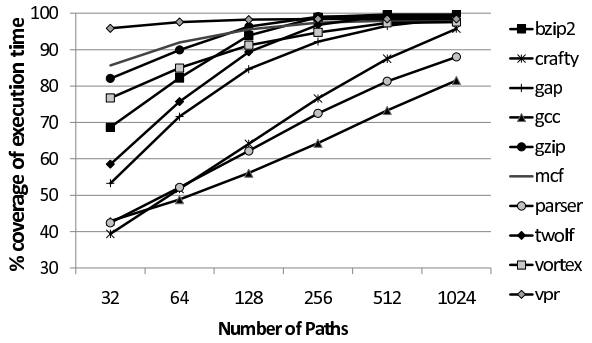

Figure 9. Path coverage for SPEC2000 benchmarks with ref input.

tem is modified to recognize the association between the lifeguard process and the application process. The application core stalls if the log buffer is full, while the lifeguard core stalls if the log buffer is empty. The detailed cache contention effects between the application and the lifeguard is modeled by the simulator. Lifeguard binaries are generated using gcc-3.4.6. Denote this set-up as *Decoupled-LBA*.

Hot Path Detection and Path Handler Generation. To simplify the prototyping effort, we use offline profiling to detect hot paths of up to 8 branches for both Decoupled-Valgrind and Decoupled-LBA. Moreover, we generate the path handlers offline for Decoupled-LBA. However, we ensure that when running the path handlers, the implementations mimic a JIT-based approach by limiting the total number of hot path handlers over an entire application to be at most 128 for any one lifeguard. We believe that the offline simplification is reasonable because hot path detection and optimization overheads have been shown to be small [2, 17].

Figure 9 shows the cumulative coverage of paths for the SPEC2000 benchmarks in our study. We see that 128 (256) hot paths cover over 85% (90%) of the execution times in 7 out of the 10 benchmarks. Therefore, we generate path handlers as follows. For each benchmark, the path optimizer estimates and ranks the optimization benefits for the 256 hottest paths. Then it generates path handlers for the 128 paths with the most benefit. The handler generation is performed offline for Decoupled-LBA, while it is performed as part of the initialization step in Decoupled-Valgrind, loading the path handlers into the code cache.

5.2 Effectiveness in Reducing Redundant Events

We start by evaluating how aliases affect the effectiveness in eliminating redundant checks. As discussed in Section 3.2, disambiguation of the metadata that are manipulated by the lifeguard event handlers in a path handler plays an essential role. Rather than directly disambiguating the indirect memory accesses to metadata, our solution *statically* analyzes the application path to disambiguate the application's direct memory accesses and then leverages the one-to-one mapping from data to metadata to disambiguate the metadata accesses.

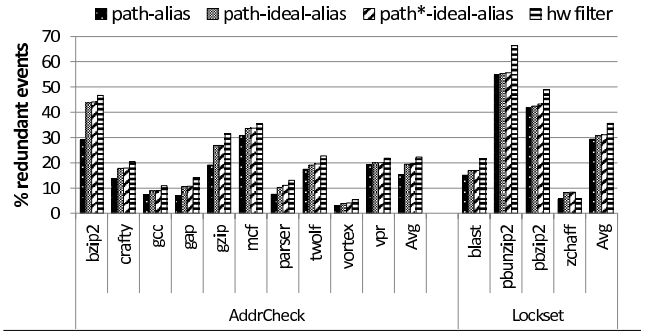

Figure 10. Impact of aliasing on redundant check detection for ADDRCHECK and LOCKSET on Decoupled-LBA.

Figure 10 compares our static alias analysis solution against more idealized techniques representing upper-bounds that can be achieved, on Decoupled-LBA. We consider three such idealized schemes, which use different amounts of dynamic information for disambiguating metadata accesses. The figure shows, for ADDRCHECK and LOCKSET, the percentage of dynamic lifeguard events in each benchmark that are identified as redundant by the different schemes. Our scheme, *path-alias*, represents the results of analyzing the path handlers corresponding to the selected hot paths of the monitored programs. We estimate the number of detected dynamic redundant events by computing the sum of the number of redundant events detected per path handler multiplied by the execution frequency of the paths. *path-ideal-alias* is similar to *path-alias* except that it is enhanced with dynamic runtime information about metadata accesses that always alias, even though they could not be determined statically by our algorithm. *path*-ideal-alias* further extends *path-ideal-alias* to detect aliases that span 2, 4 and 8 iterations of loop paths and the best results are reported. *hw filter* is similar to the proposal in [5], which uses a hardware filtering mechanism to avoid redundant metadata checks. This represents the most ideal setting because all aliases are resolved at runtime. We achieve a tighter upper bound by enabling the hardware filter only in the hot paths that were considered for static analysis. Infinite filter size is used to avoid overflows. The hardware filter states are preserved across loop path iterations but are flushed at the beginning of other new paths.

Compared to the various upper bounds, we see that our solution is quite close to the upper bounds in almost all cases. On average, *path-alias* detects 15% and 29% redundant checks for ADDRCHECK and LOCKSET, respectively. In the case of pbunzip2, it detects that over 50% of LOCKSET events are redundant. On average, *path-ideal-alias* detects 19% and 30% redundant checks for ADDRCHECK and LOCKSET, respectively. *path-ideal-alias* and *path*-ideal-alias* achieve similar benefits, suggesting that there is little additional benefits from statically detecting aliases across loop paths iterations. This is because we already take advantage of inter-iteration knowledge in our optimizations. Finally, *hw filter* is the best performer on average, because it makes full use of runtime information. Interestingly, *hw filter* was outperformed by *path-ideal-alias* and *path*-ideal-alias* for LOCKSET monitoring zchaff. This is because checks on loads do not make checks on stores redundant in LOCKSET, but not vice versa. *hw filter* has to treat loads and stores as entirely separate events. Consequently, given a path with a load preceding a store to the same address, *hw filter* cannot filter either of them. In contrast, static analysis can eliminate the load because it can scan forwards and backwards in the path.

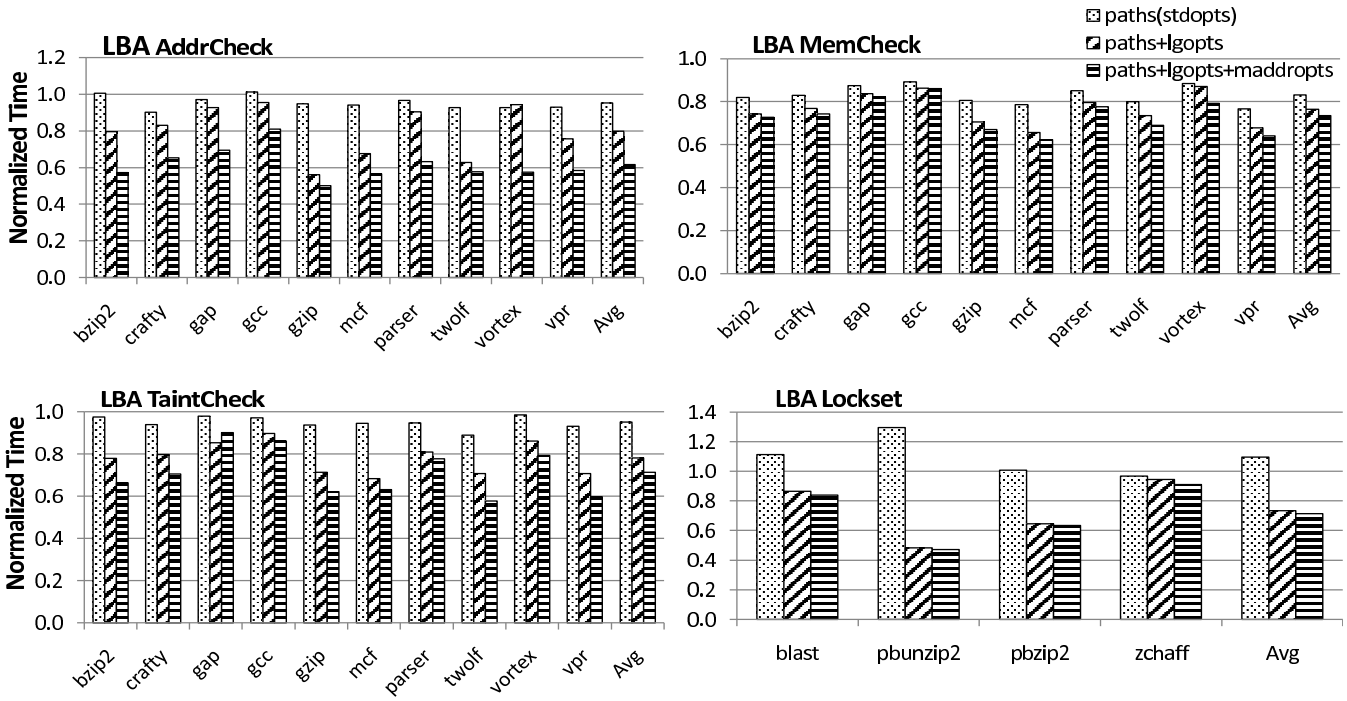

Figure 12. Lifeguard acceleration on Decoupled-LBA.

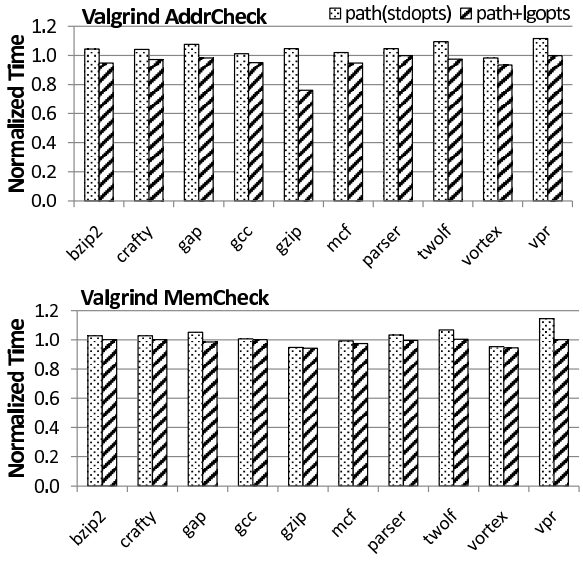

Figure 11. Lifeguard acceleration on Decoupled-Valgrind.

5.3 Lifeguard Acceleration on Decoupled-Valgrind

We now evaluate how much ADDRCHECK and MEMCHECK are accelerated using optimized path handlers on Decoupled-Valgrind. As described earlier in Section 4.1, our optimizations are limited in our current implementation in the following ways: (i) only one hot path is supported per path address, (ii) loop-dead handler calls are not eliminated, and (iii) our metadata access optimization cannot be performed. In addition, we observe that simply using path handlers without our optimizations enables the Valgrind IR optimizer to eliminate more shadow register operations.

Figure 11 shows the execution times of *path(stdopts)* and *path+lgopts* versions of each lifeguard normalized to the execution time of the baseline lifeguard not using path handlers. Every reported result is the best of five runs on a real machine. Because non-dedicated systems were used for the experiments, the best, rather than average, results are presented to limit the unpredictable effects of OS and network activities.

For ADDRCHECK, we observe that despite the limitations described above, *path+lgopts* is faster than the baseline on eight of the ten SPEC2000 benchmarks, with up to 31% reduction in the overhead of monitoring gzip. Loops account for over 95% of gzip execution on reference input. The significant improvement on gzip was due to loop-invariant handler optimizations. However, we see that without domain-knowledge optimizations, *path(stdopts)* is slower than the baseline, up to 12% on vpr, because the overhead of logging outweighs the code improvements made by Valgrind due to longer paths.

On the other hand, *path+lgopts* results in modest MEMCHECK improvements for only four benchmarks (up to 6% on gzip). Our investigations identified the following reasons why *path+lgopts* is less effective on MEMCHECK compared to ADDRCHECK: (i) for a MEMCHECK handler call to be eliminated, both its check and propagation must be redundant, and (ii) the shadow register optimizations performed by the Valgrind IR optimizer overlap with that of our technique, on the other hand ADDRCHECK has no shadow register operations. Due to the second reason, we observe that *path(stdopts)* for MEMCHECK is faster (by 5%) than the baseline on gzip and vortex. This is a result of increasing the length of paths from 3 branches to 8, which enables the Valgrind optimizer to eliminate more redundant shadow register operations. However, *path(stdopts)* is slower on six of the benchmarks (up to 15% on vpr), while *path+lgopts* is always comparable or faster than the baseline because it selects only profitable path handlers.

The lifeguard accelerations obtained using our techniques on Decoupled-Valgrind are in fact conservative because the framework currently lacks the features to support our remaining optimizations. For example, the lack of support for inlining path handlers prevents metadata optimizations.

5.4 Lifeguard Acceleration on Decoupled-LBA

We now evaluate the lifeguard performance gains from our optimizations on Decoupled-LBA. Compared to Decoupled-Valgrind, Decoupled-LBA has the following advantages: (i) hardware logging with no runtime penalty, (ii) support for multiple paths per path address leading to better coverage, (iii) a look-ahead mechanism for detecting last loop iterations, thus enabling loop-dead handler optimizations, and (iv) inlining of path handlers enabling our metadata access optimization. Consequently, it is a better platform for demonstrating the full potential of our optimizations. Figure 12 shows the relative lifeguard performance gains of *path(stdopts)*, *path+lgopts*, and *path+lgopts+maddropts* over the baseline.

As shown in Figure 12, *path+lgopts+maddropts* is consistently the fastest version, followed by *path+lgopts* indicating that our domain-knowledge and metadata access optimizations offer complimentary lifeguard accelerations. Compared to the baseline, *path+lgopts+maddropts* reduces monitoring overhead by up to 50% for single threaded programs (ADDRCHECK on gzip) and 53% for multithreaded programs (LOCKSET on pbunzip2). In addition, average overhead reductions of about 30% are observed for both monitoring scenarios. On the other hand, *path(stdopts)* without the domain-specific optimization techniques achieves much smaller gains or even incurs worse performance than the baseline.

Overall, our optimizations reduce monitoring overhead on Decoupled-LBA by 19–50% for ADDRCHECK, 14–38% for MEMCHECK, 10–42% for TAINTCHECK, and 9–53% for LOCKSET.

6. Conclusion

This paper presented a novel approach to optimizing lifeguards: decoupling the lifeguard code from the monitored program to enable hot-path lifeguard optimizations. Our solution leverages simple knowledge about lifeguards to reduce redundant lifeguard handler calls and to accelerate metadata accesses. We implemented our techniques on both a software-only lifeguard platform (Valgrind) and a hardware-assisted lifeguard platform (LBA). On Valgrind, our techniques reduce monitoring overhead by 2–31% compared to baseline lifeguards. As LBA factors out the runtime overhead of software logging and other limitations of the Valgrind implementation, our techniques achieve even better performance on LBA. The overhead of lifeguard monitoring on LBA is reduced from 1.3–13.3X down to 0.8–10.5X for single threaded programs and from 3.5–4.9X down to 1.9–4.1X for multithreaded programs. Based on the experimental results, we conclude that path optimizations enabled by decoupled lifeguards significantly reduce monitoring overhead. Future work includes fixing the limitations of our Valgrind implementation and generalizing the techniques to handle parallel applications running on multiple cores.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *PLDI*, 2000.

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

[4] 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.

[5] 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.

[6] S. Chen, M. Kozuch, P. B. Gibbons, M. Ryan, T. Strigkos, T. C. Mowry, O. Ruwase, E. Vlachos, B. Falsafi, and V. Ramachandran. Flexible Hardware Acceleration for Instruction-Grain Lifeguards. *IEEE Micro*, 29(1), 2009. *Top Picks from the 2008 Computer Architecture Conferences*.

[7] B. Cmelik and D. Keppel. Shade: A Fast Instruction Set Simulator for Execution Profiling. In *SIGMETRICS*, 1994.

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

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

[12] A. Gal, C. W. Probst, and M. Franz. HotpathVM: An effective JIT compiler for resource-constrained devices. In *VEE*, 2006.

[13] M. L. Goodstein, E. Vlachos, S. Chen, P. B. Gibbons, M. Kozuch, and T. C. Mowry. Butterfly Analysis: Adapting Dataflow Analysis to Dynamic Parallel Monitoring. In *ASPLOS*, 2010.

[14] N. D. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3), 1996.

[15] I. H. Kazi, H. H. Chen, B. Stanley, and D. J. Lilja. Techniques for obtaining high performance in java programs. *ACM Comput. Surv.*, 32 (3), 2000.

[16] P. Lee and M. Leone. Optimizing ML with Runtime Code Generation. In *PLDI*, 1996.

[17] 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.

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

[19] National Center for Biotechnology Information. ftp://ftp.ncbi.nih.gov/blast/.

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

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

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

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

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

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

[26] Parallel Bzip2. http://compression.ca/pbzip2/.

[27] Princeton Zchaff. http://www.princeton.edu/~chaff/zchaff.html.

[28] T. A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *POPL*, 1995.

[29] 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.

[30] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan. Parallelizing Dynamic Information Flow Tracking. In *SPAA*, 2008.

[31] 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.

[32] M. Tiwari, S. Mysore, and T. Sherwood. Quantifying the potential of program analysis peripherals. In *PACT*, 2009.

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

[34] E. Vlachos, M. L. Goodstein, M. A. Kozuch, S. Chen, B. Falsafi, P. B. Gibbons, and T. C. Mowry. ParaLog: Enabling and Accelerating Online Parallel Monitoring of Multithreaded Applications. In *ASPLOS*, 2010.

[35] E. Witchel and M. Rosenblum. Embra: Fast and Flexible Machine Simulation. In *SIGMETRICS*, 1996.

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