ISOLATOR: Dynamically Ensuring Isolation in Concurrent Programs

Sriram Rajamani sriram@microsoft.com
G. Ramalingam grama@microsoft.com
Venkatesh Prasad Ranganath rvprasad@microsoft.com
Kapil Vaswani kapilv@microsoft.com

September 2008
Technical Report
MSR-TR-2008-91

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

1 Introduction

Isolation is a fundamental ingredient in concurrent programs. A thread T may read and/or write certain shared variables in a critical section of code and it may be necessary to ensure that other threads do not interfere with T during this period — other threads should not observe *intermediate* values of these shared variables produced by T and other threads should not update these variables either. This property is called isolation.

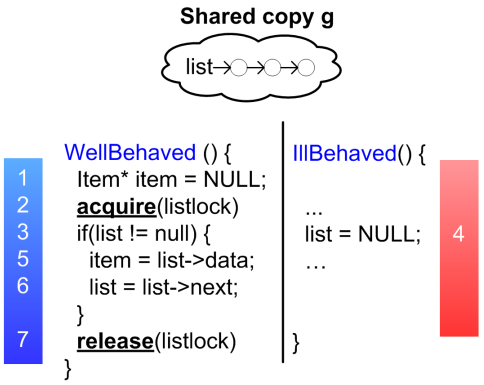
Isolation helps avoid undesirable outcomes arising out of unexpected interactions between different threads and it enables programmers to reason locally about each thread, without worrying about interactions from other threads.

Today, *locking* is the most commonly used technique to achieve isolation. Most often, programmers associate a lock with every shared variable. A *locking discipline* requires that every thread hold the corresponding lock while accessing a shared variable. We say that a thread is *well-behaved* if it follows such a discipline. If all threads are well-behaved, then the thread T holding the locks corresponding to a set of shared variables $\mathcal{V}$ will be isolated from any accesses to $\mathcal{V}$ from all other threads.

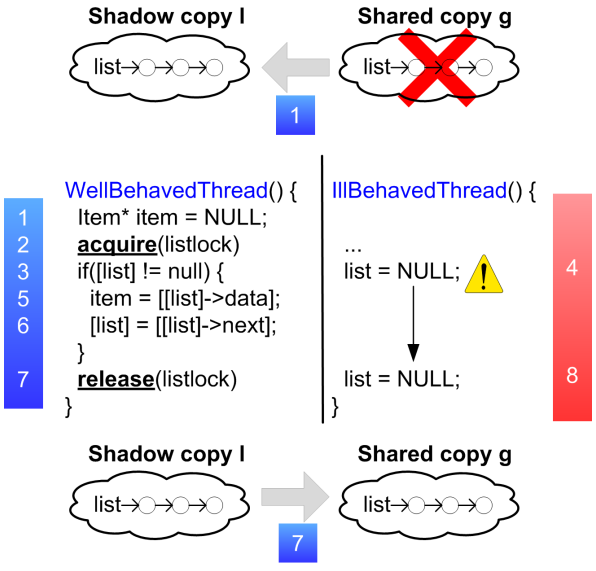
However, commonly used programming languages provide no mechanism to ensure that such locking disciplines are indeed followed by all threads in a program. Thus, even when a thread T_{well} holds a lock ℓ corresponding to a shared variable g, nothing prevents another *ill-behaved* thread T_{ill} from directly accessing g without acquiring lock ℓ, either due to programmer error or malice. Such accesses to g violate the isolation property expected by thread T_{well} and make it impossible to reason locally about the program. Such interferences leads to well-known problems such as *non-repeatable reads*, *lost updates*, and *dirty reads*.

In this paper, we propose a runtime scheme called ISOLATOR that guarantees isolation (by detecting and preventing isolation violations) for parts of a program that follow the locking discipline, even when other parts of the program fail to follow the locking discipline. One of our underlying assumptions is that the code for which we wish to provide isolation guarantees is available (for instrumentation), but the remaining code (which may cause isolation violations) is unavailable.

Motivating Example. We will use the example program fragment shown in Figure 1 to elaborate on the goals of our work. This program consists of a shared list pointed to by a shared variable `list`, protected by the lock `listlock`. The figure illustrates code fragments belonging to two different threads. The first thread T_1 executes function `WellBehaved()` that returns the first item from the list, if the list is non-empty. We refer to this thread a *well-behaved thread* as it follows the locking discipline. The second thread T_2 executes function `IllBehaved()` that removes all items from the list by setting it to be empty. We refer to this thread an *ill-behaved* thread as it does not follow the locking discipline: specifically, it does not acquire the lock `listlock` before updating the list. Because T_2 does not follow the locking discipline,



(a)



(b)

Figure 1: Execution of a program in ISOLATOR. Figure (a) shows an interleaving of threads in which the ill-behaved accesses a shared variable without acquiring the right lock, causing an isolation violation. Figure (b) shows the same interleaving with ISOLATOR. Numbers represent the order in which events occurs. Events with the same number are assumed to execute atomically.

it may execute statement 4 while T_1 is in its critical section (e.g., after the null check in T_1). This can cause T_1 to, unexpectedly, dereference a null pointer.

Goal. Our goal is to provide a defense mechanism to protect well-behaved threads, such as T_1, from interference by ill-behaved threads, such as T_2. We would like to guarantee that thread T_1 is isolated from any access to the list by other threads while T_1 holds the lock for the list. One contribution of our work is a precise and formal characterization of our desired goal. The input for our problem is a concurrent program, a specification of which lock protects which data, and a set of threads $\mathcal{W}$ for which we would like to guarantee isolation. Abstractly, our goal is a runtime mechanism Θ that alters the execution behavior of a concurrent program such that it ensures: (1) safety, (2) isolation, and (3) permissiveness. *Safety* means that every run of a program with Θ should indeed be a possible run of the program without Θ. Safety ensures that the scheme Θ does not introduce new behaviors in the program. *Isolation* means that every run of a program with Θ satisfies isolation for all threads in $\mathcal{W}$. *Permissiveness* means that every run of the program P (without Θ) that satisfies isolation (for the threads in $\mathcal{W}$) is allowed by Θ as well. Permissiveness ensures that concurrency in the program is not *unnecessarily* restricted in the pursuit of isolation.

The Basic Idea. ISOLATOR employs a custom memory allocator to associate every lock with a set of pages that is used only for variables protected by the lock and exploits page protection to guarantee isolation. For the motivating example shown in Figure 1, ISOLATOR allocates the `list` shared variable and the list objects on the page(s) associated with `listlock` (which we refer to as the *shared page(s)*). When the well behaved thread T_1 acquires a `listlock`, ISOLATOR makes a copy of the shared page(s), which we refer to as the shadow page(s), and turns on protection for the shared page(s). We refer to the copy of shared variables in the shadow page(s) as *shadow variables*. ISOLATOR also instruments all accesses to shared variables in T_1's critical section to access the corresponding shadow variable instead. In the figure, we use the notation `[list]` to refer to the shadow variable corresponding to a shared variable `list`; see Section 5 for details of ISOLATOR's instrumentation scheme. If an ill behaved thread T_2 now tries to access the list without acquiring `listlock`, a page protection exception is raised, and is caught by a custom exception handler registered by ISOLATOR. At this point, an isolation violation has been detected.

Upon catching the exception, ISOLATOR's custom exception handler code just yields control and retries the offending access later. When T_1 releases the lock `listlock`, ISOLATOR copies the shadow page(s) back to the shared page(s) and releases the page protection on the shared page(s). From this point on, an access by an ill behaved thread to the list will succeed (until some well behaved thread acquires `listlock` again). Thus, ISOLATOR essentially delays any access to a shared variable by an ill-behaved thread until it can occur with no other thread holding the corresponding lock, thus ensuring isolation. As we describe in the paper, this basic scheme can be optimized by avoiding the

copying between successive lock acquisitions by well behaved threads.

Applications. A mechanism such as ISOLATOR is useful in several circumstances. It makes a concurrent program more robust, and can keep the program running even in the presence of concurrency bugs. It may be particularly appropriate in contexts such as *debugging* and *safe-mode execution*, and when third-party plugins may be ill-behaved. In the context of debugging, the ability of ISOLATOR to identify interference the moment it happens helps identify locking-discipline violation in the ill-behaved thread. The ability of ISOLATOR to avoid interference helps continue with program execution to understand other aspects of the program's behavior, rather than be distracted by the interference. By safe-mode execution, we mean execution of a program, which we expect (e.g., from prior history) is likely to crash, in a mode that reduces the likelihood of a crash. This is particularly useful for applications or operating systems that are extended by third-party plugins that may be ill-behaved.

For example, consider an operating system where the I/O manager is heavily multi-threaded to deal with latency of I/O devices. Several data structures are shared between the I/O manager and device drivers in the system. Even though the OS vendor can ensure that the I/O manager code obeys locking discipline, there is no way to ensure that device drivers written by third parties actually follow the locking discipline. Any violation of the locking discipline can lead to violations of invariants in the I/O manager, and cause the operating system to crash.

Contributions. To summarize, our paper makes the following contributions:

- We formally define three desiderata for any runtime scheme that ensures isolation: (1) safety, (2) isolation, and (3) permissiveness.
- We present a scheme ISOLATOR and prove that ISOLATOR satisfies all the desiderata.
- We present the results of our empirical evaluation of an implementation of ISOLATOR and demonstrate that ISOLATOR can achieve isolation with reasonable runtime overheads.

Our work was inspired by TOLERACE [10], which has a similar goal of protecting well-behaved threads from ill-behaved threads. However, our work differs in the provided guarantees and in the underlying implementation mechanism (See Section 4.4).

2 Background

In this section, we describe a simple concurrent programming language and its semantics and formalize the concept of isolation.

A Concurrent Programming Language. A concurrent program is a triple $P = \langle \mathcal{G}, \mathcal{L}, \mathcal{T} \rangle$ where $\mathcal{G}$ is a finite set $\{g_1, g_2, \ldots\}$ of shared variables, $\mathcal{L}$ is a finite set $\{\ell_1, \ell_2, \ldots\}$ of locks, and $\mathcal{T}$ is a finite set $\{T_1, T_2, \ldots\}$ of threads. A thread is a triple $T = \langle K, \mathcal{I}, \mathcal{R} \rangle$ where K is a natural number such that $\{1, 2, \ldots, K\}$ are the possible values of the program counter of the thread, $\mathcal{I}$ maps each program counter value (i.e., $\{1, 2, \ldots, K\}$) to an *instruction* (defined below), and $\mathcal{R} = \{r_1, r_2, \ldots\}$ is a finite set of local variables.

A *operand* v is either a constant or the contents of some variable. An *instruction* is one of the following: (1) $\texttt{Acquire}(\ell)$ acquires the lock ℓ. This instruction *blocks* if the lock ℓ has been already acquired. (2) $\texttt{Release}(\ell)$ releases the lock ℓ if the executing thread holds the lock (and blocks otherwise). (3) $r_i = \texttt{Read}(g_j)$ reads the value of shared variable g_j into local variable r_i. (4) $\texttt{Write}(g_j, v)$ writes the value of operand v into shared variable g_j. (5) $r_i = \texttt{Op}(v_1, v_2, \ldots, v_k)$ performs an operation (such as add, subtract, and, or, xor, or any such arithmetic or logical operation) on the values of operands $v_1, v_2, \ldots, v_k$ and store the result in the local variable r_i. (6) $\texttt{JmpZ}(r_i, pc)$ performs a conditional jump to pc if the value of local variable r_i is zero.

We assume a sequentially-consistent execution semantics for a concurrent program. At any point in execution, any thread that is not blocked may execute its next instruction. The execution of an instruction by a thread increments the thread's program counter by one if the instruction does not explicitly modify the program counter. We represent the *(execution) history* of a concurrent program P by a sequence $\pi = \sigma_0, \sigma_1, \ldots$ of pairs $\sigma_i = \langle t, x \rangle$ where t is the identifier (an integer) of the thread that executes the instruction x. (Even a truly concurrent execution of multiple instructions on multiple processors can be represented by an equivalent sequence if there is an ordering on instructions that access the same shared variable. Thus, our semantics is fairly general.)

Isolation. In concurrent programs, *interference freedom* is often required while accessing (reading and/or writing) shared data to prohibit access to inconsistent data. This requirement is satisfied by performing such accesses in *isolation*. Most often, programmers achieve isolation by (1) consistently associating a lock with every shared variable and (2) ensuring that every access to a shared variable occurs only when the accessing thread holds the associated lock. This methodology is usually referred to as a *locking discipline*. For a program P, its locking discipline is represented as a function $LD : \mathcal{G} \to \mathcal{L}$. Intuitively, a thread T_t of P obeys the locking discipline LD if, for every shared variable g, T_t always holds the lock $LD(g)$ while accessing g.

More formally, given a history $\pi = \sigma_0, \sigma_1, \ldots, \sigma_n$ and a lock ℓ, a *ℓ-protected sub-history π^ℓ of π is a contiguous subsequence* $\sigma_i, \ldots, \sigma_j$ of π such that $\sigma_i = \langle t, \texttt{Acquire}(\ell) \rangle$, $\sigma_j = \langle t, \texttt{Release}(\ell) \rangle$ or $j = n$, and $\forall m.i < m < j \Rightarrow \sigma_m \neq \langle t, \texttt{Release}(\ell) \rangle$. We shall use $owner(\pi^\ell)$ to denote the identifier of the thread that owns the lock ℓ in π^ℓ. We say that a thread T_t of P obeys the locking discipline LD if, for every history π of P, for every $\sigma_k = \langle t, x \rangle \in \pi$ where instruction x accesses shared variable g, there exists a ℓ-protected sub-history π^ℓ such that $LD(g) = \ell$, $owner(\pi^\ell) = t$, and $\sigma_k \in \pi^\ell$.

Consider a locking discipline LD. Intuitively, a thread T_i *interferes* with thread T_j under LD if T_i accesses a protected shared variable g while T_j holds the lock $LD(g)$. Formally, a ℓ-protected sub-history π^ℓcontains an *interference* under the locking discipline LD if it contains an element $\sigma_i = \langle s, x\rangle$ such that x accesses a shared variable g, $LD(g)= \ell$, and $s \neq owner(\pi^\ell)$. Dually, a ℓ-protected sub-history is *isolated* under the locking discipline LD if there are no elements $\sigma_i = \langle s, x\rangle$ such that x accesses a shared variable g, $LD(g)= \ell$, and $s \neq owner(\pi^\ell)$. We say that a thread T_t executes in isolation in a history π if every ℓ-protected sub-history of π with t as the owner is isolated. A history π is *isolated* if each of its ℓ-protected sub-histories is isolated. Similarly, a program P executes in isolation if each of its histories is isolated.

Proposition 1 *Given a concurrent program $P = \langle \mathcal{G}, \mathcal{L}, \mathcal{T}\rangle$ and a locking discipline LD, program P executes in isolation if every thread $T_t \in \mathcal{T}$ obeys the locking discipline LD.*

3 Ensuring Isolation

Given a locking discipline LD, suppose the threads $\mathcal{T}$ in a program P can be partitioned into *well-behaved threads* $\mathcal{W}$ that obey LD and *ill-behaved threads* $\mathcal{T}\backslash\mathcal{W}$ that may disobey LD. Even under such circumstances, we wish to come up with an efficient and non-intrusive technique that ensures every well-behaved thread executes in isolation (without interference) in every possible history.

The end-goal of the technique is to avoid undesirable interleavings that violate isolation. The technique works by altering the state representation and modifying the interpretation of individual instructions in the program.

A technique *optimally ensures isolation* if the following conditions are satisfied:

Safety For any program P, every history π of P permitted by the technique is also a history of P(under the standard semantics). Furthermore, the state produced by the execution of π by the technique must be *equivalent* to that produced by the execution of π under the standard semantics.

Isolation For any program P and every history π of P permitted by the technique, every well-behaved thread of P executes in isolation in π.

Permissiveness For any program P, every history π of P (under the standard semantics) that is isolated with respect to well-behaved threads of P is also permitted by the technique.

We present a technique for optimally ensuring isolation that relies on alternative operational semantics for `Acquire`, `Release`, `Read`, and `Write` instructions executed by well-behaved threads and `Read` and `Write` instructions executed by ill-behaved thread.

While the requirements of safety and isolation may seem somewhat obvious, we note that there are fault-tolerance techniques (such as Tolerace [13, 10]) that

guarantee neither of these properties (see Section 4.4). As for the permissiveness criterion, it mandates that the technique should not unnecessarily forbid interleavings or concurrency that are "good" (i.e., isolated).

4 Isolator

4.1 Basic Algorithm

Input. The input to ISOLATOR consists of a concurrent program, a locking discipline LD, and a set $\mathcal{W}$ of well-behaved threads for which we would like to provide the isolation guarantee.

Requirements. We assume that the runtime system allows us to enable or disable *memory protection* for any variable v. We denote the operation that enables memory protection for variable v by *MemProtect(v)* and the operation that disables protection for the variable by *MemUnprotect(v)*. Once a variable v is protected, any access to the variable generates a *memory protection violation exception*. We assume that the runtime system allows us to register an *exception handler* that will be triggered (in the context of the thread that caused the memory protection violation) allowing ISOLATOR to take control of the execution.

For every shared variable g_j, ISOLATOR utilizes a new variable $shadow_j$, which we refer to as the *shadow variable* of g_j. The *CopyAndProtect* operation copies the value of a shared variable to its corresponding shadow variable, and enables protection for the shared variable. The *UnprotectAndCopy* operation disables protection for the shared variable, and copies the value of the shadow variable back to the shared variable. We assume that these two operations are atomic. We describe how to implement these two operations using memory protection and OS support in Section 5.

ISOLATOR **Semantics**. During the execution of a thread in $\mathcal{W}$, ISOLATOR works by interpreting the primitive instructions (namely `Acquire`, `Release`, `Read`, and `Write`) differently from the standard semantics. Our implementation realizes the ISOLATOR semantics by rewriting every occurrence of an instruction x in a well-behaved thread by the corresponding code-fragment shown in Figure 2. We use $\texttt{Acquire}_S(\ell)$ and $\texttt{Release}_S(\ell)$ to represent the standard semantics of $\texttt{Acquire}(\ell)$ and $\texttt{Release}(\ell)$. Instructions not shown in the table are interpreted as usual. Also, all instructions are interpreted as usual when threads in $\mathcal{T} \backslash \mathcal{W}$ execute.

Let $InvLD : \mathcal{L} \rightarrow \wp(\mathcal{G})$ be the inverse of the function LD. It maps every lock to the set of shared variables protected by that lock according to the locking discipline LD.

In accordance with the Isolator semantics (Figure 2), when a thread in $\mathcal{W}$ acquires a lock ℓ, ISOLATOR copies the value of shared variables protected by ℓ to the corresponding shadow variables, and enables memory protection for the

Instruction I	ISOLATOR's semantics for I
$\texttt{Acquire}(\ell)$	$\texttt{Acquire}_S(\ell)$ $foreach\ g_j \in InvLD(\ell)$ $\texttt{CopyAndProtect}(g_j, shadow_j)$ end
$\texttt{Release}(\ell)$	$foreach\ g_j \in InvLD(\ell)$ $\texttt{UnprotectAndCopy}(g_j, shadow_j)$ end $\texttt{Release}_S(\ell)$
$r_i = \texttt{Read}(g_j)$	$r_i := shadow_j;$
$\texttt{Write}(g_j, v)$	$shadow_j := v;$
$\texttt{OnException}(g_j)$	$yield()$
$\texttt{CopyAndProtect}(g_j, shadow_j)$	$atomic\{shadow_j := g_j;$ $\texttt{MemProtect}(g_j);\}$
$\texttt{UnprotectAndCopy}(g_j, shadow_j)$	$atomic\{\texttt{MemUnprotect}(g_j);$ $g_j := shadow_j;\}$

Figure 2: Isolator Semantics for various operations. $\texttt{Acquire}_S(\ell)$ and $\texttt{Release}_S(\ell)$ represent lock acquisition and release operations under standard semantics.

shared variables. Any access to a shared variable in a thread in $\mathcal{W}$ is directed to the corresponding shadow variable. When a thread in $\mathcal{W}$ releases a lock ℓ, memory protection is disabled for the shared variables protected by ℓ and the value of the shadow variables are copied back to the corresponding shared variables.

The final component of the ISOLATOR semantics relates to the treatment of instructions executed by ill-behaved threads — ISOLATOR does not alter the semantics of such instructions. However, any such instruction that accesses a protected shared variable without acquiring the corresponding lock is not *enabled for execution* if one of the well-behaved threads holds the lock. ISOLATOR ensures this, in the implementation, by installing a custom exception handler that handles access violation exceptions and forces the ill-behaved thread to back-off by temporarily yielding control. This is denoted by the instruction $\texttt{OnException}(g_j)$in Figure 2.

A discerning reader may have noted that although the core ISOLATOR algorithm enforces isolation, it does not provide any progress guarantees to ill-behaved threads. In some cases, an ill-behaved thread may starve for long durations while well-behaved threads hold locks on shared variables. But note that such behavior is identical to what can happen if the ill-behaved thread were to correctly follow the locking discipline. We discuss this further in Section 7.

4.2 Properties of ISOLATOR

First, we prove that `Acquire`(ℓ) and `Release`(ℓ) can be thought of as atomic operations even though they execute several `CopyAndProtect`(g_j, $shadow_j$) and `UnprotectAndCopy`(g_j, $shadow_j$) operations in a loop.

Theorem 2 *Consider any execution sequence π produced by a program under* ISOLATOR *semantics. We can transform π into an equivalent execution sequence π' where all the* `Acquire`(ℓ) *and* `Release`(ℓ) *operations execute atomically.*

Proof Consider any execution sequence π. Suppose thread T_t starts executing `Acquire`(ℓ). Any intervening operations by other threads that read or write any of the shared variables $g_j \in InvLD(\ell)$ can be thought of as occurring before the execution of `Acquire`(ℓ) by thread T_t. Similarly, suppose thread T_t starts executing `Release`(ℓ). Any intervening operations by other threads that read or write any of the shared variables $g_j \in InvLD(\ell)$ can be thought of as occurring after the execution of `Release`(ℓ) by thread T_t.

Next, we show that ISOLATOR has the three properties that defined our goal, namely *safety*, *isolation* and *permissiveness* (see Section 3).

Establishing these properties requires us to compare the normal execution behavior of a program (the program's *standard semantics*) with the execution behavior of the program under ISOLATOR (which we refer to as the ISOLATOR *semantics*). We first define a notion of *equivalence* between the program states used by the standard semantics and the program states used by the ISOLATOR semantics. Note that the only difference in the representation of a state under the ISOLATOR semantics is that the value of a shared variable g_j is stored in the local copy $shadow_j$ when g_j is protected, and in g_j otherwise. Let σ_i be a state in the ISOLATOR semantics. We define $lookup(\sigma_i, g_j)$ to be the value of $shadow_j$ if the lock corresponding to g_j is currently held by some well-behaved thread, and the value of g_j otherwise.

A state σ_s in the standard semantics is said to be *equivalent* to a state σ_i in the ISOLATOR semantics iff (a) For all threads T, the value of the program counter of T, the value of local variables of T, and the set of locks held by thread T are the same in σ_s and σ_i, and (b) For every shared variable g_j, the value of g_j in σ_s is equal to $lookup(\sigma_i, g_j)$.

Consider a sequence π of pairs $\langle t, x \rangle$, consisting of a thread id and an instruction. We say that such a sequence is *feasible* under the standard semantics if it is the execution history for some (possibly incomplete) program execution (under the standard semantics). We define the notion of feasibility under the ISOLATOR semantics similarly.

Theorem 3 *(1) A sequence π is feasible under the* ISOLATOR *semantics iff it is feasible under the standard semantics and is isolated (with respect to $\mathcal{W}$). (2) Furthermore, for any isolated feasible sequence π, the final state produced by the execution of π under the* ISOLATOR *semantics is equivalent to the final state produced by the execution of π under the standard semantics.*

Proof We prove the theorem by induction on the length of π. The claim is trivially true for the empty sequence π. Assume that it is true for some sequence π. Consider any sequence $\pi' = \pi.\langle t, x\rangle$.

Case 1 (Interfering instruction): Consider the case where instruction x accesses some shared variable g_m currently locked by a thread other than T_t. In this case the memory locations corresponding to g_m are protected by ISOLATOR. As a result, the execution of instruction x will cause a fault. Thus, π' is not feasible under the ISOLATOR semantics.

Case 2 (Non-interfering instruction): Consider the case where a thread T_t executes an instruction x that does not access a shared variable currently locked by another thread. In this case, the instruction execution does not cause a fault. Furthermore, all the operands of the instruction have the same values in both the standard semantics as well as the ISOLATOR semantics (from the inductive hypothesis that the states are equivalent). As a consequence, the resulting states are equivalent as well.

Corollary 4 ISOLATOR *optimally ensures isolation.*

Proof Follows from Theorem 3.

4.3 Optimized Algorithm

The naive implementation of ISOLATOR's `Acquire` and `Release` operations that copies data and enables/disables memory protection at the beginning and end of every critical section is inefficient. This is because copying data between the shared and shadow variables and enabling/disabling memory protection are both expensive operations (several thousands of cycles depending on the amount of data). In Figure 3, we show an optimized algorithm that greatly reduces copying overhead.

The optimized algorithm relies on the observation that for a given lock, as long as the lock is accessed only by threads in $\mathcal{W}$, the copying and changing protection done by `Release` and the subsequent `Acquire` operations are redundant. To eliminate this redundancy, we simply do not perform copying or disable protection during `Release` (as shown in Figure 3). On `Release`, threads perform no additional operations other than releasing the lock. This implies that the shared variables protected by the lock remains under memory protection even after the thread releases its lock.

With every lock ℓ, the optimized algorithm maintains a flag $IsShadowValid_\ell$ which indicates whether the shadow variables contain the most recent version of the shared state. The flag is initially set to $false$ and set to $true$ on every `Release`(ℓ). On `Acquire`(ℓ), threads check the $IsShadowValid_\ell$ flag. If the flag is $false$, the thread copies the value of the shared variables to the corresponding shadow variables and enables memory protection. However, if $IsShadowValid_\ell$ is $true$, the thread does not update the shadow variables since the shadow copy contains the most recent version of the shared state.

When a thread in $\mathcal{T}\backslash\mathcal{W}$ accesses a shared variable (either with or without acquiring the lock) and the shared variable is in protected mode, an access

Instruction I	Semantics for I in our implementation
$\texttt{Acquire}(\ell)$	$\texttt{Acquire}_S(\ell)$ $if\ (!IsShadowValid_\ell)$ $\ IsShadowValid_\ell := true$ $\ foreach\ g_j \in InvLD(\ell)$ $\ \ \texttt{CopyAndProtect}(g_j, shadow_j)$ $\ end$
$\texttt{Release}(\ell)$	$\texttt{Release}_S(\ell)$
$r_i = \texttt{Read}(g_j)$	$r_i := shadow_j;$
$\texttt{Write}(g_j, v)$	$shadow_j := v;$
$\texttt{OnException}(g_\texttt{j})$	$let\ \ \ell = LD(g_j)\ \ in$ $if\ (\texttt{TryAcquire}_\texttt{S}(l))$ $\ if\ (IsShadowValid_\ell)$ $\ \ IsShadowValid_\ell := false$ $\ \ foreach\ g_j \in InvLD(\ell)$ $\ \ \ \texttt{UnprotectAndCopy}(g_j, shadow_j)$ $\ \ end$ $\ \texttt{Release}_S(\ell)(l)$ $else$ $\ yield()$

Figure 3: Semantics of operations in optimized ISOLATOR

violation is raised. Unlike the exception handler described in Figure 2 that merely yields control, the exception handler in our implementation first tries to acquire the lock ℓ corresponding to the variable using a nonblocking acquire operation $\texttt{TryAcquire}_\texttt{S}(\ell)$.

This operation returns true if ℓ is held by the current thread or if ℓ was available and it was acquired (without blocking); otherwise, it returns false. When the operation returns true, we assume that the $\texttt{TryAcquire}_\texttt{S}$ operation is treated as a reentrant acquire. If the lock API does not support reentrancy, we can modify the code for $\texttt{OnException}(g_\texttt{j})$ in Figure 3 to check explicitly if the current thread already holds the lock, and handle that case separately.

If $\texttt{TryAcquire}_\texttt{S}(\ell)$ returns true, then the code for $\texttt{OnException}(g_\texttt{j})$ first checks the value of $IsShadowValid_\ell$. If $IsShadowValid_\ell$ is true, then protection is turned off on the shared variables, the shared variables are updated with the values of the corresponding shadow variables, and the $IsShadowValid_\ell$ is set to false. If the value of $IsShadowValid_\ell$ is false, then no extra operations are performed as the shared variables already have the most recent state. This case happens when two threads from $\mathcal{T}\backslash\mathcal{W}$ both access a shared variable simultaneously, and the exception handler for the first thread has already done the copying and set $IsShadowValid_\ell$ to false.

Due to this optimization, copying between shadow and shared variables happens only when ownership of the shared variable goes from a thread in $\mathcal{W}$ to a

Instruction I	Tolerace's semantics for I

$\texttt{Acquire}(\ell)$	$\texttt{Acquire}_S(\ell)$
	foreach $g_j \in InvLD(\ell)$
	  $shadow_j := g_j$;
	  $orig_j := g_j$;
	endfor

$\texttt{Release}(\ell)$	$GW = \{j \in InvLD(\ell) \mid g_j \neq orig_j\}$
	$LW = \{j \in InvLD(\ell) \mid shadow_j \neq orig_j\}$
	$if(LW \neq \{\})$
	  $if(GW \neq \{\})$
	  report "unfixable race"
	  else
	  foreach $g_j \in InvLD(\ell)$
	  $g_j := shadow_j$;
	  endif
	endif
	$\texttt{Release}_S(\ell)$

$r_i = \texttt{Read}(g_j)$	$r_i := shadow_j$;
$\texttt{Write}(g_j, v)$	$shadow_j := v$;

Figure 4: Semantics of operations in Tolerace

thread in $\mathcal{T} \backslash \mathcal{W}$, or vice-versa. If this transfer is infrequent, the program does not experience any performance overheads due to ISOLATOR. We evaluate the overheads of our implementation in more detail in Section 7.

4.4 Comparison with Tolerace

While ISOLATOR satisfies safety, isolation and permissiveness, similar algorithms that appear in the literature violate these desiderata. In this section, we briefly describe Tolerace [10] and show an example to illustrate that it can violate safety.

The left portion of Figure 4 gives the semantics of operations in Tolerace [10]. The `Acquire` and `Release` operations are very similar to that of ISOLATOR, and the `Read` and `Write` operations are exactly the same as that of ISOLATOR. However, the main difference is that memory protection is not used to detect conflicts. Instead, during the `Acquire` operation, the original value of each shared variable g_j is stored in $orig_j$. During the `Release` operation, Tolerace checks if some shared variable g_j and shadow copy $shadow_{j'}$ have been written by comparing their values with the original values. If this is the case, it simply declares that an unfixable race has been encountered. Otherwise, the `Release` operation writes back the shadow copy back to the shared variables.

Tolerace does not ensure isolation in some situations where both shadow and

Thread 1	Thread 2
$L1$: **Acquire**(ℓ_1)	
	$M1$: **Write**$(x, 5)$
	$M2$: **Write**$(a, 10)$
$L2$: **Acquire**(ℓ_2)	
$L3$: $r_2 :=$ **Read**(a);	
$L4$: **Write**(b, r_2);	
$L5$: **Release**(ℓ_2)	
$L6$: $r_1 :=$ **Read**(x);	
$L7$: **Release**(ℓ_1)	
$L8$: **Acquire**(ℓ_1)	
$L9$: **Write**(y, r_1);	
$L10$: **Release**(ℓ_1)	

Figure 5: Example showing how Tolerace violates safety

shared copies are updated. More surprisingly, in the presence of nested acquires and releases, as the example below shows, it generates behaviors that are not allowed by the standard semantics.

Consider the execution history in the right portion of Figure 4. Here the lock ℓ_1 protects shared variables x and y, and the lock ℓ_2 protects shared variables a and b. Thread 1 is well-behaved and Thread 2 is ill-behaved. Let the initial value of all shared variables be 0. During **Acquire**(ℓ_1) at line L1, Tolerace makes shadow copies of x and y. Then, Thread 2 updates x to 5 and a to 10 respectively. Then, Thread 1 acquires ℓ_2 at line L2, and makes shadow copies of a and b. Note that at this point the shadow copy of x still has the old value 0, but the shadow copy of a has the new value 10. Later, when Thread 1 executes **Release**(ℓ_1) at line L7, the shadow copy of b contains 10, which is written back to the shared copy, and the local variable r_1 contains the old value of x, namely 0. Finally, the statements at lines L8 to L10 are executed, resulting in the value 0 written to y.

Under standard semantics, if b gets the value 10, then y necessarily will get the value 5. Thus, the above execution is not allowed by the usual semantics, but it is incorrectly allowed by Tolerace.

5 Isolator for C

While the concurrent programming language in Section 2 simplifies the description of Isolator, it also masks the complexities involved in applying Isolator in the context of real world programming languages that support functions, pointers, and dynamically allocated data. In this section, we address this con-

cern by describing ISOLATOR$_C$, a realization of ISOLATOR for C.

5.1 The Input

In our earlier presentation, the input to ISOLATOR consisted of a set of well-behaved threads, represented by their code, and a specification of the locking discipline. We now generalize this and allow the input to ISOLATOR$_C$ to consist of any part P of a C program for which we wish to provide isolation guarantees, as long as P satisfies the following conditions: (a) We require P to be *closed* with respect to function calls: if P contains a call to some function f, we require P to include the code for f. (However, this requirement can be relaxed if f is guaranteed not to reference the shared data we are protecting and not to acquire or release the corresponding locks. This is convenient for handling library calls.) (b) We require the lock acquire/release operations in P to be *well-matched*: *i.e.*, for any possible execution path (by a single thread) in the whole program, the subpath from any point when execution enters P to the point when execution subsequently leaves P must have well-matched lock acquire/release operations.

Unlike in the simplified language used earlier, there is no syntactic difference between shared and thread-local data in a C program. The specification of the locking discipline identifies the shared data, as well as the locks protecting the shared data. Shared data may be either static (global) variables or dynamically allocated memory. For shared static variables, an annotation attached to the declaration of the variable (of the form "`__guarded_by (x) v`", where `x` is a static variable of type lock) indicates the lock `x` protects variable `v`. For dynamically allocated shared data, an annotation attached to the statement that allocates the memory (of the form "`malloc(...) __guarded_by (lock-expr)`") indicates the lock that protects the allocated memory. The meaning of the above annotation is that the memory allocated by an execution of the allocation statement is protected by the lock that `lock-expr` evaluates to during the execution of the statement. This allows for dynamic locks and fine-grained locking. These annotations can be automatically inferred using tools such as Locksmith [16]. The inferred annotations can be checked and refined by the programmer and then provided as input to ISOLATOR$_C$.

5.2 Shared Data Protection and Duplication

As explained earlier, ISOLATOR requires some form of memory protection mechanism to prevent ill-behaved accesses to shared variables. Most modern operating systems, including Windows [3] and Linux [2], support some form of memory protection. In ISOLATOR$_C$, we consider the Windows Virtual Memory API, which enables a process to modify access protection at the granularity of virtual memory pages. This API is commonly used by the OS to detect stack overflows and guard critical data structures against corruption.

This approach requires that every lock be associated with a set of pages used only for shared variables protected by that lock. As described in Section 4, every shared variable is also associated with a shadow variable. In ISOLATOR$_C$,

each shared variable protected by a lock ℓ is allocated on a *shared memory page* specific to ℓ and the corresponding shadow variable is allocated on the associated *shadow memory page* at a fixed offset from shared memory page. The fixed offset between the two pages of shared data allows redirection of data accesses by merely adding a fixed offset to the accessed address.

For statically allocated data, the allocation of the shared and shadow variables on appropriate memory pages can be statically achieved (with support from the compiler). For dynamically allocated data, ISOLATOR$_C$ transforms every call to `malloc` for allocating shared data into a request to a custom memory allocator that takes the associated lock as an extra parameter and allocates memory for both shared objects and shadow objects on appropriate memory pages.

5.3 Code Instrumentation

We assume that the lock acquire and release operations `Acquire` and `Release` are implemented as C functions with lock variables as parameters. ISOLATOR$_C$ replaces all calls to these functions by calls to custom operations that realize the isolator semantics as described in Section 4.

We now describe how data accesses are instrumented. We simplify the current discussion by assuming that the instrumented code is well-behaved (*i.e.*, that it accesses shared data only while holding the corresponding lock) and consider the general case later. Unlike in our simple language, there is no syntactic difference between references to local data and references to shared data in C. However, direct references (to static data) can be easily classified as a reference to local or shared data. We transform any access to a shared variable g_i into a reference to the corresponding shadow variable $shadow_i$.

Pointers to shared variables, however, introduce some complications. If a well-behaved thread contains a reference of the form "`*p`" involving pointer indirection, then ISOLATOR$_C$ needs to determine if this reference is to a shared variable to redirect the reference appropriately. We utilize a static analysis to determine whether an indirect reference "`*p`" may be a reference to a shared variable. Any points-to analysis can be adapted to compute this information, as indicated below:

- If none of the targets that `p` may point-to is a shared variable, then `*p` is not a reference to shared variable, and the access is left as is.
- If all of the targets that `p` may point-to are shared variables, then `*p` is always a reference to a shared variable, and the access is redirected to the shadow variable.
- If some, but not all, of the targets that `p` may point-to are shared variables, then we cannot statically determine whether the reference `*p` will be to a shared variable. We instrument the code to introduce a runtime check that determines if `p` points to a shared variable. If the check passes, then

the access is redirected to the shadow variable; otherwise, the access is left unchanged.

Note that we always allocate the shadow variable at the same offset from the shared variable, as explained earlier. Thus, the redirection of `*p` is achieved by transforming it into `*(p+offset)`, even when we do not know which shared variable `p` points-to. (Otherwise, the redirection will require another indirection at runtime if the referenced variable is not known at instrumentation time.)

Also note that an access to a shared variable g_i must be redirected to the corresponding shadow variable only when the access occurs in a context where the shared variable has been copied to the shadow variable. However, this precondition may not hold for a particular access to g_i in the program fragment P if the code in P is not well-behaved (and accesses the shared variable without acquiring its lock) or if the corresponding lock-acquire was done before the instrumented code P starts executing. If such situations are possible, we can handle them by using an analysis to identify accesses which may suffer from this problem and adding a runtime check in the corresponding instrumented code to determine if the access should be redirected.

5.4 Function Cloning

In general, we may have functions that are called from within critical sections in the instrumented code as well as from outside critical sections. If these functions may potentially access shared data, cloning these functions can greatly simplify the analysis required by the instrumentation, reduce the need for runtime checks added by instrumentation and reduce the runtime overhead of these checks. Specifically, one can use the original, uninstrumented, function for all calls from outside the instrumented code (which includes code in P that is guaranteed to execute outside critical sections). We can use an instrumented version of the function for all calls in the instrumented code that may execute inside a critical section.

5.5 Atomic `CopyAndProtect` and `UnprotectAndCopy`.

In the description of the ISOLATOR algorithm, we assumed that the `CopyAndProtect` and `UnprotectAndCopy` operations can be implemented atomically. We now describe atomic implementation of these operations.

- **Atomic `CopyAndProtect`.** A page level `CopyAndProtect` operation takes the address of two virtual pages V_1 and V_2 as input, copies V_1 to V_2 and marks V_1 as protected. In our implementation, `CopyAndProtect` is performed atomically by first marking V_1 as *read-only*, updating V_2, and finally marking V_1 as protected with no access. This has the same effect as performing the operation atomically. (Any concurrent read by an ill-behaving thread is equivalent to one performed before the `CopyAndProtect` operation began).

Thread T_{well}	Thread T_{ill}
`Acquire`(ℓ_1);	
	`Acquire`(ℓ_2);
$g_1 = ...$	$... = g_2$
	$g_1 = ...$
`Acquire`(ℓ_2);	

Figure 6: Interleaving illustrating a potential deadlock under ISOLATOR.

- **Atomic `UnprotectAndCopy`.** A page-level `UnprotectAndCopy` operation takes two virtual pages V_1 and V_2 as input, copies V_2 to V_1 and marks V_1 as unprotected. An atomic implementation of `UnprotectAndCopy` is harder to realize because the semantics involve writes to protected pages and disabling protection is a pre-requisite for writing. However, disabling protection before writing would leave a window of vulnerability in execution where an isolation violation from an ill-behaved thread would not be detected. The implementation of `UnprotectAndCopy` relies to OS support to ensure atomicity:

  1. Allocate a temporary virtual page V_{tmp} mapped to a physical page P_{tmp}.
  2. Copy contents of V_1 to the page V_{tmp}.
  3. Change the virtual-physical page mapping so that the virtual page V_2 maps to the physical page P_{tmp}.
  4. Disable protection on the virtual page V_2.

  The implementation guarantees that any other thread concurrently accessing either causes an access violation or observes V_2 in a state consistent with V_1.

6 Limitations and Extensions

Deadlocks and Livelocks. While ISOLATOR guarantees isolation, it has the potential to introduce deadlocks and livelocks. Consider the example in Figure 6. Let us assume ℓ_1 protects g_1 and ℓ_2 protects g_2. For the interleaving given in Figure 6 ISOLATOR attempts to delay the access to g_1 by the T_{ill} until the thread T_{well} releases ℓ_1. However, thread T_{well} waits to acquire ℓ_2, which is being held by T_{ill}. Thus, the delaying of the access to g_1 in T_{ill} by ISOLATOR introduces a deadlock. Similar examples can be constructed where ISOLATOR introduces livelocks.

A simple back-off based solution to alleviate deadlocks and livelocks is to remove the memory protection after a finite number of executions of the *yield*()operations in the exception handler. Alternatively, an arbitrary thread can be allowed to continue execution after a deadlock is dynamically detected in ISOLATOR's exception handler. While these strategies would avoid deadlocks

and livelocks, they would allow isolation violation (that can be reported to the user).

Locking Granularity. Our current isolation implementation relies on hardware and OS support for memory protection. On most existing operating systems, the granularity of protection is tightly coupled with the page size. This design, coupled with ISOLATOR's custom memory allocation scheme, may cause memory fragmentation. The fragmentation will be more pronounced if the program uses fine-grained locks. This drawback can be addressed if hardware/OS implementations decouple memory protection from page size and provide support for fine-grained memory protection. Researchers have already proposed several hardware and software extensions [20, 22, 5] that support memory protection at the granularity of individual cache lines. We note that these extensions, if supported by future hardware implementations, will also benefit several other techniques that rely on memory protection [4, 12].

Handling other synchronization primitives. Apart from locks, ISOLATOR is also capable of handling other synchronization primitives such as reader/writer locks and condition variables. For critical sections that acquire reader locks, we provide isolation by enabling read-only protection for data protected by the lock and enabling full page protection only when a writer lock is acquired. A wait on a condition variable is treated as a release of the associated lock before the wait and an acquire of the lock after the wait.

Collocated locks and data. Consider the annotated declaration of `tree` structure in Figure 7. The implementation uses one coarse-grained lock `lock` that is part of the structure to protect fields `root` and `items`. If ISOLATOR$_C$'s custom allocator allocates `lock` on the same page as the fields, accesses to the ℓ will raise access violation exceptions when memory protection is enabled. In languages like Java that do not support pointer arithmetic, this problem can be safely addressed by automatically splitting the object. However, such transformations can be unsafe in languages like C/C++. Hence, ISOLATOR$_C$ identifies such cases and expects the programmer to manually perform the transformation.

Weak memory models. As described in this paper, ISOLATOR assumes and ensures sequentially consistent semantics. However, several hardware implementations only guarantee sequential consistency for correctly synchronized accesses and provide weaker guarantees for unsynchronized accesses. Under weak memory models, isolation can be ensured if the `MemProtect` operation acts as a memory barrier for the shared variables being protected (perhaps at a higher performance cost). If the implementation of `MemProtect` does not enforce an order on shared variables, ISOLATOR's implementation may have to introduce additional barriers.

```
typedef struct _cp_tree {
 /* root node*/
  __guarded_by(lock) cp_node *root;

 /* item count */
 __guarded_by(lock) int items;

 /* lock protecting items and the tree */
 cp_lock *lock;
} cp_tree;
```

Figure 7: Declaration of a structure in which the lock and data protected by the lock are collocated.

7 Experimental Evaluation

We have implemented ISOLATOR as a compiler phase using Microsoft's Phoenix compiler infrastructure (July 2007 SDK). However, due to limitations of the Phoenix infrastructure, some aspects of our algorithm have been manually implemented. For instance, since Phoenix does not support function cloning, we preprocess our benchmark programs to identify functions called from critical sections and duplicate the functions manually. Some manual transformations were also required to overcome the lack of support for inter-procedural alias analysis and the inability to split fields of structures that contain locks. However, we believe that these tasks can be easily automated using a more powerful compiler infrastructure.

7.1 Benchmarks

We evaluate the runtime overheads of ISOLATOR using several multi-threaded microbenchmarks as well as real world programs.

- libcprops is a C prototyping library consisting of generic data structures (heaps, lists, trees, hashtables etc.) and applications levels components (thread pools, http sever/clients, etc). The library uses per-instance locks to control access to data structures. We used ISOLATOR to enforce this locking discipline. These data structures are our microbenchmarks. For each data structure, we implemented a stress testing client that creates a specified number of threads and each thread randomly performs operations on the data structure until the total number of operations exceeds a threshold (1 million). We classify the data structure operations as read and write operations and parametrize the client by the fraction of write operations to be performed. We also evaluated ISOLATOR using the modified version of httpclient, an application that creates a specified number of threads to fetch a set of URLs. httpclient uses a shared trie to store cookies and a shared stack to store data transfer requests that are performed

asynchronously.

- pfscan is a parallel file scanning utility that mimics the Unix grep utility. pfscan consists of a shared queue that maintains a set of files to be scanned for matching. The main thread creates several worker threads that dequeue file names from the queue and process each file while the main thread populates the queue with the names of files in the given path. pfscan uses a course-grained lock to protect access to the queue; we use ISOLATOR to ensure isolation for critical sections in the queue API. For our evaluation, we use pfscan to search for five keywords in 4800 files of C/C++ source code.

- lkrhash is an industry strength concurrent hash table. The hash table implementation uses a lock to protect a collection of sub-tables that are created when entries are added or removed from the hash table, when the hash table is searched, and when the tables expand or contract. We use a set of inputs provided with the implementation to measure overheads of ISOLATOR.

7.2 Experimental platform and methodology

We performed our experiments on a system with the Intel Pentium Core 2 Duo CPU (1.6Ghz) and 3 GB of RAM running Windows Vista. All our benchmarks were compiled using Microsoft's C/C++ compiler (version 15.00.21022) and use the pthreads library. To ensure that our measurements are not biased by micro-architectural effects such as cache warm-up, we estimate the execution time of a benchmark (with or without ISOLATOR) by running the benchmark 5 times, ignoring the first run and computing the average of the other 4 runs.

7.3 Experiments with microbenchmarks

Runtime overheads. We conducted two sets of experiments to evaluate ISOLATOR's runtime overheads for the microbenchmarks. In the first set of experiments, we simulate a scenario where all threads are well-behaved. We achieve this by instrumenting all critical sections in the data structure implementation. This represents ISOLATOR's best case scenario because it minimizes the amount of copying between the shared variables and the shadow variables and the number of memory protect/unprotect operations.

Figure 8 shows the execution time for the microbenchmarks for different number of threads and fraction of write operations. We observe that ISOLATOR has extremely low overheads for most benchmarks. The average overheads of ISOLATOR is 1.42% with a minimum of -9.3% and a maximum of 11.2%. We also note that the overheads of ISOLATOR do not depend on the number of threads or the fraction of write operations, which suggests that ISOLATOR's internal data structures (used for tracking the mapping between locks and the shared pages they protect) do not introduce any synchronization bottlenecks. We analyzed cases in which enabling ISOLATOR led to a speedup and found that the speedups

can be attributed to our custom memory allocator, which improves locality by allocating data protected by the same lock on the same set of pages.

In a second set of experiments, we introduce one additional thread in all the microbenchmarks; this thread executes operations with uninstrumented critical sections, simulating a scenario where the program consists of both well-behaved and ill-behaved threads. The presence of this thread forces ISOLATOR to copy data between shared and shadow pages and enable/disable memory protection on every ownership transfer between well-behaved threads and the additional thread. For our experiment, we control this interaction by restricting the number of operations performed in the well-behaved threads.

7.4 Experiments with real-world applications

Figure 9 illustrates the overheads of ISOLATOR for varying number of well-behaved threads and fraction of operations in the additional thread. As expected, we observe that the overheads of ISOLATOR increase as the fraction of operations performed in the additional thread increases. As long as execution is dominated by well-behaved threads ($< 10\%$ operations from the additional thread), ISOLATOR has reasonable overheads ($< 20\%$ for most microbenchmarks), independent of the number of threads. Beyond this threshold, ISOLATOR's overheads increase significantly, reaching about 100%. The linked list is an exception with up to 8x overheads. We attribute these overheads to small critical sections in the benchmark. As a result, ISOLATOR's acquire and release operations dominate this benchmark's execution time. We believe there is scope for other interesting optimizations to reduce these overheads and leave such optimizations for future work.

Effectiveness. During our experiments, ISOLATOR detected real isolation violations in the microbenchmarks. Figure 10, which shows a simplified fragment of code from the trie benchmark, illustrates one such isolation violation. The function `cp_trie_prefix_match`, reads the `root` object and the `leaf` field of the root object (line 4 and 5) without acquiring the lock on the data structure. The function `cp_trie_remove`, which removes keys from the trie, can potentially write to both these fields (lines 27, 35 and 37) under certain conditions. An isolation violation (a race condition) occurs when one these reads occurs simultaneously with the write. ISOLATOR detects and prevents this violation by delaying the reads until `cp_trie_remove` has released the lock. We find that isolation violations in these benchmarks are hard to reproduce and occur rarely during execution. Consequently, detecting and tolerating these violations does not add any noticeable overheads.

Figure 11 shows the execution times of the three real world applications, pfscan, httpclient and lkrhash with and without ISOLATOR. The overheads of ISOLATOR are consistently low for all these benchmarks (a maximum of 6% and 1% on average). In these benchmarks, ISOLATOR did not detect any isolation violations.

8 Related Work

There has been a lot of prior work on detecting races using static techniques [19, 7, 16, 14, 6, 9] and dynamic techniques [17, 15, 21, 1] Unlike race detection, which is useful for testing and identifying bugs, ISOLATOR is a fault toleration technique that makes the execution of a buggy program more robust. (However, our approach can also be used to identify a class of races and isolation violations.)

The idea of tolerating race conditions was first proposed in Tolerace by Kriovski et al [10, 13]. More recently, Krena et al [11] propose heuristic mechanisms for dynamically detecting and fixing race conditions. Both these works do not satisfy the semantic conditions (safety, isolation and permissiveness) satisfied by ISOLATOR.

Flanagan and Freund [8] proposed a static analysis to inject locks to fix synchronization errors in programs with annotations capturing the locking discipline. More recently, Shpeisman et al [18] propose a technique to enforce isolation for programs using STMs. Their technique statically analyses code to identify instructions outside atomic sections that can conflict with other atomic sections and inserts appropriate barriers to ensure strong atomicity. Both these works require analysis of the whole program. In Shpeisman et al's work [18], instrumentation needs to be done after static analysis on the whole program. In contrast, our work requires analysis and instrumentation of only parts of the code, and can be used to prevent other parts of the program (that we have not even seen) from violating isolation for the parts of the code we instrument.

Recently, we became aware of work by Baugh et al [5], where they propose the use of fine-grained memory protection to guarantee strong atomicity in hybrid transactional memory (HTM) implementations, and ensure that hardware TM does not access locations accessed by the STM. The idea behind ISOLATOR is similar to their work, with the main difference being that ISOLATOR targets legacy applications that use locks.

References

[1] Intel thread checker. http://www.intel.com/cd/software/products/asmona/eng/286406.htm, March 2008.

[2] Linux memory protection. http://linux.about.com/library/cmd/blcmdl2_mprotect.htm, March 2008.

[3] Memory protection Windows. http://msdn2.microsoft.com/en-us/library/aa366785(VS.85).aspx, March 2008.

[4] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the Conference on Architectural support for Programming Languages and Operating Systems (ASPLOS)*, pages 96–107, 1991.

[5] L. Baugh, N. Neelakanthan, and C. Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2008.

[6] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the Symposium on Operating systems principles (SOSP)*, pages 237–252, 2003.

[7] C. Flanagan and S. N. Freund. Type-based race detection for java. *ACM SIGPLAN Notices*, 35(5):219–232, 2000.

[8] C. Flanagan and S. N. Freund. Automatic synchronization correction. In *Electronic Proceedings of the Conference on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, 2005. https://urresearch.rochester.edu/handle/1802/2083.

[9] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *Proceedings of the Conference on Programming language design and implementation (PLDI)*, pages 1–13, 2004.

[10] D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. An oracle for tolerating and detecting asymmetric races. Technical Report MSR-TR-2007-122, Microsoft Research, 2007.

[11] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing data races on-the-fly. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, pages 54–64, 2007.

[12] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: trading address space for reliability and security. In *Proceedings of the Conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 115–124, 2008.

[13] R. Nagpal, K. Pattabiraman, D. Kirovski, and B. Zorn. Tolerace: Tolerating and detecting races. In *Proceedings of the Second Workshop on Software Tools for Multi-Core Systems (STMCS)*, 2007.

[14] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *Proceedings of the Conference on Programming language design and implementation (PLDI)*, pages 308–319, 2006.

[15] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *Proceedings of the Symposium on principles and practice of parallel programming (PPoPP)*, pages 179–190, 2003.

[16] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Context-sensitive correlation analysis for race detection. In *Proceedings of the Conference on Programming language design and implementation (PLDI)*, pages 320–331, 2006.

[17] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[18] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in stm. In *Proceedings of the Conference on Programming language design and implementation (PLDI)*, pages 78–88, 2007.

[19] J. W. Voung, R. Jhala, and S. Lerner. Relay: Static race detection on millions of lines of code. In *Proceedings of Symposium on The Foundations of Software Engineering (ESEC-FSE)*, pages 205–214, 2007.

[20] E. Witchel, J. Cates, and K. Asanovi. Mondrian memory protection. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 304–316, 2002.

[21] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the Symposium on Operating systems principles (SOSP)*, pages 221–234, 2005.

[22] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iwatcher: Efficient architectural support for software debugging. In *Proceedings of the International Symposium on Computer architecture (ISCA)*, 2004.

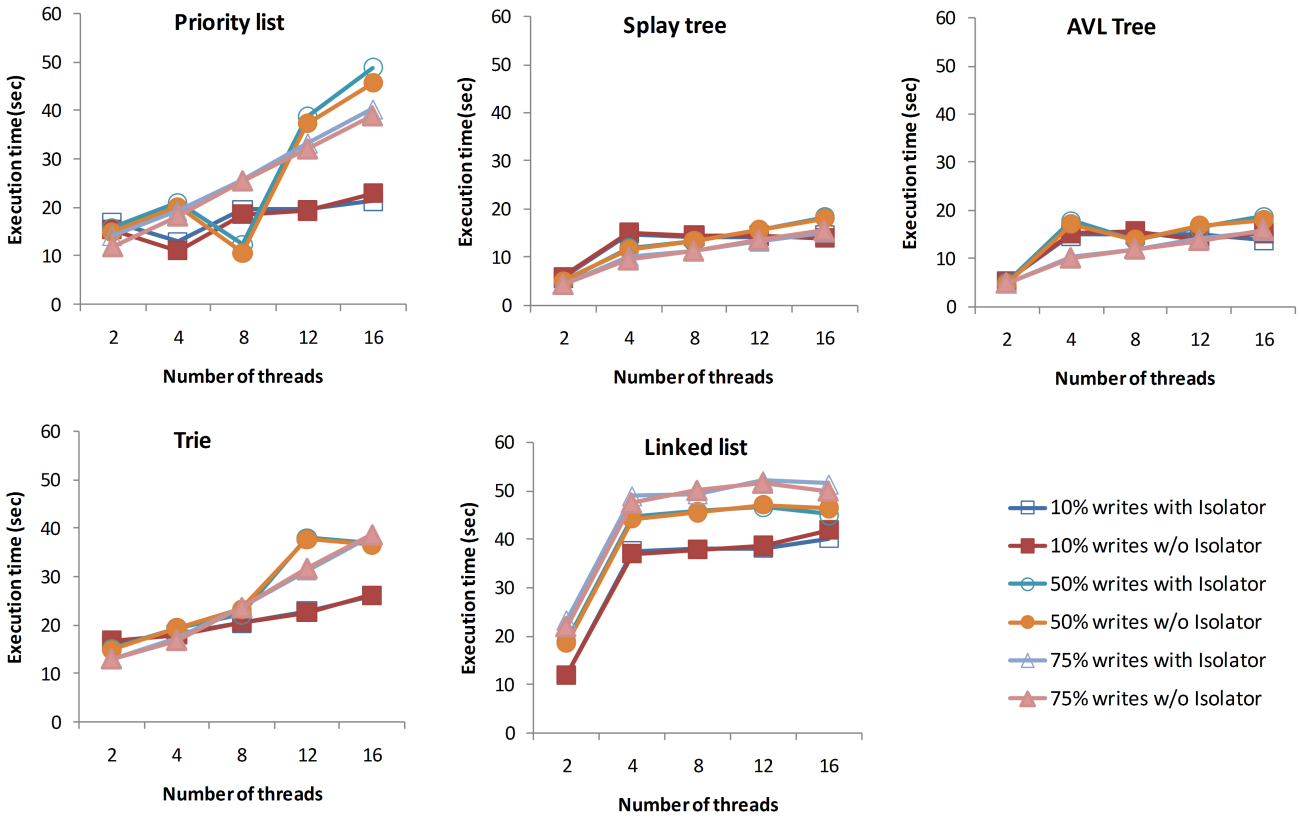


Figure 8: Execution time of various microbenchmarks from the libcprops library with and without ISOLATOR for different number of threads and different ratios of read/write operations.

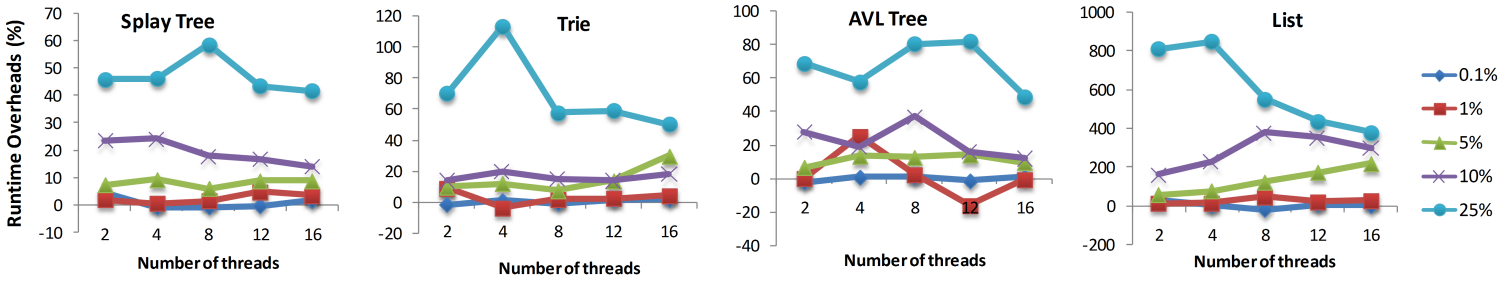


Figure 9: Overheads of ISOLATOR for various microbenchmarks from the libcprops library for different number of well-behaved threads and different fraction of operations from ill-behaved threads.

```
1 /* return longest prefix match for key */
2 int cp_trie_prefix_match(cp_trie *grp,
3  char *key, void **leaf) {
4  void *last = grp->root->leaf;
5  cp_trie_node *link = grp->root;
6  ...
7  lock(grp);
8  while ((map_node = NODE_MATCH(link, key))
9   != NULL) {
10   ...
11  }
12  *leaf = last;
13  unlock(grp);
14  return match_count;
15 }

16 /* removing mappings */
17 int cp_trie_remove(cp_trie *grp, char *key,
18  void **leaf) {
19  int rc = 0;
20  cp_trie_node *link = grp->root;
21  cp_trie_node *prev = NULL;
22  ...
23  lock(grp);
24  /* NULL keys are stored on the root */
25  if (key == NULL) {
26   if (link->leaf) {
27    link->leaf = NULL;
28   }
29   goto DONE;
30  }
31  while ((map_node = NODE_MATCH(link, key))
32   != NULL) {
33   ...
34   if (node->leaf) {
35    node->leaf = NULL;
36    ...
37    cp_trie_node_delete(grp, link);
38    ...
39    break;
40   }
41  }
42  unlock(grp);
43  return rc;
44 }
```

Figure 10: An isolation violation in the `trie` benchmark. The `prefix_match` function reads the `leaf` field of the `root` object without acquiring a lock on the trie. This read might occur while the `remove` function is removing an entry from the trie.

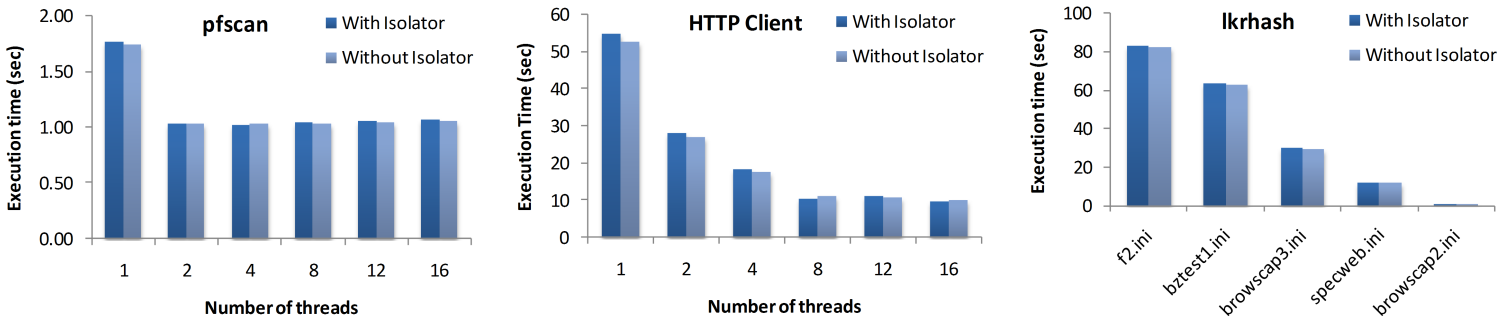


Figure 11: Execution time of three applications with and without ISOLATOR.