# Hardware Support for Enforcing Isolation in Lock-Based Parallel Programs

Paruj Ratanaworabhan
Faculty of Engineering
Kasetsart University
paruj.r@ku.ac.th

Martin Burtscher
Texas State University
burtscher@txstate.edu

Darko Kirovski
Microsoft Research
darkok@microsoft.com

Benjamin Zorn
Microsoft Research
zorn@microsoft.com

## Abstract
When lock-based parallel programs execute on conventional multicore hardware, faulty software can cause hard-to-debug race conditions in critical sections that violate the contract between locks and their protected shared variables. This paper proposes new hardware support for enforcing isolation of critical section execution. It can detect and tolerate races, allowing programs to execute race-free. Our hardware scheme targets the existing large code base of locked-based parallel programs written in type unsafe languages such as C and C++. Our approach works directly on unmodified executables. An evaluation of 13 programs from the SPLASH2 and PARSEC suites shows that the cost of the additional hardware and the impact on the overall execution time is minimal for these applications. Our mechanism is complementary to hardware transactional memory in that it uses similar structures but focuses on enhancing the reliability of existing lock-based programs.

***Categories and Subject Descriptors*** B.8.1 [Reliability, Testing, and Fault-Tolerance]

***General Terms*** Reliability, performance, experimentation

***Keywords*** Race detection and toleration, hardware support for reliability, transactional memory

## 1. Introduction
The advent of multicore hardware puts parallel programming in the spotlight. As single-threaded performance has saturated, parallel programs are the new hope for the continued performance improvement in computing and are expected to increasingly become the norm. Traditional lock-based parallel programming, however, is a difficult undertaking. It does not compose well, suffers from all the problems of sequential programming, and introduces additional sources of errors, e.g., deadlock, atomicity violation, and data races. Researchers have therefore proposed a new paradigm called transactional memory (TM) [1] to tame parallel programming. Although it is a promising technology, TM has not yet matured enough to be widely adopted. At present, parallel programmers still use lock-based synchronization, and there exists a large installed code base of lock-based parallel programs, particularly those written in unsafe languages such as C or C++ with add-on libraries for threading and synchronization. Microsoft is reported to have over 50 million lines of source code in its repository written in this fashion.


*ICS'12*, June 25–29, 2012, Venice, Italy.


This paper focuses on enhancing the reliability of lock-based parallel programs. It proposes a hardware extension to enforce isolation of critical section execution. Normally, when lock-based programs execute, they are vulnerable to race conditions. When a critical section executes, accesses to its protected shared variables may not be mutually exclusive as intended. There may be buggy threads that acquire no locks or the wrong locks, thus concurrently accessing the same "protected" shared variables, which may lead to a race condition. Our proposed scheme, which enforces isolation of critical section execution, can allow a racy program that would otherwise suffer from an isolation violation to execute race-free.

Isolation is a desirable property because it allows programmers to reason about parallel program execution with semantics that match their intuition. With guaranteed isolation, program execution always preserves the data-race-free model (DRF0 to be precise) semantics [2]. This model observes sequentially consistent execution for all synchronization operations while allowing the underlying hardware to be weakly ordered. Violation of isolation can give rise to unexpected behavior. Consider the example in Figure 1. A programmer would expect the value of `baseScript` to be either `default` or `gScript` after the critical section execution. However, Thread 2 can reset `gScript` to `NULL` after Thread 1 has evaluated the `if` condition comparing `gScript` with `NULL` but before executing the `else` body. This unexpected behavior (referred to as Nonrepeatable Read) results from a data race and can crash the program when Thread 1 passes `NULL` to the compile function. With guaranteed isolation, the system forces the resetting of `gScript` to `NULL` to either happen before or after the critical section execution, thus restoring the expected behavior and allowing the execution to proceed correctly. Other unexpected behaviors that could result from isolation violation due to data races in lock-based programs include Intermediate Lost Updates and Intermediate Dirty Reads [3].

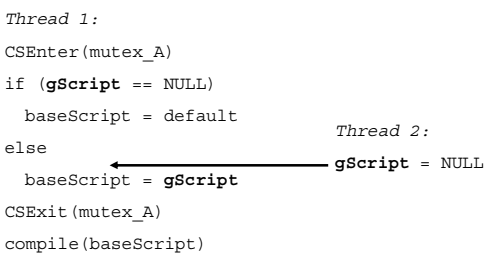



**Figure 1: Isolation violation resulting in a nonrepeatable read**

This paper makes the following contributions.

1. We propose a hardware extension to enforce isolation of critical section execution. This mechanism builds on top of existing private caches in multicore chips. Preexisting lock-based executables can run on this hardware without modification.
2. We evaluate the proposed hardware on parallel applications taken from the SPLASH2 and PARSEC suites and show that the additional hardware cost is minimal for these programs.
3. We describe how hardware TM can readily be retrofitted to support our mechanism, thus making hardware TM useful not only

for new transaction-based code but also for conventional lock-based programs.

## 2. Theoretical Framework

This section investigates the theoretical framework for handling isolation violations. We first consider harmful interleavings between accesses to shared variables of safe threads and those of non-safe threads. A safe thread only accesses shared variables inside of critical sections that are guarded by the correct locks whereas a non-safe thread might access the same shared variables outside of any critical section or use the wrong lock to guard them. Then, we present a mechanism that enforces isolation by preventing all harmful interleavings. We first consider cases where a single variable is protected and accessed in a non-nested critical section. Then, we extend this theoretical framework to cover cases involving multiple variables and overlapped critical sections. The proposed framework is based on ToleRace [4], a software tool and runtime environment that prevents asymmetric data races.

Let r, w, and x denote read, write, and don't-care operations, respectively, and let lower case letters represent accesses of non-safe threads and upper case letters represent accesses of safe threads. r+ denotes a sequence of at least one read and r* indicates zero or more reads. The operators + and * are equally defined for writes and don't-cares. We note that there are only three ways in which a sequence of operations from a single thread can interact with a single variable: by reading it only (r+), by setting its value regardless of its prior (wx*), and by setting its value based upon its prior (r+wx*). For the r+wx* sequence, we assume that w is dependent upon the value retrieved by r.

Suppose that one of the three possible sequences of operations from a non-safe thread slices the operations of a safe thread into two parts. This results in 27 possible interleavings among the three sequences from both threads, i.e., three possible sequences for the first part of the safe thread, times three sequences for the non-safe thread, times three sequences for the second part of the safe thread. Some of these interleavings are harmful and result in an isolation violation. We classify them in terms of race cases in the first two columns of Table 1. Note that the sequences in capital letters represent the safe thread operations and are combinations of two or more of the three sequences mentioned above. For example, R+X* in race case I corresponds to either an R+ or an R+WX* sequence. The notation used for the sequences in race case VI is slightly overloaded; it denotes the set of interleavings that can result from an r+wx* intervening sequence from the non-safe thread that does not already belong to cases IV and V.

**Table 1: Data races that result in an isolation violation and the outcome of the resolution function *f* (see below), attempting to guarantee isolation in the presence of each type of race**

| race type | | short-hand | serializable | *f*(V, V', V") | schedule |
|---|---|---|---|---|---|
| I | X+ wx* R+X* | XwR | Yes | V | XRw |
| II | R*WX* r+ R*WX* | WrW | Yes | V' | rWW |
| III | R+X* wx* WX* | RwW | Yes | V | RWw |
| IV | R+ r+wx* R+ | RrwR | Yes | V | RRrw |
| V | WX* r+wx* X+ | WrwX | Yes | V' | rwWX |
| VI | X+ r+wx* X+ - {IV + V} | RrwW | No | N.A. | N.A. |

Column 3 of Table 1 provides a short-hand notation for each race case. It only includes the access operations that matter. For example, in race case II, isolation violation occurs because the read operations from the non-safe thread observe the "dirty" value produced by the first write instead of the second. Operations that are relevant in this case are the read from the non-safe thread and the two writes from the safe thread that sandwich the read (hence the WrW short-hand notation). The motivation for these short-hand notations will become apparent when we explore the mechanism for ensuring isolation next.

Note that our theoretical framework needs to be concerned only with race condition involving 2 threads. Any race condition among K threads where K is greater than 2 can always be reduced to one of the above race cases between two threads shown in Table 1 [4].

### 2.1 Idealized Mechanism for Ensuring Isolation

The core approach to guaranteeing isolation of critical section execution is to replicate the protected shared state so that the thread that acquires a lock on the shared state has an exclusive copy. This thread continues reading from and writing to this copy until it releases the lock. When the lock is released, the system determines which, if any, isolation violation occurred and decides whether to propagate the value of the local (exclusive) copy or the global copy upon releasing the lock. Isolation can be guaranteed as long as the access operations involved in a race are serializable. If they are not, our system is still able to detect the isolation violation and functions as a race detector in this case. The idealized mechanism for enforcing isolation is described below and shown diagrammatically in Figure 2. Our mechanism is only concerned with enforcing serializable execution and, therefore, cannot fully handle parallel programs that require "stronger" semantics such as linearizability.

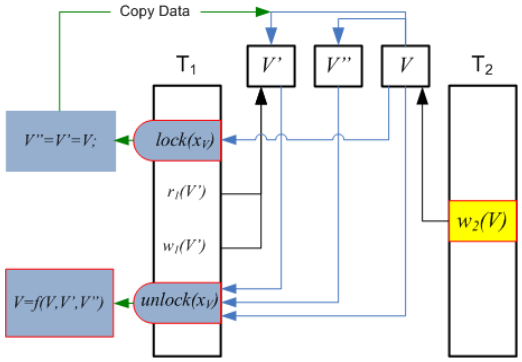



**Figure 2: Ensuring isolation by replicating shared state and propagating appropriate copies using the resolution function *f***

*Initialization and Finalization*: Assume that the binding of lock $x_V$ to shared variable $V$ is known before the critical section in the safe thread $T_1$ is entered and that storage for two additional copies ($V'$, $V''$) of variable $V$ is available. This assumption is only valid with the idealized mechanism. Our implementation relaxes this assumption a great deal.

*Lock (Entry)*: When lock $x_V$ is acquired by $T_1$, copying $V$ to $V'$ and $V''$ ($V''=V'=V$) is performed atomically. Note that the copying and the lock acquire may not be performed atomically.

*Reads and Writes inside the Critical Section*: All instructions in the critical section of $T_1$ use $V'$ instead of $V$. $V'$ is the local copy of $V$ for $T_1$ that cannot be accessed by other threads, not even due to a race. All other threads $T_2$ are unchanged and continue using $V$ for all accesses. Copy $V''$ is not accessed by any thread until $T_1$ exits the critical section.

*Unlock (Exit)*: When $T_1$ exits the critical section by releasing the acquired lock, the system analyzes the content of $V'$, the original value $V''$, and the value $V$ that could have been altered by other threads as a consequence of a race. Depending on the relationship of the values in $\{V, V', V''\}$ and knowledge about the specific race case from Table 1 that has occurred, the resolution function $V = f(V, V', V'')$ defines the value of $V$ after $T_1$ finishes its critical section. The resolution function is executed atomically.

Combining the mechanism outlined above and the knowledge of each race case in Table 1, we can reason about which cases the sys-

tem can handle and enforce isolation. The last two columns of Table 1 summarize the definition of *f* and indicate the resulting serializable schedule of operations for cases where isolation can be enforced. We see that a system with this type of resolution function can guarantee isolation in all race cases except the RrwW case. Here, the safe thread and the interleaved part of the non-safe thread both read the value of the shared variable after the safe thread has entered the critical section, and then they execute in parallel. Both threads see the same value returned by the read, which would not be possible if the safe thread had executed its critical section in isolation. This case is, therefore, not serializable under our theoretical framework. (Section 6 discusses the possibility of tolerating this case with re-execution.)

## 2.2 Multiple Variables and Nested Critical Sections

So far, we have only considered multithreaded, single-variable, non-nested critical section contexts. We now extend our framework to handle all cases, including multiple variables and nested critical sections. Local copies and the resolution function need to be made and executed atomically for multiple variables. Nested critical sections share their local copies with the outermost critical section. However, they have their own resolution function to resolve races for their protected variables.

In general, this mechanism to ensure isolation may lead to inconsistent execution. If it does, the system cannot enforce isolation and reverts to isolation violation detection mode instead. Inconsistent execution arises when the system reorders operations of a non-safe thread such that the operations do not follow their original program order and there are dependencies among the operations that must be observed. This can occur because the proposed mechanism resolves races to each variable independently. Here, dependencies refer to data dependences, i.e., when a write to a given variable depends on a read of another variable.

To understand the general cases involving multiple variables and overlapped critical sections, it suffices to consider a race involving two variables P and Q. Here we provide a summary of the actions involved in each race case. More detailed explanation can be found elsewhere [4].

Let a non-nested critical section protect both variables in a safe thread. In a non-safe thread, let an intervening sequence to P come before an intervening sequence to Q in program order, but the two may overlap each other. Table 2 enumerates all possible P and Q intervening combinations from the non-safe thread. The third column indicates whether the system reorders the intervening operations to P and Q.

**Table 2: Enumeration of intervening sequences to P and Q; trailing x* and r+ of P sequence may overlap with Q sequence**

| P | Q | reorderd by system | dependency from P to Q | enforcing isolation |
|---|---|---|---|---|
| r+ | r+ | No | No | Yes |
| wx* | r+ | Yes | No | Yes |
| r+wx* | r+ | If race IV to P | No | Yes |
| r+ | wx* | No | maybe | Yes |
| wx* | wx* | No | maybe | Yes |
| r+wx* | wx* | No | maybe | Yes |
| r+ | r+wx* | No | maybe | Yes |
| wx* | r+wx* | If race V to Q | maybe | No if reordered, Yes otherwise |
| r+wx* | r+wx* | If race IV to P and V to Q | maybe | No if reordered, Yes otherwise |

Note that the presented framework does not disallow orderings that violate sequential memory consistency (SC). If a programmer wants SC to be honored on weakly-ordered hardware, he or she has to use explicit synchronization operations to restrict the ordering of operations.

# 3. HEI Implementation

Given the ubiquity of multicore chips, it seems expedient to implement our proposed *Hardware for Enforcing Isolation* (HEI) with this microprocessor trend in mind. As we shall see, not much new hardware is needed to realize HEI in a multicore setting. Preexisting multicore components lend themselves well to embrace HEI's functionality. Recall that the mechanism presented in Section 2 involves making thread-local copies of shared variables and operating on those copies. The private data cache in each core naturally serves as thread-local storage. Thus, the central idea behind the design of HEI is to use existing cache components in multicore chips and leverage the already present coherence mechanism that maintains coherency among the private caches of the individual cores. In addition, we want the underlying hardware to be transparent to the running program. The program's executable should be able to run on top of HEI without any modification or user intervention.

The proposed design assumes write-back caches and leverages the MSI snoopy invalidation-based cache coherence protocol. To support the HEI functionality, the cache coherence protocol is augmented with extra states and transitions. We believe that invalidation-based protocols are preferred over update-based protocols and write-back caches are favorable to write-through caches in a multicore environment because both require markedly less inter-core communication bandwidth relative to their counterparts. Hence, we propose the design of HEI per the above assumptions. As the HEI mechanism is tied to bus-based multicore architectures, the scalability of HEI will be limited by it. At present, we target HEI for processors with up to 16 cores.

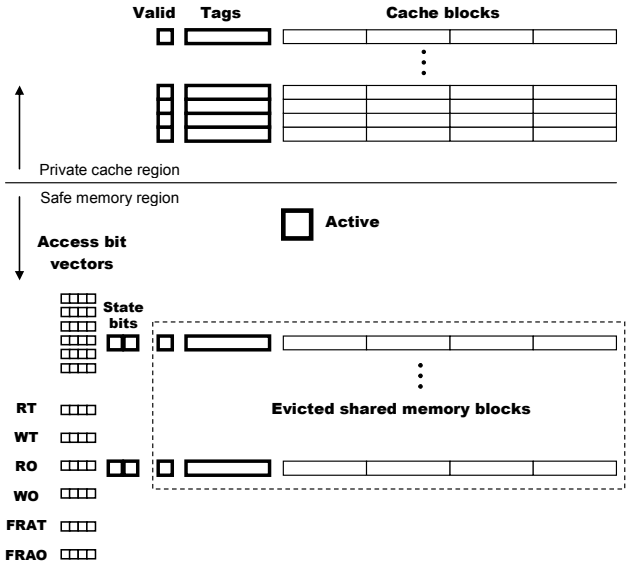



**Figure 3: Basic structures of the proposed hardware for enforcing isolation (HEI)**

## 3.1 Basic Design and Operation

The basic structure of the proposed HEI is shown in Figure 3. The main components represent the safe memory region. This safe memory extends the private cache and is placed at the cache level that handles coherence traffic from other cores. Its structure is similar to that of a victim cache [5]. The design leaves the existing pri-

vate cache unaltered, but the coherence protocol needs to be augmented.

There are four basic components in the safe memory: the evicted cache block region, the active bit, sets of Access Bit Vectors (ABV), and two status bits.

*Evicted cache block region:* The general structure of this component is the same as the private cache. The key difference is that an entry is searched associatively. Basically, a block is transferred from the private cache to this evicted region in the safe memory whenever the CPU of the core in consideration accesses potentially shared memory locations that reside in this block.

*Active Bit:* The active bit specifies whether there are valid entries in the safe memory. When this bit is set, HEI signals to the cache controller to also look for cache entries in this region. Whenever the controller receives a request from the CPU or from the bus, it first searches the private cache area. If a miss occurs, it continues the search in the safe memory before proceeding to the lower levels of the memory hierarchy. For performance reasons, searches may be launched simultaneously in multiple levels.

*Access Bit Vectors (ABV):* There are six cache block bit vectors that serve as bookkeeping mechanism for the types of accesses to each byte in an evicted cache block. They are called: Read This (RT), Write This (WT), Read Others (RO), Write Others (WO), First This Access Read (FTAR), and First Others Access Read (FOAR). The width in bits of each ABV is equal to the size of the cache block in bytes. The purpose of each ABV is explained in Table 3. "This" refers to this core, "Others" to all other cores.

*State Bits:* The two state bits of each entry reflect the state of the corresponding cache block in the private caches of other cores.

**Table 3: Description of each Access Bit Vector**

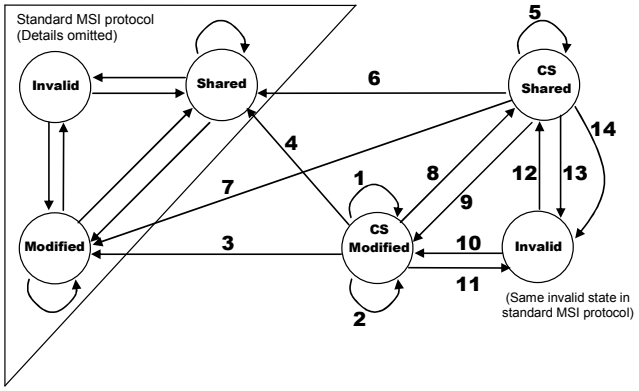
| Access Bit Vectors (ABV) | Descriptions |
|---|---|
| Read This (RT) | Bit i of RT is set when a **read** access from **this** core touches the $i^{th}$ byte |
| Write This (WT) | Bit i of WT is set when a **write** access from **this** core touches the $i^{th}$ byte |
| Read Others (RO) | Bit i of RO is set when a **read** access from the **other** cores touches the $i^{th}$ byte |
| Write Others (WO) | Bit i of WO is set when a **write** access from the **other** cores touches the $i^{th}$ byte |
| First This Access Read (FTAR) | Bit i of FTAR is set when the **first access** that touches the $i^{th}$ byte from **this** core is a **read** |
| First Others Access Read (FOAR) | Bit i of FOAR is set when the **first access** that touches the $i^{th}$ byte from the **other** cores is a **read** |

### 3.1.1 Modifications to the Standard MSI Cache Coherence Protocol

The modification depicted in Figure 4 involves (1) adding a new Modified (M) and a new Shared (S) state called Critical Section Modified (CSM) and Critical Section Shared (CSS), which are almost exact replica of the M and S states, (2) modifying some state transition actions between and within the original M and S states, and (3) adding a new set of state transition actions between the CSM and CSS states that are analogous to those of the M and S states. The details pertaining to the standard MSI protocol are omitted for clarity. Interested readers are referred to, e.g., Culler et al. [6]. The underlined actions in Figure 4 relate directly to HEI's functionality. The key points in this augmented protocol are:

1. It adds the two new states CSS and CSM, as mentioned above.
2. It adds transitions from the Invalid (I) state to the two new states, CSS and CSM (transitions 10 and 12 in Figure 4).
3. It dictates that a read hit while in the CSS state needs to propagate a read message on the shared bus (transition 5). In the S state, a read hit does not generate additional bus traffic and is confined to the private cache of the core in consideration.
4. It requires that a read hit and a write hit while in the CSM state place corresponding read and invalidation messages on the bus (transitions 1 and 2). The corresponding transitions in the M state do not propagate such read and invalidation messages.

The changes noted in items 3 and 4 above allow the safe memory in a given cache to snoop for intervening reads or writes that would otherwise be confined to the private caches of the other cores. Note that the cache coherence engine only runs in the private cache region, never in the safe memory region.




1: CPU read hit; **place read message on bus**
2: CPU write hit; **place invalidation message on bus**
3: CPU write miss; **place write miss on bus and write back**
4: CPU read miss; **place read miss on bus and write back**
5: CPU read hit; **place read message on bus**
6: CPU read miss; **place read message on bus**
7: CPU write miss; **place write miss on bus**
8: **Read miss for block received; write back; place read message on bus**
9: CPU write hit; **place invalidation message on bus**
10: CPU write miss and block found in safe memory; **place write miss on bus**
11: **Write miss for block received; write back**
12: CPU read miss and block found in safe memory; **place read miss on bus**
13: **Write miss for block received**
14: **Invalidation for block received**

**Figure 4: Augmenting the MSI protocol to enable HEI; bus requests are in bold; CPU requests in normal font; underlined actions are the key differences from the MSI protocol**

### 3.1.2 Basic Operation

Initially, the safe memory in each core is not active; all the ABVs and the valid bit for each block in the evicted cache region are cleared. When a core detects that:

1. the program starts to execute a critical section, i.e., acquires a lock, and
2. accesses possibly shared memory locations, i.e., non-stack locations,

the hardware evicts the cache block being accessed from the private cache region and sends the block to an entry in the evicted cache region in the safe memory. Detecting the first condition requires hardware that recognizes, for example, the test&test&set lock idiom similar to that used in SLE [7]. Detecting the second condition requires hardware that filters out instructions whose addressing mode is indirect off the stack and frame pointers. We assume that shared variables are accessed via registers other than these two registers.

Once the block is in the safe memory, HEI broadcasts an invalidation message to nullify all other copies of this block in this and all other cores. It also sets the Active Bit in the safe memory.

When the Active Bit is on, there are two cases to consider:

1. subsequent accesses to this block coming from this core, and
2. subsequent accesses to this block coming from another core.

In the first case, the following happens:

1. The accesses never bring the block back into the private cache; they always take a miss in the private cache and use the block in the safe memory. As a consequence, whenever the Active Bit is

on, any misses in the private cache need to search the safe memory for a matching entry before going to the next level of the memory hierarchy.
2. The accesses set the appropriate bits in each of the RT, WT, and FTAR bit vectors. Recall that these sets of ABVs are for this core and are updated based on this core's requests.

In the second case, we may potentially have a race. The following happens:

1. The accesses are subject to the augmented cache coherence protocol described above as they are accessing the private cache.
2. The (active) safe memory in each core snoops on the bus for messages that may be relevant and sets the appropriate bits in each of the RO, WO, and FOAR vectors.

Upon exiting from a critical section, HEI resolves races according to the table in Figure 5, which is based on Table 1 in Section 2. Races are resolved at byte granularity using the information from the six Access Bit Vectors. The information from the RT, WT, RO, and WO bit vectors are sufficient to infer some race types. However, for the entries labeled A, B, and C in Figure 5, we need the information stored in the FOAR and FTAR bit vectors to pin down the exact race type. To resolve the race correctly, HEI retrieves the latest block copy that resides outside of the safe memory. This block should be in one of the three states I, CSS, or CSM. To do so, the hardware consults the state bits. If the block is in either the I or CSS state, it obtains the latest copy from the main memory; otherwise, it requests the block from the core that has the block in the CSM state. Upon receiving the latest copy, the hardware inspects the block byte by byte. Based on the race resolution in Figure 5, it determines for each byte whether to pass on the value from the block in the safe memory or from the outside block.

After successfully resolving any possible races for each byte in a block, HEI processes the next valid block in the safe memory until all valid blocks in the evicted cache region have been processed. Then, it resets the valid bits, the ABVs, and the Active Bit. Note that the hardware needs to ensure that this process happens atomically. A straightforward way of ensuring this is to lock the bus so that only bus transactions from the core that is resolving the race are allowed during this time period. As we will show in the next section, critical section memory accesses constitute only a very small fraction of the total memory accesses, so this simple but seemingly costly way of ensuring atomicity should work well in practice. Furthermore, for most critical sections, the number of entries in the evicted cache region is small.

When encountering the RrwW race case that HEI cannot tolerate, the hardware reverts to detection mode and terminates program execution. The HEI hardware allows logging of quite detailed information including addresses, type, and sequence of accesses involved in the race. Such information can be a helpful debugging aid to identify the cause of the data race offline.

### 3.1.3 Nested and Overlapped Critical Sections

The HEI mechanism described thus far is designed to handle non-nested non-overlapped critical sections. It may seem like the simplest way to extend the basic hardware to cope with nested critical sections is to flatten them and to add a counter that keeps track of the nesting level. Unfortunately, this simple scheme does not faithfully preserve the original lock-based program semantics (see the example in Section 7) and does not correctly handle overlapped critical sections.

To remedy this situation, we need the ability to perform race resolution at every critical section exit. Thus, we have to associate a lock variable with each access (the term lock variable here refers to the memory location used in the test&test&set operation, which marks the start of a critical section). Since HEI resolves races at byte granularity, having a lock variable associated with each byte in a cache block is potentially costly. If we take a slightly less ambitious approach and decide not to support arbitrary levels of nesting or overlapping, we can scale back the hardware considerably. Based on the benchmark programs we studied, nesting levels greater than two occur rarely. Hence, we believe supporting up to four nesting levels will accommodate the majority of lock-based programs.

| WT | RO | WO | Race type (if any) | Serializable execution enforced By HEI | |
|---|---|---|---|---|---|
| 0 | 0 | 0 | No race | None | |
| 0 | 0 | 1 | No race | None | |
| 0 | 1 | 0 | No race | None | |
| 0 | 1 | 1 | No race | None | |
| 1 | 0 | 0 | No race | None | |
| 1 | 0 | 1 | No race | None | |
| 1 | 1 | 0 | WrW | rWW | |
| 1 | 1 | 1 | WrwX, No race | rwWX, None | A |
| 0 | 0 | 0 | No race | None | |
| 0 | 0 | 1 | XwR | XRw | |
| 0 | 1 | 0 | No race | None | |
| 0 | 1 | 1 | RrwR, XwR | RRrw, XRw | B |
| 1 | 0 | 0 | No race | None | |
| 1 | 0 | 1 | XwR | XRw | |
| 1 | 1 | 0 | WrW | rWW | |
| 1 | 1 | 1 | WrwX, RrwW, XwR, RwW | rwWX, Not serializable, XRw, RWw | C |

Case A:

| FOAR | Race type |
|---|---|
| 0 | No race |
| 1 | WrwX |

Case B:

| FOAR | Race type |
|---|---|
| 0 | XwR |
| 1 | RrwR |

Case C:

| FTAR | FOAR | Race type |
|---|---|---|
| 0 | 0 | XwR |
| 0 | 1 | WrwX |
| 1 | 0 | RwW |
| 1 | 1 | RrwW |

**Figure 5: HEI resolution table based on Access Bit Vectors**

### 3.1.4 Fallback Mechanism

If the HEI resources are exhausted while the program is inside of a critical section, there are two possibilities to consider: (1) a race has already been detected, or (2) a race has not yet occurred. In the former case, the fallback is simply to stop program execution and report the race. Resolving the race in the middle of a critical section is not defined in HEI. The focus of the fallback mechanism, therefore, is on the second case where program execution must be allowed to continue. Abruptly stopping the program is not an acceptable solution in this case. After all, the introduction of HEI should not interfere with legitimate race-free runs of programs. The basic idea behind the recovery is to reconcile the copies of each of the active cache blocks in the safe memory region and to update the shared memory before program execution resumes. Consider each cache block residing in the safe memory region. Instances of these cache blocks may exist in (1) one other processor in the CSM state or (2) one or more other processors in the CSS or invalid state. The recovery machinery needs to flush all instances of these cache blocks to memory and make sure that each byte in the blocks holds the latest value from the last write to this byte. To do this correctly, the machinery only needs to consult the two write access bit vectors, WT and WO, associated with each cache block in the safe memory region. To ensure atomicity of the recovery process, the shared bus is locked until the flushing is completed.

## 4. Discussion

This section compares and contrasts the two other isolation enforcement techniques most closely related to ours, transactional memory (TM) and PACMAN [8]; the former is a well-studied technique whereas the latter is quite new.

*General approach:* Pacman is a hardware technique that tolerates races by stalling the unsafe threads whose accesses interfere with those in the safe thread that is executing in a critical section. At the heart of Pacman is a centralized piece of hardware (which can be made distributed) called SigTable. An entry in the SigTable stores the signature of the addresses being touched by the safe thread as well as its PID. Pacman monitors accesses from all other threads

through coherence transactions to check whether their addresses are in the stored signature. If so, there is a potential race and the access requests from other threads are nacked, thus effectively stalling those threads. Pacman's hardware is placed at the network level so it is unintrusive in the sense that it minimally interferes with the normal processor-cache operations. Pacman could also be used as a race detector, but it is not designed for this purpose as the SigTable holds limited access information and the use of signatures may alias addresses that could result in false positives.

HEI is more intrusive as it is placed directly at the cache level that handles coherence transactions. HEI is distributed among each core's private cache. There is no explicit stalling of threads in HEI; race toleration occurs only at the exit of safe threads' critical sections. HEI readily supports detection and toleration modes as the safe memory contains more access information down to byte granularity and uses full addresses. If it is desirable to have Pacman's style stall-based tolerance, HEI can be modified to emulate it by nacking all external requests that hit in the safe memory.

The HEI mechanism is similar to TM with a lazy versioning policy and lazily detecting conflicts among transactions. However, it is not based on optimistic synchronization as TM is; there is no notion of abort-and-rollback, nor is there a need for contention management. Whereas handling side effect operations and nested transactions are still open issues with TM, HEI handles all I/O operations as well as overlapped critical sections transparently, preserving the semantics of the original lock-based program.

Pacman can also be viewed as pessimistic TM that has no notion of versioning or resolving conflicts (as it prevents them in the first place). Hence, the TM issues mentioned above become non-issues in Pacman as well.

*Isolation enforcement capability:* there exist TM mechanisms that guarantee isolation among transactions and between transactions and non-transactions [9]. Pacman, being a pessimistic TM, can also fully enforce isolation in lock-based code. HEI, on the other hand, fails to enforce isolation when it encounters RrwW race cases. In Section 6, we outline a modification to the standard HEI implementation that can tolerate RrwW races.

*Speculative execution:* TM has this at heart, and, hence, it needs sophisticated techniques to deal with rollback and contention management. In addition, there are open issues with I/O operations. Pacman and HEI never perform speculative execution, so both never have non-undoable actions like I/Os. Encountering of conflicting accesses result in a thread stall in Pacman whereas HEI allows all threads involved in a conflict to progress if it can tolerate the race or terminates the execution and reports the race otherwise.

*Starvation and deadlock:* this is a major issue in TM that often needs sophisticated contention management to resolve. It is also an issue in Pacman. A simple deadlock case involves two threads using different locks to guard the same protected shared variables, nacking each other while both execute inside of the critical sections. Pacman requires a rather elaborated scheme to deal with this situation. HEI, on the other hand, never encounters such an issue. It always makes forward progress when it can tolerate the race or terminates program execution and reports the race if it cannot. Even when multiple threads exhaust their HEI hardware and need to perform flushing, deadlock cannot occur in HEI as the only resource each thread needs is the bus, which acts as a serialization point that each thread needs to acquire through arbitration before flushing. Once a thread acquires the bus, it does not need to wait for any other resources to perform flushing.

*Hardware complexity:* it is clear that optimistic concurrency, which needs to satisfy both performance and isolation goals in TM, requires more complex hardware than both Pacman and HEI. The major components in Pacman and HEI, the safe memory and the SigTable, are both regular, table-like structures. The SigTable is a bit more complicated as it works with encoded addresses. However, it is the control logic that makes Pacman more complex than HEI. Pacman's control logic needs to handle deadlock and requires some intrusive hardware addition to be made to the cache hierarchy. In addition, placing the SigTable at the network level makes it tricky to reason about atomicity.

*Scalability:* HEI has limited scalability as its mechanism is tied to bus-based schemes. Pacman can be made more scalable with a distributed SigTable. Scalable TM has been demonstrated by, for example, Chafi et al. [10].

*Executables:* Pacman and HEI operate largely on unmodified lock-based executables. They can run code that uses lock-free data structures that contain intentional races and treat them as harmless since these races involve accesses to shared data only outside of lock regions. TM, on the other hand, operates on transactional code. Lock-based code can run on TM hardware, but it first needs to be transactified, i.e., converted from lock-based to transaction-based code. This conversion, however, is not trivial [11].

# 5. Evaluation

## 5.1 Benchmarks

We use 13 applications from the SPLASH2 [12] and PARSEC [13] benchmark suites for our evaluation. The eight programs from the SPLASH2 suite were chosen per the minimum set recommended by the suite's guidelines. They are *cholesky*, *fft*, *lu*, *radix*, *barnes*, *ocean*, *radiosity*, and *water*. We replaced the SPLASH2 suite's PARMAC macros with a pthreads library implementation. For each of the eight programs, the default inputs were used. We selected the five programs from the PARSEC suite that use the pthreads library. They are *dedup*, *facesim*, *ferret*, *fluidanimate*, and *x264*. The PARSEC suite aims to provide up-to-date multithreaded programs that focus on workloads in recognition, mining, and synthesis. They are run with the simlarge inputs.

## 5.2 System and Compiler

All benchmarks are compiled and run on a 32-bit system with a four-core 2.8 GHz Xeon CPU with 16 kB L1 data cache per core, a 2 MB unified L2 cache, and 2 GB of main memory. The operating system is Red Hat Enterprise Linux Release 4 and the compiler is gcc version 3.4.6. We compiled the SPLASH2 and PARSEC programs per each suite's guideline.

## 5.3 Quantitative Assessment of HEI

This subsection investigates quantitatively the characteristics of the safe memory and the potential overhead when running the benchmark applications under HEI. We first focus on the amount of hardware required to protect the critical sections. Then, we look at the frequency of critical section executions and the effect on the execution time of each application given certain overheads incurred by HEI.

Figure 6 shows, for each application, the fraction of critical section executions that are fully covered by HEI when the block size is 64 bytes and the number of safe memory entries is 32, 64, or 128. HEI fully covers a critical section execution when, for a given amount of hardware, it does not cause overflow in the safe memory. For 64 entries, over 93% of the critical section executions in each application are covered. When doubling the number of entries to 128, the coverage increases to 99%. Figure 7 shows, for varying block sizes, the average number of safe memory entries each application operates on. For a 64-byte block, all applications except *dedup* and *ferret* often process less than 8 entries. To obtain the result in Figures 6 and 7, we use our in-house cache simulator that implements the modified MSI protocol as described in the previous section. The simulator is implemented as a Pintool plug-in [14] that runs with the Pin dynamic binary instrumentor.

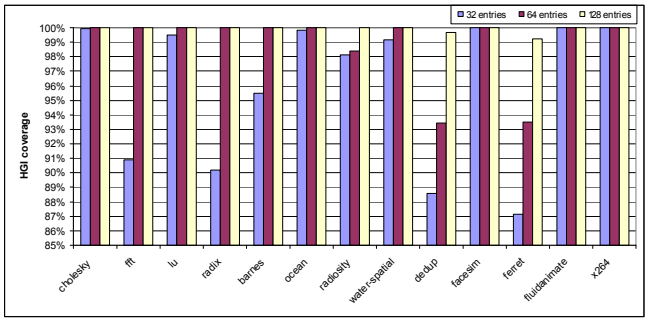



**Figure 6: HEI coverage of critical section execution for 64-byte blocks (y-axis not zero based)**

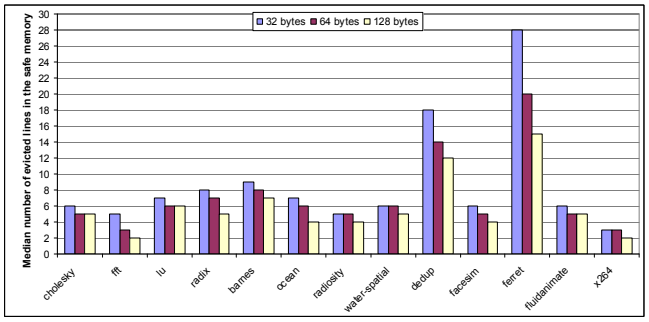



**Figure 7: Median number of entries in the evicted cache of the safe memory required by HEI for different block sizes**

**Table 4: Critical section execution characteristics of each benchmark application**

| | Avg. Uniq. Mem. Loc. Touched | % Mem. Access | % Time 4-core | % Time 8-core | % Time 16-core |
|---|---|---|---|---|---|
| **cholesky** | 14.34 | 0.06% | 0.22% | 0.44% | 0.88% |
| **fft** | 6.83 | < 0.001% | 0.004% | 0.007% | 0.014% |
| **lu** | 20.93 | < 0.001% | 0.02% | 0.03% | 0.06% |
| **radix** | 22.57 | < 0.001% | 0.001% | 0.003% | 0.006% |
| **barnes** | 43.05 | 0.43% | 10.94% | 19.72% | 32.95% |
| **ocean** | 20.99 | < 0.01% | 0.03% | 0.06% | 0.12% |
| **radiosity** | 4.92 | 0.49% | 23.62% | 38.21% | 55.30% |
| **water-spatial** | 15.70 | < 0.01% | 0.02% | 0.03% | 0.07% |
| **dedup** | 121.30 | 0.73% | 4.47% | 8.55% | 15.76% |
| **facesim** | 15.40 | < 0.01% | 0.07% | 0.15% | 0.29% |
| **ferret** | 92.76 | 1.22% | 6.93% | 12.96% | 22.94% |
| **fluidanimate** | 15.00 | 1.36% | 4.32% | 8.29% | 15.30% |
| **x264** | 6.47 | < 0.01% | 0.01% | 0.02% | 0.03% |

From the simulation result in Figures 6 and 7, we see that covering the majority of critical section execution does not require an excessive amount of hardware. In addition, most critical section executions exercise only a small fraction of the hardware.

Next, we look at the characteristics of critical section executions with a focus on memory accesses. Table 4 summarizes the results. In the first data column, we show the average number of memory locations touched within a critical section. These are the possibly shared locations accessed by each thread. Recall from the previous section that we exclude all stack accesses via frame and stack pointers. Most applications access no more than 20 locations on average. The number for *radix* is slightly above 20 and for *barnes* it is 43. *ferret* and *dedup* perform significantly more accesses than the other applications. These two benchmarks execute loops inside of critical sections whereas all the other programs loop over their critical sections. This access information is in line with the results for the amount of hardware and the coverage shown in Figure 6.

The second column of Table 4 shows the fraction of memory accesses inside critical sections as a percentage of all memory accesses. As we can see, accesses inside of critical sections are relatively infrequent events. The percentage for all the benchmarks except *ferret* and *fluidanimate* is less than 1%. After all, parallel programs try to avoid serial execution, the part where Amdahl's law limits performance. The quantities in the first and second columns are obtained from dynamic program analysis using Pin; we dynamically instrumented all machine instructions – inside and outside of critical sections – that have at least one memory operand not accessed via the stack or frame pointer.

The third, fourth, and fifth columns show the fraction of total execution time spent in critical sections for 4, 8, and 16-core machines. We set the limit at 16 cores as we do not project HEI to be applicable beyond this point. To measure the time, we inserted a call to the clock_gettime() function at each critical section entry and exit point. In case of nested critical sections, we flatten them and measure the time for the outermost critical section. clock_gettime() provides an interface to a high-resolution timer with roughly nanosecond accuracy. To get accurate timing measurement, we configure all the programs to execute with only a single thread. Then, we estimate the time spent outside of the critical sections by assuming that those portions can be fully parallelized. Hence, if each program is to run on an $n$-core machine, this portion of time is divided by $n$. The fraction of time spent in the critical sections is calculated as:

$$\frac{(time\ spent\ in\ CS)}{\frac{(time\ spent\ outside\ CS)}{n} + (time\ spent\ in\ CS)}$$

We see that only *radiosity*, *barnes*, *dedup*, *ferret*, and *fluidanimate* have a noticeable fraction of time in critical sections, ranging from around 4% to 55%. All the other applications spend less than 1% of their execution time in critical sections. As expected, when the number of cores increases, the fraction of time inside of critical sections becomes more significant, especially when we assume that the code outside of critical sections can be fully parallelized.

The results obtained so far seem to suggest that the HEI overhead should not be significant. The safe memory hardware is small and most of the time only a small fraction of it is exercised. In addition, memory accesses in critical sections are relatively infrequent and most parallel applications spend little time in critical sections.

## 5.4 Qualitative Argument for HEI Overhead

This subsection offers qualitative arguments as to why the HEI overhead is unlikely to be excessive. We focus on two factors: the HEI memory bandwidth requirement and the HEI overhead when executing inside of critical sections.

To understand the bandwidth requirement, we need to look at the modified MSI protocol in Figure 4. The bottom line is that a system with HEI consumes about as much bandwidth as a system without it. The only time that a system with HEI generates extra coherence messages from cores not executing in critical sections is when the other cores have cache blocks in either CS modified or CS shared states (Figure 4 messages 1, 2, and 5). Since this usually signals a race, it should occur rarely.

Next we look at HEI's critical section execution overhead. The performance penalty of HEI comes primarily from two sources, accessing safe memory and resolving races. As the result in Section 5.3 indicates, the majority of critical section executions exercises fewer than 8 entries in the safe memory. Therefore, the associative search associated with each safe memory access should require only a couple of cycles more than a normal L1 access. This may be a significant overhead for accesses that would have hit in the L1. However, such an access is more likely to first miss in the L1 (after all, it is a shared variable), in which case the overhead becomes insignificant.

The resolution hardware is made of combinational logic that requires only simple AND gates followed by OR gates to process the access bit vector and "filter" bytes in a safe memory entry to pass on. It should take less than one cycle per resolution. If we need faster race resolution, more hardware can be added to resolve races in parallel.

# 6. Enforcing Isolation for the RrwW Race with Re-execution

As described, HEI is not able to guarantee isolation for the race case RrwW. However, this case can be serialized as rwRW if we have the ability to re-execute the RW operations in the safe thread after having seen the intervening rw operations from the non-safe thread. Assuming we have a mechanism for re-execution, guaranteeing isolation for this race case becomes viable. This section quantifies the potential to enhance HEI through critical section re-execution on the benchmark programs.

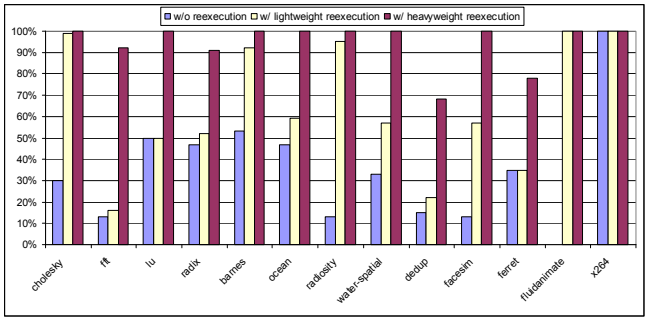



**Figure 8: Worst-case measurement for fraction of shared accesses where isolation can be guaranteed**

First, consider the re-execution capability of a particular critical section execution. We consider three categories: non-reexecutable, lightweight re-executable, and heavyweight re-executable. A non-re-executable critical section always involves interactive I/O operations that cannot be undone or repeated. A lightweight re-executable critical section does not have calls to library functions that invoke operating system services. Therefore, it can be re-executed by checkpointing the register and memory reads in the critical sections in user code and address space. A heavyweight re-executable critical section contains calls that invoke operating systems services that can potentially be undone by checkpointing the full memory state in both user and OS space, including calls to threading and synchronization operations in the pthread library.

We then measure the fraction of shared accesses where HEI can guarantee isolation in each benchmark application. The results of this experiment for all critical section executions are shown in Figure 8. This is the worst-case measurement, i.e., at least this fraction can be guaranteed. The reading at 50% means that, on average, for a given memory location accessed inside of critical sections, at least half of the accesses can be guaranteed isolation, i.e., they can be serialized with racing accesses.

The first bar in Figure 8 shows, for each application, the measurement without any re-execution. In essence, this bar indicates the fraction of all accesses to a given shared memory location inside of critical sections that do not start with a read that is later followed by a write (which is indicative of RW operations). Recall that when this is the case, it guarantees that race case RrwW cannot happen, and, hence, isolation can always be guaranteed. We find two extreme cases. *fluidanimate* has zero potential whereas *x264* has 100%. *barnes* hovers just above 50% and most of the benchmarks have less than 50% potential. When considering lightweight re-execution, the potential shoots up dramatically in a number of applications, particularly in *fluidanimate* where it jumps from 0% to 100%. Other benchmarks with large improvements are *cholesky*, *barnes*, *radiosity*, and *facesim*. Expectedly, when considering heavyweight re-execution, we see 100% potential in all benchmark applications except those that contain non-reexecutable critical sections, i.e., *fft, radix, dedup, and ferret.*

# 7. Enabling HEI with Hardware TM

Anticipating the arrival of hardware TM [15], it seems expedient to fit the HEI mechanism into this framework. This section investigates how to accommodate the HEI functionality in this way. A related work albeit with a different focus by Gupta et al. [16] leverages the conflict detection in TM to detect data races.

Because piggybacking on hardware TM requires a minimal increase in hardware budget and complexity, processor manufacturers may be inclined to support both HEI and TM at the same time with the former geared towards enhancing reliability of existing lock-based programs and the latter towards transaction-based parallel applications.

As HEI is designed to work transparently with existing lock-based program binaries, a pre-processing system is needed to "transactify" lock-based programs into transaction-based programs where lock-based critical sections are transformed into atomic blocks marked by transaction constructs recognized by the hardware TM. Note that essentially all this system does is replace lock-based constructs with some constructs recognized by the TM hardware. No knowledge about HEI or TM semantics is incorporated at this point. An example of such a system is HyTM [17]. The following are some necessary conditions for HEI to function on top of hardware TM.

*1. Use deferred (a.k.a. lazy) update:* To be compatible with HEI, the hardware TM must not modify the shared data directly. Most proposed hardware TMs that buffer updates in private caches satisfy this requirement. However, there exist hardware TM systems such as Wisconsin's LogTM [9] that do not. LogTM uses an eager update protocol that is not compatible with HEI.

*2. Support open-nested semantics:* To accommodate nested critical sections and follow their lock-based program semantics faithfully, the hardware TM needs to provide support for open nesting. Simple flattening is not sufficient to embrace nested critical sections as it may prevent forward progress in the original lock-based program. To make this more concrete, consider the following (somewhat contrived) example.

```
P = 4;
Lock(mutex_X);
Q++;
Lock(mutex_Y);
P++;
Unlock(mutex_Y);
do {} while (P < 5);
Unlock(mutex_X);
```

In this example, the lock variable mutex_X protects the update to variable Q whereas mutex_Y protects the update to variable P. The mutex_Y critical section is nested in the outer mutex_X critical section and, just before exiting from the outer critical section, there is a do-while loop that spins on the condition (P < 5). If the underlying hardware TM uses a deferred update protocol and flattens nested transactions, when the example code is "transactified", it will keep spinning on the do-while loop after executing the inner critical section as the updated value of the shared variable P will not be made visible until the outer critical section completes.

Unfortunately, recently proposed hardware TM systems handle nested transactions by flattening them. Therefore, for the current generation of TM hardware to accommodate nested critical sections in the way HEI requires, it may need assistance from special hardware as outlined in Section 3.1.3.

*3. Disable concurrent transactions:* For HEI, there is no notion of concurrent execution of critical sections. To enforce this condition, the hardware TM must disallow concurrent transactions that originate from the same mutex variable. Disabling a given transaction

should not be hard to accomplish as the TM hardware already contains structures to track a new transaction that is about to execute in the middle of an outstanding transaction. However, recognizing which transaction to disable requires special treatment. During the "transactification", an atomic block derived from a corresponding critical section needs to be augmented with the mutex variable that is associated with the critical section. When the TM hardware processes a given atomic block, it checks whether the atomic block is generic, i.e., coming from purely transaction-based code, or derived, i.e., coming from lock-based code. It imposes no restriction on the concurrent execution of the former, but prohibits concurrent execution of the latter.

*4. Detect conflicts from non-transaction executions:* If the hardware TM is only concerned with conflicts among transactions, it needs to be augmented to detect conflicts from non-transaction execution to support HEI. The contention management hardware is not needed when in HEI mode because HEI always guarantees forward progress. However, additional resolution hardware is needed. This is simple hardware whose primary function is to propagate memory writes based on the type of conflict detected.

# 8. Related Work

## 8.1 Race Toleration

Krena et al. [18] propose two schemes to dynamically heal data races for Java programs. In one scheme, they reduce the probability of races happening by forcing threads that are about to cause racy accesses to yield. This is done at the byte-code level through yield() calls. In the other scheme, they add extra locks to some common code patterns that are likely to result in races. Rajamani et al. [19] propose a run-time system called Isolator that enforces isolation through page protection. The idea is to protect the pages containing shared variables (that are protected by a lock) so that accesses to them can be intercepted. Then, accesses to those variables that observe the proper locking discipline are redirected to a local copy of the corresponding page. Any improper access will be to the original page and hence raise a page protection fault. Similarly, Abadi et al. [20] use page-level protection to guarantee strong atomicity in software transactional memory. Lucia et al. [21] tolerate some degree of atomicity violation with implicit atomicity by grouping consecutive memory operations into atomic blocks. To avoid concurrency bugs, Yu and Narayanasamy [22] constrain thread interleavings in production runs with Lifeguard Transactions (LifeTx). Legitimate interleavings are determined during the testing phase and LifeTx avoids untested interleavings, which are the sources of concurrency bugs.

## 8.2 Hardware Transactional Memory

HEI is analogous to hardware TM, which was first proposed by Herlihy and Moss [1]. Their scheme works on a fully associative transaction cache and leverages the cache coherence protocol to detect conflicts. Rajwar and Goodman proposed Speculative Lock Elision (SLE) [7], a form of optimistic synchronization that elides locks in parallel programs and, thus, allows multiple critical sections that were mutually exclusive to execute concurrently. Their follow-up work, Transactional Lock Removal (TLR) [23], extends SLE to address issues with starvation as well as transactional semantics. The broad introduction of multicore chips around the year 2004 reinvigorated research in hardware transactional memory. Transactional Coherence and Consistency (TCC) from Stanford [24] is a hardware TM scheme that provides strong isolation and employs deferred policies for both update operations and conflict detections. In contrast to TCC, LogTM [9] from Wisconsin uses eager update and eager conflict detection protocols. LogTM's authors argue that commits happen more often than aborts so adopting all eager policies make the common cases fast. Ceze et al. [25] describes a hardware TM that does not rely on cache coherence to track and detect conflicting memory accesses. All the hardware TM schemes described thus far are bounded hardware TM. Cases for unbounded TMs have been described by Ananian et al. [26] and Rajwar et al. [27].

# 9. Conclusions

This paper presents HEI, a hardware technique for enforcing isolation in lock-based parallel programs that leverages existing multicore chip components and the cache coherence protocol. We especially target programs written in type unsafe languages, which are prevalent in today's parallel programming world. HEI runs the original, unmodified executables and allows a racy program to execute race-free whenever it can enforce isolation; in the cases where it cannot, it reverts to detecting isolation violations. We evaluate HEI on 13 programs from the SPLASH2 and PARSEC suites and show the additional hardware cost and the overhead to be small. With increasing silicon real estate on a chip, we believe HEI to be a good hardware investment to enhance reliability of lock-based parallel program execution.

# 10. Acknowledgement

We would like to thank the anonymous reviewers for their insightful comments on this paper. The Computer Systems Laboratory at Cornell University provided some of the computing resources used to obtain the results for this work. Martin Burtscher is supported by grants and gifts from NVIDIA and Intel. Paruj Ratanaworabhan is supported by grants from the Faculty of Engineering, Kasetsart University as well as a gift from the KSIP laboratory.

# 11. References

[1] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures" *International Symposium on Computer Architecture*, 1993.

[2] S. V. Adve and M. D. Hill, "Weak Ordering - A New Definition" *International Symposium on Computer Architecture*, 1990.

[3] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. Hudson, K. Moore, and B. Saha, "Enforcing Isolation and Ordering in STM" *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[4] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman, "Detecting and Tolerating Asymmetric Races" ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2009.

[5] N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers" *International Symposium on Computer Architecture*, 1990.

[6] D. Culler, J. P. Singh, and A. Gupta, Eds., *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998.

[7] R. Rajwar and J. R. Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution" *International Symposium on Microarchitecture,* 2001.

[8] S. Qi, N. Otsuki, N. Nogueira, A. Muzahid, and J. Torrellas, "Pacman: Tolerating Asymmetric Data Races with Unintrusive Hardware" *International Symposium on High Performance Computer Architecture*, 2012.

[9] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: Log-Based Transactional Memory" *International Symposium on High-Performance Computer Architecture,* 2006.

[10] H. Chafi, J. Casper, B. Carlstrom, A. McDonald, C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, "A Scalable, Non-Blocking Approach to Transactional Memory" *International Symposium on High Performance Computer Architecture,* 2007.

[11] C. Blundell, C. Lewis, and M. Martin, "Deconstructing Transactional Semantics: The Subtleties of Atomicity" *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2005.

[12] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations" *International Symposium on Computer Architecture*, 1995.

[13] C. Bienia, S. Kumar, J. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications" *International Conference on Parallel Architectures and Compilation Techniques,* 2008.

[14] C. 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" *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[15] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early Experience with a Commercial Hardware Transactional Memory Implementation" *International Conference on Architectural Support for Programming Languages and Operating Systems,* 2009.

[16] S. Gupta, F. Sultan, S. Cadambi, F. Ivancic, and M. Rotteler, "Using Hardware Transactional Memory for Data Race Detection," *International Parallel & Distributed Processing Symposium,* 2009.

[17] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid Transactional Memory" *International Conference on Architectural Support for Programming Languages and Operating Systems,* 2006.

[18] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar, "Healing Data Races On-The-Fly" *ACM Workshop on Parallel and Distributed Systems: Testing and Debugging*, 2007.

[19] S. Rajamani, G. Ramalingam, V. Ranganath, and K. Vaswani, "ISOLATOR: Dynamically Ensuring Isolation in Concurrent Programs" *International Conference on Architectural Support for Programming Languages and Operating Systems,* 2009.

[20] M. Abadi, T. Harris, and M. Mehrara, "Transactional Memory with Strong Atomicity using Off-the-Shelf Memory Protection Hardware" *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* 2009.

[21] B. Lucia, J. Devietti, K. Strauss, and L. Ceze, "Atom-Aid: Detecting and Surviving Atomicity Violations" *International Symposium on Computer Architecture*, 2008.

[22] J. Yu and N. S., "Tolerating Concurrency Bugs Using Transactions as Lifeguards" *International Symposium on Microarchitecture*, 2010.

[23] R. Rajwar and J. R. R. Goodman, "Transactional Lock-Free Execution of Lock-Based Programs" *International Conference on Architectural Support for Programming Languages and Operating Systems,* 2002.

[24] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional Memory Coherence and Consistency" *International Symposium on Computer Architecture*, 2004.

[25] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval, "Bulk Disambiguation of Speculative Threads in Multiprocessors" *International Symposium on Computer Architecture*, 2006.

[26] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded Transactional Memory" *International Symposium on High Performance Computer Architecture*, 2005.

[27] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing Transactional Memory" *International Symposium on Computer Architecture*, 2005.