Drinking from Both Glasses: Adaptively Combining Pessimistic and Optimistic Synchronization for Efficient Parallel Runtime Support [*]

Man Cao
Ohio State University
caoma@cse.ohio-state.edu

Minjia Zhang
Ohio State University
zhanminj@cse.ohio-state.edu

Michael D. Bond
Ohio State University
mikebond@cse.ohio-state.edu

Abstract

It is notoriously challenging to achieve parallel software systems that are both scalable and reliable. Parallel runtime support—such as multithreaded record & replay, data race and atomicity violation detectors, transactional memory, and support for stronger memory models—helps achieve these goals, but existing commodity solutions slow programs substantially in order to capture (track or control) the program's cross-thread dependences accurately. Capturing cross-thread dependences using "pessimistic" synchronization slows every program access, while "optimistic" synchronization allows for lightweight instrumentation of most accesses but dramatically slows accesses involved in cross-thread dependences.

This paper introduces (1) a hybrid of pessimistic and optimistic synchronization and (2) an adaptive policy that enables fine-grained switching between pessimistic and optimistic synchronization based on program behavior. The adaptive policy uses online profiling and a cost–benefit model to inform its decisions. We design a dependence recorder on top of our approach to demonstrate its feasibility as a framework for efficient parallel runtime support.

We have implemented our approach in a high-performance Java virtual machine and show that it outperforms parallel runtime support based solely on pessimistic or optimistic synchronization. These results show the potential for adaptive, hybrid synchronization for efficient parallel runtime support in commodity systems.

1. Introduction

Software must become more parallel in order to scale with successive microprocessor generations that provide more—instead of faster—cores. However, writing and debugging parallel programs is notoriously difficult. Programmers want both scalability and correctness, but modern general-purpose programming languages and systems provide poor support. Programmers are stuck using locks, making it hard to achieve both correctness and scalability.

Researchers have developed dynamic program analyses and software systems that help provide scalable, reliable parallelism. Notable examples include data race detectors (e.g., [8]), software transactional memory (e.g., [10]), region serializability enforcement (e.g., [17]), atomicity violation checkers (e.g., [9]), and multithreaded record & replay systems (e.g., [24]). However, existing approaches are *impractical* because they slow programs by several times or more, rely on unrealistic custom hardware, or have other serious limitations.

Existing runtime support that targets commodity systems (often called *software-only*) adds instrumentation to virtually every program access to *capture* (track or control) *cross-thread dependences*, i.e., data dependences (write–write, write–read, and read–write dependences) that involve two different threads. Critically, the runtime support adds synchronized instrumentation at any access that might be involved in a data race in order to capture cross-thread dependences soundly.

Many existing analyses perform synchronization at all accesses, or a substantial fraction of all accesses, that might possibly be involved in data races (e.g., [8–10, 14, 15]). (Identifying or eliminating racy accesses is itself an unsolved problem [6, 8, 15, 26].) We refer to this kind of synchronization as "pessimistic" since it conservatively assumes any access will be involved in a cross-thread dependence. This frequent synchronization, which typically takes the form of a small critical section implemented with an atomic operation and a memory fence, slows programs dramatically by preventing compiler and hardware reordering and triggering remote cache misses at otherwise mostly-read-shared memory accesses.

Alternatively, synchronization can be *optimistic* by avoiding performing synchronization on accesses *not* involved in cross-thread dependences, while still preserving soundness when accesses *are* involved in cross-thread dependences. Examples of optimistic synchronization include biased locking [7, 12, 19], the Shasta distributed shared memory system [20], sharing detection [25], and the Octet dynamic analysis framework [5]. Notably, Octet captures cross-thread dependences soundly, but accesses to objects *not* involved in cross-thread dependences do *not* require synchronization, while accesses involved in cross-thread dependences require expensive coordination between threads. While Octet adds very low overhead for non-conflicting patterns, it can add significant overhead even for programs with moderate communication (0.1–1% of all accesses being communicating) [5]. Similarly, enforcing region serializability, supporting transactional memory, detecting atomicity violations, and recording dependences add substantial overhead for high-communication programs [3, 5, 21, 27].

Contributions. This paper introduces a new approach that adaptively combines pessimistic and optimistic synchronization to get the benefits of both. We develop a hybrid model that combines pessimistic and optimistic synchronization and allows switching shared-memory objects between pessimistic and optimistic synchronization on the fly. Then we present a cost–benefit model that decides, based on program behavior, whether an object should use pessimistic or optimistic synchronization. We develop an adaptive policy for switching between pessimistic and optimistic states based on efficient online profiling that approximates the cost–benefit model.

At run time, our approach establishes happens-before relationships [13] that transitively imply all of an execution's cross-thread dependences. We demonstrate that parallel runtime support can build on the approach by designing a dependence recorder that is suitable for multithreaded record & replay.

[*] Appears at the Workshop on Determinism and Correctness in Parallel Programming (WoDet), March 2014. This material is based upon work supported by the National Science Foundation under Grants CAREER-1253703 and CSR-1218695.

We have implemented the hybrid state model, adaptive policy, and dependence recorder in a high-performance Java virtual machine. We compare our approach to purely pessimistic and optimistic approaches and find that our adaptive approach outperforms both: it always outperforms the pessimistic-only approach, and it adds low overhead over an optimistic-only approach for low-conflict programs, while significantly outperforming the optimistic-only approach for high-conflict programs. The dependence recorder built on our adaptive approach does *not* outperform a purely optimistic approach on average across all programs, but it significantly improves performance for high-conflict programs.

2. Background, Motivation, and Related Work

Dynamic analyses and systems that check and enforce concurrency correctness generally need to *capture* cross-thread dependences, encompassing one of the following:

Detecting (i.e., tracking) cross-thread dependences. Examples include data race detectors, atomicity violation detectors, and dependence recorders (e.g., for multithreaded record & replay).

Controlling cross-thread dependences. This runtime support often takes the form of lightweight per-object locks. Examples include transactional memory, enforcement of strong memory models, and deterministic execution.

Both cases require instrumentation at every program memory access. This instrumentation typically maintains metadata for each object[1] that helps to capture cross-thread dependences. This paper uses the following metadata values, called *states* (analogous to cache coherence states [18]):

- WrEx_T: Write exclusive for thread T. T can read and write an object in this state, but other accesses require a state change.
- RdEx_T: Read exclusive for T. T can read but not write an object in this state.
- RdSh_c: Read shared. Any thread can read the object, but writes require changing the state. The counter value c helps detect write–read dependences: a thread can compare c to its per-thread value to determine whether it has previously read an object in RdSh_c state.

At each object access, instrumentation (1) checks the object's state and the current access to detect cross-thread dependences, and it may perform additional client-specific actions such as computing happens-before relationships (e.g., data race detection) or triggering conflict detection and resolution (e.g., transactional memory); (2) updates the state; and (3) performs the program access. All of these actions must happen atomically because the access might be involved in a data race. (Data races are difficult to detect or eliminate [6, 8, 15, 16, 26].) Most existing runtime support guarantees atomicity by performing synchronization at every memory access, which we call "pessimistic" synchronization. Alternatively, "optimistic" synchronization eschews synchronization for non-communicating memory accesses, but requires a heavyweight coordination protocol for communicating memory accesses.

Pessimistic synchronization. Pessimistic synchronization essentially creates a small critical section around each access and its instrumentation. This critical section "locks" the object's state by atomically changing it to a special *intermediate* state Int_T (where T is the current thread), preventing other threads from accessing the object's metadata simultaneously.

[1] This paper uses the term "object" to refer to any unit of shared memory.

These frequent atomic operations slow program execution substantially by triggering remote cache misses and serializing execution. In our experiments, basic pessimistic synchronization slows programs by more than 5X on average (Section 7.2).

We note that existing approaches can often avoid performing atomic operations on every memory access. For example, repeated accesses to the same object can avoid synchronization if they occur in the same transaction (for transactional memory systems [11]) or synchronization-free region (for data race detectors [8]). Software transactional memory systems can typcially avoid atomic operations for *loads* by validating that they do not conflict, instead requiring only memory fences (e.g., [22]). Other approaches avoid explicit capturing of cross-thread dependences but incur other costs, e.g., DoublePlay runs two instances of the program simultaneously [24].

Optimistic synchronization. In contrast, optimistic synchronization avoids performing explicit synchronization operations as much as possible. As long as an access does not trigger a state change, it does not need to perform synchronization. For example, a thread T1 may repeatedly read or write to an object in $\mathsf{WrEx}_\mathsf{T1}$ state without performing any synchronization. However, before a thread T2 can access the object, it must *coordinate* with T1 to prevent T1 from continuing to access the object.

Coordination can be *explicit* or *implicit*. Explicit coordination involves T2 sending a *request* to T1, which T1 responds to only when it is at a *safe point*—a point that is not between an access and its accompanying instrumentation. Implicit coordination occurs if T2 atomically observes T1 in a "blocked" state, in which case T2 can change the object's state without waiting.

This tradeoff—synchronization-free in the common case but requiring expensive coordination in the uncommon case—works well in practice. We evaluate and build on an existing optimistic approach called Octet [5]. Table 1 shows that the vast majority of accesses do not change the metadata state (*Same state*). The remaining accesses are either *Conflicting*, requiring coordination between threads; or *Upgrading or fence*, which require an atomic operation or memory fence but not coordination. This optimistic approach adds only 26% overhead on average across large Java programs (Section 7.2). However, for programs with a relatively high fraction of accesses conflicting (e.g., pjbb2005 with 0.72%), overhead can exceed 100% because coordination is expensive.

We have attempted to measure the costs of different kinds of access instrumentation. Table 2 shows the geomean result across all programs (Section 7 describes experimental methodology). The average time in cycles (measured via the IA-32 TSC register) for instrumentation using *Pessimistic* synchronization is a few hundred cycles on average. For optimistic synchronization, a non-communicating access (*Same state*) costs only several dozen cycles by avoiding synchronization and keeping the instrumentation limited to a simple check. However, instrumentation at *Conflicting* accesses costs 2–3 orders of magnitude more than same-state accesses with *Explicit* coordination protocol, since it incurs the latency of roundtrip communication. *Implicit* coordination's cost is closer to the cost of pessimistic instrumentation.

Our goal is to develop a hybrid of pessimistic and optimistic synchronization that uses optimistic synchronization for most accesses, but avoids most coordination by using pessimistic synchronization at most conflicting accesses. At the same time, this hybrid approach needs to be able to provide efficient runtime support for both detecting and controlling cross-thread dependences.

Prior work on adaptive mechanisms. Prior work has used adaptive techniques for program synchronization. Usui et al. use online profiling and a cost–benefit model to adaptively choose between lock-based mutual exclusion and software transactional memory

	Same state	Conflicting	Upgrading or fence
eclipse6	1.2×10^{10}	1.3×10^{5}	6.3×10^{4}
hsqldb6	6.1×10^{8}	9.0×10^{5}	5.5×10^{5}
lusearch6	2.5×10^{9}	4.4×10^{3}	2.7×10^{3}
xalan6	1.1×10^{10}	1.8×10^{7}	1.2×10^{7}
avrora9	6.0×10^{9}	6.0×10^{6}	6.2×10^{6}
jython9	5.0×10^{9}	6.7×10^{1}	1.5×10^{1}
luindex9	3.3×10^{8}	3.7×10^{2}	2.1×10^{2}
lusearch9	2.4×10^{9}	2.9×10^{3}	2.6×10^{3}
pmd9	5.7×10^{8}	4.3×10^{4}	3.7×10^{4}
sunflow9	1.7×10^{10}	7.5×10^{3}	1.9×10^{4}
xalan9	1.0×10^{10}	1.9×10^{7}	1.2×10^{7}
pjbb2000	1.7×10^{9}	9.5×10^{5}	9.1×10^{5}
pjbb2005	6.6×10^{9}	4.8×10^{7}	3.5×10^{7}

Table 1. The run-time frequency of optimistic state transitions, divided into three categories that require different levels of synchronization.

	Pessimistic	Optimistic		
		Same state	Conflicting	
			Explicit	Implicit
geomean	1.5×10^{2}	4.7×10^{1}	9.2×10^{3}	3.6×10^{2}

Table 2. Average CPU cycles across all programs, for pessimistic and optimistic transitions.

(STM) for enforcing atomicity of critical sections [23]. Abadi et al. present an STM that adaptively changes how it detects conflicts for non-transactional accesses, depending on whether transactions access the same objects [1].

3. Hybrid State Model

This section introduces a new hybrid state model that combines pessimistic and optimistic states and enables transitions between pessimistic and optimistic states. We refer to objects in pessimistic and optimistic states as *pessimistic objects* and *optimistic objects*, respectively. At run time, the application's heap contains a mix of pessimistic and optimistic objects.

States. Our hybrid model combines the $\mathsf{WrEx_T}$, $\mathsf{RdEx_T}$, and $\mathsf{RdSh_c}$ states from both pessimistic and optimistic state models. We differentiate pessimistic and optimistic states with a superscript: $\mathsf{WrEx_T^{Pess}}$, $\mathsf{RdEx_T^{Pess}}$, $\mathsf{RdSh_c^{Pess}}$, $\mathsf{WrEx_T^{Opt}}$, $\mathsf{RdEx_T^{Opt}}$, $\mathsf{RdSh_c^{Opt}}$.

Both pessimistic and optimistic states use the same intermediate state $\mathsf{Int_T}$. All pessimistic transitions use $\mathsf{Int_T}$ to create a small critical section around the instrumentation and access. Conflicting transitions use $\mathsf{Int_T}$ to help perform the coordination protocol.

Transitions. Table 3 shows the complete set of state transitions when a thread attempts to perform an access. The top half of the table shows pessimistic state transitions, and the bottom half shows optimistic state transitions. Transitions between pessimistic and optimistic states are also possible. Pessimistic transitions can transition to an optimistic state only in cases where the state does not actually need to change. Only conflicting transitions can change from an optimistic to a pessimistic state. Although the model could support other transitions between pessimistic and optimistic states, limiting transitions to these cases makes sense because (1) optimistic states are advantageous for same-state accesses and (2) pessimistic states are advantageous for conflicting accesses.

4. Adaptive Policy

This section first introduces a cost–benefit model for deciding whether an object should be in a pessimistic or optimistic state. It then presents an efficient adaptive policy that uses online profiling, approximating the cost–benefit model.

4.1 Cost–Benefit Model

Our cost–benefit model estimates whether an object should be in pessimistic or optimistic states. Applying the model exactly as presented would require oracle knowledge, since decisions are based on future program behavior. Section 4.2 uses profile-guided feedback and gives up accuracy for better performance.

The basic idea of the cost–benefit model is that if the total time spent in optimistic transitions would exceed the total time spent in pessimistic transitions, the object should be in a pessimistic state; otherwise, it should be in an optimistic state.

Formalized model. Let N_{pess} be the number of pessimistic transitions that would occur if an object were in pessimistic states only. All transitions on pessimistic states are pessimistic transitions, so N_{pess} counts the total number of accesses. Let N_{confl} and $N_{nonConfl}$ be the numbers of conflicting and non-conflicting (same-state, upgrading, and fence) transitions, respectively, that would occur if the object were in optimistic states only. Since together N_{confl} and $N_{nonConfl}$ count all accesses,

$$N_{pess} = N_{nonConfl} + N_{confl} \tag{1}$$

Let $T_{nonConfl}$, T_{confl}, and T_{pess} be the average time costs for non-conflicting,[2] conflicting,[3] and pessimistic transitions, respectively. The model considers these values to be (platform-specific) constants that can be computed ahead of time, e.g., from Table 2.

According to the model, an object should be in optimistic states if and only if the following inequality holds:

$$T_{pess} \times N_{pess} \geq T_{nonConfl} \times N_{nonConfl} + T_{confl} \times N_{confl} \tag{2}$$

The left-hand side is the total time spent on state transitions if the object were in pessimistic states. The right-hand side is the total time on state transitions if the object were in optimistic states.

Applying (1) into (2) and transforming the formula yields:

$$N_{nonConfl} \geq \frac{T_{confl} - T_{pess}}{T_{pess} - T_{nonConfl}} \times N_{confl} \tag{3}$$

We define the coefficient as K_{confl} (since it essentially measures the penalty from conflicting transitions):

$$K_{confl} = \frac{T_{confl} - T_{pess}}{T_{pess} - T_{nonConfl}} \tag{4}$$

and (3) becomes:

$$N_{nonConfl} \geq K_{confl} \times N_{confl} \tag{5}$$

Thus, using the cost–benefit model requires only the numbers of conflicting and non-conflicting transitions—or simply their ratio—that would occur if the object were only in optimistic states.

4.2 Profile-Guided Adaptive Policy

Using the cost–benefit model to change objects to optimistic or pessimistic states at run time, presents several challenges that we address as follows.

- The cost–benefit model relies on knowing the ratio of conflicting to non-conflicting transitions for an object *a priori*. Our approach uses *online* profiling, i.e., decisions about future state transitions are based on (recent) past behavior.
- Counting non-conflicting transitions would be expensive because they are common. Our profiling approach instead counts transitions that *would* be non-conflicting (if the object were in optimistic states)—only while an object is in pessimistic states.

[2] The model computes the time for non-conflicting transitions as simply the time for same-state transitions, a reasonable approximation because upgrading and fence transitions incur a cost similar to pessimistic transitions.

[3] T_{confl} is the time for a conflicting transition using the *explicit* protocol.

Trans. type	**Old state**	**Program access**	**New state**	**Sync. needed**	**Cross-thread dependence?**
Pessimistic	$\mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Pess}}$	R or W by T	$\mathsf{Int}_{\mathsf{T}} \rightarrow \mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Pess}}$ **or** $\mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Opt}}$		
or	$\mathsf{RdEx}_{\mathsf{T}}^{\mathsf{Pess}}$	R by T	$\mathsf{Int}_{\mathsf{T}} \rightarrow \mathsf{RdEx}_{\mathsf{T}}^{\mathsf{Pess}}$ **or** $\mathsf{RdEx}_{\mathsf{T}}^{\mathsf{Opt}}$	CAS	No
Pess $\rightarrow$ Opt	$\mathsf{RdSh}_{\mathsf{c}}^{\mathsf{Pess}}$	R by T if $\mathsf{T.rdShCount} \geq \mathsf{c}$	$\mathsf{Int}_{\mathsf{T}} \rightarrow \mathsf{RdSh}_{\mathsf{c}}^{\mathsf{Pess}}$ **or** $\mathsf{RdSh}_{\mathsf{c}}^{\mathsf{Opt}}$		
Pessimistic	$\mathsf{RdEx}_{\mathsf{T}}^{\mathsf{Pess}}$	W by T	$\mathsf{Int}_{\mathsf{T}} \rightarrow \mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Pess}}$	CAS	No
	$\mathsf{RdEx}_{\mathsf{T1}}^{\mathsf{Pess}}$	R by $\mathsf{T2}$	$\mathsf{Int}_{\mathsf{T2}} \rightarrow \mathsf{RdSh}_{\mathsf{gRdShCount}}^{\mathsf{Pess}}$		Maybe
Pessimistic	$\mathsf{RdSh}_{\mathsf{c}}^{\mathsf{Pess}}$	R by T if $\mathsf{T.rdShCount} < \mathsf{c}$	$\mathsf{Int}_{\mathsf{T}} \rightarrow$ Same ($\mathsf{T.rdShCount} = \mathsf{c}$)	CAS	Maybe
	$\mathsf{WrEx}_{\mathsf{T1}}^{\mathsf{Pess}}$	W by $\mathsf{T2}$	$\mathsf{Int}_{\mathsf{T2}} \rightarrow \mathsf{WrEx}_{\mathsf{T2}}^{\mathsf{Pess}}$		
Pessimistic	$\mathsf{WrEx}_{\mathsf{T1}}^{\mathsf{Pess}}$	R by $\mathsf{T2}$	$\mathsf{Int}_{\mathsf{T2}} \rightarrow \mathsf{RdEx}_{\mathsf{T2}}^{\mathsf{Pess}}$	CAS	Maybe
	$\mathsf{RdEx}_{\mathsf{T1}}^{\mathsf{Pess}}$	W by $\mathsf{T2}$	$\mathsf{Int}_{\mathsf{T2}} \rightarrow \mathsf{WrEx}_{\mathsf{T2}}^{\mathsf{Pess}}$		
	$\mathsf{RdSh}_{\mathsf{c}}^{\mathsf{Pess}}$	W by T	$\mathsf{Int}_{\mathsf{T}} \rightarrow \mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Pess}}$		
	$\mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Opt}}$	R or W by T	Same		
Same state	$\mathsf{RdEx}_{\mathsf{T}}^{\mathsf{Opt}}$	R by T	Same	None	No
	$\mathsf{RdSh}_{\mathsf{c}}^{\mathsf{Opt}}$	R by T if $\mathsf{T.rdShCount} \geq \mathsf{c}$	Same		
Upgrading	$\mathsf{RdEx}_{\mathsf{T}}^{\mathsf{Opt}}$	W by T	$\mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Opt}}$	CAS	No
	$\mathsf{RdEx}_{\mathsf{T1}}^{\mathsf{Opt}}$	R by $\mathsf{T2}$	$\mathsf{RdSh}_{\mathsf{gRdShCount}}^{\mathsf{Opt}}$		Maybe
Fence	$\mathsf{RdSh}_{\mathsf{c}}^{\mathsf{Opt}}$	R by T if $\mathsf{T.rdShCount} < \mathsf{c}$	($\mathsf{T.rdShCount} = \mathsf{c}$)	Memory fence	Maybe
Conflicting	$\mathsf{WrEx}_{\mathsf{T1}}^{\mathsf{Opt}}$	W by $\mathsf{T2}$	$\mathsf{Int}_{\mathsf{T2}} \rightarrow \mathsf{WrEx}_{\mathsf{T2}}^{\mathsf{Opt}}$ **or** $\mathsf{WrEx}_{\mathsf{T2}}^{\mathsf{Pess}}$		
or	$\mathsf{WrEx}_{\mathsf{T1}}^{\mathsf{Opt}}$	R by $\mathsf{T2}$	$\mathsf{Int}_{\mathsf{T2}} \rightarrow \mathsf{RdEx}_{\mathsf{T2}}^{\mathsf{Opt}}$ **or** $\mathsf{RdEx}_{\mathsf{T2}}^{\mathsf{Pess}}$	Roundtrip coordination	Maybe
Opt $\rightarrow$ Pess	$\mathsf{RdEx}_{\mathsf{T1}}^{\mathsf{Opt}}$	W by $\mathsf{T2}$	$\mathsf{Int}_{\mathsf{T2}} \rightarrow \mathsf{WrEx}_{\mathsf{T2}}^{\mathsf{Opt}}$ **or** $\mathsf{WrEx}_{\mathsf{T2}}^{\mathsf{Pess}}$		
	$\mathsf{RdSh}_{\mathsf{c}}^{\mathsf{Opt}}$	W by T	$\mathsf{Int}_{\mathsf{T}} \rightarrow \mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Opt}}$ **or** $\mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Pess}}$		

Table 3. All possible state transitions for the hybrid model. The top half of the table shows transitions starting in pessimistic states, and the bottom half shows transitions starting in optimistic states. Instances of bold "**or**" indicate cases where a state can potentially transition between pessimistic and optimistic states.

- Profiling groups of objects and aggregating information about them could be beneficial. For example, aggregating profiling data by objects' allocation sites would enable identifying allocation sites that should put objects into optimistic versus pessimistic states. This approach would add extra space overhead to track objects' allocation sites and require dynamic recompilation of sites. Our current approach does not aggregate profiling information; it profiles individual objects at run time. Newly allocated objects start in the $\mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Opt}}$ state (where T is the allocating thread) since most objects are mostly non-conflicting.

As mentioned above, objects allocated by thread T start in the $\mathsf{WrEx}_{\mathsf{T}}^{\mathsf{Opt}}$ state. Instead of counting both non-conflicting and conflicting transitions for optimistic objects, the approach counts only conflicting transitions, which are relatively infrequent. If an object executes enough conflicting transitions (a heuristic value $\mathit{Cutoff}_{\mathit{confl}}$), i.e., if the following condition is satisfied,

$$\mathit{ConflTrans}(o) \geq \mathit{Cutoff}_{\mathit{confl}} \tag{6}$$

the approach switches the object to an appropriate pessimistic state. Switching according to (6) does *not* work well *by itself* because different objects need different $\mathit{Cutoff}_{\mathit{confl}}$ values. Thus, the approach sets $\mathit{Cutoff}_{\mathit{confl}}$ to a low constant value and profiles the object further after it becomes pessimistic.

All pessimistic transitions count whether the transition would be non-conflicting or conflicting *if* the object were actually in an optimistic state. This information helps inform the cost–benefit model about whether to switch the object back to an optimistic state. Our approach uses the following formula, derived from equation (5), to make this decision:

$$\mathit{Inertia} + K_{\mathit{confl}} \times N_{\mathit{confl}} \leq N_{\mathit{nonConfl}} \tag{7}$$

The heuristic value $\mathit{Inertia}$ helps avoid switching an object back to an optimistic state before collecting enough profiling information.

The profiling approach can compute (7) efficiently at run time using a single per-object value as follows. After the first optimistic-to-pessimistic transition, assign $\mathit{Inertia}$ to the value. For every pessimistic transition, if it would be non-conflicting, decrement the value by 1; otherwise increment the value by K_{confl}. If the value becomes negative, switch the object to an optimistic state.

Our current approach switches each object from optimistic to pessimistic at most once, to avoid putting an object repeatedly into pessimistic states if it incurs conflicting transitions but should ideally be in optimistic states.

Experiments (not shown) find that performance is not very sensitive to the values of K_{confl} and $\mathit{Inertia}$. Various values of K_{confl} (8–512) and $\mathit{Inertia}$ (8–1024) are effective.

5. Using the Hybrid State Model as a Framework

An important requirement for the hybrid state model is for it to be a framework for runtime support that tracks or controls cross-thread dependences. We demonstrate *tracking* of cross-thread dependences by designing and implementing a dependence recorder that records enough information about cross-thread dependences to enable exact replay. (However, we are not yet able to demonstrate replay, due to other sources of nondeterminism that are hard to control.) Future work will show how to apply the hybrid state model to *controlling* cross-thread dependences.

Our dependence recorder builds on a recorder from prior work that uses optimistic synchronization [5]. It stores information about cross-thread dependences in per-thread logs. For optimistic transitions, our recorder uses the same approach as prior work. To some extent, the recorder can use the same approach for pessimistic transitions: after all, in our model, pessimistic and optimistic transitions have similar characteristics (e.g., $\mathsf{RdEx}_{\mathsf{T}}^{\mathsf{Pess}} \rightarrow \mathsf{RdSh}_{\mathsf{c}}^{\mathsf{Pess}}$ versus $\mathsf{RdEx}_{\mathsf{T}}^{\mathsf{Opt}} \rightarrow \mathsf{RdSh}_{\mathsf{c}}^{\mathsf{Opt}}$), allowing cross-thread dependences to be identified soundly for pessimistic transitions in the same way as for optimistic transitions.

The exception is pessimistic transitions that *would* be conflicting if they involved optimistic states (the last four transitions in the top half of Table 3). Suppose that thread $\mathsf{T1}$ was the last to access an object o, and $\mathsf{T2}$ performs a conflicting access, i.e., if o is in an optimistic state, then the access would trigger a conflicting transition, which requires coordination. For optimistic states, the coordination protocol stops both threads, enabling recording of a happens-before edge by recording the source and sink of the edge in T1 and T2's logs, respectively. The source and sink are each a *dynamic accesses location* (DAL), which is a unique execution point identified by (1) static program location and (2) a per-thread counter called a *yield-point counter* that is incremented at every method entry and loop back edge. T1 records the source DAL when it responds to T2. T2 records the sink DAL for the access that detects the conflict.

However, for a pessimistic transition that *would* be conflicting if o were in an optimistic state, it is difficult to identify and record a DAL on T1 that is the source of a happens-before edge that implies the cross-thread dependence. This source DAL needs to be (1) after the last access to the object on T1, to capture the dependence, and (2) no later than the current DAL of T1 (which can be actively changing since only T2 is stopped), or else replay may deadlock.

Our recorder addresses this challenge using the following strategy. Each pessimistic transition maintains a per-thread *pessimistic-timestamp counter* that is the value of the thread's yield-point counter at the *most recent* pessimistic transition. Thus T1's last pessimistic transition will have updated this value. If, at the conflicting access, T2 observes that T1's yield-point counter now exceeds T1's pessimistic-timestamp counter, then T2 can simply record T1's yield-point counter value, since replay can wait for this same value in order to replay the happens-before edge. Otherwise (if T1's yield-point counter and pessimistic-timestamp counter have the same value), the approach must record a more precise source DAL. Our current approach is for T2 to acquire a lock on T1's log and write an event into the log that records the DAL of T1's most recent pessimistic transition (these values can be observed racily but safely), so during replay T1 can replay the source of the happens-before edge. This approach requires that *all* accesses to per-thread logs acquire locks, which adds noticeable overhead. We are exploring alternative strategies that might provide better performance.

6. Implementation

We have implemented the hybrid state model and adaptive policy in Jikes RVM 3.1.3, a high-performance Java virtual machine [2]. Our implementation builds on the publicly available Octet implementation in Jikes RVM.[4]

Each object and state field has a metadata word for its state. Three low bits enable representing the seven possible states, and the other bits represent a thread address or the read-shared counter.

Jikes RVM's dynamic just-in-time compilers insert instrumentation before every memory access in the application and Java libraries. The instrumentation performs the "fast path," which checks if the object is in an optimistic state and needs only a same-state transition. If the fast-path check fails, the instrumentation performs the "slow path," which handles the other possible state transitions.

To handle pessimistic transitions, the compiler inserts instrumentation *after* every memory access to "unlock" the state by changing it from Int_T to the new state. To avoid the cost of conditionally performing this unlock operation, the compilers generate two code paths: one executes only the program access, and the other executes the program access followed by changing the state. The instrumentation before the program access jumps to the latter code path if and only if the slow path indicates a transition from a pessimistic state. While in theory this configuration should add no overhead for objects in optimistic states, in practice it adds noticeable overhead, presumably because the possibility of executing the pessimistic code path affects the compiler's optimizations, e.g., uses of virtual registers on the pessimistic code path affect register allocation decisions in the rest of the method.

[4] `http://jikesrvm.org/Research+Archive`

7. Evaluation

This section evaluates the run-time characteristics and performance of our hybrid state model and adaptive policy, compared with purely pessimistic and optimistic approaches.

Methodology. We evaluate our implementation on the DaCapo Benchmarks, versions 2006-10-MR2 and 9.12-bach (2009) [4], distinguished with suffixes 6 and 9; and fixed-workload versions of SPECjbb2000 and SPECjbb2005.[5] We include only multithreaded programs and exclude programs that (unmodified) Jikes RVM cannot execute.

Experiments execute on a machine with 4 AMD Opteron 6272 16-core processors and 128 GB memory, running Linux 2.6.32. We configure experiments to utilize only 32 cores of the 64-core system (using the Linux taskset command). On 64 cores, instrumented code outperforms uninstrumented code for a few benchmarks. We do not fully understand this anomalous result but have determined that it is affected by Linux scheduling domain policies.

We build a high-performance configuration (FastAdaptiveGenImmix) of Jikes RVM. We let the garbage collector adjust the heap size automatically at run time. Each performance result is the median of 25 trial runs; we also show the mean as the center of 95% confidence intervals. Table 4's results use the mean of 5 trials.

We use the following values for the parameters of the adaptive policy, except when stated otherwise:

$Cutoff_{confl}$	K_{confl}	$Inertia$
4	200	100

We choose a low value for $Cutoff_{confl}$ in order to transition optimistic objects to pessimistic states aggressively; pessimistic-to-optimistic transitions can remedy pessimistic objects that should actually be in optimistic states. As Section 4.2 mentioned, performance is not very sensitive to the values of K_{confl} and $Inertia$.

7.1 Run-Time Characteristics

Table 4 shows the frequency of different state transitions under the adaptive policy. The columns show the counts of various transition categories from Table 3. The table breaks down *Optimistic* into three types since different optimistic transitions incur substantially different costs.

For *Same State* and *Conflicting* transitions, the number of transitions from the optimistic-only state model (Table 1) is shown in parentheses. The *Conflicting* column measures how well the adaptive policy achieves its primary goal of reducing conflicting transitions. The reduction is substantial for programs that execute a relatively high fraction of conflicting transitions when using purely optimistic states. Section 7.2 shows that performance improves significantly for these programs.

The *Same state* column measures the downside of the adaptive policy: some transitions that would have been same-state transitions (requiring no synchronization) become pessimistic transitions (requiring synchronization). The table shows that at most 1–2% of same-state transitions become pessimistic. Achieving this result relies on the adaptive policy's profiling of pessimistic objects to identify objects to put back into optimistic states.

Although more same-state transitions than conflicting transitions become pessimistic, it does not imply a performance loss, since a conflicting transition costs 2–3 orders of magnitude more than a same-state transition.

For these programs at least, the adaptive policy achieves its goal of eliminating most of the conflicting transitions (and thus most of the expensive coordination protocol overhead) for high-conflict

[5] `http://spec.org/, http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005`

	Optimistic			Pessimistic	Optimistic to Pessimistic	Pessimistic to Optimistic
	Same state	Conflicting	Upgrading or fence			
eclipse6	$(1.2\times10^{10} \rightarrow)$ 1.2×10^{10}	$(1.3\times10^{5} \rightarrow)$ 1.4×10^{5}	6.7×10^{4}	1.6×10^{6}	2.3×10^{2}	1.5×10^{2}
hsqldb6	$(6.1\times10^{8} \rightarrow)$ 6.0×10^{8}	$(9.0\times10^{5} \rightarrow)$ 2.6×10^{5}	2.2×10^{5}	8.2×10^{6}	1.3×10^{4}	2.2×10^{2}
lusearch6	$(2.5\times10^{9} \rightarrow)$ 2.5×10^{9}	$(4.4\times10^{3} \rightarrow)$ 4.2×10^{3}	2.6×10^{3}	1.6×10^{3}	4.0×10^{0}	0
xalan6	$(1.1\times10^{10} \rightarrow)$ 1.0×10^{10}	$(1.8\times10^{7} \rightarrow)$ 4.4×10^{5}	9.0×10^{4}	2.0×10^{8}	5.3×10^{2}	1.1×10^{2}
avrora9	$(6.0\times10^{9} \rightarrow)$ 6.0×10^{9}	$(6.0\times10^{6} \rightarrow)$ 8.2×10^{5}	7.9×10^{4}	1.9×10^{7}	1.5×10^{5}	4.7×10^{1}
jython9	$(5.0\times10^{9} \rightarrow)$ 5.0×10^{9}	$(6.7\times10^{1} \rightarrow)$ 6.6×10^{1}	1.5×10^{1}	0	0	0
luindex9	$(3.3\times10^{8} \rightarrow)$ 3.3×10^{8}	$(3.7\times10^{2} \rightarrow)$ 3.7×10^{2}	2.1×10^{2}	0	0	0
lusearch9	$(2.4\times10^{9} \rightarrow)$ 2.4×10^{9}	$(2.9\times10^{3} \rightarrow)$ 2.1×10^{3}	1.9×10^{3}	6.2×10^{3}	1.3×10^{1}	1.0×10^{0}
pmd9	$(5.7\times10^{8} \rightarrow)$ 5.8×10^{8}	$(4.3\times10^{4} \rightarrow)$ 1.7×10^{4}	1.2×10^{4}	2.1×10^{5}	1.2×10^{2}	9.6×10^{0}
sunflow9	$(1.7\times10^{10} \rightarrow)$ 1.7×10^{10}	$(7.5\times10^{3} \rightarrow)$ 6.2×10^{3}	1.6×10^{4}	3.2×10^{4}	1.1×10^{1}	3.0×10^{0}
xalan9	$(1.0\times10^{10} \rightarrow)$ 9.9×10^{9}	$(1.9\times10^{7} \rightarrow)$ 2.7×10^{5}	5.3×10^{4}	1.9×10^{8}	8.2×10^{2}	1.2×10^{2}
pjbb2000	$(1.7\times10^{9} \rightarrow)$ 1.7×10^{9}	$(9.5\times10^{5} \rightarrow)$ 8.3×10^{5}	7.9×10^{5}	7.0×10^{6}	5.5×10^{4}	4.0×10^{3}
pjbb2005	$(6.6\times10^{9} \rightarrow)$ 6.5×10^{9}	$(4.8\times10^{7} \rightarrow)$ 8.4×10^{5}	3.3×10^{5}	1.3×10^{8}	3.8×10^{3}	3.7×10^{3}

Table 4. State transitions for the adaptive policy. For optimistic transitions that are *Same State* or *Conflicting*, the number in parentheses is the number of transitions when using the optimistic-only model, i.e., the same value as in Table 1.

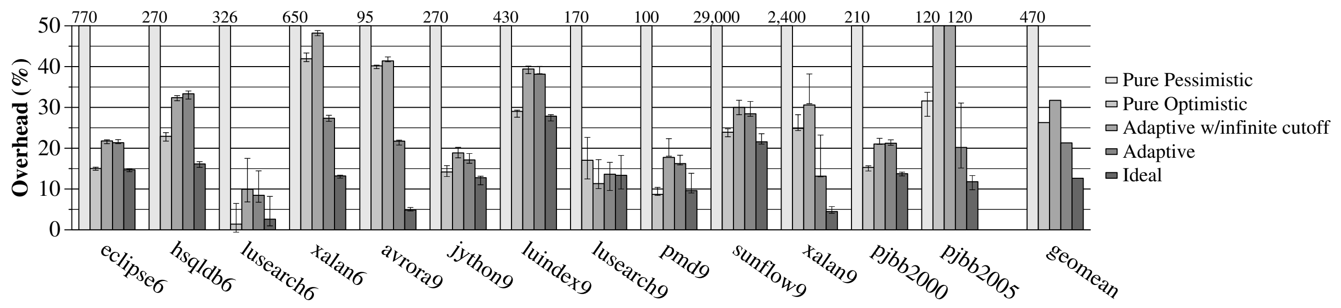

Figure 1. Run-time overhead of purely optimistic and pessimistic state models, compared with the hybrid state model using the adaptive policy. The ranges are 95% confidence intervals centered at the mean.

programs, while keeping the percentage of pessimistic transitions low.

7.2 Performance

Figure 1 compares the performance of our adaptive policy with policies that use only pessimistic or optimistic states. Each bar shows the run-time overhead added over unmodified Jikes RVM.

Pure Pessimistic evaluates using only pessimistic states, which is expensive, as prior work also shows [5]. This configuration adds 470% overhead on average (excluding sunflow9, the geomean is 310%), showing that pessimistic states must be applied judiciously.

Pure Optimistic is the overhead of optimistic states only (i.e., Octet [5]). It adds 26% average overhead, but a few programs incur substantial costs.

The two *Adaptive* configurations use the adaptive policy to switch between optimistic and pessimistic states. *Adaptive w/infinite cutoff* sets $Cutoff_{confl}$ to ∞, so no object actually transitions to a pessimistic state; this configuration measures only the costs (e.g., from instrumentation; Section 6), not the benefits, of the adaptive policy. These costs are 5.5% (relative to baseline execution) over a purely optimistic approach.

Adaptive uses the default values for $Cutoff_{confl}$ and other parameters. The adaptive policy significantly improves the performance of several programs that perform poorly with a purely optimistic model, which are the same programs that have many conflicting transitions eliminated by the adaptive policy (Table 4). The adaptive policy reduces overhead by 15% (42% $\rightarrow$ 27%) for xalan6, by 18% (40% $\rightarrow$ 22%) for avrora9, and by 98% (from 118% $\rightarrow$ 20%) for pjbb2005. Despite reducing conflicting transitions significantly for hsqldb6 (Table 4), the adaptive policy has little impact because hsqldb6's conflicting transitions mainly use the implicit protocol, which costs about as much as a pessimistic transition.

Ideal is the overhead of optimistic states only, but without using the coordination protocol. It should cost about the same as if all conflicting transitions became pessimistic, but all same-state transitions remained optimistic. This unsound configuration, which adds 13% on average, is essentially a lower bound on the overhead that the adaptive protocol might be able to provide.

The adaptive policy adds 21% overhead on average. It reduces overhead (relative to baseline execution) by 10% over *Adaptive w/infinite cutoff*, recovering most of the 13% overhead difference between the purely optimistic approach and the ideal, unsound configuration. However, the overhead added by the adaptive policy's mechanism cuts into this performance improvement. With more effort, we should be able to eliminate most of the overhead from the adaptive policy's mechanism.

While an optimistic approach provides the best performance for low-conflict programs, our adaptive approach provides significantly better performance for high-conflict programs. On average, the adaptive approach adds lower overhead than either purely pessimistic or optimistic approaches.

Dependence recorder. Figure 2 shows the performance of the dependence recorder based on purely optimistic and adaptive approaches (Section 5). The adaptive approach improves the recorder's performance considerably for the high-conflict programs xalan6, xalan9 and pjbb2005, but incurs more overhead than prior work for other programs, adding 1–2% more overhead on average than the purely optimistic recorder. The adaptive recorder adds more overhead mainly because it must lock its per-thread logs. The hybrid model reduces the number of conflicting transitions but not

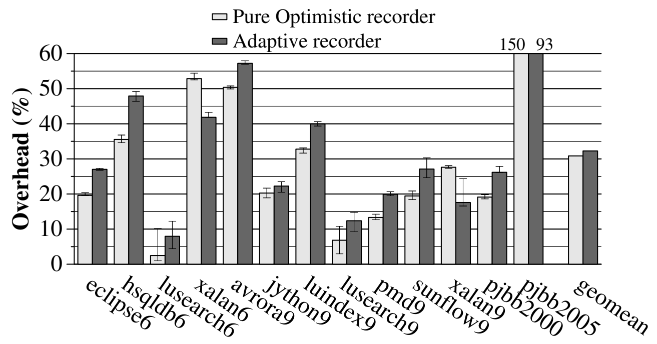

Figure 2. Run-time overhead of the dependence recorder built on top of the hybrid state model using the adaptive policy, compared with a purely optimistic state model.

the number of detected cross-thread dependences, so the recorder still needs to log as much as the purely optimistic recorder.

8. Conclusion and Future Directions

We present a hybrid state model and adaptive policy that achieve better performance than purely pessimistic or optimistic approaches. Our results suggest that our hybrid, adaptive approach is promising, especially for high-conflict applications for which a purely optimistic approach incurs high coordination costs.

Here we discuss three opportunities for future work. (1) We plan to improve performance further by reducing the costs of the adaptive policy itself. (2) The adaptive policy profiles objects individually, which is not reactive enough if many objects each experience one or few conflicting transitions; these objects will not transition to pessimistic states soon enough. We will investigate aggregate profiling (e.g., by allocation site) of objects in order to make profiling more reactive (e.g., so some allocation sites put newly allocated objects directly into pessimistic states). (3) We have focused on showing how to apply the hybrid state model and adaptive policy to tracking cross-thread dependences by building a dependence recorder. We plan to demonstrate that our approach is also a suitable framework for *controlling* cross-thread dependences.

Acknowledgments

We thank Swarnendu Biswas, Jipeng Huang, Milind Kulkarni, and Aritra Sengupta for helpful discussions; and the anonymous reviewers for insightful feedback on the text.

References

[1] M. Abadi, T. Harris, and M. Mehrara. Transactional Memory with Strong Atomicity Using Off-the-Shelf Memory Protection Hardware. In *PPoPP*, pages 185–196, 2009.

[2] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Research Virtual Machine Project: Building an Open-Source Research Community. *IBM Systems Journal*, 44:399–417, 2005.

[3] S. Biswas, J. Huang, A. Sengupta, and M. D. Bond. DoubleChecker: Efficient Sound and Precise Atomicity Checking. In *PLDI*, 2014. To appear.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*, pages 169–190, 2006.

[5] M. D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. Fathi Salmi, S. Biswas, A. Sengupta, and J. Huang. Octet: Capturing and Controlling Cross-Thread Dependences Efficiently. In *OOPSLA*, pages 693–712, 2013.

[6] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, pages 211–230, 2002.

[7] M. Burrows. How to Implement Unnecessary Mutexes. In *Computer Systems Theory, Technology, and Applications*, pages 51–57. Springer–Verlag, 2004.

[8] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *PLDI*, pages 121–133, 2009.

[9] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *PLDI*, pages 293–303, 2008.

[10] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *OOPSLA*, pages 388–402, 2003.

[11] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, 2nd edition, 2010.

[12] K. Kawachiya, A. Koseki, and T. Onodera. Lock Reservation: Java Locks Can Mostly Do Without Atomic Operations. In *OOPSLA*, pages 130–141, 2002.

[13] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 21(7):558–565, 1978.

[14] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE TOC*, 36:471–482, 1987.

[15] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy. Chimera: Hybrid Program Analysis for Determinism. In *PLDI*, pages 463–474, 2012.

[16] M. Naik and A. Aiken. Conditional Must Not Aliasing for Static Race Detection. In *POPL*, pages 327–338, 2007.

[17] J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. ...and region serializability for all. In *HotPar*, 2013.

[18] M. S. Papamarcos and J. H. Patel. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *ISCA*, pages 348–354, 1984.

[19] K. Russell and D. Detlefs. Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing. In *OOPSLA*, pages 263–272, 2006.

[20] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *ASPLOS*, pages 174–185, 1996.

[21] A. Sengupta, S. Biswas, M. Zhang, M. D. Bond, and M. Kulkarni. Hybrid Static–Dynamic Analysis for Region Serializability. Technical Report OSU-CISRC-11/12-TR18, Computer Science & Engineering, Ohio State University, 2013.

[22] 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 *PLDI*, pages 78–88, 2007.

[23] T. Usui, R. Behrends, J. Evans, and Y. Smaragdakis. Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In *PACT*, pages 3–14, 2009.

[24] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *ASPLOS*, pages 15–26, 2011.

[25] C. von Praun and T. R. Gross. Object Race Detection. In *OOPSLA*, pages 70–82, 2001.

[26] C. von Praun and T. R. Gross. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. In *PLDI*, pages 115–128, 2003.

[27] M. Zhang, J. Huang, M. Cao, and M. D. Bond. LarkTM: Efficient, Strongly Atomic Software Transactional Memory. Technical Report OSU-CISRC-11/12-TR17, Computer Science & Engineering, Ohio State University, 2013.