Adaptive Tracking of Cross-Thread Dependences

Ohio State CSE technical report #OSU-CISRC-7/13-TR15, July 2013

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

OCTET is a framework for dynamic analysis that soundly captures cross-thread dependences in parallel programs. It optimistically assumes that most accesses do not conflict, enabling the instrumentation to not perform synchronization at non-conflicting accesses. However, OCTET's performance can suffer substantially if an application triggers more than a small fraction of conflicting accesses: on the order of 0.1% or more of all accesses, according to our investigations.

This paper introduces an adaptive approach to replace as many heavyweight conflicting transitions as possible with so-called "pessimistic" transitions. An adaptive policy determines if an object should switch from an optimistic to a pessimistic state, or from pessimistic to optimistic state, based on online profiling. Experimental results show that this approach can reduce the overhead of OCTET by 30–40% for one program, while adding low overhead for applications that do not have many conflicting accesses.

Note to the reader. This technical report (TR) describes recent preliminary innovations and results. In its current form, this TR may not stand on its own, and readers may need to read our prior work on OCTET for background [3]. Questions, suggestions, and feedback are welcome.

1. Background and Motivation

Writing and debugging parallel programs has been a longstanding challenge. Researchers have proposed various solutions for guaranteeing reliable concurrency, among which dynamic, software-only analyses have perhaps the most potential to be practical. To check and enforce concurrency correctness, these analyses rely on tracking cross-thread dependences (i.e., conflicting accesses to shared memory). A fundamental problem is how to track cross-thread dependences soundly and efficiently.

OCTET soundly captures cross-thread dependences in an optimistic way: its analysis adds low overhead at non-conflicting accesses, but conflicting accesses require expensive communication between threads [3]. The analysis tracks the "locality state" of each object; instrumentation at loads and stores uses the state to identify conflicting accesses and updates the state if needed. OCTET's optimistic design optimizes the common case, based on the observation that most accesses are compatible with the state. It slows programs by 26% on average [3], which is significantly faster than comparable prior work targeting commodity systems.

However, for an application that performs many conflicting accesses, although these accesses are still not the majority of all accesses, OCTET's overhead increases drastically and is much slower than the naïve pessimistic model [3] or von Praun and Gross's state model [8]. We find empirically that if the ratio of conflicting transitions to all accesses is approximately at least 0.1%, the roundtrip coordinations could incur significant overhead. Another important factor (that we have not yet included in our model) is that some invocations of the coordination protocol are more expensive than others, e.g., the explicit protocol is more expensive than the implicit protocol, and RdSh $\rightarrow$ WrEx are more expensive than other conflicting transitions.

Table 1 shows the ratio of state transitions for each category. We have collected these results on a 4-core system using the same methodology as in Section 4. For the 13 applications we have tested, 12 have less than 0.2% of all state transitions conflicting. pjbb2005 has 0.51% conflicting transitions and OCTET experiences a 1.8X slowdown (Section 4). Executing on another platform using 32 cores, this

	Same state	Conflicting	Upgrading or fence
eclipse6	99.9984%	0.0011%	0.00050%
hsqldb6	99.73%	0.16%	0.11%
lusearch6	99.99971%	0.00018%	0.00011%
xalan6	99.80%	0.12%	0.080%
avrora9	99.81%	0.095%	0.098%
jython9	99.9999985%	0.0000012%	0.00000030%
luindex9	99.99982%	0.00011%	0.000065%
lusearch9	99.99985%	0.00011%	0.000040%
pmd9	99.988%	0.0068%	0.0047%
sunflow9	99.999920%	0.000036%	0.000045%
xalan9	99.84%	0.094%	0.063%
pjbb2000	99.89%	0.055%	0.052%
pjbb2005	99.16%	0.51%	0.33%

Table 1. The fraction of all accesses that trigger each kind of OCTET state transition (including "same state" transitions). We round each percentage x as much as possible such that x and $100\% - x$ each have two significant digits.

	Fast path	Conflicting			Pessimistic
		*Ex→*Ex		RdSh→	
		Explicit	Implicit	WrEx	
eclipse6	2.9×10^1	2.8×10^3	1.2×10^2	1.5×10^3	1.2×10^2
hsqldb6	2.9×10^1	1.4×10^4	1.3×10^2	4.7×10^3	1.0×10^2
lusearch6	7.9×10^1	1.5×10^5	1.9×10^2	8.4×10^6	1.4×10^2
xalan6	3.2×10^1	3.5×10^4	2.8×10^2	3.6×10^5	9.2×10^1
avrora9	3.0×10^1	1.8×10^4	2.4×10^2	2.6×10^5	4.6×10^3
jython9	2.9×10^1	3.2×10^3	2.4×10^2	N/A	N/A
luindex9	3.0×10^1	1.8×10^3	1.3×10^2	1.4×10^3	N/A
lusearch9	3.0×10^1	3.6×10^4	1.8×10^2	1.1×10^3	2.3×10^2
pmd9	3.1×10^1	3.9×10^4	1.7×10^2	1.4×10^4	1.1×10^2
sunflow9	3.0×10^1	1.5×10^4	1.3×10^2	1.6×10^5	4.0×10^2
xalan9	3.0×10^1	1.0×10^4	2.8×10^2	1.3×10^4	9.2×10^1
pjbb2000	3.5×10^1	8.9×10^4	1.0×10^2	2.4×10^5	1.0×10^2
pjbb2005	3.5×10^1	1.2×10^4	1.9×10^2	1.5×10^5	1.2×10^2

Table 2. Average CPU cycles for each type of state transition. Conflicting transitions are divided into transitions involving just two threads (both implicit and explicit coordination protocols) and transitions involving all threads. *N/A* means the program executes no transitions of that type.

slowdown is as high as 2.1X—much larger than the 1.3X for the naïve pessimistic model [3].

We have reason to believe that applications with a high ratio of conflicting transitions are not rare. For example, the STAMP benchmarks [4] have a conflicting transition rate as high as 1–4% [9].

To confirm our intuition about the relative costs of different state transitions, we have quantitatively measured the time (in CPU cycles) required to perform a fast-path state check, a conflicting transition, and a naïve pessimistic state transition. As Table 2 shows, a conflicting transition can be hundreds of times slower than a fast path. We have measured these times *using the hybrid model and adaptive policy* introduced in this paper; using a different model and policy would lead to different state transitions and different results.

With the above observations, one can naturally ask if we can combine the benefits of OCTET and the pessimistic state model (or von Praun and Gross's model [8], which turns out to benefit from the very same pessimistic state transition for highly conflicted applications). Certainly the *happens-before* relationship [5] established by a conflicting transition could also be established by a pessimistic transition. The key challenge is when and how to change a conflicting transition into an equivalent pessimistic transition, without breaking the correctness of OCTET.

2. Adaptive State Transition Model

This section introduces a new state model that we call the *hybrid model*. At a high level, we add three new states to the OCTET state model: $\mathsf{PessiWrEx_T}$, $\mathsf{PessiRdEx_T}$, and $\mathsf{PessiRdSh_c}$. These new states are essentially the pessimistic version of the original three optimistic states, and they carry similar meaning. In addition, we introduce a special pessimistic state LOCKED, analogous to the existing intermediate states ($\mathsf{WrEx^{Int}}$ and $\mathsf{RdEx^{Int}}$), which is required for the correctness of pessimistic state transitions. The LOCKED state indicates a thread is already executing a pessimistic state transition on the object. We call an object in a pessimistic state a *pessimistic object*, and an object in an optimistic state an *optimistic object*. At run time, the application's heap contains a mix of optimistic and pessimistic objects.

The *adaptive policy* dynamically decides whether an object should become pessimistic or optimistic. The policy can choose to put an optimistic object into pessimistic state, in order to reduce conflicting transitions. It can also convert a pessimistic object back into optimistic state, if it decides the pessimistic state is not beneficial for the object.

2.1 State Transitions

Table 3 shows the complete set of state transitions when a thread attempts to perform an access. The first 10 rows are the same as in OCTET, covering all possible transitions between two optimistic states. The remaining 18 rows are transitions that involve pessimistic states. $toPessi()$ and $toOpti()$ are two decision-making functions of the adaptive policy, which Section 2.3 describes in detail. The model supports transitions between optimistic and pessimistic state independently from the adaptive policy's decisions.

In order to easily cooperate with the optimistic state model, the transition between two pessimistic states has exactly the same condition as the corresponding optimistic states. For example, $\mathsf{PessiWrEx_T}$ will stay in $\mathsf{PessiWrEx_T}$ if the access is a read or write by T; $\mathsf{PessiRdEx_{T1}}$ will change to $\mathsf{PessiRdSh_{gRdShCount}}$ if the access is a read by T2. Even though dedicated pessimistic states exist for reads ($\mathsf{PessiRdEx_T}$ and $\mathsf{PessiRdSh_c}$), pessimistic read accesses still synchronize with the LOCKED state. This requirement is due to the fact that another thread could try to perform a write access to a $\mathsf{PessiRdSh_c}$ object at any point, so a read access must hold the lock until it finishes reading.

For transitions from a pessimistic state to an optimistic state, our model only supports transitioning to optimistic $\mathsf{WrEx_T}$ or $\mathsf{RdEx_T}$, but not $\mathsf{RdSh_c}$. The sole purpose for this design is to simplify the model. Allowing transition from $\mathsf{PessiRdSh_c}$ directly to $\mathsf{RdSh_c}$ is a correct alternative. The difference between these two designs is negligible, due to the infrequency of pessimistic-to-optimistic transitions and the low cost of upgrading and fence transitions (compared to conflicting transitions).

2.2 Instrumentation

Figure 1 shows the barrier for adaptive state transitions. For simplicity, we only show the barrier for a write access. A read barrier is more complicated because it involves handling $\mathsf{RdSh_c}$ and $\mathsf{PessiRdSh_c}$ states.

The fast path (line 1) remains the same as purely optimistic OCTET. The instrumentation only extends the slow path. The method slowPath() (lines 2 and 12) returns a

Code path	Trans. type	Old state	Access	New state	Sync. needed	Cross-thread dependence?
Fast	Same state	WrEx_T	R or W by T	Same	None	No
		RdEx_T	R by T	Same		
		RdSh_c	R by T if $T.\text{rdShCount} \geq c$	Same		
Slow	Upgrading	RdEx_T	W by T	WrEx_T	CAS	No
		RdEx_{T1}	R by T2	$\text{RdSh}_{\text{gRdShCount}}$		Maybe
	Fence	RdSh_c	R by T if $T.\text{rdShCount} < c$	$(T.\text{rdShCount} = c)$	Memory fence	Maybe
	Conflicting	WrEx_{T1}	W by T2	$\text{WrEx}^{\text{Int}}_{T2} \rightarrow \text{WrEx}_{T2}$	Roundtrip coordination	Maybe
		WrEx_{T1}	R by T2	$\text{RdEx}^{\text{Int}}_{T2} \rightarrow \text{RdEx}_{T2}$		
		RdEx_{T1}	W by T2	$\text{WrEx}^{\text{Int}}_{T2} \rightarrow \text{WrEx}_{T2}$		
		RdSh_c	W by T	$\text{WrEx}^{\text{Int}}_{T} \rightarrow \text{WrEx}_{T}$		
		WrEx_{T1}	W by T2 if $toPessi(o)$	$\text{WrEx}^{\text{Int}}_{T2} \rightarrow \text{PessiWrEx}_{T2}$		
		WrEx_{T1}	R by T2 if $toPessi(o)$	$\text{RdEx}^{\text{Int}}_{T2} \rightarrow \text{PessiRdEx}_{T2}$		
		RdEx_{T1}	W by T2 if $toPessi(o)$	$\text{WrEx}^{\text{Int}}_{T2} \rightarrow \text{PessiWrEx}_{T2}$		
		RdSh_c	W by T if $toPessi(o)$	$\text{WrEx}^{\text{Int}}_{T} \rightarrow \text{PessiWrEx}_{T}$		
	Pess.	PessiWrEx_T	R or W by T	$\text{LOCKED} \rightarrow \text{PessiWrEx}_T$	CAS	No
		PessiRdEx_T	R by T	$\text{LOCKED} \rightarrow \text{PessiRdEx}_T$		
		PessiRdSh_c	R by T if $T.\text{rdShCount} \geq c$	$\text{LOCKED} \rightarrow \text{PessiRdSh}_c$		
		PessiRdEx_T	W by T	$\text{LOCKED} \rightarrow \text{PessiWrEx}_T$	CAS	No
		PessiRdEx_{T1}	R by T2	$\text{LOCKED} \rightarrow \text{RdSh}_{\text{gRdShCount}}$		Maybe
		PessiRdSh_c	R by T if $T.\text{rdShCount} < c$	$\text{LOCKED} \rightarrow \text{PessiRdSh}_c$ $(T.\text{rdShCount} = c)$	CAS	Maybe
		PessiWrEx_{T1}	W by T2	$\text{LOCKED} \rightarrow \text{PessiWrEx}_{T2}$	CAS	Maybe
		PessiWrEx_{T1}	R by T2	$\text{LOCKED} \rightarrow \text{PessiRdEx}_{T2}$		
		PessiRdEx_{T1}	W by T2	$\text{LOCKED} \rightarrow \text{PessiWrEx}_{T2}$		
		PessiRdSh_c	W by T	$\text{LOCKED} \rightarrow \text{PessiWrEx}_{T}$		
		PessiWrEx_T	R or W by T if $toOpti(o)$	$\text{LOCKED} \rightarrow \text{WrEx}_T$	CAS	No
		PessiRdEx_T	R by any thread T′ if $toOpti(o)$	$\text{LOCKED} \rightarrow \text{RdEx}_{T'}$		
		PessiRdEx_T	W by T if $toOpti(o)$	$\text{LOCKED} \rightarrow \text{WrEx}_T$		
		PessiRdSh_c	R by T if $toOpti(o)$	$\text{LOCKED} \rightarrow \text{RdEx}_T$		

Table 3. The hybrid model's state transitions fall into five categories, including adaptively switching to and from pessimistic states.

nonzero value if and only if the new state is from a pessimistic state transition.

The method slowPath() (line 12) contains two nested loops. The outer loop (lines 14–17 and lines 34–35) and the code after it (lines 36–42) are the same as handling a conflicting transition in purely optimistic OCTET, except that the $toPessi()$ function determines if it should change to a pessimistic state (line 42). Inside the outer loop, the instrumentation checks if the current state is pessimistic (line 18). If it is, the instrumentation tries to CAS it into the LOCKED state and uses the decision from $toOpti()$ to perform a pessimistic state transition (lines 19–31). If the CAS is successful, slowPath() simply returns the new state (line 31) to be used by line 6. If the object is changed to an optimistic state by another thread, the instrumentation breaks out of the inner loop of the pessimistic transition, and continues the outer loop for a conflicting transition (line 26). Two important invariants of slowPath() are: (i) when the current thread is about to execute line 34, oldState must be an optimistic state; (ii) when it is about to exectute line 19, oldState must be a pessimistic state.

2.3 Adaptive Policy

The ultimate goals for the adaptive policy are (i) to eliminate as many conflicting transitions as possible, while (ii) continuing to use optimistic states for most accesses that cannot create a cross-thread dependence. To achieve the first goal, the policy puts an object into a pessimistic state if it speculates that there will soon be some conflicting transitions on this object. For the second part of the goal, we apply our cost–benefit model to approximate if a pessimistic state for a particular object is expected to improve performance; if not, that object is converted back into optimistic. We add extra per-object metadata (o.adpinfo) that maintains information needed for the adaptive policy.

In our current policy, an object can be switched from optimistic to pessimistic at most once for the following reason. If an object was switched back from pessimistic to optimistic, then we have reason to believe that it is not worthwhile to put the object in pessimistic state, so there is no more need to speculate on future conflicting transitions of this object.

2.3.1 Cost–Benefit Model

We design a cost–benefit model that the adaptive policy uses to decide if a pessimistic state is beneficial for an object. The

Fast path for a write access:

```
1  if (obj.state != WrEx_T) {
2    newState = slowPath(obj);
3    if (newState != 0) { // slowpath() returns a valid state
4      obj.f = ... ; // duplicate program write
5      memfence;
6      obj.state = newState; // unlock and update metadata for
             pessimistic transition
7      goto L11; // jump after the original write
8    }
9  }
10 obj.f = ...; // program write
11
```

Slow path for a write access:

```
12 int slowPath(Object obj) {
13 outer:
14   do {
15     // non-blocking safe point:
16     if (requestsSeen()) { handleRequests(); }
17     oldState = obj.state;
18     if ( isPessimisticState (oldState)) {
19       while (oldState == LOCKED ||
20               !(&obj.state, oldState, LOCKED)) {
21         // non-blocking safe point:
22         if (requestsSeen()) { handleRequests(); }
23         oldState = obj.state;
24         if (! isPessimisticState (oldState)) {
25           // Another thread has changed the object to an
                 optimistic state
26           continue outer;
27         }
28       }
29       newState = toOpti(o) ? WrEx_T : PessiWrEx_T ;
30       // newState is determined by the adaptive policy
31       return newState;
32     }
33   }
34   while (oldState == any intermediate state ||
35           !CAS(&obj.state, oldState, WrEx_T^Int));
36   handleRequestsAndBlock(); // start blocking safe point
37   response = request(getOwner(oldState));
38   while (!response) {
39     response = status(getOwner(oldState));
40   }
41   resumeRequests(); // end blocking safe point
42   obj.state = toPessi(o) ? PessiWrEx_T : WrEx_T ;
43   // newState is determined by the adaptive policy
44   return 0;
45 }
```

Figure 1. The barrier for adaptive state transitions for a write access. obj.state is per-object metadata; $toOpti()$ and $toPessi()$ are the decision-making functions of the adaptive policy (Sections 2.3).

basic idea is to calculate the total time spent on state transitions if an object becomes pessimistic versus if it stays optimistic, or vice versa. A pessimistic state is beneficial if and only if the total time spent on pessimistic state transitions is less than the total time of optimistic state transitions.

The cost–benefit model simply counts upgrading and fence transitions as fast path, since their quantity is low compared to fast paths and their cost is low compared to conflicting transitions.

We now formalize the cost–benefit model as follows:

Let T_{fpath}, T_{confl}, and T_{pessi} be the average time costs for a fast path, a conflicting transition, and a pessimistic transition, respectively.

Let N_{fpath} and N_{confl} be the numbers of fast paths and conflicting transitions for an object if it is optimistic; and N_{pessi} be the number of pessimistic transitions if the object is pessimistic. Clearly:

$$N_{pessi} = N_{fpath} + N_{confl} \tag{1}$$

The model considers a pessimistic state to be no longer beneficial if the following equation is satisified:

$$T_{pessi} \times N_{pessi} \geq T_{fpath} \times N_{fpath} + T_{confl} \times N_{confl} \tag{2}$$

In (2), the left-hand side is the total time spent on pessimistic state transitions if the object is pessimistic. The right-hand side is the total time on state transitions if the object stays optimistic, including time on fast paths and conflicting transitions.

Applying (1) into (2) and transforming the formula, we get:

$$N_{fpath} \geq \frac{T_{confl} - T_{pessi}}{T_{pessi} - T_{fpath}} \times N_{confl} \tag{3}$$

We define the coefficient as K:

$$K = \frac{T_{confl} - T_{pessi}}{T_{pessi} - T_{fpath}} \tag{4}$$

and (3) becomes:

$$N_{fpath} \geq K \times N_{confl} \tag{5}$$

For a given application on a specific hardware and OS platform, it is reasonable to consider T_{fpath}, T_{confl}, and T_{pessi} as constants, which means K is also an application-and-platform dependent constant.[1] Thus if we know the value of K, the policy can tell if a pessimistic state would be beneficial by counting the number of fast paths and conflicting transitions.

Nevertheless, it is challenging to directly apply the cost–benefit model to decide if an object should become pessimistic, without affecting performance significantly. It is expensive to perform online profiling on an optimistic object: counting the fast paths is costly because they occur frequently. Offline profiling is possible but undesirable.

Instead, we use a simple heuristic that only relies on counting conflicting transitions to decide if an object should become pessimistic. Then, while the object is in the pessimistic state, we apply the cost–benefit model to determine whether an object should be switched back to an optimistic state.

[1] This assumption ignores the fact that different instances of the same kind of transition—particularly conflicting transitions—can have significantly different costs.

2.3.2 Transitioning to a Pessimistic State

In order to design a policy that makes predictions about future conflicting transitions of an object at run time, we have studied the patterns of conflicting transitions in several applications from the DaCapo, SPECjbb2000, SPECjbb2005, and STAMP benchmarks [2, 4, 6, 7]. We have observed two main types of locality effects with conflicting transitions:

- *Object conflict locality*: An object that has a lot of conflicting transitions in the past is likely to have conflicting transitions in the future.
- *Program site conflict locality*: A site (line of code) in a program that has caused a lot of conflicting transitions in the past is likely to have conflicting transitions in the future. A special case is the allocation site of an object. Objects allocated at a particular site can have many more conflicting transitions than objects allocated at other sites.

Our current policy exploits only *object conflict locality*. At run time, it counts the number of conflicting transitions that ever happened on an optimistic object, and stores the counter in per-object metadata (o.adpinfo). At every conflicting transition, the decision-making function $toPessi()$ simply checks if the following equation is satisfied:

$$\text{o.adpinfo} \geq \mathit{Cutoff}_{\mathit{confl}} \qquad (6)$$

Here $\mathit{Cutoff}_{\mathit{confl}}$ is a constant threshold value. If (6) is satisfied, the policy switches the object to pessimistic state. Note that optimistic-to-pessimistic-state transitions are conflicting transitions themselves.

This heuristic can effectively reduce the number of conflicting transitions for some applications, such as pjbb2005. However, performance can be quite sensitive to $\mathit{Cutoff}_{\mathit{confl}}$ for some applications because of the tradeoff between reducing conflicting transitions versus sacrificing fast paths. For example, experiments on 32 cores of a 64-core machine show that avrora9 has the best performance when $\mathit{Cutoff}_{\mathit{confl}}$ is 4, while sunflow9 highly favors a $\mathit{Cutoff}_{\mathit{confl}}$ of 4096. For this reason, the adaptive policy allows an object to switch back to optimistic (Section 2.3.3).

It is possible to implement *program site conflict locality* purely at run time, but it could add considerable overhead to track conflicting transitions at either all program sites, or only allocation sites. Furthermore, it could require dynamic recompilation of sites in order to change their behavior.

2.3.3 Transitioning Back to an Optimistic State

The decision-making function $toOpti()$ utilizes our cost–benefit model (Section 2.3.1) to determine if it is beneficial to keep an object in pessimistic state. The per-object metadata o.adpinfo stores the *benefit value* of a pessimistic object.

When an object first becomes pessimistic, we assume there exists sufficient evidence to believe that the pessimistic state is likely to be beneficial, i.e., that the object will soon experience conflicting transitions. Our model thus incorporates an "initial benefit" $Inertia_o$ into the $toPessi()$ function.

Since the optimistic and pessimistic states both include write-exclusive, read-exclusive, and read-shared states, profiling can easily determine whether the current pessimistic transition *would* take a fast path, upgrading/fence, or conflicting transition if the object *were* in an optimistic state. In this sense, N_{fpath}, N_{confl} are the numbers of *would-be* fast paths and *would-be* conflicting transitions since an object became pessimistic. Let N_{pessi} be the actual number of pessimistic transitions for the object.

At every pessimistic transition that is a would-be fast path, the function $toOpti()$ computes whether the following formula, derived from equation (5), is satisfied:

$$Inertia_o + K \times N_{\mathit{confl}} - N_{\mathit{fpath}} \leq 0 \qquad (7)$$

If the formula is satisfied, the policy transitions from the pessimistic state to an optimistic state (WrEx_T or RdEx_T).

We can implement (7) efficiently using a single value: after the first optimistic-to-pessimistic transition, assign $Inertia_o$ to o.adpinfo; for every pessimistic transition, if it is a would-be fast path, decrement o.adpinfo by 1; if it is a would-be conflicting transition, increment o.adpinfo by K.

Experiments (not shown) show that the results are not sensitive to the values of K and $Inertia_o$. Most objects can be distinctly classified as optimistic or pessimistic, so any "reasonable" K (8–512) and $Inertia_o$ (8–1024) effectively identifies suitable states for objects.

3. Implementation

We have implemented the hybrid state model and adaptive policy on top of our OCTET prototype [3] inside Jikes RVM 3.1.3 [1].

We modify the representation of OCTET state metadata so that it uses the lowest three (instead of two) bits to denote eight states: three optimistic states and the intermediate state; and three pessimistic states and the LOCKED state. $\mathsf{PessiRdSh}_\mathsf{c}$ is soundly and efficiently represented using the unused range [0x00000000, 0x3fffffff).

The implementation adds a metadata word per object and static field (in addition to the OCTET state word) to maintain o.adpinfo.

We have not yet implemented the fast path as shown in Figure 1 so that the barrier unlocks the pessimistic metadata only after the program access. Instead, the barrier currently unlocks pessimistic metadata *before* the program access.

4. Evaluation

This section evaluates the performance and run-time characteristics of our adaptive extensions to OCTET.

Methodology. We evaluate our modified Jikes RVM on the DaCapo Benchmarks [2] versions 2006-10-MR2 and

9.12-bach (2009), and fixed-workload versions of SPECjbb-2000 and SPECjbb2005 called pjbb2000 and pjbb2005 [2, 6, 7].[2] We include only multithreaded benchmarks, and we exclude a few benchmarks that (unmodified) Jikes RVM cannot exeute correctly. DaCapo benchmark names from the 2006 and 2009 versions have suffixes of 6 and 9, respectively.

Experiments execute on a machine with an Intel i5-2400 4-core CPU and 4GB memory, running Linux 2.6.32. We build a high-performance configuration FastAdaptiveGenImmix of Jikes RVM. We let the garbage collector adjust the heap size automatically at run time. The performance result is the median of 25 trial runs. We also show the mean as the center of 95% confidence intervals.

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

- $\mathit{Cutoff}_{\mathit{confl}}$: 4
- K: 200
- $\mathit{Inertia}_o$: 100

As mentioned previously, performance is not sensitive to the values of K and $\mathit{Inertia}_o$. We choose a low value for $\mathit{Cutoff}_{\mathit{confl}}$, so the policy transitions optimistic objects to pessimistic states aggressively, since pessimistic-to-optimistic transitions can remedy pessimistic objects that should actually have optimistic behavior.

4.1 State Transitions

Table 4 shows the percentage of each type of state transition in adaptive OCTET. The last two columns show the percentages of conflicting and same-state transitions eliminated by our adaptive policy, compared with original OCTET (Table 1). The more conflicting transitions eliminated, the more performance improves, as Section 4.2 shows. The tradeoff is that some fast paths become pessimistic transitions. Our policy works quite well for applications that have a high ratio of conflicting transitions. It reduces conflicting transitions by 72–97% for hsqldb6, xalan6, avrora9, xalan9, and pjbb2005. It provides little or no improvement for applications with few conflicting transitions—but these applications incur little overhead anyway from conflicting transitions.

These results indicate that the solution derived from our object conflict locality assumption is effective for applications with a high rate of conflicting accesses. To reduce the number of conflicting transitions in other applications, our future work will explore different adaptive policies.

4.2 Performance

Figure 2 compares the performance of adaptive OCTET with alternatives that use only optimistic and pessimistic states. Each bar shows normalized execution time, normalized to unmodified Jikes RVM (*Base*).

[2] `http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005`

The two *Pure Optimistic* configurations show the overhead added by original OCTET (i.e., with optimistic states only). *Pure Optimistic w/o coord*, which adds 22% overhead on average, essentially shows only the fast-path-only overhead (since it unsoundly skips the coordination protocol), so it can be considered an approximate lower bound on the overhead that the adaptive policy might be able to provide.

The two *Adaptive* configurations use the hybrid model and adaptive policy to switch between optimistic and pessimistic states. *Adaptive w/infinite cutoff* sets $\mathit{Cutoff}_{\mathit{confl}}$ to ∞, so no object actually transitions to a pessimistic state, so this configuration measures the cost but not the benefit of adaptive OCTET. These costs are 1.6% (relative to baseline execution) over purely optimistic OCTET. They come from adding a metadata word per object, and maintaining this word and comparing it to $\mathit{Cutoff}_{\mathit{confl}}$ on each conflicting transition.

Adaptive uses the default values for $\mathit{Cutoff}_{\mathit{confl}}$ and other constants and adds 30% overhead on average. It significantly improves the performance of several programs that perform poorly with a purely opimistic model. Unsurprisingly, the benefit is greater for programs that have many conflicting transitions eliminated by the adaptive policy, as shown in Table 4. Adaptive OCTET reduces overhead by 42% (from 75% to 33%) for avrora9, and by 29% (from 77% to 48%) for pjbb2005. Adaptive OCTET also improves the performance of xalan6. Despite reducing conflicting transitions significantly for hsqldb6 (Table 4), adaptive OCTET does not improve its performance much because most of its conflicting transitions use the implicit protocol, which have a similar cost to pessimistic transitions.

Our adaptive policy never significantly degrades performance relative to purely optimistic OCTET, and it sometimes improves performance substantially.

For completeness, *Pure Pessimistic* evaluates using only pessimistic states, which prior work also shows is expensive [3]. This configuration slows programs by 3.7X on average, showing that pessimistic states must be applied judiciously.

5. Conclusion

We present a novel approach that adaptively switches between optimistic and pessimistic states, in order to more efficiently capture cross-thread dependences for applications with high ratios of conflicting accesses. An evaluation of our implementation on top of OCTET shows that it significantly improves performance over the previous, purely optimistic approach, especially for highly conflicting applications.

References

[1] 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-

	Transitions						Reductions	
	Same state	Conflicting	Upgrading or fence	Pessimistic	Optimistic to Pessimistic	Pessimistic to Optimistic	Conflicting eliminated	Same state sacrificed
eclipse6	99.9971%	0.0011%	0.00050%	0.0013%	0.0000012%	0.0000011%	0%	0.0013%
hsqldb6	98.7%	0.045%	0.039%	1.2%	0.0021%	0.000038%	71.88%	1.03%
lusearch6	99.99965%	0.00018%	0.00010%	0.000064%	0.00000016%	0%	0%	0.000060%
xalan6	98.0%	0.0030%	0.00063%	1.9%	0.0000050%	0.00000098%	97.5%	1.80%
avrora9	99.74%	0.017%	0.027%	0.21%	0.0025%	0.00000067%	82.11%	0.07%
jython9	99.9999984%	0.0000013%	0.00000030%	0%	0%	0%	0%	0%
luindex9	99.99982%	0.00011%	0.000064%	0%	0%	0%	0%	0%
lusearch9	99.99966%	0.000081%	0.000025%	0.00023%	0.00000047%	0.000000086%	26.36%	0.00019%
pmd9	99.961%	0.0033%	0.0013%	0.034%	0.000018%	0.00000081%	51.57%	0.027%
sunflow9	99.99982%	0.000033%	0.000044%	0.0013%	0.0000012%	0.0000011%	8.33%	0.0001%
xalan9	98.5%	0.010%	0.0011%	1.5%	0.0000048%	0.0000011%	89.36%	1.34%
pjbb2000	99.49%	0.047%	0.045%	0.42%	0.0031%	0.00023%	14.55%	0.40%
pjbb2005	98.0%	0.013%	0.0050%	2.0%	0.000058%	0.000056%	97.45%	1.17%

Table 4. Adaptive OCTET state transitions statistics. The rightmost two columns are the percentages of conflicting and same-state transitions reduced by our adaptive approach, compared with Table 1.

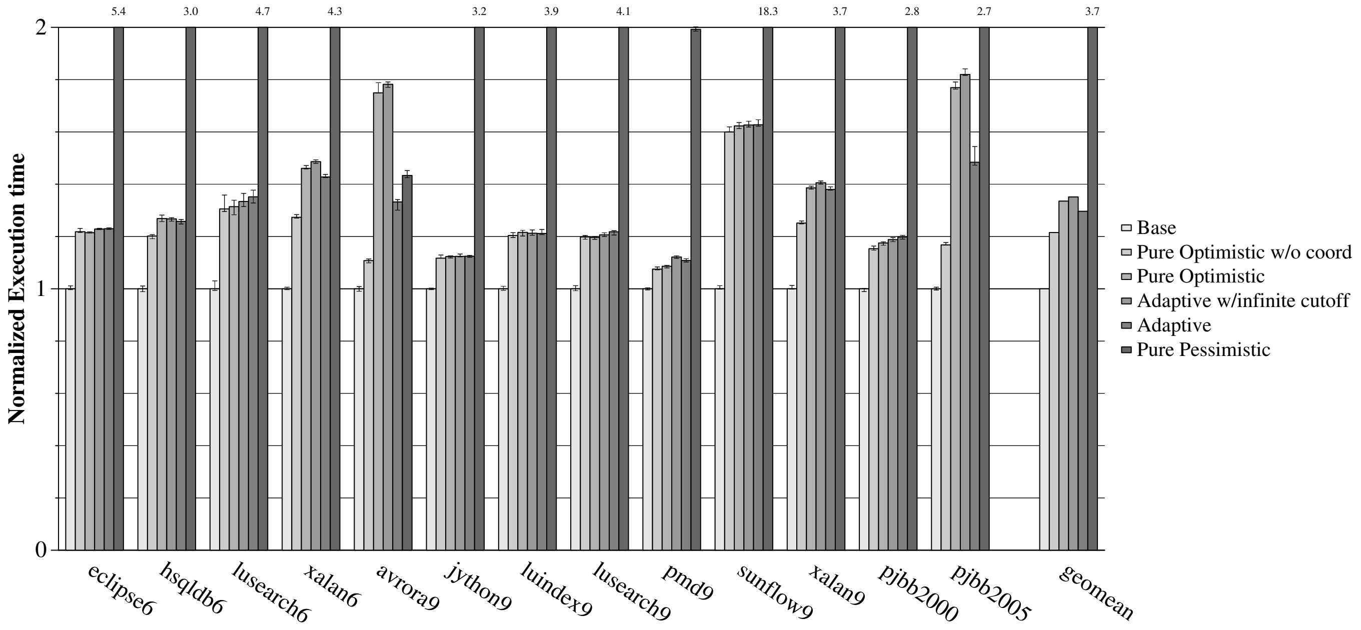

Figure 2. Run-time performance of our hybrid model and adaptive policy, compared with purely optimistic and pessimistic states. The ranges are 95% confidence intervals centered at the mean.

Source Research Community. *IBM Systems Journal*, 44:399–417, 2005.

[2] 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.

[3] 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*, 2013. To appear.

[4] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IEEE International Symposium on Workload Characterization*, 2008.

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

[6] Standard Performance Evaluation Corporation. *SPECjbb2000 Documentation*, release 1.01 edition, 2001.

[7] Standard Performance Evaluation Corporation. *SPECjbb2005 Documentation*, release 1.04 edition, 2005.

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

[9] M. Zhang, J. Huang, M. Cao, and M. D. Bond. LarkTM: Efficient, Strongly Atomic Software Transactional Memory. Submitted.