Legato: End-to-End Bounded Region Serializability Using Commodity Hardware Transactional Memory [*]

Aritra Sengupta Man Cao Michael D. Bond
Ohio State University (USA)
{sengupta,caoma,mikebond}@cse.ohio-state.edu

Milind Kulkarni
Purdue University (USA)
milind@purdue.edu

This tech report extends the CGO 2017 paper with Appendices A and B

Abstract

Shared-memory languages and systems provide strong guarantees only for well-synchronized (data-race-free) programs. Prior work introduces support for memory consistency based on region serializability of executing code regions, but all approaches incur serious limitations such as adding high run-time overhead or relying on complex custom hardware.

This paper explores the potential for leveraging widely available, commodity hardware transactional memory to provide an end-to-end memory consistency model called *dynamically bounded region serializability* (DBRS). To amortize high per-transaction costs, yet mitigate the risk of unpredictable, costly aborts, we introduce dynamic runtime support called *Legato* that executes multiple dynamically bounded regions (DBRs) in a single transaction. Legato varies the number of DBRs per transaction on the fly, based on the recent history of committed and aborted transactions. Legato outperforms existing commodity enforcement of DBRS, and its costs are less sensitive to a program's shared-memory communication patterns. These results demonstrate the potential for providing always-on strong memory consistency using commodity transactional hardware.

1. Introduction

It is notoriously challenging to achieve both correctness and scalability in the context of shared-memory languages and systems. A fundamental challenge is that, for performance reasons, languages and systems provide intuitive, well-defined semantics only for *well-synchronized* executions. For *ill-synchronized* executions (executions with data races), languages and systems provide weak, unexpected, and sometimes undefined behaviors [1, 15, 41].

Prior work has introduced compiler, runtime, and system support for stronger memory consistency models [3, 4, 20, 27, 28, 34, 35, 38, 42, 43, 51, 58, 61, 63, 64, 67]. However, existing work suffers from serious limitations: it is either not *end-to-end*, i.e., it does not provide guarantees with respect to the original program; adds high run-time overhead; or it relies on complex custom hardware support (Section 2). A promising direction in this space is support for *serializability of bounded regions*, i.e., an execution is equivalent to some serial execution of bounded regions of code [4, 20, 39, 42, 58, 63]. While existing support for bounded region serializability suffers from the aforementioned limitations, it offers the potential for a practical balance between performance and strength.

This paper's approach provides support for an end-to-end memory model called *dynamically bounded region serializability* (DBRS). DBRS guarantees atomic execution for regions of code between well-defined program points: loop back edges, method calls and returns, and synchronization operations. This guarantee with respect to the original program has the potential to simplify the job of analyses, systems, and developers. Prior work called *EnfoRSer* provides end-to-end support for a closely related memory model called *statically bounded region serializability* [58], but EnfoRSer requires complex compiler transformations, and its performance is sensitive to a program's memory access communication patterns (Sections 2.2 and 5.3).

This paper introduces an approach called *Legato* that provides end-to-end DBRS using commodity *hardware transactional memory* (HTM) [30, 31, 71]. Our implementation targets Intel's *Transactional Synchronization Extensions* (TSX) [71], which is widely available on mainstream commercial processors. While enforcing DBRS with commodity HTM seems straightforward, commodity HTM has two relevant limitations. First, the cost of starting and stopping a transaction is relatively high [55, 71]. We find that a naïve implementation of DBRS that executes each dynamically bounded region (DBR) in its own transaction, slows programs on average by 2.7X (Section 5.3). Second, commodity HTM is "best effort," meaning that *any* transaction might abort, requiring a non-transactional fallback.

Legato overcomes the first challenge (high per-transaction costs) by merging multiple DBRs into a single transaction. However, longer transactions run into the second challenge, since they are more likely to abort for reasons such as memory access conflicts, private cache misses, and unsupported instructions and events. Legato thus introduces a dynamic, online algorithm that decides, on the fly, how many DBRs to merge into a transaction, based on the history of recently at-

[*] This material is based upon work supported by the National Science Foundation under Grants CSR-1218695, CAREER-1253703, CCF-1421612, CCF-1422178, and XPS-1629126.

tempted transactions, considering both *transient* and *phased* program behavior. For the (rare) single-DBR transactions that commodity HTM cannot commit, Legato falls back to a global lock to ensure progress.

Our evaluation compares our implementation of Legato with the closest related prior work, EnfoRSer [58], on benchmarked versions of large, real, multithreaded Java applications [12]. In addition to outperforming EnfoRSer on average, Legato provides two key advantages over EnfoRSer. First, Legato provides stable performance across programs, whereas EnfoRSer's approach is sensitive to an execution's amount of shared-memory communication. For programs with more communicating memory accesses, EnfoRSer slows them considerably, and Legato provides significantly lower overhead. Second, Legato adds significantly less compiler complexity and costs than EnfoRSer—reducing execution costs for just-in-time compilation. Overall, Legato demonstrates the potential benefits—and the inherent limits—of using commodity HTM to enforce strong memory consistency.

2. Background and Motivation

This section first motivates and explains the memory model enforced by Legato, dynamically bounded region serializability (DBRS). We then describe Intel's hardware transactional memory (HTM) and explain why its limitations make it challenging to provide DBRS.

2.1 Memory Models

Memory models define the possible behaviors of a multithreaded program execution [1–3, 15, 41]. If a memory model's guarantees are with respect to the original source program, then the memory model is *end-to-end*, and both the compiler and hardware must respect it. In contrast, hardware memory models provide guarantees with respect to the *compiled* program only [18, 34, 40, 65, 66].

DRF0 and its variants. Memory consistency models for shared-memory languages, including C++ and Java's memory models [15, 41], are based on the *DRF0* memory model introduced by Adve and Hill in 1990 [3]. DRF0 assumes that executions do not have any data races.[1] By assuming data race freedom, DRF0 permits compilers and hardware to perform optimizations that ignore possible inter-thread effects, as long as they do not cross synchronization boundaries.

As long as an execution is free of data races, DRF0 (and its variants) provide a guarantee of *sequentially consistency* [3, 33]. This guarantee in turn provides an even stronger property: synchronization-free regions (SFRs) of code appear to execute atomically; that is, the execution is equivalent to some serial execution of SFRs [3, 38]. However, for executions with data races, DRF0 provides few, if any, guarantees [15, 16, 19, 41, 60].

[1] A *data race* exists if two accesses to the same variable are *concurrent*—not ordered by the *happens-before relation*, which is the union of synchronization and program order [32]—and *conflicting*—executed by different threads and at least one is a write.

Sequential consistency. Much work has focused on providing *sequential consistency* (SC) as a memory model [1, 4, 20, 34, 35, 43, 54, 61, 64, 67]. Under SC, operations appear to interleave in program order [33]. Despite much effort, it is not clear that SC provides a compelling tradeoff between strength and performance [1]. Enforcing end-to-end SC generally requires restricting optimizations in both the compiler and hardware. Many SC-enforcing systems only target hardware [28, 34, 35, 54] or the compiler [43], lacking end-to-end guarantees. Further, SC's guarantees are relatively weak and unintuitive for programmers, and SC does not eliminate many concurrency bugs [1, 58].

2.2 Region Serializability

Prior work introduces memory models based on *region serializability* (RS), in which regions of code appear to execute atomically and in program order [4, 11, 20, 38, 39, 42, 51, 58, 63]. RS is appealing not only because it has the potential to be stronger than SC, but also because it permits compiler and hardware optimizations within regions.

Some prior work provides serializability of *unbounded regions*: regions bounded by synchronization or enclosed in transactions [7, 11, 13, 14, 21, 29, 30, 38, 46, 51, 52, 70]. Unbounded RS incurs high run-time overhead or relies on complex custom hardware support, in order to detect and resolve conflicts between arbitrarily long regions of code. Furthermore, unbounded RS leads to potentially unintuitive progress guarantees: some programs guaranteed to terminate under SC cannot terminate under unbounded RS [51, 58].

Bounded RS. In contrast, approaches that support *bounded RS*—in which each executed region performs a bounded amount of work—have the potential to avoid the cost and complexity associated with supporting unbounded regions. This paper's focus is on *end-to-end* bounded RS, which several existing approaches provide by enforcing atomic execution and restricting cross-region optimizations in both the compiler and hardware [4, 42, 58, 63]. Nonetheless, most of these approaches (with one exception [58], described below) choose region boundary points arbitrarily and thus cannot provide any guarantee beyond end-to-end SC; furthermore, they require custom hardware extensions to caches and coherence protocols [4, 42, 63].

In this paper, we focus on a memory model that provides end-to-end guarantees, with respect to the source program, that are strictly stronger than SC. We call this memory model *dynamically bounded region serializability* (DBRS). Under DBRS, regions of code that are guaranteed to be dynamically bounded (called *DBRs*) execute atomically. In particular, we define the boundaries of DBRs to be loop back edges and method calls and returns, since an execution cannot execute an unbounded number of steps without reaching a back edge, call, or return. Synchronization operations are also DBR boundaries, to ensure progress.

In a DBRS-by-default world, analyses, systems, and developers can make simplifying assumptions. Analyses and systems such as model checkers [47] and multithreaded record & replay [68] can consider many fewer interleavings, compared with SC or weaker models. Developers can reason easily about regions of code that will execute atomically.

Furthermore, DBRS improves reliability automatically, by providing atomicity for regions of code that programmers may *already assume* to be atomic. Prior work shows that DBRS (called SBRS in that work, but essentially with the same region boundaries) eliminates erroneous behaviors due to data races—even errors that are possible under SC [58].

Prior work providing bounded RS in commodity systems. Among prior approaches providing bounded RS, only one, EnfoRSer [58], does not rely on custom hardware. In this paper, our focus is also on supporting commodity systems—which now provide best-effort HTM. While EnfoRSer has relatively low overhead (36% on average in prior work [58] and 40% on average in our experiments), it suffers from several serious drawbacks. First, it uses complex compiler analyses and transformations that may limit adoption and affect execution time in a just-in-time compilation environment. Second, EnfoRSer achieves low overhead by employing a form of lightweight, *biased* reader–writer lock [17] that adds low overhead for programs with relatively few inter-thread dependences. For programs with even modest levels of communication, the biased locks slow programs substantially [17, 58]. (In general, software approaches that do *not* use biased locking to provide RS, such as STMs, slow programs by 2–3X or more [24, 62].)

To our knowledge, EnfoRSer is the only existing approach that provides end-to-end bounded RS guarantees with respect to the original program. EnfoRSer's static and dynamic components rely on regions being both *statically and dynamically* bounded; *statically bounded* means that when the region executes, *each static instruction executes at most once*. EnfoRSer thus introduces and enforces a memory model called *statically bounded region serializability* (SBRS), which bounds regions at loop back edges and method calls and returns, as well as synchronization operations. This paper's implementation uses the same boundaries for *DBRS* as EnfoRSer used for SBRS, thus enforcing the same memory model. DBRS requires *only* dynamic boundedness and in theory need not be as strict as SBRS. For example, one could extend DBRs by omitting region boundaries at leaf method calls and returns and at loop back edges for loops with bounded trip counts.

2.3 Commodity Hardware Transactional Memory

Recent mainstream commodity processors support hardware transactional memory (HTM) [22, 71].[2] We focus on Intel's *Transactional Synchronization Extensions* (TSX), which provide two interfaces: *hardware lock elision* (HLE) and *restricted transactional memory* (RTM). RTM allows programmers to provide fallback actions when a transaction aborts, whereas HLE can only fall back to acquiring a lock [26, 71], so we use RTM in order to have finer control.

[2] Non-commodity processors, such as IBM's Blue Gene/Q [69] and Azul's systems [23], provided HTM somewhat earlier.

RTM extends the ISA to provide XBEGIN and XEND instructions; upon encountering XBEGIN, the processor attempts to execute all instructions transactionally, committing the transaction when it reaches XEND. The XBEGIN instruction takes an argument that specifies the address of a fallback handler, from which execution continues if the transaction aborts. When a transaction aborts, RTM does the following: reverts all architectural state to the point when XBEGIN executed; sets a register to an error code that indicates the abort's proximate cause; and jumps to the fallback handler, which may re-execute the transaction or execute non-transactional fallback code. RTM also provides XABORT, which explicitly aborts an ongoing transction, and XTEST, which returns whether execution is in an RTM transaction.

2.4 Commodity HTM's Limitations and Challenges

Intel's commodity HTM has two main constraints that limit its direct adoption for providing region serializability with low overhead: best-effort completion and run-time overhead. Prior work that leverages Intel TSX for various purposes has encountered similar issues [8, 36, 37, 45, 50, 55, 56, 71, 72]. (Prior work uses TSX for purposes that are distinct from ours, e.g., optimizing data race detection. An exception is observationally cooperative multithreading [50]; Section 6.)

Best-effort speculative execution. The TSX specification explicitly provides no guarantees that *any* transaction will commit. In practice, transactions abort for various reasons:

Data conflicts: While a core executes a transaction, if any other core (whether or not it is in a transaction) accesses a cache line accessed by the transaction in a conflicting way, the transaction aborts.

Evictions: In general, an eviction from the L1 or L2 private cache (depending on the TSX implementation) triggers an abort. Thus a transaction can abort simply because the footprint of the transaction exceeds the cache.

Unsupported instructions: The TSX implementation may not support execution of certain instructions, such as CPUID, PAUSE, and INT within a transaction.

Unsupported events: The TSX implementation may not support certain events, such as page faults, context switches, and hardware traps.

Other reasons: A transaction may abort for other, potentially undocumented, reasons.

Overheads of transactional execution. A transactional abort costs about 150 clock cycles, according to Ritson and Barnes [55]—which is on top of the costs of re-executing the transaction speculatively or executing fallback code.

Even in the absence of aborts, simply executing in transactions incurs overhead. Prior work finds that each transaction (i.e., each non-nested XBEGIN–XEND pair) incurs fixed "startup" and "tear-down" costs approximately equal to the overhead of three uncontended atomic operations (e.g., test-and-set or compare-and-swap) [55, 71]. Prior work also finds that transactional reads have overhead [55], meaning that executing a transaction incurs non-fixed costs as well.

3. Legato: Enforcing DBRS with HTM

Enforcing the DBRS memory model with HTM appears to be a straightforward proposition. The compiler can enclose each dynamically bounded region (DBR) in a hardware transaction; the atomicity of hardware transactions thus naturally provides region serializability. Unfortunately, this straightforward approach has several challenges. First, the operations to begin and end a hardware transaction are quite expensive—nearly the cost of three atomic operations per transaction [55, 71] (Section 2.4). Because DBRs can be quite short, incurring the cost of three atomic operations per region can lead to significant overhead; indeed, on the benchmarks we evaluate, we find that simply placing each DBR in a hardware transaction leads to 175% overhead over an unmodified JVM (see Section 5.3). On the other hand, if a transaction is large, a second problem arises: commodity HTM is best-effort, so transactional execution of the DBR is not guaranteed to complete, e.g., if the memory footprint of a transaction exceeds hardware capacity (Section 2.4). Third, even short regions of code may contain operations such as page faults that cannot execute inside an HTM transaction.

3.1 Solution Overview

We propose a novel solution, called *Legato*,[3] to enforce DBRS at reasonable overheads, overcoming the challenges posed by expensive HTM instructions. Our key insight lies in amortizing the cost of starting and ending transactions by merging multiple DBRs into a single hardware transaction. Note that merging DBRs into a larger atomic unit does not violate the memory model: if a group of DBRs executes atomically, the resulting execution is still equivalent to a serialization of DBRs.

Rather than merging together a fixed number of regions into a single transaction (which could run afoul of the other problems outlined above), we propose an adaptive merging strategy that varies the number of regions placed into a hardware transaction based on the history of recent aborts and commits, to adapt to transient and phased program behavior. As transactions successfully commit, Legato increases the number of regions placed into a single transaction, further amortizing HTM overheads. To avoid the problem of large transactions repeatedly aborting due to HTM capacity limitations, Legato responds to an aborted transaction by placing fewer regions into the next transaction. Section 3.2 describes Legato's merging algorithm in detail.

Legato's merging algorithm addresses the first two issues with using HTM to enforce DBRS. The third issue, where HTM is unable to complete a single region due to incompatible operations, can be tackled by using a fallback mechanism that executes a DBR using a global lock, which is acceptable since fallbacks are infrequent.

3.2 Merging Regions to Amortize Overhead

If Legato is to merge DBRs together to reduce overhead, the obvious question is how many regions to merge together to execute in a hardware transaction. At a minimum, each region can be executed separately, but as we have discussed, this leads to too much overhead. At the maximum, regions can be merged until "hard" boundaries are hit: operations such as thread fork and monitor wait that inherently interrupt atomicity. While this maximal merging would produce the lowest possible overhead from transactional instructions, it is not possible in practice, due to practical limitations of transactional execution, as well as specific limitations introduced by commodity HTMs, such as Intel's TSX. We identify three issues that exert downward pressure on the number of regions that can be executed as a single transaction:

1. Conflicts between transactions will cause one of the transactions to abort and roll back, wasting any work performed prior to the abort. Larger transactions are likely to waste more work. Predicting statically when region conflicts might happen is virtually impossible.
2. Intel's HTM implementation has capacity limits: while executing a hardware transaction, caches buffer transactional state. If the read and write sets of the transaction exceed the cache size, the transaction aborts *even if there is no region conflict*. In general, larger transactions have larger read and write sets, so are more likely to abort due to capacity limits. While in principle it might be possible to predict when a transaction can exceed the HTM's capacity limits, in practice it is hard to predict the footprint of a transaction *a priori*.
3. Finally, there are some operations that Intel's HTM cannot accommodate within transactions. We call these operations "HTM-unfriendly" operations. Any DBR that contains an HTM-unfriendly instruction not only cannot be executed in a transaction (and hence must be handled separately), but it clearly cannot be merged with other regions to create a larger transaction. Some of these HTM-unfriendly operations can be identified statically, but other HTM-unfriendly operations, such as hard page faults, are difficult to predict.

Note that each of these issues is, essentially, dynamic. It is hard to tell whether merging regions together into a transaction will trigger an abort. Moreover, the implications of each of these issues for region merging is different. Region conflicts are unpredictable and inherently transient, meaning that an abort due to a region conflict is *nondeterministic*. While executing larger transactions may result in more wasted work, it is often the case that simply re-executing the transaction will result in successful completion (though the transaction may inherently be high conflict, in which case merging fewer regions and executing a smaller transaction may be cost effective, or even necessary to make progress).

On the other hand, if transactions abort due to capacity limits or HTM-unfriendly instructions, re-executing exactly the same transaction will likely still result in an abort. If a transaction aborts due to a capacity constraint, it may be necessary to merge fewer regions into the transaction prior to attempting to re-execute the transaction. Furthermore, if a

[3] In music, "legato" means to play smoothly, with no breaks between notes.

transaction aborts due to an HTM-unfriendly instruction, it might be necessary to execute *just that region* using a non-HTM fallback mechanism (see Section 4) to make progress.

While all of these issues push for merging fewer regions into each transaction, minimizing overhead argues for merging as many regions as possible into a single transaction.

3.2.1 Basic Merging Algorithm

Putting it all together, mitigating the overhead of HTM while accounting for its limits suggests the following dynamic approach to merging regions:

- Begin a transaction prior to executing a region. Execute the region transactionally.
- If execution reaches the end of the region without aborting, execute the *following* region as part of the *same* transaction.
- If a transaction aborts and rolls back, determine the reason for its abort. If the abort was transient, such as due to a page fault or a region conflict, retry the transaction. If the abort was due to capacity limits or HTM-unfriendly instructions, retry the transaction but *end the transaction prior to* the region that caused the problem. Begin a new transaction before executing the next region.

While this basic merging algorithm is attractive, and greedily merges as many regions as possible into each transaction, there are several practical limitations. First, note that a transaction has to abort at least once before it can commit, which will waste a lot of work. Second, when a transaction aborts, Intel's HTM implementation *provides no way to know* ***when or where*** *the transaction aborted*. In other words, we cannot know which DBR actually triggered the abort—there is no way to communicate this information back from an aborted transaction—which means that we do not know when to end the re-executed transaction and begin the next one.

Thus we cannot rely on tracking the abort location to determine when to stop merging regions and commit a transaction. Instead, we need a *target* number of regions to merge together. While executing a transaction, if the target number of regions is met, the transaction commits (even if more regions could have been merged into the transaction), and the next region executes in a new transaction.

Figure 1 shows the instrumentation that Legato uses at region boundaries to implement this target-based merging strategy. T.regionsExec is a per-thread integer that tracks how many more DBRs to merge together in the current transaction. Once the target number of regions have been merged, the transaction commits, and the instrumentation queries a *controller* to determine the next target (line 3). If a transaction aborts, the instrumentation queries the controller for a new target (line 7).

Note that the controller logic executes only at transaction boundaries (lines 2–9). In contrast, each region boundary executes only an inexpensive decrement and check (line 1).

So how should the controller determine the merge target? In general, if transactions are successfully committing, then

```
1  if (−−T.regionsExec == 0) {
2    XEND();
3    T.regionsExec = T.controller.onCommit();
4    _eax = −1; // reset HTM error code register
5    abortHandler:
6     if (_eax != −1) { // reached here via abort?
7       T.regionsExec = T.controller.onAbort();
8     }
9    XBEGIN(abortHandler);
10   // if TX aborts: sets _eax and jumps to abortHandler
11 }
```

Figure 1. Legato's instrumentation at a DBR boundary. T is the currently executing thread. When a transaction has merged the target number of regions (T.regionsExec == 0) or aborts (control jumps to abortHandler), the instrumentation queries the controller (Figure 2) for the number of DBRs to merge in the next transaction.

it is probably safe to be more aggressive in merging regions together. On the other hand, if transactions are frequently aborting, Legato is probably being too aggressive in merging and should merge fewer regions into each transaction. Next we describe Legato's controller design.

3.2.2 A Setpoint Controller

Because different programs, as well as phases during a program's execution, have different characteristics—e.g., different region footprints and different likelihoods of region conflicts—we cannot pick a single merge target throughout execution. Instead, our approach should try to infer this target dynamically. Borrowing from the control theory literature, we call the target number of regions the *setpoint*, and the algorithm that selects the setpoint the *setpoint algorithm*.

The key to the setpoint algorithm is that there are two targets: the setpoint, setPoint, which is the "steady state" target for merging regions together, and the current target, currTarget, which is the target for merging regions together *during the current transaction's execution*. (The controller returns currTarget's value to the instrumentation in Figure 1.)

We distinguish between these two targets for a simple reason. Legato attempts to execute regions together in a transaction until it hits the setpoint, at which point the transaction commits. But what happens if the transaction aborts before hitting the setpoint? There are two possible inferences that could be drawn from this situation:

1. The abort is a temporary setback, e.g., due to a capacity limit exceeded by this combination of regions. It may be necessary to merge together fewer regions to get past the roadblock, but there is no need to become less aggressive in merging regions overall. In this case, it might be useful to temporarily reduce currTarget, but eventually increase currTarget so that currTarget = setPoint.
2. The abort reflects a new phase of execution for the program—for example, moving to a higher-contention phase of the program—where it might be prudent to be less aggressive in merging regions. The right approach then is to lower setPoint, in order to suffer fewer aborts.

```
STEADY
    On Commit:
        1. setPoint $\leftarrow$ setPoint $+ S_{inc}$
        2. currTarget $\leftarrow$ setPoint
    On Abort:
        1. setPoint $\leftarrow$ setPoint $- S_{dec}$
        2. currTarget $\leftarrow \min($currTarget/$C_{decr}$, setPoint$)$
        3. $\Rightarrow$ PESSIMISTIC state
PESSIMISTIC
    On Commit:
        1. $\Rightarrow$ OPTIMISTIC state
    On Abort:
        1. currTarget $\leftarrow$ currTarget / $C_{decr}$
OPTIMISTIC
    On Commit:
        1. currTarget $\leftarrow \min($currTarget $\times C_{incr}$, setPoint$)$
        2. if currTarget = setPoint then $\Rightarrow$ STEADY state
    On Abort:
        1. currTarget $\leftarrow$ currTarget / $C_{decr}$
        2. $\Rightarrow$ PESSIMISTIC state
```

Figure 2. A state machine describing transitions in the setpoint algorithm when onCommit and onAbort are called by the instrumentation. In all cases, the state machine returns currTarget.

These considerations lead to the design of Legato's control algorithm, which is expressed as a finite state machine with three states, described in Figure 2. Each thread uses its own instance of the state machine. In the STEADY state, whenever a transaction commits, the controller views this as evidence that merging can be more aggressive, so it increases setPoint and sets currTarget to setPoint. When a transaction aborts, the controller decreases both setPoint and currTarget. An abort moves the controller into PESSIMISTIC state, which continues to decrease currTarget if more aborts occur. If a transaction *commits* while the controller is in PESSIMISTIC state, the controller does not assume that it is safe to become more aggressive immediately. Instead, the controller keeps the current target, and it moves to OPTIMISTIC state, where it can be aggressive again.

Note that currTarget increases and decreases *geometrically*, while setPoint increases and decreases *arithmetically*. In other words, the current target fluctuates quickly to capture transient effects such as capacity limits, while the setpoint, which represents the steady-state merge target, fluctuates more slowly, to capture slower effects such as changing phases of the program.

Our experiments use a value of 2 for C_{decr}. The idea is that, given the lack of knowledge about the abort location, it is equally probable that a transaction aborted during its first half as its second half. Our experiments use fixed values for S_{inc} and S_{dec} chosen from a sensitivity study (Section 5.3).

3.3 Designing a Fallback

Legato requires a fallback mechanism when it encounters a single DBR that cannot be executed transactionally (e.g., if the region contains an HTM-unfriendly instruction). The fallback mechanism must allow the region to execute in a non-speculative manner *while still maintaining atomicity* with respect to other, speculatively executing regions. Because we find that the fallback is needed infrequently, Legato uses a simple, global spin lock to provide atomicity in these situations. If a single-DBR transaction (i.e., currTarget = 1) aborts, Legato's instrumentation acquires a global lock for the duration of the DBR, ensuring atomicity with respect to other, non-speculative regions. To ensure atomicity with respect to other, *speculatively executing* regions, each speculative region must check whether the global lock is held, and abort if so. Existing speculative lock elision approaches use similar logic to elide critical sections [26, 53, 57, 71, 73].

We modify the instrumentation from Figure 1 so that, immediately prior to committing a transaction, Legato checks whether the global lock is held. If it is, Legato conservatively aborts the transaction.

3.4 Discussion: Extending Legato to Elide Locks

Although individual DBRs are bounded by synchronization operations such as lock acquire and release operations, a multi-DBR transaction may include lock acquires and releases. Thus, it is entirely possible for an executed transaction to include both the start and end of a critical section. Therefore, with modest extensions to its design, Legato can naturally execute the critical section *without* acquiring the lock, in a generalization of lock elision [26, 53, 57, 71, 73]. Legato's approach can even execute several critical sections, including nested critical sections, inside a single transaction. Although extending the design is minor, changing the implementation's handling of program locks would involve major changes to the locking and threading subsystems, so we leave this exploration for future work.

4. Implementation

We have implemented Legato by modifying Jikes RVM 3.1.3 [5, 6], a Java virtual machine that is comparable in performance to efficiently tuned commercial JVMs [11]. We have made our Legato implementation publicly available on the Jikes RVM Research Archive.

We extend Jikes RVM to generate TSX operations by borrowing from publicly available patches by Ritson et al. [56].

Although Legato's approach is distinctly different from EnfoRSer's [58], we have built Legato in the same JVM instance as our publicly available implementation of EnfoRSer, to minimize any irrelevant empirical differences between the two implementations.

Compilation and instrumentation. Jikes RVM uses two just-in-time compilers at run time. The first time a method executes, Jikes RVM compiles it with the *baseline* compiler, which generates unoptimized machine code directly from Java bytecode. When a method becomes hot, Jikes RVM compiles it with the *optimizing* compiler at successively higher optimization levels. We modify both compilers to insert Legato's instrumentation. The prior software-only work, EnfoRSer, modifies only the optimizing compiler, due to the complexities of transforming programs to provide the memory model; it modifies baseline-compiled code to simulate the performance of enforcing DBRS [58].

Jikes RVM's dynamic compilers automatically insert *yield points*—points in the code where a thread can yield, e.g., for stop-the-world garbage collection or for online profiling [10]—into compiled code. Yield points are typically at each method's entry and exit and at loop back edges. Thus they demarcate *statically and dynamically* bounded regions, which EnfoRSer uses to enforce statically bounded region serializability (SBRS). Unlike EnfoRSer, Legato does not require *statically* bounded regions (Section 2.2). Nevertheless, Jikes RVM's yield points provide convenient region boundaries, so Legato adopts them. Legato instruments all yield points and synchronization operations as DBR boundaries. For efficiency and to avoid executing the yield point handler in a transaction, Legato inserts its region boundary instrumentation so that the yield point executes in the instrumentation slow path (see Figure 1). This instrumentation performs the yield point logic only after ending a transaction and before starting the next transaction.

Handling HTM-unfriendly events and instructions. For instructions and events that we can identify at compile time as HTM unfriendly (Section 3.2), which are often in internal JVM code, the implementation forcibly ends the current transaction, and starts another transaction after the instruction. For unexpected HTM-unfriendly instructions and events, Legato already handles them by falling back to a global lock after a single-DBR transaction aborts.

Our implementation executes application code, including Java library code called by the application, in transactions. Since compiled application code frequently calls into the VM, transactions may include VM code. If the VM code is short, it makes sense to execute it transactionally. However, the VM code may be lengthy or contain HTM-unfriendly instructions. We thus instrument VM code's region boundaries (yield points) like application boundaries—except that VM code does not start a new transaction after committing the current transaction. Since some VM code is "uninterruptible" and executes no yield points, Legato unconditionally ends the current transaction at a few points we have identified that commonly abort (e.g., the object allocation "slow path"). Whenever a thread ends a transaction in VM code, it starts a new transaction upon re-entering application code.

5. Evaluation

This section evaluates the performance and run-time characteristics of Legato, compared with alternative approaches.

5.1 Setup and Methodology

EnfoRSer implementation. EnfoRSer provides multiple compiler transformations for ensuring atomicity; our experiments use EnfoRSer's *speculation transformation* because it performs best [58].

EnfoRSer can use the results of whole-program static race detection analysis [58, 59]. However, to avoid relying on the "closed-world hypothesis" (Section 6) and to make EnfoRSer more directly comparable with Legato (which does not use whole-program static analysis), our experiments do *not* use any whole-program static analysis.

	Threads		Dynamic events		
	Total	Live	Accesses	DBRs	Acc. / DBR
eclipse6	18	12	1.6×10^{10}	5.0×10^{9}	3.1
hsqldb6	402	102	6.8×10^{8}	4.4×10^{8}	1.5
lusearch6	65	65	3.1×10^{9}	9.7×10^{8}	3.2
xalan6	9	9	1.3×10^{10}	5.2×10^{9}	2.5
avrora9	27	27	7.9×10^{9}	1.4×10^{9}	5.6
jython9	3	3	6.6×10^{9}	4.7×10^{9}	1.4
luindex9	2	2	3.9×10^{8}	1.5×10^{8}	2.6
lusearch9	c	c	3.0×10^{9}	8.9×10^{8}	3.4
pmd9	5	5	6.4×10^{8}	3.7×10^{8}	1.7
sunflow9	$2\times c$	c	2.3×10^{10}	3.4×10^{9}	6.9
xalan9	c	c	1.2×10^{10}	4.7×10^{9}	2.6
pjbb2000	37	9	2.6×10^{9}	1.2×10^{9}	1.7
pjbb2005	9	9	9.1×10^{9}	3.6×10^{9}	2.5

Table 1. Dynamic execution characteristics. Three programs spawn threads proportional to the number of cores c, which is 14 in our experiments. The last three columns report memory accesses executed, dynamically bounded regions executed, and their ratio.

Environment. The experiments run on a single-socket Intel Xeon E5-2683 system with 14 cores and one hardware thread per core (we disable hyperthreading), running Linux 3.10.0. Although Intel has disabled TSX in this processor because of a known vulnerability, the vulnerability does not affect non-malicious code, so we explicitly enable TSX.

Benchmarks. In our experiments, modified Jikes RVM executes (1) the large workload size of the multithreaded DaCapo benchmarks [12] versions 2006-10-MR2 and 9.12-bach (2009), distinguished by suffixes 6 and 9, using only programs that Jikes RVM can run, and (2) fixed-workload versions of SPECjbb2000 and SPECjbb2005.[4]

Execution methodology. For each configuration of EnfoRSer and Legato, we build a high-performance Jikes RVM executable, which adaptively optimizes the application as it runs and uses the default high-performance garbage collector, which adjusts the heap size automatically.

Each performance result is the mean of 15 runs, with 95% confidence intervals shown. Each reported statistic is the mean of 8 statistics-gathering runs.

5.2 Run-Time Characteristics

The *Threads* columns in Table 1 report the total number of threads spawned and maximum number of live threads for each evaluated program. The *Dynamic events* columns report the number of executed memory accesses, the number of executed dynamically bounded regions (i.e., the number of region boundaries executed), and the average number of memory accesses executed per DBR. Each program executes between hundreds of millions and tens of billions of memory accesses. The average DBR size for each program varies from 1.4 to 6.9 executed memory accesses.

[4] `http://www.spec.org/jbb200{0,5}`, `http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005`

	Staccato		Legato							
				DBRs /	Acc. /	Abort rate				
	Trans.	Abort rate	Trans.	trans.	trans.	Total	(conf	cap	fallb	other)
eclipse6	5.0×10^{9}	$<$0.1%	4.4×10^{7}	110	360	14.4%	(2.1%	3.3%	$<$0.1%	9.0%)
hsqldb6	4.2×10^{8}	$<$0.1%	8.8×10^{6}	48	77	13.8%	(1.7%	2.3%	$<$0.1%	9.8%)
lusearch6	8.9×10^{8}	0.8%	3.5×10^{7}	25	88	31.0%	(24.9%	0.6%	0.2%	5.4%)
xalan6	5.1×10^{9}	0.1%	2.3×10^{8}	22	57	10.5%	(7.2%	0.6%	$<$0.1%	2.7%)
avrora9	1.4×10^{9}	3.2%	1.1×10^{8}	13	71	41.7%	(13.6%	0.4%	3.7%	24.0%)
jython9	4.5×10^{9}	$<$0.1%	3.5×10^{7}	130	190	8.0%	(0.8%	5.1%	$<$0.1%	2.1%)
luindex9	1.5×10^{8}	$<$0.1%	1.5×10^{6}	100	260	16.7%	(0.9%	4.9%	$<$0.1%	10.9%)
lusearch9	8.2×10^{8}	0.2%	2.8×10^{7}	29	110	22.7%	(15.9%	0.6%	$<$0.1%	6.1%)
pmd9	3.5×10^{8}	1.2%	1.5×10^{7}	23	43	41.2%	(35.5%	1.5%	0.2%	4.0%)
sunflow9	3.3×10^{9}	0.3%	5.3×10^{7}	62	440	20.8%	(17.3%	2.5%	$<$0.1%	1.0%)
xalan9	4.5×10^{9}	0.2%	2.2×10^{8}	20	55	14.8%	(8.5%	0.7%	$<$0.1%	5.7%)
pjbb2000	1.2×10^{9}	$<$0.1%	3.9×10^{7}	31	53	7.2%	(3.4%	0.53%	$<$0.1%	3.2%)
pjbb2005	3.6×10^{9}	0.4%	5.0×10^{8}	7.2	18	28.1%	(14.8%	0.1%	$<$0.1%	13.2%)

Table 2. Transaction commits and aborts for Staccato and Legato, and average DBRs and memory accesses per transaction for Legato.

Table 2 reports committed and aborted transactions for the default configuration of Legato that uses the setpoint-based merging algorithm (right half), compared with a configuration of Legato that does no merging, which we call *Staccato*[5] (left half). The two *Trans.* columns show the number of committed transactions for Staccato and Legato, respectively. The *DBRs / trans.* column shows the number of DBRs executed per committed transaction for Legato. Note that Staccato executes one DBR per transaction.[6] As the table shows, on average Legato reduces the executed transactions by 1–2 orders of magnitude compared with Staccato. The *Acc. / trans.* column shows that a typical transaction in Legato executes dozens or hundreds of memory accesses.

While the reduction in committed transactions represents the *benefit* provided by Legato's merging of DBRs into transactions, Legato's *Abort rate* columns represent the *cost* of merging. The *Total* column shows the total number of aborts, as a percentage of total committed transactions (the *Trans.* column). Most programs have an abort rate between 10% and 30%, which is substantial. In contrast, the abort rate for Staccato is significantly lower: typically less than 1% and at most 3.2%. The values in parentheses show the breakdown of aborts into four categories (again as a percentage of committed transactions): conflict and capacity aborts; explicit aborts when the global fallback lock is held (Section 3.3); and other reasons (Section 2.4). For most programs, conflicts are the primary cause of aborts; conflicts are difficult to predict since they involve cross-thread interactions. In general, it is worthwhile for Legato to "risk" conflicts in order to merge transactions and avoid per-transaction costs. The second-most common cause of aborts is "other"; although RTM's abort error code provides no information about these aborts, by using the Linux perf tool we find that most of these aborts are due to system calls inside the VM, existing exceptions in the program, and system interrupts.

[5] In music, "staccato" means to play each note disconnected from the others.

[6] Although one might expect the number of transactions executed by Staccato to be exactly equal to the number of regions per benchmark from Table 1, they differ because (i) the measurements are taken from different runs, and (ii) in Staccato, not every region executes in a transaction, due to the fallback mechanism.

5.3 Performance

Figure 3 compares the performance of three approaches that provide DBRS: EnfoRSer [58]; Staccato, which executes each DBR in its own transaction; and the default Legato configuration. Each result is the run-time overhead over baseline execution (unmodified Jikes RVM). Since EnfoRSer adds dramatically different overhead depending on the amount of cross-thread communication between memory accesses (Section 2.2), the figure divides the programs into whether they have low or high rates of communicating memory accesses, leading to low or high EnfoRSer overhead, respectively. We categorize programs as "high communication" if $\geq$0.1% of all memory accesses trigger biased reader–writer locks' "explicit communication" protocol [17, 58].

For the four high-communication programs, EnfoRSer adds nearly 60% overhead or more; for pjbb2005, EnfoRSer adds over 120% overhead. Unsurprisingly, pjbb2005 has the highest percentage of accesses involved in cross-thread communication, 0.5% [17].

In contrast, the run-time overhead of Staccato is not correlated with cross-thread communication, but is instead dominated by the costs of starting and ending transactions. On average, Staccato adds 175% run-time overhead due to the high cost of frequent transaction boundaries. This result shows that applying commodity HTM naïvely to provide DBRS is a nonstarter, motivating Legato's contributions.

Legato improves over both Staccato and EnfoRSer by amortizing per-transaction costs and by avoiding EnfoRSer's highly variable costs tied to communicating memory accesses. By merging transactions, Legato reduces average run-time overhead to 33%, compared with 175% without merging, despite incurring a significantly higher abort rate than Staccato (Table 2). Legato's overhead is relatively stable across programs, and is at most 50% for any program. On average, Legato adds 32% and 35%, respectively for the low- and high-communication programs. Notably, Legato's

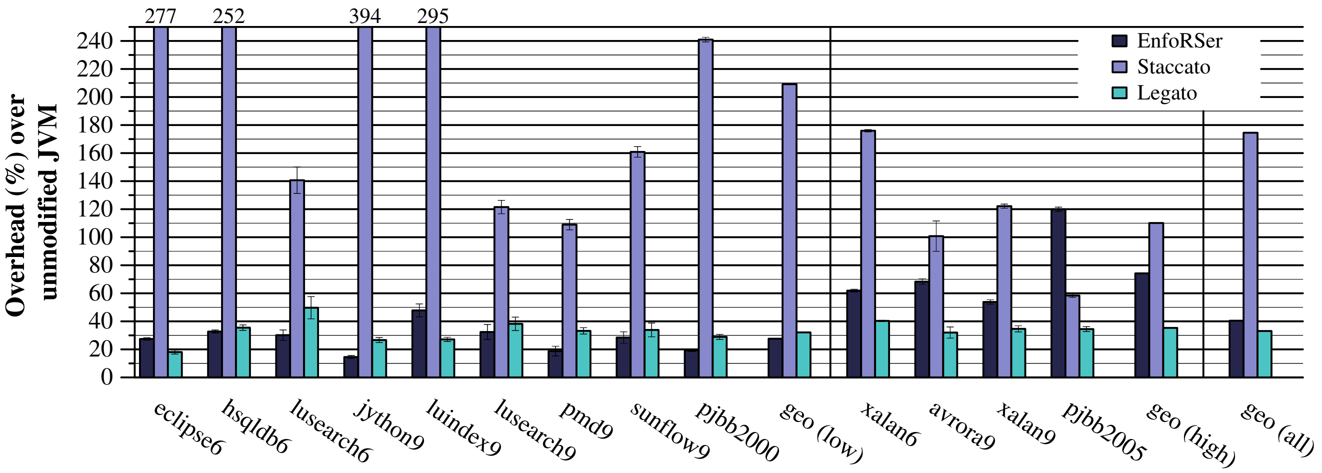

Figure 3. Run-time overhead of providing DBRS with three approaches: software-based EnfoRSer; Staccato, which executes each DBR in a transaction; and the default Legato configuration that merges multiple DBRs into transactions. The programs are separated into low- and high-communication behavior, with component and overall geomeans reported.

average overhead for high-communication programs (35%) is significantly lower than EnfoRSer's average overhead for these programs (74%). Not only does Legato significantly outperform EnfoRSer on average (33% versus 40%, respectively), but Legato's overhead is more stable and less sensitive to shared-memory interactions.

Legato's performance breakdown. We used partial configurations to find more insights into the 33% run-time overhead of Legato. Instrumentation at region boundaries adds an overhead of 12%. The remaining overhead of 21% comes from beginning and ending transactions, aborting and running inside transactions. We have found it diffcult to concretely separate out these costs.

Sensitivity study. As presented in Section 3.2.2, Legato maintains a per-thread setpoint, setPoint, that it adjusts in response to transaction commits and aborts. The setpoint algorithm's state machine (Figure 2) uses S_{inc} and S_{dec} to adjust setPoint. By default, Legato uses the parameters $S_{inc} = 1$ and $S_{dec} = 10$; our rationale for these choices is that the marginal cost of an aborted transaction is significantly more than the cost of executing shorter transactions.

Here we evaluate sensitivity to the parameters S_{inc} and S_{dec}. Figure 4 shows the run-time overhead of Legato with each combination of 1 and 10 for these values. The first configuration, *Legato S_inc = 1, S_dec = 10*, corresponds to the default Legato configuration used in the rest of the evaluation. The second configuration, *Legato S_inc = 10, S_dec = 1*, shows that performance is sensitive to large changes in these parameters; its high overhead validates the intuition behind keeping the S_{inc}/S_{dec} ratio small. The last two configurations both set $S_{inc} = S_{dec}$; their overheads fall between the overheads of the first two configurations.

The *magnitudes* of S_{inc} and S_{dec} represent how quickly the algorithm adjusts to phased behavior. We find that for other values of S_{inc} and S_{dec} within 2X of the default values (results not shown), the magnitudes do not significantly affect average performance. Instead, the results suggest that the S_{inc}/S_{dec} *ratio* is most important, and a ratio of about 0.1 provides the best performance on average.

Dynamic vs. fixed setpoints. Legato uses a dynamic setpoint algorithm that adjusts the setpoint based on recent execution behavior. An alternative approach is to use a fixed setpoint throughout the entire execution, which cannot handle differences between programs or a program's phases. Appendix A evaluates this alternative, finding that the best fixed setpoint differs from program to program. While the best fixed setpoint for each application provides similar performance to Legato's dynamic setpoint algorithm, Legato does not require per-application tuning.

Compilation time. For both EnfoRSer and Legato, the dynamic, just-in-time compilers in Jikes RVM insert instrumentation into the compiled code. EnfoRSer performs complex compiler analyses and transformations that generate significantly more code than the original program; unlike Legato, EnfoRSer analyzes, transforms, and instruments memory accesses, branches, arithmetic instructions, and other instructions. In contrast, Legato limits its instrumentation to region boundaries. In a just-in-time compilation setting, the additional compilation time translates into extra execution time for several reasons: (1) the extra compilation time itself (although Jikes RVM uses a separate optimizing compiler thread); (2) the effect on downstream optimizations and transformations that must handle more code; and (3) the opportunity cost of not being able to optimize code as quickly, and thus executing less-optimized code overall.

Figure 5 shows the compilation time added by EnfoRSer and Legato for the same executions used for Figure 3. We emphasize that since compilation occurs at run time, these costs are already reflected in overall execution time; in fact, Legato's compilation overhead advantage is somewhat muted by the fact that Jikes RVM's adaptive optimization system [9] naturally performs less compilation as compila-

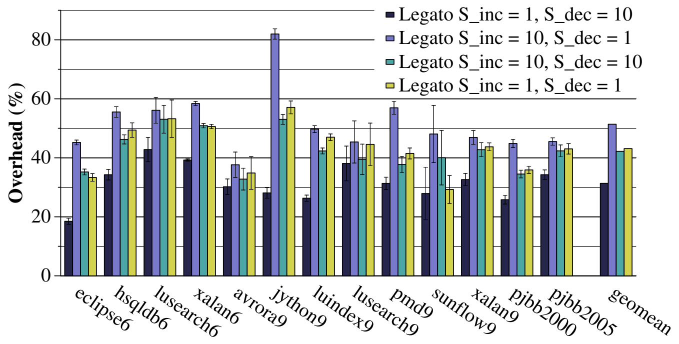

Figure 4. Sensitivity of run-time performance to the parameters to Legato's dynamic setpoint algorithm.

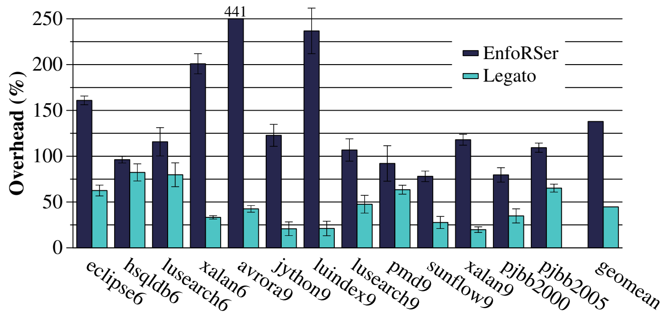

Figure 5. Just-in-time compilation overhead of providing DBRS with EnfoRSer versus Legato.

tion becomes more expensive (i.e., Legato optimizes more methods than EnfoRSer). The figure shows that on average Legato adds one-third as much compilation overhead over the baseline (unmodified Jikes RVM) as EnfoRSer: 138% versus 45%. These results show that the best-performing existing approach that targets commodity systems, relies on complex, costly compiler analyses and transformations. In contrast, Legato uses only relatively simple instrumentation at region boundaries (cf. Figure 1).

Space overhead. Appendix B reports space overhead of EnfoRSer and Legato over an unmodified JVM. EnfoRSer and Legato incur average space overhead of 29% and 2%, respectively, mainly because EnfoRSer adds per-object metadata while Legato does not.

6. Related Work

This section considers prior work *other than* work on memory models and commodity HTM, which Section 2 covered.

Using whole-program static analysis. Whole-program static analysis can identify potential data races between static program memory accesses [48, 49]. However, sound (no missed races) static analysis suffers from two main problems. First, its precision scales poorly with program size and complexity. EnfoRSer uses the results of sound static analysis to avoid instrumenting definitely data-race-free accesses, reducing average overhead from 36% to 27% [58]. In follow-up work, we extend EnfoRSer to provide atomicity of low-contention regions with coarse-grained pessimistic locks, using static pairs of potentially racy accesses to identify regions that should use the same lock [59]. Second, whole-program static analysis relies on the so-called "closed-world hypothesis"—that all of the code is available at compile time—which fails to hold for dynamically loaded languages such as Java. In contrast, Legato does not rely on whole-program static analysis or the closed-world hypothesis.

Another serializability-based model. O'Neill and Stone use TSX to provide speculation-based atomicity for *observationally cooperative multithreading* (OCM) [50]. In OCM, all program code is in atomic regions, but the regions are not dynamically bounded. The OCM implementation uses TSX's HLE implementation, which automatically falls back to a global lock for transactions that abort; the main performance challenge is serialized execution resulting from regions that abort when executed as TSX transactions. In contrast, Legato's regions generally fit in a TSX transaction; the main challenge is merging regions into transactions to balance competing costs.

Dynamic atomic regions. Prior work proposes customized hardware to execute code in atomic regions, enabling aggressive speculative optimizations that can be rolled back if necessary [25, 44]. Mars and Kumar address one challenge in such settings: data conflicts can cause atomic optimization regions to roll back excessively [44]. They introduce a framework to mitigate the effect of shared-memory conflicts that lead to misspeculations or "squashes." Their technique, called *BlockChop*, uses a composition of retrials, delay, translation to finer-grained regions, and fallback to conflict-free interpretation mode to significantly reduce the number of misspeculations, exploiting the right tradeoff between the costs of these mechanisms. While BlockChop shows the cost of conflicting accesses to be a significant contributor to overheads, Legato shows that in current commercial hardware the cost of start and end of transactions is prohibitive. Essentially, because of their different objectives and constraints, BlockChop favors reducing region sizes to reduce conflicts while Legato favors increasing region sizes to reduce overhead.

7. Conclusion

Legato provides a novel direction to provide bounded region serializability at reasonable overheads on commodity systems, overcoming the limitations and complexity of prior work. Unlike prior approaches, Legato maintains consistent overhead even in the face of applications with many communicating threads, making it more suitable for applications with diverse communication patterns, and giving it more predictable performance overall. Legato hence advances the state of the art in efficiently enforcing stronger memory models in commodity systems.

Acknowledgments

We thank James Cownie, Roman Dementiev, Melissa O'Neill, and Carl Ritson for help with using TSX and integrating it into Jikes RVM. We thank the anonymous reviewers for helpful suggestions on the text.

References

[1] S. V. Adve and H.-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *CACM*, 53:90–101, 2010.

[2] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29:66–76, 1996.

[3] S. V. Adve and M. D. Hill. Weak Ordering—A New Definition. In *ISCA*, pages 2–14, 1990.

[4] W. Ahn, S. Qi, M. Nicolaides, J. Torrellas, J.-W. Lee, X. Fang, S. Midkiff, and D. Wong. BulkCompiler: High-performance Sequential Consistency through Cooperative Compiler and Hardware Support. In *MICRO*, pages 133–144, 2009.

[5] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[6] 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.

[7] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *HPCA*, pages 316–327, 2005.

[8] T. A. Anderson, M. O'Neill, and J. Sarracino. Chihuahua: A Concurrent, Moving, Garbage Collector using Transactional Memory. In *TRANSACT*, 2015.

[9] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *OOPSLA*, pages 47–65, 2000.

[10] M. Arnold and D. Grove. Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines. In *CGO*, pages 51–62, 2005.

[11] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia. Valor: Efficient, Software-Only Region Conflict Exceptions. In *OOPSLA*, pages 241–259, 2015.

[12] 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.

[13] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory. In *ISCA*, pages 24–34, New York, NY, USA, 2007.

[14] J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood. TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. In *ISCA*, pages 127–138, 2008.

[15] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, pages 68–78, 2008.

[16] H.-J. Boehm and B. Demsky. Outlawing Ghosts: Avoiding Out-of-Thin-Air Results. In *MSPC*, pages 7:1–7:6, 2014.

[17] 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.

[18] J. Burnim, K. Sen, and C. Stergiou. Testing Concurrent Programs on Relaxed Memory Models. In *ISSTA*, pages 122–132, 2011.

[19] M. Cao, J. Roemer, A. Sengupta, and M. D. Bond. Prescient Memory: Exposing Weak Memory Model Behavior by Looking into the Future. In *ISMM*, pages 99–110, 2016.

[20] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *ISCA*, pages 278–289, 2007.

[21] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A Scalable, Non-blocking Approach to Transactional Memory. In *HPCA*, pages 97–108, 2007.

[22] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière. Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack. In *EuroSys*, pages 27–40, 2010.

[23] C. Click. Azul's Experiences with Hardware Transactional Memory. In HP Labs – Bay Area Workshop on Transactional Memory, Jan. 2009.

[24] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *PPoPP*, pages 67–78, 2010.

[25] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing^{TM} Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. In *CGO*, pages 15–24, 2003.

[26] D. Dice, A. Kogan, Y. Lev, T. Merrifield, and M. Moir. Adaptive Integration of Hardware and Software Lock Elision Techniques. In *SPAA*, pages 188–197, 2014.

[27] K. Gharachorloo and P. B. Gibbons. Detecting Violations of Sequential Consistency. In *SPAA*, pages 316–326, 1991.

[28] C. Gniady and B. Falsafi. Speculative Sequential Consistency with Little Custom Storage. In *PACT*, pages 179–188, Washington, DC, USA, 2002.

[29] 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. In *ISCA*, pages 102–113, 2004.

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

[31] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, pages 289–300, 1993.

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

[33] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Computer*, 28:690–691, 1979.

[34] C. Lin, V. Nagarajan, and R. Gupta. Efficient Sequential Consistency Using Conditional Fences. In *PACT*, pages 295–306, 2010.

[35] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram. Efficient Sequential Consistency via Conflict Ordering. In *ASPLOS*, pages 273–286, 2012.

[36] Y. Liu, J. Gottschlich, G. Pokam, and M. Spear. TSXProf: Profiling Hardware Transactions. In *PACT*, pages 75–86, 2015.

[37] Y. Liu, Y. Xia, H. Guan, B. Zang, and H. Chen. Concurrent and Consistent Virtual Machine Introspection with Hardware Transactional Memory. In *HPCA*, pages 416–427, Feb 2014.

[38] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *ISCA*, pages 210–221, 2010.

[39] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *ISCA*, pages 277–288, 2008.

[40] S. Mador-Haim, R. Alur, and M. M. Martin. Specifying Relaxed Memory Models for State Exploration Tools. In *EC*^{2}, 2009.

[41] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *POPL*, pages 378–391, 2005.

[42] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. DRFx: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *PLDI*, pages 351–362, 2010.

[43] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. A Case for an SC-Preserving Compiler. In *PLDI*, pages 199–210, 2011.

[44] J. Mars and N. Kumar. BlockChop: Dynamic Squash Elimination for Hybrid Processor Architecture. In *ISCA*, pages 536–547, 2012.

[45] H. S. Matar, I. Kuru, S. Tasiran, and R. Dementiev. Accelerating Precise Race Detection Using Commercially-Available Hardware Transactional Memory Support. In *WoDet*, 2014.

[46] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *HPCA*, pages 254–265, 2006.

[47] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *PLDI*, pages 446–455, 2007.

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

[49] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. In *PLDI*, pages 308–319, 2006.

[50] M. E. O'Neill and C. A. Stone. Making Impractical Implementations Practical: Observationally Cooperative Multithreading Using HLE. In *TRANSACT*, 2015.

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

[52] S. H. Pugsley, M. Awasthi, N. Madan, N. Muralimanohar, and R. Balasubramonian. Scalable and Reliable Communication for Hardware Transactional Memory. In *PACT*, pages 144–154, 2008.

[53] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *MICRO*, pages 294–305, 2001.

[54] P. Ranganathan, V. Pai, and S. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *SPAA*, pages 199–210, 1997.

[55] C. G. Ritson and F. R. Barnes. An Evaluation of Intel's Restricted Transactional Memory for CPAs. In *CPA*, pages 271–292, 2013.

[56] C. G. Ritson, T. Ugawa, and R. E. Jones. Exploring Garbage Collection with Haswell Hardware Transactional Memory. In *ISMM*, pages 105–115, 2014.

[57] A. Roy, S. Hand, and T. Harris. A Runtime System for Software Lock Elision. In *EuroSys*, pages 261–274, 2009.

[58] A. Sengupta, S. Biswas, M. Zhang, M. D. Bond, and M. Kulkarni. Hybrid Static–Dynamic Analysis for Statically Bounded Region Serializability. In *ASPLOS*, pages 561–575, 2015.

[59] A. Sengupta, M. Cao, M. D. Bond, and M. Kulkarni. Toward Efficient Strong Memory Model Support for the Java Platform via Hybrid Synchronization. In *PPPJ*, pages 65–75, 2015.

[60] J. Ševčík and D. Aspinall. On Validity of Program Transformations in the Java Memory Model. In *ECOOP*, pages 27–51, 2008.

[61] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *TOPLAS*, 10(2):282–312, 1988.

[62] 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.

[63] A. Singh, D. Marino, S. Narayanasamy, T. Millstein, and M. Musuvathi. Efficient Processor Support for DRFx, a Memory Model with Exceptions. In *ASPLOS*, pages 53–66, 2011.

[64] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi. End-to-End Sequential Consistency. In *ISCA*, pages 524–535, 2012.

[65] C. SPARC International, Inc. *The SPARC Architecture Manual: Version 8*. 1992.

[66] C. SPARC International, Inc. *The SPARC Architecture Manual: Version 9*. 1994.

[67] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. Padua. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *PPoPP*, pages 2–13, 2005.

[68] 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.

[69] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *PACT*, pages 127–136, 2012.

[70] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling

Hardware Transactional Memory from Caches. In *HPCA*, pages 261–272, 2007.

[71] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance Computing. In *SC*, pages 19:1–19:11, 2013.

[72] T. Zhang, D. Lee, and C. Jung. TxRace: Efficient Data Race Detection Using Commodity Hardware Transactional Memory. In *ASPLOS*, pages 159–173, 2016.

[73] L. Ziarek, A. Welc, A.-R. Adl-Tabatabai, V. Menon, T. Shpeisman, and S. Jagannathan. A Uniform Transactional Execution Environment for Java. In *ECOOP*, pages 129–154, 2008.

A. Fixed Versus Dynamic Setpoints

Figure 7 evaluates the performance of Legato's dynamic setpoint algorithm, compared with an alternative strategy of using a fixed setpoint for the entire execution. *Legato default (dynamic setpoint)* is the default Legato configuration evalauted in Section 5. The figure shows a horizontal line indicating the overhead of default Legato. *Legato w/fixed setpoint* uses a fixed setpoint (i.e., a constant value of setPoint for the entire execution). The figure plots the run-time overhead of *Legato w/fixed setpoint* for various fixed setpoint values, from 2 to 4096. Each point is the mean of 10 trials, with the interval showing 95% confidence intervals.

Most programs have a fixed setpoint at which they incur the lowest run-time overhead. However, the best-performing setpoint is not constant across benchmarks. Legato's default algorithm closely replicates the peformance of the best fixed setpoint on certain instances and fails on other occasions. The absence of a best-perfoming fixed setpoint across benchmarks proves that rigorous per-application profiling is required to use *Legato w/fixed setpoint* effectively. In contrast, the default Legato algorithm can avoid per-application profiling and yet provide performance close to the best-known fixed setpoint. On average, across benchmarks, the default algorithm improves performance compared to any single fixed setpoint (Figure 7(n)).

B. Space Overhead

This section provides a comparison of space overhead added by EnfoRSer and Legato. We define the memory usage of an execution as the maximum heap memory in use after any full-heap garbage collection (GC). The experiments use the default, high-performance, generational GC in Jikes RVM and allow GC to adjust the heap size automatically (Section 5.3).

Figure 6 shows that the memory usage overhead incurred (over an unmodified JVM) by EnfoRSer (29%) is significantly higher than by Legato (2%). We believe EnfoRSer's space overhead is higher mainly because EnfoRSer adds a word to each object's header to act as a lightweight biased reader–writer lock [17, 58]. In contrast, Legato relies on the intrinsic conflict detection capabilities of commodity HTM. Another potential factor is that EnfoRSer's compiler passes use heavyweight analysis and transformations at run time, generating many objects and triggering more frequent GCs, although we expect most of these objects to be short-lived and not survive full-heap GCs. Legato provides repeatedly lower space overhead than the *baseline* for pjbb2000, presumably because Legato's allocation behavior leads to triggering GC at points when the working set size is relatively smaller than the points when GC occurs in the baseline.

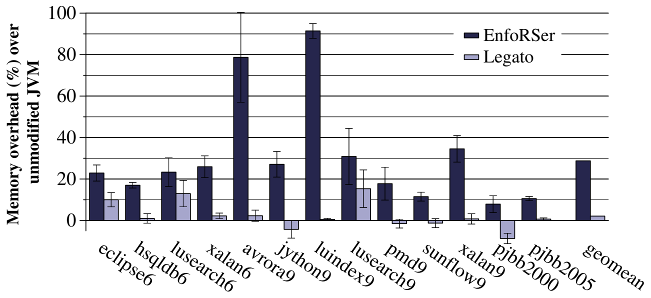

Figure 6. Space overhead of providing DBRS with EnfoRSer versus Legato.

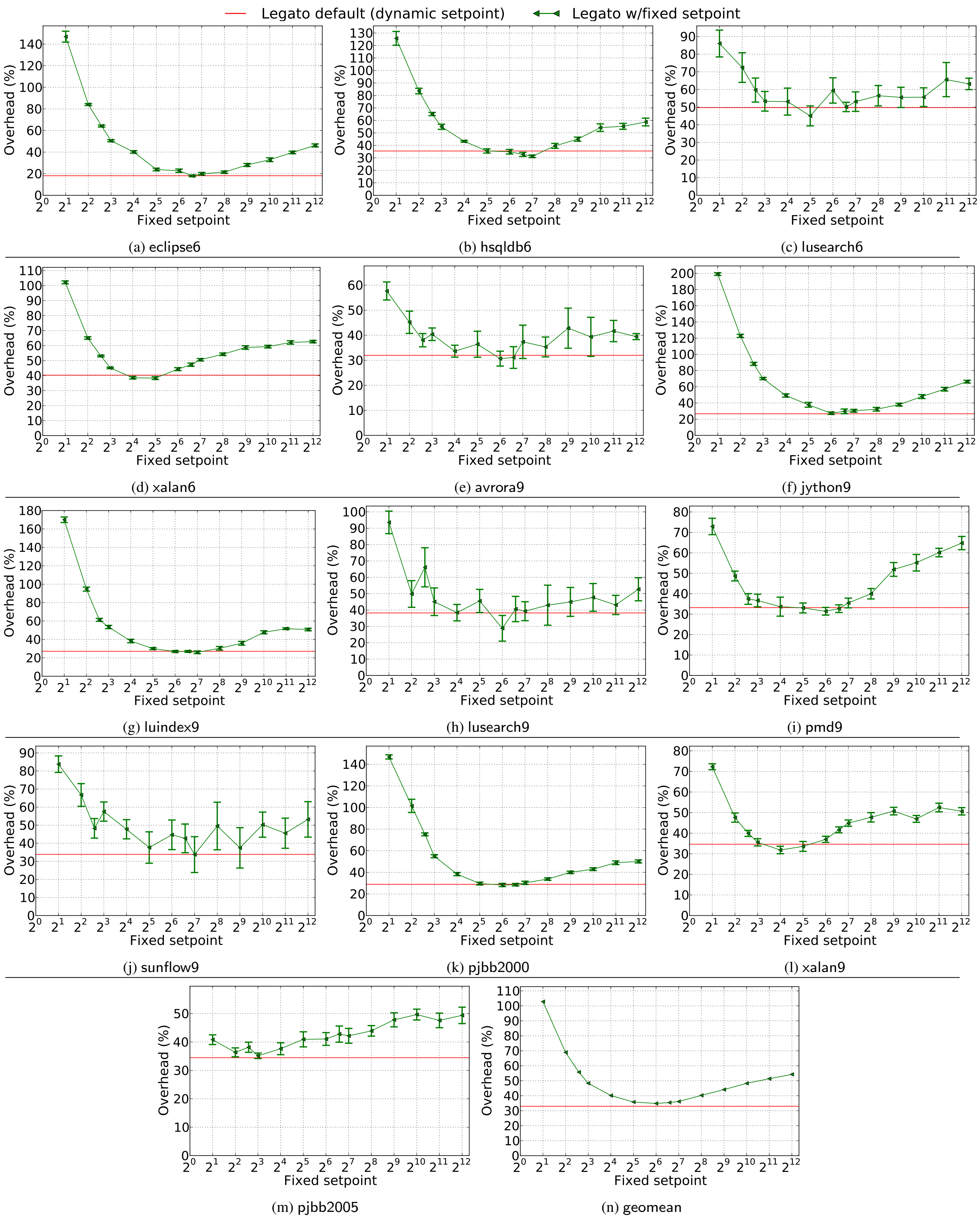

Figure 7. Performance of Legato's dynamic setpoint algorithm compared with Legato using various fixed setpoints, for each benchmark and across benchmarks. X-axis plots fixed set points.