A Case for Deconstructing Hardware Transactional Memory Systems

Mark D. Hill, Derek Hower, Kevin E. Moore,
Michael M. Swift, Haris Volos, and David A. Wood

Department of Computer Sciences, University of Wisconsin–Madison
{markhill, drh5, kmoore, swift, hvolos, david}@cs.wisc.edu
http://www.cs.wisc.edu/multifacet

Abstract

Major hardware and software vendors are curious about transactional memory (TM), but are understandably cautious about committing to hardware changes.

Our thesis is that deconstructing transactional memory into separate, interchangeable components facilitates TM adoption in two ways. First, it aids hardware TM refinement, allowing vendors to adopt TM earlier, knowing that they can more easily refine aspects later. Second, it enables the components to be applied to other uses, including reliability, security, performance, and correctness, providing value even if TM is not widely used. We develop some evidence for our thesis via experience with LogTM variants and preliminary case studies of scalable watchpoints and race recording for deterministic replay.

1 Introduction

Transactional memory (TM) [17,22] has been proposed to ease concurrent programming. TM systems may be implemented completely in software *(Software TMs* or *STMs*) [15,16,48], as software-hardware *HybridTMs* [10,20,44,45], or largely in hardware *(Hardware TMs* or *HTMs*). Example HTMs include *HMTM* [17], *TCC* [14,26,27], *UTM/LTM* [2], *VTM* [44], *LogTM* [29,30,58], and *Bulk* [8]. HTMs mitigate most STM overheads, because they (i) augment hardware with state to track read/write sets, (ii) leverage cache coherence to detect conflicts, and (iii) use caches or write buffers to hold tentative writes. For these reasons, HTMs can execute existing lock-based programs and micro-benchmarks as fast or faster than multi-threaded programs using fine-grain locking [29,42,43].

Major hardware and software vendors have exhibited considerable curiosity regarding both hardware and software TMs as a way to ease their customers' transition to multicore systems. Nevertheless, vendors are understandably cautious about committing to hardware changes, in part, because (a) TM implementations are still in flux, and (b) ultimate success of TM is not yet assured.

Our Thesis. *We should facilitate TM adoption by deconstructing HTMs into separate, interchangeable components.* Section 2 provides an example TM deconstruction into five components that can be developed independently. We find two arguments supporting this thesis.

Deconstruction Aids HTM Refinement. Deconstruction into interchangeable components eases HTM refinement relative to monolithic HTMs. It also encourages clean interfaces between components to support independent evolution. In addition, deconstruction supports *hybrid* TM systems, in which not all components are provided by hardware. It also allows vendors to adopt TM earlier, knowing that they can more easily refine aspects later (mitigating concern (a)). Section 3 provides evidence for this benefit from LogTM variants and other systems.

Deconstruction Enables Use Beyond TM. Deconstruction also allows TM components to be applied toward solving problems beyond TM. This encourages vendors to adopt TM components earlier, knowing that they have value even it TM fails (mitigating concern (b)). In addition, deconstruction encourages early interface refinement to support these other uses, as opposed to discovering them after interfaces have been set. Section 4 outlines potential uses in reliability, security, performance, and correctness.

We support using HTM components with several preliminary case studies. Section 5.1 presents scalable watchpoints constructed from transactional memory components. Section 5.2 presents a transactional race recorder that enables deterministic replay.

Summary. We find evidence that deconstructing HTMs may speed TM adoption by reducing the risks of adopting non-optimal HTM components (because they can be improved later) and HTM components at all (because they have uses beyond TM). Moreover, we find that these benefits are more easily obtained if components are designed for multiple uses *a priori* rather than applied *ex post facto*. We believe that similar benefits are likely from deconstructing hybrid TMs as well.

2 Deconstructing HTMs

The first HTM proposal, HMTM, from Herlihy and Moss [17], proposed a single mechanism that provides the entire transaction capability. The system places data accessed by a transaction into a *transactional cache* and detects conflicts when another transaction attempts to access that data. However, the use of a single cache for all TM functionality prevents this system from supporting transactions that do not fit in the cache or from surviving interrupts or other context switches.

Our group's work on HTMs began with an attempt to remedy the size limitation with a coarse deconstruction of HTM into version management and conflict detection. *Version man-*

Table 1. Evolution of Wisconsin HTM Systems from HMTM

		HMTM[17]	LogTM [29]	Nested LogTM [30]	LogTM-SE [58]
GRAAS Components	**Grouping**	Xact load, store, validate, commit instructions	Begin, end, & abort instructions	*Closed/open* begin, commit, & abort instructions	Same as Nested LogTM
	Rollback	Flush transactional cache	Flat log, hardware fill, software abort handling	*Segmented log*, hardware fill, software abort handling	Segmented log, *holds pushed signatures*, hardware fill, SWaborts
	Access Summary	Transactional cache states	Cache R/W bits & sticky coherence	*Replicated* cache R/W bits & sticky coherence	*R/W signatures* & sticky coherence
	Access Check	On read/write, coherence protocol checks cache state	On read/write, coherence protocol finds R/W bits	Same as LogTM	On read/write, coherence protocol finds *signatures*
	Scheduling	Fixed in hardware	Fixed in hardware	Same as LogTM	HW + *SW conflict handler hooks*

agement handles the simultaneous storage of both *new* data (for commit) and *old* data (for abort). *Conflict detection* signals an overlap between the *write set* (data written) of one transaction and the write set or *read set* (data read) of other concurrent transactions.

Our first proposal, LogTM [29] combines innovative version management with extant conflict detection methods—extending cache tags with transaction state. On transactional writes, LogTM saves old values in per-thread *logs* and writes new values "in place," in contrast to HMTM, SLE [42], TCC, Bulk and others that buffer speculative new values and replace old values only at commit.

What's important here is not whether LogTM is better, but that deconstruction enables a separation of concerns: with LogTM, version management and conflict detection use different mechanisms that may be evolved separately. We will argue that separation accelerates both refining HTMs (Section 3) and allowing HTM mechanisms to be applied to other purposes (Section 4).

To these ends, our current best hypothesis for an HTM deconstruction is the *GRAAS components* (pronounced "grass"). For each component, we describe its function and prove a few sample mechanisms:

1) **Grouping**: How the TM system is informed of what instructions should form a transaction. This may be via explicit instructions that begin and end a transaction [14] or inferred from other instructions [7,17].

2) **Rollback**: How the TM system "undoes" a transaction's tentative execution to support abort. This may entail flushing new values from a cache [14] or walking a log to restore old values to memory [29].

3) **Access Summary**: How the TM system records a transaction's read/write sets. This information is needed to detect when transactions conflict and are not serializable. HTMs have provided this function with a separate cache [17], with bits on existing cache lines [2,14,29], with bloom-filter like signatures [8,28,58], and with memory tags [2,6].

4) **Access Check**: When and how the TM system checks for conflicts (i.e., access summary overlaps). Some HTMs check access lazily, by broadcasting the write set at commit to other processors [8]. Others perform access checks eagerly, as part of the coherence protocol [17,29].

5) **Scheduling**: How the TM system seeks concurrency, liveness, and fairness in the presence of conflicts. This generally takes the form of a conflict handler that may stall, abort, or queue a transaction. Scheduling has been implemented as a simple hardware conflict resolution policy [14,29] or as a software handler that can support more flexible policies [60].

In the next section, we provide evidence for the value of the GRAAS components. Nevertheless, GRAAS is a work-in-progress that should be interpreted as an example deconstruction, not the final word.

3 Deconstruction Aids HTM Refinement

Here, we examine how deconstructing HTMs into separate components facilitates improving HTMs. We first present our experience with LogTM as a case study and then touch upon how others have productively used deconstruction.

As discussed above, our group first developed LogTM. This system separated out and replaced version management (now renamed the GRAAS component *rollback)*, but left other components unchanged from prior HTMs (see first two columns of Table 1).

Our second HTM, *Nested LogTM* [30], adds support for closed [32] and open nested transactions [33,51]. Deconstruction separated implementation changes into three separate concerns. *Grouping* was refined so that commit instructions indicated closed or open nesting. *Rollback* used a segmented, rather than flat, log to enable partial abort. *Access summary* was enhanced by replicating read/write bit (R/W) bits on each cache block (e.g., 4 times). This last change, which increased hardware costs and limited the number of nesting levels to the replication of R/W bits, inspired our third HTM.

Our third HTM, *LogTM Signature Edition (LogTM-SE)* [58], reduces support needed in L1 caches, supports unbounded nesting, and (independently) enables better conflict management. Once again, decomposition made implementation more straightforward. The most important change is that the access summary of each transaction is maintained via a compact signature (e.g., 128 bytes) [5,31,39,47]. This change necessitated small changes to support rollback (to save/restore signatures on the log) and access check (check signatures rather than in-cache R/W bits). In addition, LogTM-SE supports trapping on transaction conflicts to enable flexible scheduling [46].

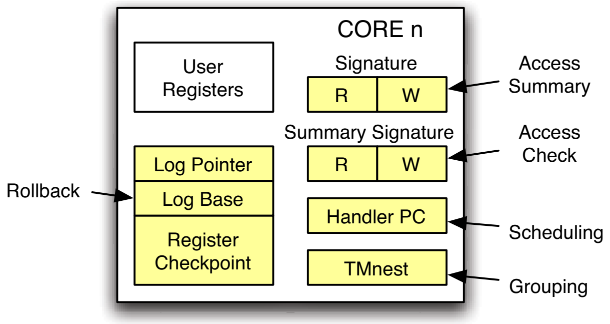

Figure 1: Additional state for a single core of a LogTM-SE CMP with the GRAAS components labeled.

The GRAAS deconstruction enabled these LogTM improvements (see Table 1). Understanding that access summaries are different than access checks, for example, allowed LogTM-SE to change access summaries to signatures without changing access checking via coherence.

Without deconstruction, the components of an HTM are tightly integrated, making it difficult to address a problem with one component without impacting the others. For example, when a single hardware structure, such as the cache, provides both access checks and access summaries, it is difficult for the OS to deschedule the transaction, because the access check mechanism is not available once the thread has been suspended. LogTM-SE solves this problem with an additional variety of access check, *summary signatures*. These are local access summaries representing descheduled threads that are checked on every memory reference.

3.1 Deconstructing LogTM-SE

To demonstrate the utility of deconstructing HTM, we provide several examples based on a deconstruction of LogTM-SE [58]. LogTM-SE supports the GRAAS components using a combination of hardware mechanisms and low-level software. Figure 1 shows the additional hardware state associated with each component.

Grouping. LogTM-SE hardware implements grouping with the TMnest register, which is the transaction nesting depth (0 means non-transactional). The *xbegin* and *xend* instructions increment and decrement this register.

Rollback. Rollback is implemented by a user-level software handler that restores old values from the transaction log, the bounds of which are defined by the Log Pointer and Log Base registers. Hardware writes old values to the log prior to store instructions.

Access Summary. Access summary is provided by hardware signatures (read and write). In LogTM-SE, the access summary for an active transaction is collected in the read and write signature and may be saved/restored by software.

Access Check. Access check is implemented by hardware checks of processors' primary and summary signatures. An extension to the coherence mechanism checks each memory request for conflicts with the primary signatures on all other processors. An additional hardware check tests for a conflict with the executing processor's local summary signature.

Scheduling. Scheduling in LogTM-SE is provided by (1) a fixed transaction conflict resolution policy implemented in the coherence mechanism and (2) a software handler for conflicts with descheduled transactions. The conflict handler may stall (and later resume) an active transaction, abort the current transaction and (later) restart it, or switch to an alternate execution path.

3.2 Deconstruction in Other HTMs

Several other HTMs implicitly use deconstruction to aid HTM refinement. For example, Bulk [8] enhances a design similar to TCC [14] by changing the access check mechanism and access summary mechanism. Similar to LogTM-SE, Bulk uses signatures to over-approximate the read- and write-sets of a transaction. But, Bulk detects conflicts by broadcasting the signatures during commit. Another example of limited deconstruction is to use OS scheduling primitives to resolve conflicts [60].

Deconstruction is also useful for constructing hybrid TM systems, where software and hardware cooperatively provide transactions. For example, SigTM relies on software for grouping, rollback, and scheduling, but uses hardware signatures for access checks and summarization [28].

4 Deconstruction Enables Use Beyond TM

Transactional memory was originally proposed to ease high-performance concurrent programming. However, deconstructing TM led us to realize that the GRAAS components may be independently useful to solve other problems. Components that support other uses may encourage hardware vendors to implement transactional memory, as the mechanisms are useful even if TM is not widely adopted.

Applying HTM components to other problem areas has the additional benefit of refining the interfaces. In many cases, small semantic changes or additional features may greatly improve the utility of a component. As we show in Section 5.1, the ability to continue a transaction after a conflict is simple to implement and useful for debugging purposes, but may not have been considered solely for transactional memory.

We identify four problem areas in which transactional components may be of use:

- **Reliability:** handling hardware and software failures
- **Security:** providing fine-grained access control
- **Performance:** speculating on fast-path code
- **Correctness:** finding bugs, including concurrency problems

In each of these areas, the GRAAS components provide new capabilities that are not possible with monolithic TMs or are prohibitively slow. Table 2 lists eight potential applications of the GRAAS components for addressing critical issues outside of TM.

Table 2. Mechanisms Supported by Transaction Components

Mechanism	Use	Grouping	Rollback	Access Summary	Access Check	Scheduling
Fine-grained Isolation	Reliability, Security	Around component	Abort	Writes	Against white / black list	
Failure Recovery	Reliability, Security	Around component	Abort			
Information Flow	Security	Around component		Reads / writes		
Resource Limits	Reliability, Security	Around component	Abort			Limit resource consumption
Speculation	Performance, security	Around code block	Retry alternate	Reads / writes	Against preset list	
Watch Points	Correctness				Against watch locations	Invoke debugger
Race Detection	Correctness	Races and sequential		Reads / writes	Against conflicts	Raise error when sequential
Replay	Correctness	Around races				Replay commit order

4.1 Improving Reliability

Three key problems in improving reliability are detecting failures, isolating faulty modules from correct modules, and recovering from failure. Fine-grained access control mechanisms developed for reliability [49,50,53], which detect when a faulty modules writes to data it does not control, are similar to the access check and access summary components of TM. With small modifications, a processor that implements the GRAAS components can also provide fine-grained memory isolation. The access summary for a module can be compared against either a white list of known-good locations or a black list of known-bad locations. A conflict signals a (possibly) illegal operation by the module. In addition, transactional access checks can prevent other code from observing corruption and subsequently failing. The access check component can optionally detect such violations automatically. Once a failure has been detected, either through hardware or software checks, rollback provides a mechanism to recover the system to a safe state.

For example, the Nooks driver isolation system [49] could be implemented more easily and execute faster with the GRAAS components. Nooks isolates drivers using page-level protection on virtual memory and recovers by reloading and re-initializing the driver. Rather than change page tables on every invocation of a device driver, a transactional isolation system would instead install a memory summary against which to check accesses by the driver. Similarly, rather than roll the driver back completely, transactional rollback allows only the last invocation of the driver to be rolled back, a much faster recovery process. For bugs that are detected quickly, this would greatly improve the availability of the system.

4.2 Improving Security

Fine-grained memory access control is also helpful for improving security, to prevent untrusted code from reading or writing sensitive data structures. A major challenge is ensuring that untrusted code cannot disable the access summarization and checks, which can be ensured through code inspection [50]. When the access check detects a conflict, this indicates a potential security problem that must be verified. The recovery strategy previously described can keep an application running in the presence of a security breach by rolling back corrupted data.

Two further problems where the transaction components can improve code security are denial of service and information flow tracking. Denial-of-service attacks plague Internet-facing servers. Frequently, an attacker will discover an algorithm that is polynomial or exponential in the size of an input, and then send a request exploiting this algorithm. Detecting an attack is difficult, because there is no resource accounting for small portions of a program: in most OSs the thread is the smallest granularity of accounting [4]. Recovery is also difficult, because it is not safe to kill a thread that may be holding locks. However, transactions must, by definition, be finite. We can use transactions to enforce limits on the length of transactions, in cycles or addresses referenced, and the scheduler can detect an attack when a transaction exceeds this limit. Rollback allows the program to recover to a point where data structures are unlocked and consistent.

A third security problem that may benefit from GRAAS components is information flow tracking [38, 40, 56, 57]. Systems that enforce information flow policies must record at a fine grain the memory read and written by a program, to ensure that secure data is not disclosed to low-security entities. The access summary component of transactions provides a convenient mechanism to track the memory read and written by a piece of code. The system can speculate that code does not violate the information flow policy. At commit, it can check whether the code referenced high or low security areas. If so, it may rollback the transaction and re-execute on a slower code path that tracks detailed information flow (at a finer grain than the access summary).

4.3 Improving Performance

TM hardware has often evolved from thread-level speculation hardware [8,14]. However, the speculation capability provided by rollback can be used separately for other purposes. For example, compilers often support optimizations that cannot be used because they are not safe in the presence of aliasing [13]. At run time, the cost of detecting aliasing may be too high to make the optimization useful [9]. The transactional components provide the opportunity to speculatively execute aggressively optimized code and cheaply detect aliasing at runtime. Similar to the memory speculation hardware on the Itanium processor [25], transactional access checks can detect when a value cached in a register has been accessed through a pointer. If aliasing occurs, the code rolls back and executes a less optimized version. This depends on the same speculation capability previously used for tracking fine-grained information flow, although the access check is against potential aliases. Rollback alone has also been proposed to simplify aggressive optimization [36]. Due to the overhead of beginning and ending transactions, performance gains are most likely to come from removing aliasing and other checks from loop bodies.

Speculative optimization with rollback is also useful to dynamic translation systems [1], such as the Transmeta code-morphing software [11]. These systems can take advantage of rollback to implement precise exception handling in the presence of aggressive optimizations during translation.

4.4 Improving Correctness

It is often difficult to determine what a program is doing at runtime because of the expense of complete monitoring. Previous work has addressed the use of leveraging the TM system to debug code within transactions [24], but when the mechanisms are decomposed, they may be used for all code. We find three interesting uses of the access check and summary components as fast mechanisms for monitoring program memory behavior.

First, access summaries and checks can detect when a program improperly uses transactions. For example, a program may not include all race conditions within transactions. However, by maintaining access summaries and access checks for code outside explicit transactions, races can be detected when two different threads access a memory block. Compiler support may be required to detect safe sharing patterns, for example when a memory block is reallocated between threads [19].

A second use of transactional components is to implement scalable watchpoints [59]. Current architectures support only a small number of watchpoints (e.g, four on the Pentium). The GRAAS access check component can provide an arbitrary number by checking against watchpoint addresses instead of (or in addition to) the transaction's access summary. When a check fails, software can first determine if it is a false positive, and if not, execute the watch point code. In Section 5.1, we demonstrate a simple implementation of scalable watchpoints relying on transactional components.

A third use of transactional components is efficient replay of multithreaded code. Flight Data Recorder (FDR) is a mechanism to provide deterministic replay by logging a subset of coherence requests [54,55,56]. However, if all sharing takes place through transactions, it is sufficient to record instead a global order of transaction commits, e.g., with a shared counter. The transactional scheduler can use this order during replay to ensure the same execution. We discuss a sample implementation of a transactional replay mechanism in Section 5.2.

4.5 Summary

Once deconstructed, the GRAAS components of an HTM may be useful in solving many software problems. Some of these solutions require additional hardware support, such as the ability to limit transaction size for denial of service prevention. Thus, these alternate uses should be considered when component interfaces are designed.

5 Case Studies

This section describes two preliminary case studies where GRAAS components solve software problems beyond transactions: a watchpoint mechanism that executes a handler when a specified address is about to be accessed, and a recording mechanism that supports deterministic replay of transactional programs.

We implement both mechanisms on top of LogTM-SE [58]. However, the mechanisms depend on the GRAAS components and not on the details of LogTM-SE. The watchpoint mechanism relies on access checks and scheduling and the recorder depends on the grouping and scheduling components.

5.1 Scalable Watchpoints

Most architectures provide a limited number of memory watchpoints to assist programmers in monitoring memory locations. The processor generates a trap when a program accesses a watched address. However, each watchpoint requires a separate hardware register, which limits the number of watchpoints, e.g. four in Intel x86. Recently, Zhou, et al. proposed new special-purpose architectural support to implement large numbers of watchpoints [59].

The *access check* component of GRAAS can provide an arbitrary number of watchpoints without special-purpose hardware. Rather than checking access against another transaction, a watchpoint mechanism can use this mechanism to check accesses against a list of watchpoints. When a check fails, hardware traps into *scheduling* software that executes the watchpoint code. On LogTM-SE, which supports strong isolation and hence checks access for all memory operations, the watchpoint mechanism works for both transactional and non-transactional code.

As a proof of concept, we have implemented a watchpoint mechanism based on LogTM-SE. The watchpoint software module is implemented as a library that extends LogTM-SE's runtime. The library provides `add_watchpoint` and `remove_watchpoint` functions. The interface allows the programmer to specify the address and size of the watched address region and to associate a monitoring function with each watchpoint.

Watchpoints leverage LogTM-SE's *summary signature*, which normally provides access checks for descheduled transactions. Every processor checks its local summary signature on

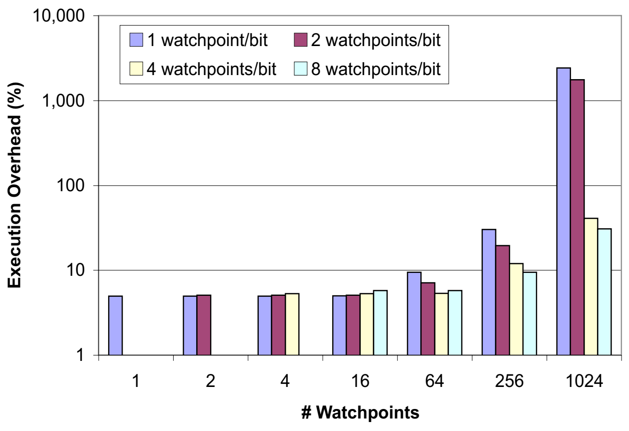

Figure 2: Execution overhead for 1 true watchpoint and 0-1023 false watchpoints with and without signature aliasing.

every memory request, trapping to software to resolve conflicts. We implement watchpoints by adding the watched addresses to a thread's summary signature.

While the summary signature detects when watched addresses are accessed, it is not sufficient to provide watchpoints. First, the signatures in LogTM-SE may raise false conflicts due to the compact encoding. For TM, it suffices to treat these as true conflicts and either abort the transaction or stall until the transaction causing the conflict completes. However, after reaching a watchpoint, we frequently want to continue execution while continuing to watch the same address. Second, LogTM-SE does not provide hardware to manipulate signatures in software. The watchpoint mechanism requires this to add individual addresses to a signature.

We extend the LogTM-SE interface with a *get_index* operation, which returns a compact representation of the indexes of the summary signature bits to which a virtual address hashes. The watchpoint mechanism uses this operation to add and remove watched addresses.

Our second extension is a non-privileged *watchpoint flag* that causes hardware to skip the summary signature access check for the next memory request. This allows the watchpoint mechanism to continue execution after a watchpoint executes without triggering another access check conflict.

The software watchpoint module manages the list of watchpoints and dispatches monitoring code. When the hardware detects a conflict, it traps into the software conflict handler. As there may be false positives, the handler compares the faulting address against the list of watched regions. If one (or more) are found, the library executes the monitor function. In all cases, it sets the watchpoint flag before continuing execution. To support watchpoints and virtualized transactions simultaneously, the library only executes watchpoint code when it detects that there was *no* transactional conflict.

We evaluated the watchpoint mechanism with the LogTM-SE simulator [58] on the bzip2 program from SPECint2006. We measure the overhead of (1) executing a watchpoint and (2) false positives due to LogTM-SE's signatures. We insert one true watchpoint that triggers 46,000 times per second and between 0 and 1023 false watchpoints (addresses that are never referenced). Executing a single true watchpoint takes 200 cycles, mostly due to the time to trap. Figure 2 shows the overhead from false conflicts as the number of watchpoints grow from 1 to 1024. The rate increases as the signature, which is only 1024 bits, saturates and causes nearly every memory reference to trap. The figure also shows the overhead for different levels of aliasing in the signature. When a bit in the signature represents more than one watchpoint, fewer false positives occur because the signature is less populated. Overall, LogTM-SE's access check component provides a simple and lightweight mechanism for scalable watchpoints.

5.2 Transactional Flight Data Recorders

Effectively debugging of multithreaded software is critical to the success of emerging multicore chips. Valuable to any debugger, *deterministic replay* enables a developer to re-execute the (buggy) program and zero in on bugs that faithfully reappear. Moreover, deterministic replay can be useful for fault detection/recovery [41] and intrusion detection [12].

A key challenge for deterministic replay is *recording memory races,* where it is sufficient to record the outcomes of all conflicting memory accesses. Two accesses (reads or writes) *conflict* if they are from different threads, access the same memory block, and at least one of them is a write. Since extant software race recorders slow down program execution tremendously [23,37], researchers have proposed hardware implementations [3,54,55,35,34,56].

The current *Flight Data Recorder (FDR)* [56], for example, supports multicore designs using sequential consistency (SC) or total store order (TSO), which is x86-like. As depicted in Figure 3, FDR augments each core with a dynamic *instruction counter (IC)* and local *timestamp memory (TSM).* FDR piggybacks timestamps on some coherence messages and exploits transitivity to add modest runtime overhead while logging only about one byte per thousand instructions executed. Selected logging of *values (Val)* supports TSO executions that are not SC executions.

While hardware vendors have adopted neither HTM nor FDR, combining the two in a *Transactional Flight Data Recorder (XFDR)* can reap synergistic benefits to promote adoption of both in two scenarios. All XFDR variants leverage the grouping component to reduce the logging requirement and scheduling to support replay.

Scenario 1: Races occur only among transactions. This scenario occurs if it is *"all transactions, all the time"* [14] or non-transactional memory races are handled separately (or cause deterministic replay to fail).

With this scenario, it is sufficient for XFDR to record the order that transactions commit, and it is not necessary to explicitly track any memory references. A naive implementation uses a *global counter* (protected by a lock). Each thread has a private log. On commit, the thread atomically increments the counter and logs the current value.

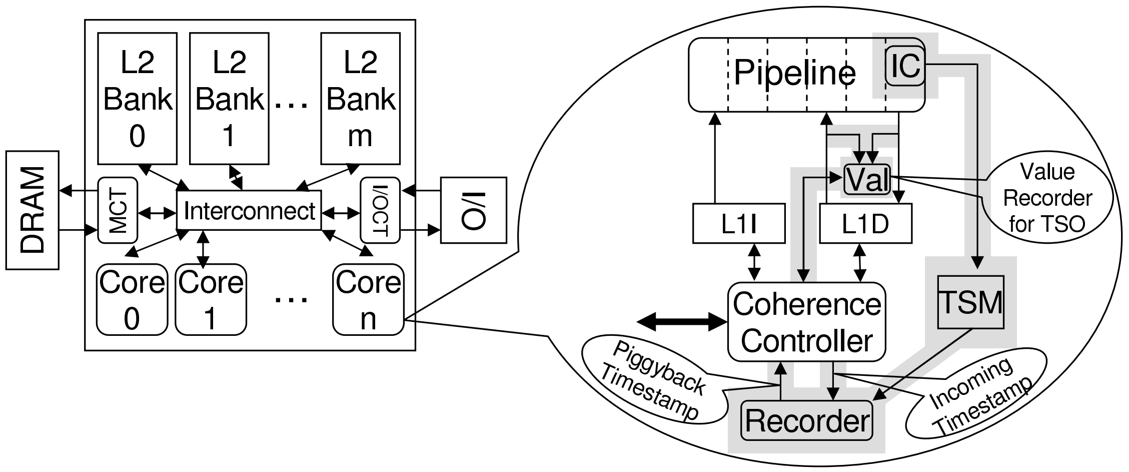

Figure 3. A base multicore (unshaded) supplemented with FDR memory race recording hardware (shaded). Used with permission from Xu et al. [56].

A better implementation uses a *scalar timestamp* [21]. Each thread remembers a single scalar timestamp. Coherence responses carry other threads's timestamps that the requesting thread uses to update its timestamp. On commit, the thread records its current timestamp in its per-thread log without any coordination with others.

This latter design improves on FDR by reducing the timestamp memory (TSM) from 24 Kbytes to 8 bytes and eliminating the instruction count (IC) and value recorder logic (Val). Nevertheless, we expect similar logging performance.

Scenario 2: Races occur anywhere. Here we must enable deterministic replay even when non-transactional memory accesses race with transactions or each other.

With this scenario, we see three initial designs. First, we can use FDR as is. Second, we can augment FDR with the scalar timestamp logic above to potentially reduce logging. Third, we can use augmented FDR, but greatly reduce TSM size, because it may be unimportant to optimize logging of non-transactional memory accesses.

Preliminary work. To date we have implemented two versions of the global-counter XFDR. The first is an all-software version that runs on a Sun T1 (Niagara) multicore system (where the TM system is implemented with a simple global lock). The second is a hardware implementation based on LogTM [29] simulated using GEMS [52]. We exercise both XFDR implementations (and the corresponding replayers) with a multithreaded program, *racey*, whose final output is sensitive to the order of its frequent data races [54]. In particular, racey computes a signature using a multiplicative congruential pseudo-random number generator [18]. After addressing several minor bugs, the signatures of racey replays match that those of the corresponding recorded execution. This builds confidence in both XFDR implementations, but does not prove them correct.

Future work. We plan to implement all three Scenario-2 variants, so we can establish that XFDR can improve upon FDR performance, reduce hardware cost, or both. In doing so, XFDR could facilitate the adoption of both hardware transaction memory and deterministic replay support.

6 Conclusion

The success of hardware transactional memory depends on convincing chip vendors of its long-term value. Decomposing transactional memory increases the likelihood of its eventual adoption in two ways. First, the components may evolve independently, allowing faster innovation. Second, the components provide more value if exposed separately, because they may be additionally targeted at other problem areas.

In this paper, we presented a decomposition of TM into five GRAAS components: grouping, rollback, access check, access summary, and scheduling. Although the GRAAS components are merely one example of how to decompose transactional memory, we have shown that they have already helped refine one HTM (LogTM-SE) and that they may be applied constructively to other important problems.

7 References

[1] Erik Altman, Kemal Ebcioglu, Michael Gschwind, and Sumedh Sathaye. Advances and Future Challenges in Binary Translation and Optimization. *Proceedings of the IEEE*, 89(11):1710–1722, November 2001.

[2] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *HPCA 11*, February 2005.

[3] David F. Bacon and Seth Copen Goldstein. Hardware-Assisted Replay of Multiprocessor Programs. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, pages 194–206, 1991.

[4] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation*, pages 45–58, February 1999.

[5] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[6] Colin Blundell, Joe Devietti, E Christopher Lewis, and Milo M.K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *ISCA 34*, June 2007.

[7] Luis Ceze, Pablo Montesinos, Christoph von Praun, and Josep Torrellas. Colorama: Architectura Support for Data-Centric Synchronization. In *HPCA 13*, February 2007.

[8] Luis Ceze, James Tuck, Calin Cascaval, and Josep Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *ISCA 33*, June 2006.

[9] Xiaoru Dai, Antonia Zhai, Wei-Chung Hsu, and Pen-Chung Yew. A General Compiler Framework for Speculative Optimizations Using Data Speculative Code Motion. In *CGO'05*, pages 280–290, March 2005.

[10] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchango, Mark Moir, and Daniel Nussbaum. Hybrid Transactional Memory. In *ASPLOS 12*, October 2006.

[11] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta Code Morphing Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO'03*, pages 15–24, March 2003.

[12] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *OSDI 5*, pages 211–224, December 2002.

[13] Manel Fernández and Roger Espasa. Speculative Alias Analysis for Executable Code. In *PACt 2002*, pages 222–231, September 2002.

[14] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional Memory Coherence and Consistency. In *ISCA 31*, June 2004.

[15] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA 2003*, October 2003.

[16] Maurice Herlihy, Victor Luchangco, Mark Moir, and William Scherer III. Software Transactional Memory for Dynamic-Sized Data Structures. In *Twenty-Second ACM Symposium on Principles of Distributed Computing, Boston, Massachusetts*, July 2003.

[17] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA 20*, pages 289–300, May 1993.

[18] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[19] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From Uncertainty to Belief: Inferring the Specification Within. In *OSDI 7*, November 2006.

[20] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid Transactional Memory. In *PPoPP'06*, pages 209–220, March 2006.

[21] Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[22] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2006.

[23] Thomas J. Leblanc and John M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[24] Yossi Lev and Mark Moir. Debugging with Transactional Memory. In *TRANSACT 2006*, June 2006.

[25] Jin Lin, Tong Chen, Wei-Chung Hsu, and Pen-Chung Yew. Speculative register promotion using Advanced Load Address Table (ALAT). In *CGO'03*, pages 125–134, March 2003.

[26] Austen McDonald, JaeWoong Chung, Brian Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural Semantics for Practical Transactional Memory. In *ISCA 33*, June 2006.

[27] Austen McDonald, JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Brian D. Carlstrom, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun. Characterization of TCC on Chip-Multiprocessors. In *PACT 2005*, September 2005.

[28] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen Mcdonald, NAthan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *ISCA 34*, June 2007.

[29] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-Based Transactional Memory. In *HPCA 12*, pages 258–269, February 2006.

[30] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting Nested Transactional Memory in LogTM. In *ASPLOS 12*, pages 359–370, October 2006.

[31] Andreas Moshovos, Gokhan Memik, Babak Falsafi, and Alok Choudhary. JETTY: Filtering Snoops for Reduced Power Consumption in SMP Servers. In *HPCA 7*, January 2001.

[32] J. Eliot B. Moss. *Nested transactions: an approach to reliable distributed computing*. PhD thesis, Massachusetts Institute of Technology, 1981.

[33] J. Eliot B. Moss. Open Nested Transactions: Semantics and Support. In *Workshop on Memory Performance Issues*, February 2006.

[34] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. Recording Shared Memory Dependencies Using Strata. In *ASPLOS 12*, pages 229–240, October 2006.

[35] Satish Narayanasamy, Gilles Pokam, and Brad Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *ISCA 32*, pages 284–295, June 2005.

[36] Naveen Neelakantam, Ravi Rajwar, Suresh Srinivas, Uma Srinivasan, and Craig Zilles. Hardware Atomicity for Reliable Software Speculation. In *ISCA 34*, June 2007.

[37] Robert H. B. Netzer. Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging (PADD)*, pages 1–11, 1993.

[38] James Newsome and Dawn Song. Dynamic Taint Analysis: Automatic Detection, Analysis, and Signature Generation of Exploit Attacks on Commodity Software. In *Proceedings of the Network and Distributed Systems Security Symposium*, February 2005.

[39] Jih-Kwon Peir, Shih-Chang Lai, Shih-Lien Lu, Jared Stark, and Konrad Lai. Bloom Filtering Cache Misses for Accurate Data Speculation and Prefetching. In *Proceedings of the 2002 International Conference on Supercomputing*, pages 189–198, June 2002.

[40] Feng Qin, Zhenmin Li, Yuanyuan Zhou, Cheng Wang, Ho seop Kim, and Youfeng Wu. (LIFT): A Low-Overhead Practical Information Flow Tracking System for Detecting General Security Attacks. In *MICRO 39*, December 2006.

[41] Feng Qin, Joe Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies — a safe method to survive software failure. In *Proceedings of the 20th ACM Symposium on Operating System Principles*, October 2005.

[42] Ravi Rajwar and James R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *MICRO 34*, December 2001.

[43] Ravi Rajwar and James R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *ASPLOS 10*, October 2002.

[44] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing Transactional Memory. In *ISCA 32*, June 2005.

[45] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural Support for Software Transactional Memory. In *MICRO 39*, December 2006.

[46] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *PODC 24*, July 2005.

[47] Simha Sethumadhavan, Rajagopalan Desikan, Doug Burger, Charles R. Moore, and Stephen W. Keckler. Scalable Hardware Memory Disambiguation for High ILP Processors. In *MICRO 36*, December 2003.

[48] Nir Shavit and Dan Touitou. Software Transactional Memory. In *PODC 14*, pages 204–213, August 1995.

[49] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *SOSP 19*, pages 207–222, October 2003.

[50] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *SOSP 14*, pages 203–216, December 1993.

[51] Gerhard Weikum and Hans-Jorg Schek. *Concepts and Applications of Multilevel Transactions and Open Nested Transactions*. Morgan Kaufmann, 1992.

[52] Wisconsin Multifacet GEMS Simulator. http://www.cs.wisc.edu/gems/.

[53] Emmett Witchel, Josh Cates, and Krste Asanovic. Mondrian memory protection. In *ASPLOS 10*, pages 304–316, October 2002.

[54] Min Xu, Rastislav Bodik, and Mark D. Hill. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. In *ISCA 30*, pages 122–133, June 2003.

[55] Min Xu, Rastislav Bodik, and Mark D. Hill. A Regulated Transitive Reduction (RTR) for Longer Memory Race Recording. In *ASPLOS 12*, pages 49–60, October 2006.

[56] Min Xu, Rastislav Bodik, and Mark D. Hill. A Hardware Memory Race Recorder for Deterministic Replay. *IEEE Micro*, 27(1), Jan/Feb 2007.

[57] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th (USENIX) Security Symposium*, August 2006.

[58] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *HPCA 13*, pages 261–272, February 2007.

[59] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *ISCA 31*, pages 224–237, June 2004.

[60] Craig Zilles and Lee Baugh. Extending Hardware Transactional Memory to Support Non-busy Waiting and Non-transactional Actions. In *TRANSACT 2006*, June 2006.