Procrastination Beats Prevention

Timely Sufficient Persistence for Efficient Crash Resilience

Faisal Nawab[*,†] Dhruva R. Chakrabarti[†]

Terence Kelly[†] Charles B. Morrey III[†]

[*]CS Dept., UC Santa Barbara [†]HP Labs, Palo Alto, CA

ABSTRACT

Preserving the integrity of application data across updates in the presence of failure is an essential function of computing systems, and byte-addressable non-volatile memory (NVM) broadens the range of fault-tolerance strategies that implement it. NVM invites database systems to manipulate durable data directly via LOAD and STORE instructions, but overheads due to the widely used mechanisms that ensure consistent recovery from failures impair performance, e.g., the logging overheads of transactions. We introduce the concept of *Timely Sufficient Persistence* (TSP) mechanisms, which is relevant to both conventional and emerging computer architectures. For a broad spectrum of fault-tolerance requirements, satisfactory TSP mechanisms typically involve lower overheads during failure-free operation than their non-TSP counterparts; hardware and OS support can facilitate TSP mechanisms. We present TSP variants of programs representing two very different classes of shared-memory multi-threaded software that store application data in persistent heaps: The first employs conventional mutexes for isolation, and TSP substantially reduces the overhead of a fault-tolerance mechanism based on fine-grained logging. The second class of software employs lock-free and wait-free algorithms; remarkably, TSP is very easy to retrofit onto a non-resilient design and enjoys *zero runtime overhead*. Extensive experiments confirm that TSP yields robust crash resilience with substantially reduced overhead.

1. INTRODUCTION

Runtime failures such as process crashes, operating system kernel panics, and power outages can corrupt or destroy application data unless effective measures protect application data integrity. Both disk-based and main-memory database systems running on conventional hardware with volatile byte-addressed memory and non-volatile block storage employ sophisticated techniques to preserve the integrity of application data across updates in the presence of failures [13]. Unfortunately these techniques sometimes suffer painful runtime overheads due to the performance characteristics of hard drives and solid state drives.

Emerging hardware promises durability with greatly improved performance [14]. Byte-addressable non-volatile memory (NVM) is becoming available [1] and has enabled new database system designs with improved performance [17, 21, 26]. NVM has sparked increased interest in main-memory databases that store all data in memory and directly manipulate durable data in persistent heaps via LOAD and STORE instructions rather than through database or filesystem interfaces [25]. Such approaches offer superior performance compared to disk- and SSD-based systems with comparable fault tolerance, but they suffer noticeable overheads during failure-free operation [14].

We begin by introducing a conceptual framework that encompasses both conventional and emerging hardware. We then systematically characterize as a function of fault-tolerance requirements the circumstances under which runtime overheads may safely be postponed until failures actually occur and/or eliminated outright, and we tailor the specific measures taken to the available hardware. The result is the concept of *Timely Sufficient Persistence* (TSP). Loosely speaking, a TSP fault-tolerance mechanism eschews costly preventive measures in favor of minimalist remediation when failure is imminent, which typically reduces runtime overheads substantially. It furthermore helps us to identify new hardware and OS support to facilitate new fault tolerance mechanisms. We restrict ourselves in this paper to the context of a single computer, and the following types of failures: process crashes, kernel panics, and power outages.

The remainder of this paper is organized as follows: Section 2 reviews emerging hardware architectures, describes corresponding fault tolerance mechanisms and application programming styles, and discusses how these new developments relate to traditional fault-tolerance objectives. Section 3 defines Timely Sufficient Persistence and describes how it can be implemented on both conventional and emerging hardware. Section 4 presents two case studies illustrating the benefits of TSP: In one case, TSP imbues a program with crash resilience while adding *zero* runtime overhead; in another case, TSP substantially reduces the runtime overhead of an existing fault-tolerance technique. Section 5 presents experimental results confirming both the fault tol-

erance properties and performance advantages of TSP in our two case studies, and Section 6 concludes with a discussion.

2. NEW HARDWARE & SOFTWARE

Database management systems are designed to survive severe failures, e.g., power outages. When they run on conventional hardware with volatile DRAM memory, they must therefore write data to block-addressed storage devices, which perform poorly for random writes. Write-ahead logging and grouping partially mitigate the storage I/O bottleneck [13]. The need to synchronously commit such writes to block storage may limit overall database performance to storage bandwidth [10]. Solid state drives (SSDs) enable databases with improved performance compared with disk-based designs [19]. However, SSDs still suffer from the same fundamental drawback as HDDs: both kinds of storage devices require databases to synchronously commit data via a relatively slow block I/O interface [19]. Main-memory databases are optimized for the case where all data fits in memory [22]. To this end, systems such as HBase [9] and Silo [24] redesign recovery and locking mechanisms so as to minimize the impact of these bottlenecks. However, main-memory databases that have not been re-architected for the case that all data fits in durable memory still suffer from the I/O bottleneck of persisting on stable storage. Traditional filesystems running on conventional hardware provide an alternative means of manipulating durable data, but they suffer the same storage bottlenecks that afflict databases [4].

Byte-addressable non-volatile memory (NVM) has recently become available. Conventional DRAM is approaching density scaling limits [1], and the most promising replacement technologies—phase change memory, spin-torque transfer memory, and memristors—all happen to be non-volatile [26]. If any of these technologies architecturally supplants or supplements DRAM it will provide inherently non-volatile random-access memory (NVRAM). Meanwhile, hybrid DRAM/flash memory DIMM packages backed by batteries or supercapacitors (NVDIMMs) implement NVM by persisting the contents of DRAM to flash when power is lost [14]. Non-volatile CPU caches have been proposed to complement NVM [28]. Another way to preserve the contents of volatile DRAM across utility power outages is to fail over to an uninterruptible power supply, a traditional building block of fault-tolerant systems [12]. Regardless of the underlying technology, all forms of NVM share several advantages over block-addressed storage devices. NVM is installed on the memory bus and enjoys access latencies and bandwidth comparable to DRAM. NVM is accessed at cache-line granularity via LOAD and STORE instructions. In contrast to the relatively coarse, slow, mediated updates offered by database and file systems atop block storage devices, NVM enables fast, fine-grained, direct updates by application software.

The potential of NVM has been explored in the context of disk-based and main-memory databases [17, 21, 26] and filesystems [5]. However the arrival of new forms of NVM has also renewed interest in a style of application programming that has long been possible but that has remained outside the mainstream until recently. Since the days of MULTICS, some operating systems have offered application programs the *illusion* that LOAD and STORE instructions operate upon durable data [6]; file-backed memory mappings provide this illusion on modern POSIX systems [23]. Atop such mechanisms it is possible to layer higher-level abstractions ranging from straightforward persistent heaps [18] to sophisticated object databases [27].

Compared with the more mainstream approach in which applications manipulate durable data via filesystem or database interfaces, the "NVM style" of direct manipulation offers several attractions. The most obvious is that in-memory data structures and algorithms are sometimes more convenient and more natural than the storage-oriented alternatives. A related issue is that translating between in-memory and serial data formats can be cumbersome. Translation between serial and in-memory formats can also be slow and error-prone; parsers, for example, are notorious for harboring bugs.

Supporting fault-tolerant "NVM-style programming" in the age of genuine NVM presents interesting new opportunities and challenges. Failures that abruptly terminate program execution can leave application data in NVM in an inconsistent state, so the challenge is to ensure that recovery can always restore consistency to data that survives the failure. Recent research has proposed transactional updates of persistent heaps, where transactions are defined either explicitly by the programmer [25] or are automatically inferred from the target program's use of mutual exclusion primitives [2, 3]. In the latter approach, synchronously flushing UNDO log entries to NVM immediately before STORE instructions execute enables recovery code to roll back transactions as necessary to restore the persistent heap to a consistent state, and such synchronous flushing adds noticeable overhead during failure-free operation.

Fortunately, some characteristics of emerging hardware work to our advantage when addressing specific kinds of failures. For example, the time and energy costs of flushing volatile CPU cache contents to the safety of NVM are miniscule compared to the corresponding costs of evacuating data in volatile DRAM to block storage [14]—a crucial difference that helps enormously if we must quickly panic-halt a faulty OS kernel. In general, the key to finding the best designs for meeting given fault-tolerance requirements on emerging hardware is to systematically consider the costs of moving data out of harm's way and to devise contingency plans that replace burdensome migrations during failure-free operation with guarantees of last-minute rescue. We shall see that emerging architectures sometimes reward procrastination handsomely.

3. TIMELY SUFFICIENT PERSISTENCE

Application requirements must distinguish tolerated failures from non-tolerated failures. Process crashes and kernel panics resulting from software or hardware errors are frequently placed in the former category, as are power outages. Application requirements must furthermore specify what subset of *critical* application data must survive tolerated failures. For example, requirements might declare that the entire state of a process is critical; more selective requirements might instead deem the process heap to be critical but permit thread execution stacks to be lost. Requirements might even designate different fault tolerance requirements for different subsets of application data. Requirements must also distinguish between fail-stop failures that abruptly halt process/thread execution and failures that first corrupt ap-

plication data. For example, when a process on a POSIX system receives a `SIGKILL` signal, all threads merely halt; the same is sometimes true when a process triggers a trap, e.g., by executing illegal instructions. By contrast, memory corruption errors in C/C++ programs often corrupt critical application data.

Fault-tolerance strategies typically move data from places where tolerated failures threaten corruption or destruction to places beyond the reach of tolerated failures; we respectively refer to such locations as *vulnerable* and *safe*. Safety can be defined only with respect to fault-tolerance requirements and is orthogonal to hardware characteristics such as volatility. For example, ordinary volatile DRAM can be safe with respect to process crashes, but even hard disks may be deemed vulnerable if we must tolerate catastrophes that wipe out entire data centers. Finally, we must ask whether we have adequate notice of tolerated failures to move critical data from vulnerable locations to safe ones. If so, we may seek improved performance while still meeting fault-tolerance requirements by trading runtime guarantees that critical data *is* in a safe location for guarantees that the data *will be moved to safety* should the need ever arise.

Timely Sufficient Persistence (TSP) describes fault-tolerance mechanisms that make such tradeoffs. A TSP design satisfies its requirements by moving a *minimal* amount of data (typically only critical data) to a location that is *adequately* safe (typically no safer) and does so in a *timely* manner (typically "just in time"). For example, Whole System Persistence [14] is an ingenious two-stage TSP design that protects the entire state of a computer from power outages by flushes the contents of volatile CPU registers and caches into volatile DRAM using residual energy stored in the system power supply and then evacuating the contents of DRAM into flash storage using energy stored in supercapacitors. This design completely avoids any overhead during failure-free operation. Presently we shall consider other TSP designs that tolerate a wider range of failures (e.g., due to software errors), that protect critical data more selectively, and that offer similarly attractive performance.

In our experience it is instructive to ask simple questions about the minimum support needed to satisfy given fault tolerance requirements—if only just barely—and to ask what "hidden" support may be present in the hardware and systems we are already using. Such exercises have more than once led the authors to insights that in turn informed improved TSP designs that, in retrospect, had been right under our noses but that we had overlooked before we began to seek TSP solutions explicitly. For example, consider the requirement that critical data that is explicitly placed in memory allocated through a special interface must survive process crashes only. A naïve approach might begin with the observation that physical memory allocated to a process is promptly reclaimed by the OS when the process crashes, with no opportunity for the process to rescue its contents. This line of reasoning might then conclude that crash tolerance in this context requires preemptively (and perhaps synchronously) committing data to durable media during failure-free operation.

A better approach begins by asking what minimal degree of "durability" suffices to survive process crashes: POSIX calls it "kernel persistence," and files in memory-backed filesystems have this property. We then consider what happens when such a file is memory mapped into the address space of a process that STOREs data into the mapping region and then crashes. The modified physical memory page frames corresponding to the mapping are also pages in the backing file and are *not* reclaimed by the OS when the process crashes. Furthermore STOREd data in the CPU cache at the time of the crash will eventually be evicted into the memory-backed file and meanwhile will be visible from the cache to any process that reads the file. Therefore if the process places critical data in memory corresponding to a memory-mapped file from a DRAM-backed file system, following a crash the file will contain all data STOREd by the process up to the instant of the crash, and we obtain this guarantee with no overhead during failure-free operation. (Our technical report provides additional detail and references on the interaction between process crashes and file-backed memory mappings [15].) Of course, additional measures may be required to ensure that application data stored in the file can be restored to a consistent state following a crash; we consider two different ways of ensuring consistent recoverability in Section 4. The important point is that seeking a TSP solution has gotten us halfway to our goal with zero runtime overhead.

Different kinds of failures call for different TSP designs. If we are required to tolerate kernel panics, for example, we must arrange for the dying OS to flush volatile CPU caches to memory. This suffices to meet the requirement if memory is non-volatile (or if the machine architecture preserves the contents of memory across "warm reboots" [16]). If memory is volatile and is not preserved across OS re-starts, the contents of memory must be written to stable storage before the panic'd OS shuts down the machine. An HP team has implemented the required support in the Linux kernel's panic handler, which required a relatively small amount of straightforward code. Power outages admit a spectrum of TSP designs ranging from mundane uninterruptible power supplies to sophisticated and resourceful strategies for storing and scrounging just enough energy to rescue critical data [14]. Emerging non-volatile memories can dramatically reduce the time and energy cost of keeping a machine running long enough to rescue critical data after utility power fails.

Conventional relational database management systems allow the user to trade consistency for performance via configuration parameters. For example, serializability and snapshot isolation offer different performance and consistency guarantees. TSP designs provide a wider range of applications with analogous tradeoffs among failure toleration requirements, hardware and system software support, and performance during failure-free operation.

4. CASE STUDIES

We now consider in detail two approaches to ensuring consistent recovery of application data in multi-threaded programs that manipulate persistent heaps via CPU LOAD and STORE instructions. Both approaches share several features in common: the programming model is convenient, familiar, and readily implementable in mainstream programming languages such as C++; the programmer obtains access to address space regions backed by durable media via a conventional memory allocation interface (e.g., `malloc` for C/C++); and the programmer assists recovery by ensuring that all live application data in the persistent heap are

reachable from a heap-wide root pointer manipulated via simple `get_root()` and `set_root()` interfaces. Finally, in both approaches the application programmer must ensure that concurrent threads access shared data in an orderly manner, free of data races and other concurrency bugs. In one of our approaches, multithreaded isolation depends upon conventional synchronization primitives (e.g., Pthread mutexes); the other relies upon non-blocking algorithms. We describe how TSP enables both kinds of multithreaded software to ensure consistent recovery of the persistent heap without high-latency CPU cache flushing during failure-free operation.

Implementations of both approaches on emerging architectures featuring NVRAM or NVDIMM memory offer substantial advantages, but implementation on conventional hardware (volatile DRAM and block-addressed storage) is also possible. To tolerate process crashes only, it suffices to ensure that the persistent heap is backed by a memory-mapped file in an ordinary filesystem or even a file in a DRAM-backed file system (e.g, `/dev/shm`). To tolerate kernel panics, the kernel must flush volatile CPU caches to memory; if the latter is volatile, memory regions corresponding to persistent heaps must be written to durable storage. To tolerate power outages, sufficient standby power must be available to flush CPU caches and move persistent heap data to durable media. As noted in Section 1, NVRAM and NVDIMMs dramatically reduce the time and energy cost of tolerating both kernel panics and power outages.

Sections 4.1 and 4.2 describe the principles underlying the two approaches; Section 5 describes corresponding implementations and our empirical evaluation of their correctness and performance overheads.

4.1 Zero-Overhead Atomic Updates

This section presents the remarkable observation that a well-known class of multi-threaded *isolation* mechanisms, together with TSP, guarantee *consistent recovery* from crashes without the need for any additional mechanisms or precautions whatsoever.

Following the terminology of Fraser & Harris [8], we say that an algorithm that ensures orderly multi-threaded access to shared in-memory data is *non-blocking* if the suspension or termination of any subset of threads cannot prevent remaining active threads from continuing to perform correct computation. Non-blocking algorithms cannot employ conventional mutual exclusion because a mutex held by a terminated thread will never be released, which prevents all surviving threads from accessing data protected by the mutex. Threads in non-blocking algorithms typically employ atomic CPU instructions such as compare-and-swap to update shared memory while precluding the possibility that other threads may observe inconsistent states of application data. *Lock-free* algorithms, a special case of non-blocking algorithms, offer the stronger guarantee that forward progress occurs even in the presence of contention for shared data. *Wait-freedom* is yet a stronger guarantee that a bounded number of operations is needed to complete an operation. All wait-free algorithms are lock-free. We employ lock-free and wait-free sub-species of non-blocking algorithms, using the latter term for brevity. One additional definition helps us to reason about the effects of crashes: Following Pelley et al. [20], we imagine a thread called the *recovery observer* that is created at, and observes the state of program memory at, the instant when all other threads in a program abruptly halt due to a crash.

Consider a program whose application-level data resides in a persistent heap and is manipulated with a non-blocking algorithm. The heap is furthermore updated in TSP fashion, i.e., in the event of a crash due to any tolerated failure, data in volatile locations (e.g., CPU caches or DRAM) will be flushed to durable media (NVRAM/NVDIMMs or stable storage) as necessary. We shall see that under these assumptions, a crash cannot prevent consistent recovery of the application data in the persistent heap.

Consider a crash that abruptly terminates all of the program's threads. We imagine a recovery observer created at the instant of the crash and consider its view of memory. Thanks to TSP, practical/implementable recovery code will have precisely the same view of memory as our hypothetical recovery observer. In particular, TSP ensures that the state of recovered memory will reflect a strict prefix of the STORE instructions issued by the terminated threads. By definition of non-blocking algorithm, the termination of the program's threads by the crash cannot prevent the recovery observer from making correct progress based on its view of memory, regardless of what the recovery observer intends to do. In particular, the recovery observer may traverse application data in the persistent heap by starting at the heap's root pointer; again by the definition of non-blocking algorithm, the recovery observer will never thereby encounter corrupt or inconsistent application data. Identical reasoning applies to any number of recovery observers, which collectively could resume correct execution from the consistent state of application data that they find in the persistent heap.

The main advantage of the approach outlined above is that it requires relatively little additional effort for the class of software to which it applies. Unlike whole-system persistence (WSP) [14], our technique does not simply resume thread execution where a crash suspended it—which would be fine for power outages but which isn't the right remedy for crashes induced by software bugs. Instead, we require application code to resume execution from a consistent state of the persistent heap. However our technique is potentially applicable to a broader range of failures, including not only the power outages handled by WSP but also software failures including kernel panics and process crashes, so long as the failures do not corrupt the persistent heap. One restriction of the approach outlined above is the requirement that applications manipulate data in persistent heaps exclusively via non-blocking algorithms. Such algorithms may offer excellent performance, but they are less general, more complex, and less widely used than alternative approaches. We now consider how TSP enables efficient support for consistent recoverability in a much wider class of software.

4.2 Mutex-Based Software

Atlas is a system that employs compile-time analysis and instrumentation, run-time logging, and sophisticated recovery-time analysis to imbue conventional mutex-based multithreaded software with crash resilience [2,3]. Atlas operates upon multi-threaded programs that correctly employ mutexes to prevent concurrency bugs and ensure appropriate inter-thread isolation but that take no measures whatsoever to ensure consistent recovery from durable media. Atlas is nearly transparent, requiring minimal changes to target programs: durable data must reside in a persistent heap

and all active data structures in the persistent heap must be reachable from the persistent heap's root pointer. Atlas guarantees that recovery will restore the persistent heap to a consistent state and that crashes cannot corrupt the integrity of data within it. We explain how TSP improves performance during failure-free operation after briefly reviewing the workings of Atlas; previous publications supply the details [2, 3].

Atlas leverages the fact that shared heap data may be modified within critical sections protected by mutexes and assumes that each outermost critical section (OCS) in the target program both finds and leaves the heap in a consistent state according to application-level integrity criteria. Therefore each OCS represents a bundle of changes to the persistent heap that should be applied failure-atomically. Atlas instruments target programs with logging mechanisms to ensure that an OCS interrupted by a crash can be rolled back during recovery. Furthermore, subtle interactions among OCSes can produce situations where OCSes that completed *prior* to a crash must nonetheless be rolled back upon recovery (see Section 2.3 of [2]); Atlas recovery code correctly handles such situations. Finally, it is possible for crashes to cause Atlas-fortified software to leak memory; Atlas recently incorporated a recovery-time garbage collector to reclaim leaked memory.

Compared with the approach to consistent recovery of programs that employ non-blocking algorithms described in Section 4.1, Atlas offers several advantages: Atlas operates upon more general classes of software that employ familiar isolation mechanisms, as opposed to more restricted and much more esoteric non-blocking algorithms. Furthermore, because Atlas rolls back critical sections interrupted by crashes, it can tolerate failures that cause data corruption within such critical sections; thus Atlas-fortified software is robust against a wider range of failures.

Timely Sufficient Persistence brings substantial performance benefits to Atlas-fortified software. Atlas employs UNDO logging at run time to retain the ability to roll back OCSes during recovery: Before allowing a STORE instruction in the target program to alter a persistent heap location for the first time in an OCS, Atlas first adds an entry to its UNDO log. If TSP is not available, Atlas must *synchronously* flush the UNDO log entry from the CPU cache into memory before allowing the STORE to occur. This synchronous flushing adds considerable overhead beyond the unavoidable Atlas overhead of logging. However if TSP is available, synchronously flushing CPU caches is no longer necessary because TSP guarantees that recovery will read the most recent state of all persistent memory locations, regardless of what tolerated failure has occurred. The details of how TSP delivers on this guarantee will of course depend on the details of *how* TSP tolerates failures (Section 3).

5. EXPERIMENTS

We performed fault-injection experiments to confirm that both of the approaches described in Section 4 do indeed ensure consistent recovery of persistent heap data. We also measured the overhead of the logging required by Atlas (Section 4.2) and of the failure-free cache flushing that Atlas would require if TSP were not available. Previously published experiments applying Atlas to real applications (OpenLDAP and `memcached`) and benchmarks (Splash2) have shown a 3× performance overhead of logging alone and 5× overhead when both logging and synchronous flushing are enabled [3]. The more recent results in Section 5.2 below extend and confirm our earlier findings.

5.1 Map Interface & Implementations

Our experiments employ two different multi-threaded implementations of the familiar "map" interface, i.e., a local key-value store that in the present case maps integer keys to integer values. We divide the key space into a small lower range L used for integrity checks and the remaining much larger higher range H. Each thread $t \in [1 \dots T]$ maintains in the map two private counters indexed with keys $c_{1,t}$ and $c_{2,t}$ in L. Iteration i of the main loop of each worker thread performs three steps as atomic and isolated operations: it first sets the value associated with $c_{1,t}$ to i, then increments the value associated with a key drawn with uniform probability from H, then sets the value associated with $c_{2,t}$ to i. The correctness invariants of the map are the following two inequalities:

$$\sum_{t=1}^{T} c_{1,t} - \sum_{t=1}^{T} c_{2,t} \quad \leq \quad T \tag{1}$$

$$\sum_{t=1}^{T} c_{1,t} \quad \geq \quad \sum_{\text{key}\, k \in H} \text{map}[k].\text{value} \quad \geq \quad \sum_{t=1}^{T} c_{2,t} \tag{2}$$

Our non-blocking map implementation is based on a lock-free skip list by Herlihy & Shavit [11]. We employ a mature and stable C implementation by Dybnis that is believed to be bug-free [7]. We wrote our own mutex-based map implementation in C. It employs a separate-chaining hash table and moderate-grain locking (one mutex per 1000 buckets).

Our fault-injection methodology mimics the effects of a sudden process crash caused by an application software error, e.g., a segmentation violation, illegal instruction, or integer divide-by-zero. We abruptly and simultaneously terminate all threads in a running process by sending the process a `SIGKILL` signal, which cannot be caught or ignored. Recovery code then attempts to locate the map in the persistent heap by starting from the heap's root pointer, traverse the contents of the map, and verify the integrity of the map by testing the invariants of Equations 1 and 2.

5.2 Results

Both our map implementations recovered completely successfully after hundreds of injected process crashes, consistent with previous findings concerning Atlas [3] and with the reasoning in Section 4.1. Similar results would occur under other kinds of non-corrupting failures.

We measured the performance of four variants of our map implementations, where the metric used is "total number of iterations of all worker threads per second" (recall from Section 5.1 that each iteration performs three atomic operations). The throughput of our native unmodified mutex-based code is compared with two Atlas-fortified variants of the same code, one with UNDO logging alone and one with both logging and synchronous CPU cache flushing. We can thus quantify the overhead of logging alone, which is sufficient for consistent recovery if TSP is available, and of synchronous flushing, which is necessary for consistent recovery if TSP is not available. We include the performance of the non-blocking map for completeness, noting that com-

Hardware Platform				Throughput (millions iter/sec)			
	CPU type	hardware		Mutex-Based			
Computer	@ GHz	threads	DRAM	no Atlas	log only	log + flush	Non-Blocking
ENVY Phoenix 800 Desktop	i7-4770 @ 3.4	8	32 GB	3.66	2.36	1.58	2.54
DL580 Gen8 Server	E7-4890v2 @ 2.8	30	1.5 TB	2.13	1.50	1.06	2.00

Table 1: Hardware platforms & experimental results. All computers are HP, all CPUs Intel.

parisons with the mutex-based map are problematic because the two maps employ different data structures (hash table vs. skip list). All performance and fault-injection experiments were conducted on the HP/Intel computers described on the left-hand side of Table 1.

The right-hand side of Table 1 presents our performance results. In all cases we report results for runs with eight worker threads. For the server experiment we pinned all software threads to a single one of the DL580's four CPU sockets; each socket has 15 cores and 30 hardware threads. Running Atlas in "TSP mode" (logging enabled but synchronous flushing disabled) compared with unfortified code reduces throughput by roughly 35% on the desktop and by roughly 30% on the server. This is the price we pay for using Atlas to ensure consistent recovery when TSP is available. When TSP is *not* available Atlas must synchronously flush log entries, and the throughput reduction resulting from Atlas fortification increases to 57% on the desktop and 50% on the server. Comparing the throughput of TSP vs. non-TSP modes of Atlas, we see that TSP increases throughput by 49% on the desktop machine and 42% on the server.

6. CONCLUSIONS

Timely Sufficient Persistence brings substantial benefits when application fault tolerance requirements and available hardware and system software support enable TSP. Our experience with both real applications [3] and small benchmarks (Section 5.2) shows that TSP designs outperform their non-TSP counterparts by wide margins. Remarkably, readily implementable TSP designs for non-blocking algorithms can sometimes completely eliminate runtime overheads while satisfying stringent fault tolerance requirements. Looking forward, we believe that TSP points the way to efficient tradeoffs among runtime overheads, fault tolerance objectives, and hardware and system software support.

7. REFERENCES

[1] G. W. Burr et al. Overview of candidate device technologies for storage-class memory. *IBM J. of Research & Development*, 52(4.5), 2008.

[2] D. R. Chakrabarti and H.-J. Boehm. Durability semantics for lock-based multithreaded programs. In *Hot Topics in Parallelism (HotPar)*, 2013.

[3] D. R. Chakrabarti et al. Atlas: Leveraging locks for non-volatile memory consistency. In *OOPSLA*, 2014.

[4] V. Chidambaram et al. Optimistic crash consistency. In *SOSP*, 2013.

[5] J. Condit et al. Better I/O through byte-addressable, persistent memory. In *SOSP*, 2009.

[6] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the Multics system. In *Fall Joint Computer Conference, Part I*. ACM, 1965.

[7] J. Dybnis. Non-blocking data structures library for x86 and x86-64, Apr. 2009. `https://code.google.com/p/nbds/`.

[8] K. Fraser and T. Harris. Concurrent programming without locks. *ACM TOCS*, 25(2), May 2007.

[9] HBase. `http://hbase.apache.org`.

[10] G. Heiser et al. Rapilog: reducing system complexity through verification. In *EuroSys*, pages 323–336, 2013.

[11] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008. Pp. 339–349.

[12] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *SOSP*, 1997.

[13] C. Mohan et al. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1), 1992.

[14] D. Narayanan and O. Hodson. Whole-system persistence. In *ASPLOS*, 2012.

[15] F. Nawab et al. Procrastination Beats Prevention. Technical Report HPL-2014-70, HP Labs, 2014.

[16] W. T. Ng and P. M. Chen. The systematic improvement of fault tolerance in the Rio file cache. In *IEEE FTCS*, 1999.

[17] I. Oukid et al. Instant recovery for main-memory databases. In *CIDR*, 2015.

[18] S. Park, T. Kelly, and K. Shen. Failure-atomic `msync()`. In *EuroSys*, 2013.

[19] S. Pelley et al. Do query optimizers need to be SSD-aware? In *ADMS@VLDB*, 2011.

[20] S. Pelley et al. Memory persistency. In *ISCA*, 2014.

[21] S. Pelley et al. Storage management in the NVRAM era. In *VLDB*, 2014.

[22] M. Stonebraker et al. The end of an architectural era:(it's time for a complete rewrite). In *VLDB*, 2007.

[23] The Open Group. *Portable Operating System Interface (POSIX) Base Specifications, Issue 7, IEEE Standard 1003.1*. IEEE, 2008. See line 43041 on page 1310 of the PDF version of the standard for the semantics of writes to shared memory mappings.

[24] S. Tu et al. Speedy transactions in multicore in-memory databases. In *SOSP*. ACM, 2013.

[25] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS*, 2011.

[26] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10), 2014.

[27] S. J. White and D. J. DeWitt. Quickstore: A high performance mapped object store. In *VLDB*, 1995.

[28] J. Zhao et al. Kiln: Closing the performance gap between systems with and without persistence support. In *MICRO*, 2013.