PfTouch: Concurrent Page-Fault Handling for Intel Restricted Transactional Memory

Rubén Titos-Gil[1], Ricardo Fernández-Pascual, Alberto Ros, Manuel E. Acacio

Dept. de Ingeniería y Tecnología de Computadores, Universidad de Murcia, Spain

[1]Corresponding author at : Facultad de Informática, Campus de Espinardo, 30100 Murcia, Spain. E-mail address: rtitos@um.es (R. Titos-Gil).

PfTouch: Concurrent Page-Fault Handling for Intel Restricted Transactional Memory

Rubén Titos-Gil[1], Ricardo Fernández-Pascual, Alberto Ros, Manuel E. Acacio

Dept. de Ingeniería y Tecnología de Computadores, Universidad de Murcia, Spain

Abstract

Page faults occurring within transactions jeopardize concurrency in Intel Restricted Transactional Memory (RTM). To make progress in spite of page-fault-induced aborts, the program must resort to the non-speculative fallback path and re-execute the affected transaction. Since the atomicity of a non-speculative transaction is guaranteed by impeding the execution of any other speculative transactions until the former completes, taking the fallback path is particularly harmful for performance. Therefore, such page-fault-induced aborts currently lead to thread serialization during the potentially long period of time taken to resolve them. In this work we propose PfTouch, a simple extension to RTM that allows page-fault handling to be moved out of non-speculative transactional execution in mutual exclusion. Our proposal sidesteps taking the fallback path in these cases and thus avoids its associated performance loss, by triggering page faults in the abort handler while other speculative transactions can run concurrently. PfTouch requires minimal modifications in the Intel RTM specification and keeps the OS unaltered. Through full-system simulation, we show that PfTouch achieves average reductions in execution time of 7.7% (up to 24.4%) for the STAMP benchmarks, closely matching the performance of the more complex suspended transactional mode in the IBM Power ISA.

Keywords:
Parallel programming, Transactional memory, Intel TSX, Page fault, Serialization, Fallback path

[1]Corresponding author at : Facultad de Informática, Campus de Espinardo, 30100 Murcia, Spain. E-mail address: rtitos@um.es (R. Titos-Gil).

1. Introduction

Hardware Transactional Memory (HTM) was proposed long ago to get over the hurdles of traditional lock-based synchronization, namely priority inversion, convoying, and deadlock [1]. Early HTM implementations appeared in Azul [2] and Rock [3] processors. Not until early this decade, however, has Hardware Transactional Memory support made its way into the commercial processor arena, when firstly IBM [4], and later on Intel [5], announced its adoption in the BlueGene/Q and Haswell chips, respectively. Although its inclusion in commercial processors could be seen as a first step forward to help simplify parallel programming, current HTM support is still too rudimentary to fulfill the Transactional Memory promise of bringing parallel programming to a general audience, and some parallel applications require extensive modifications to get benefit from current HTM support [6].

Particularly, current Intel's HTM support, called Intel Restricted Transactional Memory (RTM), is best-effort, i.e, the architecture provides no guarantee that a hardware transaction will ever succeed [7]. Aborts in these HTM systems not only arise from conflicting memory accesses amongst concurrent transactions, but also from lack of buffering resources, interrupts, and page faults. Therefore, a non-speculative *fallback path* in software must be combined with the HTM support to ensure forward progress. To make things even more complicated, Intel RTM does not provide escape actions [8]. Without escape actions, non-transactional instructions cannot be executed within the context of a hardware transaction, thus restricting the ways in which a software fallback path can interact with the HTM [9].

By resorting to the fallback path, a transaction becomes non-speculative (also known as *irrevocable*), and can always complete execution, freeing itself from hardware limits. Although the hardware does not prohibit the concurrent execution of speculative and irrevocable transactions, the typical fallback path design recommended by the RTM specification [7] uses a single global lock (*fallback lock*) to enforce mutual exclusion between irrevocable and speculative transactions. In this way, the TM runtime ensures that concurrent speculative transactions do not access the same data as the irrevocable one in order to guarantee atomicity. While other more complex schemes are possible, such as using multiple locks on the fallback path, they complicate the programming model as the programmer/compiler must determine under which circumstances allowing concurrency with an irrevocable transactions does not threaten its atomicity.

Regardless of the specific implementation of the fallback path, irrevocability often entails thread serialization and missed opportunities for parallel execution: All concurrent transactions whose read-write set potentially overlaps with that of the irrevocable transaction must wait. Thus, performance suffers if transactions frequently resort to irrevocability. We have found out that for an Intel RTM-like HTM running STAMP [10], 25.1% on average of the total executed cycles are due to serialization caused by irrevocable transactions (Section 5 details our evaluation methodology). Deeper inspection of the results revealed that among the different causes for switching to irrevocability, page faults occurring within the boundaries of a transaction represent a major contributor. In STAMP, page faults are the cause for 30.9% of the waiting cycles associated to the fallback path, on average. The reason for this is how page faults are managed in RTM (via irrevocability), coupled with the coarse-grained synchronization style that characterises the STAMP benchmarks. On the contrary, in-transaction page faults are rare in parallel programs that employ fine-grained synchronization, such as many programs from HTMBench [11] in which fine-grained locks were replaced by transactions. From these results, we observe that without adequate hardware support, page faults can be an important performance bottleneck in those programs that have been written from scratch keeping the TM paradigm in mind (its promise of making parallel programming easier through more coarse-grained transactions[12]). Other authors have also found page faults to be a limiting factor [13][14][15].

According to the Intel *Transactional Synchronization Extensions* (TSX) specification for the RTM interface, every time a page fault occurs within a transaction, it is suppressed as if the faulting memory access had never occurred. The transaction aborts and the hardware informs the abort handler that the transaction may not succeed on retry. The abort handler then determines that it must take the fallback path, and proceeds to abort all other concurrent speculative transactions — ensuring that no new transactions can begin. Subsequently, the irrevocable transaction re-executes everything up to the point where the faulting memory access was found. This time, the transaction is not speculative and thus triggers the page fault: once handled by the operating system (OS), the transaction resumes its execution.

Handling page faults in this simple manner not only avoids the need for non-trivial hardware support to pause/resume speculative transactions while the kernel executes, but also makes HTM support completely transparent to the OS. The downside of this approach is that it can have a significant impact on performance [16]: page faults occurring within transactions preclude concurrency, as no transaction but the irrevocable one can make progress. Since such OS events may take thousands to millions of cycles to be resolved (for instance, when contents must be loaded from disk), performance may drop dramatically as no other transaction can execute in the meantime.

In this work we propose PfTouch, a simple technique that allows page-fault handling in RTM out of the transactional execution in mutual exclusion. Our proposal moves page-fault triggering to the abort handler, sidestepping the switch to irrevocability and therefore eliminating its associated performance loss. In particular, the hardware informs the abort handler that the transaction has aborted because of a page fault and provides the address that triggered the page fault. The abort handler then touches the same address, causing a page fault that will be handled in the usual way by the OS. Other transactions can concurrently do useful work while the OS handles the page fault. After it has been resolved, the transaction restarts speculative execution.

The modifications required in the Intel RTM specification to support this optimization are minimal: additional information about the abort cause and the causing address must be returned to the abort handler via the RAX register. In this way, the abort handler can detect that the abort was due to a page fault, and use the faulting address reported by the hardware in order to fire the page fault. Note that the OS remains unchanged, oblivious to the HTM support. Also, exposing the faulting address to the abort handler does not open up new side channels, as the same information could be inferred with the existing interface.

Through detailed simulations of a 16-core CMP running a rich set of transactional programs (including the STAMP benchmarks), we show that PfTouch achieves noticeable reductions in execution time when in-transation page faults happen (7.7% on average for the STAMP benchmarks and up to 24.4% in Vacation-h). We compare PfTouch against the suspended transactional mode in the IBM Power ISA [17], and observe that our proposal closely matches the performance of this more complex alternative. Our evaluation also shows that PfTouch improves performance in workloads where page faults have diverse sources, gains which are unattainable with software-only solutions such as heap pre-faulting [13]. We would like to highlight that the simple changes involved in PfTouch would relieve programmers from having to worry about the occurrence of in-transaction page-faults, thus paving the way for efficient support of more coarse-grained transactions. This is, in our view, imperative if the promise of Transactional Memory of simpler parallel programming is to be fulfilled.

The rest of the manuscript is organized as follows. Section 2 presents some background on how page-faults are managed in Intel RTM. Then, Section 3 gives the details of the modifications required to support PfTouch and their implications. Some current alternatives to handle page faults in transactions without the need of aborts are discussed in Section 4. The simulation environment used to evaluate PfTouch is explained in Section 5 and the obtained results are explaind in Section 6. Finally, Section 7 contains the main conclusions of this work and avenues for future work.

2. Background: Page-fault handling in Intel RTM

The typical software implementation of the functions to begin and commit a transaction when using the Intel RTM instructions is shown in Listing 1. The *beginTransaction* function will attempt to execute the transaction speculatively using the hardware support, and fall back to irrevocability (i.e., non-speculative execution with mutual exclusion enforced by a global lock) if necessary. Following TSX recommendations [7], speculative transactions must perform eager subscription on a single global lock [18], hence invariably present in their read set: transactions check the value of the lock (line 6 in Listing 1) immediately after the `xbegin` instruction and only proceed if the lock is free. This way, when a thread determines that it must resort to non-speculative execution, it achieves mutual exclusion by acquiring the lock (line 15), simultaneously meeting the two necessary conditions to maintain atomicity: i) no other transaction can execute non-speculatively (the fallback lock will be locked for as long as another non-speculative transaction executes); and ii) the write to the lock variable causes the immediate abort of all other speculative transactions due to a transactional conflict on a block in their read set. Race conditions during lock acquisition with newly started speculative transactions which had not yet subscribed to the lock, are resolved by explicitly aborting when the fallback lock is found acquired (line 8). Note that atomicity would be at risk if speculative transactions are allowed to run concurrently with a non-speculative transaction, as the latter would not be able to prevent conflicting accesses made by the former. To avoid the *lemming effect*, threads wait for the fallback lock to be unlocked (line 11) after a speculative transaction has been aborted, before they can retry the transaction.

Listing 1: Recommended software implementation of wrappers to begin/commit a transaction on Intel RTM.

```
 1 void beginTransaction() {
 2   int ret, nretries = 0;
 3   do {
 4     ret = _xbegin();
 5     if (transactionHasStarted(ret)) {
 6       if (!isLocked(fallbackLock))
 7          return; // Execute speculatively
 8       else _xabort();
 9     }
10     // Abort handler starts here
11     while (isLocked(fallbackLock)) idle();
12     ++nretries;
13   } while (maySucceedOnRetry(ret) &&
14              !tooManyRetries(nretries)));
15   acquireLock(fallbackLock);
16   // Execute non-speculatively
17 }
18 void commitTransaction() {
19   if (isLocked(fallbackLock))
20     releaseLock(fallbackLock);
21   else
22     _xend();
23 }
```

The current RTM specification reports the cause of the abort through a set of flags in the 6 least significant bits of the EAX register (*ret* in Listing 1). Bit 0 is set for explicit aborts, i.e., those caused by the execution of the `_xabort` instruction. Of particular interest is bit 1: when set, the transaction may succeed on retry (*maySucceedOnRetry* in line 13 of Listing 1 will be true). Bit 1 unset means that the fallback path must be taken to make progress (this is the case of a page fault). Bit 2 indicates that the abort was caused by a conflict with another thread. Bit 3 warns about the transaction having overflowed hardware resources. Bit 4 is set in case of a debug breakpoint has been encountered, while bit 5 reports that the transaction was aborted within a nested transaction.

When a page fault occurs within a transaction, the hardware simply suppresses it and execution continues at the abort handler (line 10) with bit 1 in the EAX register unset (*maySucceedOnRetry* in line 13 will be false). Note that retrying speculatively is very likely to trigger again the same page fault and thus another abort. Therefore, the thread must acquire the fallback lock (line 15) and re-execute the transaction non-speculatively in mutual exclusion. If the page fault is triggered again, it will be handled as usual by the OS while the affected thread is holding the fallback lock. Although managing page faults in this way simplifies the design of both the hardware and the OS, it could entail serious performance degradation for benchmarks with long-running transactions. This is not only due to the large amount of work discarded as a result of the conflict on the fallback lock, but most importantly because of the long time that other threads must block on the lock while page faults are being handled and then until the faulting transactions complete and release the lock (i.e., page faults occurring within transactions cause the serialization of all threads attempting to execute transactions). Figure 1 (left) sketches this situation. Here, transaction T0 suffers a page fault while transaction T1 is executing. The acquisition of the fallback lock by T0 implies: i) aborting T1 as result of a conflict on the cache line that contains the lock, since T1 is amongst its sharers after lock subscription (line 6 in Listing 1); and ii) preventing any speculative transactions (T2) from executing, since transactions explicitly abort when the fallback lock is found held during lock subscription (line 8 in Listing 1). Then, T0 restarts execution from the beginning in the irrevocable mode, while T1 and T2 keep waiting until the fallback lock is released (thus precluding concurrency for all the time taken to resolve the page fault). Once T0 finishes, T1 and T2 can restart executing again.

3. PfTouch: Enabling concurrency in the presence of page faults for Intel RTM

To reduce the high cost of taking the fallback path to resolve page faults inside transactions, we propose in this work *PfTouch*, a straight-forward extension to RTM that enables handling of such page faults in the abort handler: by providing it with information indicating that the transaction has aborted due to a page fault, including the address that caused it, the abort handler only needs to touch the reported address and let the OS handle it in the usual way. After the page fault is resolved, the transaction can be retried again speculatively. With a very high probability, the transactional code will access again the same address that previously caused the page fault, but this time the access will not fault. This way the transaction does not need to be-

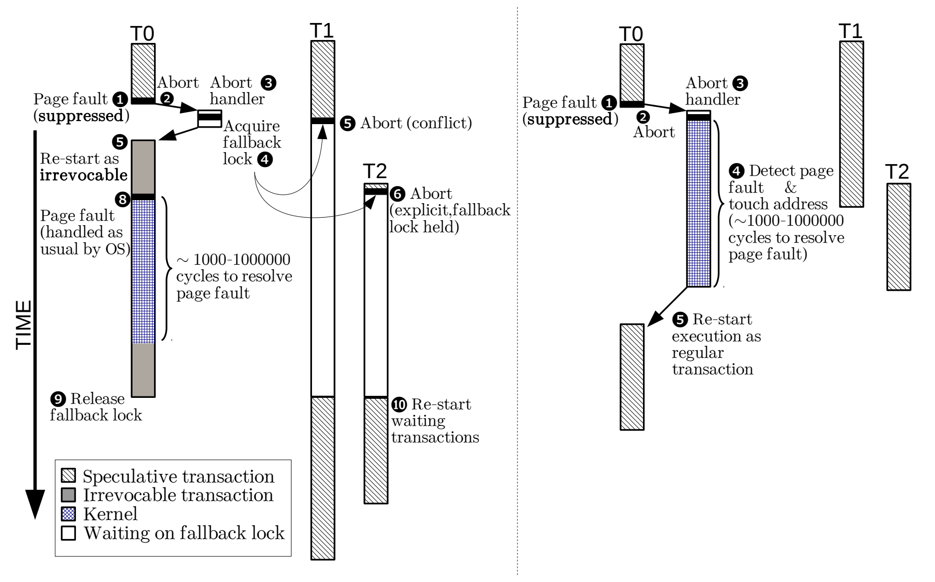
T0
T1
T2
Page fault 1 (suppressed)
Abort 2
Abort handler 3
Acquire fallback lock 4
5 Abort (conflict)
5 Re-start as irrevocable
6 Abort (explicit,fallback lock held)
8 Page fault (handled as usual by OS)
~ 1000-1000000 cycles to resolve page fault
TIME
9 Release fallback lock
10 Re-start waiting transactions
Speculative transaction
Irrevocable transaction
Kernel
Waiting on fallback lock
4 Detect page fault & touch address (~1000-1000000 cycles to resolve page fault)
5 Re-start execution as regular transaction

Figure 1: Page-fault handling in Intel RTM (left) vs PfTouch (right).

come irrevocable (i.e., the fallback lock is not taken) and other transactions can concurrently do useful work while the OS handles the page fault.

This scheme does not guarantee that the transaction will not be aborted again due to another page fault: though unlikely, the transaction may not access the exact same memory locations when it re-executes, and such newly accessed locations may trigger additional page faults. If a transaction aborts repeatedly due to page faults, it will be necessary to take the fallback lock in the end (as currently done with other kind of aborts), although this should be very infrequent.

3.1. Hardware modifications

The modifications required to support PfTouch are minimal. The current Intel RTM specification returns the abort status via the EAX register (32 bits), broadly indicating the cause of the abort using the 6 less significant bits (e.g., explicit, capacity, or conflict). However, page faults do not have a category on its own. We propose modifying the RTM interface to inform the handler whether the abort was due to a page fault and in that case, the faulting address. We advocate for the use of the 58 most significant bits of RAX (the 32 most significant bits, currently unmodified, and bits 31:6 of EAX, currently reserved in non-explicit aborts). We propose the use of bit 6 as a new status bit, set if the abort was caused by a page fault, in which case bit 7 would indicate whether the faulting access was a read or a write operation. Finally, bits 63:8 would contain the faulting virtual address, excluding its 8 least significant bits.

Listing 2: Recommended implementation of wrappers to begin/commit a transaction on Intel RTM. Changes to leverage PfTouch are highlighted.

```
1  void beginTransaction() {
2    int ret, nretries = 0;
3    do {
4      ret = _xbegin();
5      if (transactionHasStarted(ret)) {
6        if (!isLocked(fallbackLock))
7          return; // Execute speculatively
8        else _xabort();
9      }
10     // Abort handler starts here
11     if (abortedByPageFault(ret)) {
12       assert(!maySucceedOnRetry(ret));
13       touch(getFaultAddr(ret), getFaultType(ret));
14       // Do not exit do..while loop
15       setMaySucceedOnRetryMask(&ret);
16     }
17     while (isLocked(fallbackLock)) idle();
18     ++nretries;
19   } while (maySucceedOnRetry(ret) &&
20            !tooManyRetries(nretries)));
21   acquireLock(fallbackLock);
22   // Execute non-speculatively
23 }
24 void commitTransaction() {
25   if (isLocked(fallbackLock))
26     releaseLock(fallbackLock);
27   else
28     _xend();
29 }
```

3.2. Abort handler modifications

Assuming the ISA extensions described above, the highlighted code in Listing 2 (lines 11 to 16) shows the required changes in the abort handler. After architectural state is restored upon abort, the program counter points at the instruction following `xbegin`, and RAX (*ret*) has been set with the abort status (*transactionHasStarted* will be false). Then, the abort handler checks if the abort was due to a page fault and, if so, touches the faulting memory page extracting the virtual address from *ret* (bits 63:8), as well as the faulting access type (bit 7). If the faulting access was of read type (load or instruction fetch), the memory address is simply read. Otherwise, a compare and swap using identical values —*cas(addr, 0, 0)*— is executed, so that the contents of the page are not modified, but a write operation is performed. Then, it modifies the *ret* value (line 15), enabling bit 1 (*maySucceedOnRetry* in line 19 will be true) so that the transaction is retried speculatively, not through the fallback path. The OS remains totally oblivious to this change: all software modifications are confined to the abort handler. Figure 1 (right) illustrates how with PfTouch the page fault occurring in transaction T0 does not affect the execution of transactions T1 and T2, which can continue to execute while the OS resolves it.

3.3. Security implications

While it has been shown how the Intel RTM HTM support can be leveraged in order to improve the bandwidth of side channel attacks such as Meltdown [19], it has also been shown that this HTM support can be used for just the opposite, to help prevent them [20] [21].

In this regard, the proposed change to the RTM interface does not open up new side channels [22], as the same information can be inferred with the existing interface. Relying on the current RTM interface, a userspace process can already determine whether a particular virtual memory page is mapped in memory or not, without mapping it. This can be done by using a custom abort handler, and accessing the memory page from a single thread inside a small transaction. With conflict- and capacity-induced aborts ruled out, and ensuring that the transaction does not execute RTM-unfriendly instructions, the only remaining abort reasons are interrupts and page faults, which can be distiguished by looking at bit 1 in the returned abort status (XABORT RETRY).

4. Current alternatives

Current proposals for handling page faults in transactions without aborting them rely in much more complicated hardware and OS support. Fully virtualized HTM systems found in the literature like LogTM-SE [23] provide *escape actions* [8] allowing the transactional code to *escape* to selected points of run-time libraries, virtual machines, or the OS. Code in a escape action bypasses transaction version management and conflict detection. Escape actions can be invoked explicitly via new instructions or implicitly as part of a trap and, this way, can be used to allow the OS to handle page faults without aborting a transaction, in addition to system calls and other interrupts. A similar mechanism has been implemented in the best-effort HTM support in IBM's Power ISA [17] with the name of *suspended transactional mode*. This mode is entered as the result of an interrupt (like a page fault) or an explicit instruction. While in this mode, memory accesses from the thread are not added to its read or write set (allowing the execution of the OS code), but accesses from other threads are still observed to detect conflicts. If the transaction detects a conflict or other cause of failure, the abort is deferred until its execution is resumed (after the OS handles the fault/system call).

The performance degradation of page faults ocurring inside transactions can also be mitigated in certain cases using software-only strategies. When page faults happening within transaction boundaries are caused by heap accesses to newly allocated memory, transaction aborts can be avoided by touching, before speculation begins, the region of memory that will be returned by subsequent calls to *malloc* [13]. As opposed to the above hardware techniques, this software approach —henceforth referred to as *heap pre-faulting*— has limited applicability since it only addresses a specific type of page faults, and is not helpful for other important sources such as memory-mapped I/O [24]. Unlike pre-faulting, hardware-based schemes can avoid a sudden performance drop when the working set exceeds physical memory. Another important shortcoming of pre-faulting besides its limited applicability is that it adds a fixed overhead to every transaction, and thus it may slow down certain programs.

Both hardware alternatives described above allow handling the page-fault without aborting the transaction and without precluding concurrency with other transactions, in contrast to our proposal that aborts the transaction, handles the page fault, and then retries the transaction from the beginning. This means that they can achieve better performance, but at the cost of higher complexity in hardware and/or intrusive OS support. Our evaluation compares PfTouch against IBM suspended transactions as well as heap pre-faulting.

5. Simulation Environment

We have extended the widely used Gem5 simulator [25] with transactional memory support in order to model an Intel RTM-like best-effort solution [7], our baseline design. The full-system simulation mode of Gem5 is employed in order to capture the effects of *RTM-unfriendly* OS events on the performance of transactional workloads. We employ the detailed timing model for the memory subsystem provided by Ruby, combined with the simple single-issue, in-order processor model known as *TimingSimpleCPU*. Gem5 provides functional simulation of the x86-64 ISA and boots an unmodified Gentoo Linux. We perform our experiments on a 16-core tiled CMP system. Memory and network parameters are detailed in Table 1. The results presented in Section 6 are for 16-thread runs in all benchmarks except QuakeTM, whose game server allows at most 8 threads. Results correspond to the parallel part of the applications. We account for the variability of parallel applications by gathering average statistics over 10 randomized runs with a random jitter of up to 1 extra cycle to the DRAM response time, except for the performance analysis using vacation-ws-dr, in which we run 20 simulations for each input size, given the higher variability of execution time when major faults arise.

5.1. HTM Systems evaluated

Table 2 summarizes the HTM systems evaluated in Section 6. Our RTM-like baseline (*Base*) uses *speculatively modified* (SM) bit-annotations in the L1 data cache to track write sets, and a *perfect signature* to track read sets. Thus, it can maintain much larger read-sets than write-sets, following features seen in commercial Intel chips [13], where write-set size cannot exceed L1 size (32 KiB) but read-sets of several MiB are allowed. As existing best-effort HTM implementations do, the L1 data cache supports speculative versioning, and a requester-wins policy is used to resolve conflicts. The MESI coherence protocol was augmented to support speculative versioning as follows:

- Silent invalidation of M-state blocks is supported. On abort, the L1 data cache discards speculative updates through gang-invalidation of SM blocks. Thus, exclusive ownership information at the directory may become outdated. To handle this, a former exclusive L1 cache can send a negative acknowledgment (*nack*) to the L2 directory upon forwarded request from a remote L1 cache, indicating that it no longer has the block, in which case the request is serviced with data from L2.

Table 1: System parameters.

MESI Directory-based CMP	
Core Settings	
Cores	16, single issue in-order, non-memory IPC=1
Memory Settings	
L1 I&D caches	Private, 32KiB, split 8-way, 1-cycle hit latency
L2 cache	Shared, 8 MiB, unified 16-way, 24(tag)+12(data)-cycle latency
Memory	2 GiB, 200-cycle latency
Cache coherence	Directory-based, MESI states
Network Settings	
Topology and Routing	2-D mesh (4×4), X-Y
Flit size	16 bytes
Message size	5 flits (data), 1 flit (control)
Link latency	1 cycle
Link bandwidth	1 flit per cycle

- Speculative writes to M-state blocks whose SM bit is unset must write back L1 data to the L2 cache before the store completes in L1. Thus, the L1 cache controller must be able to read the value of the SM bit before deciding on the actions to perform. SM bits are typically stored alongside coherence state bits, so this can be done at no extra cost. Like regular replacements, these writebacks that retain exclusive ownership are not in the critical of the originating write.

Our baseline model uses the abort handler implementation shown in Listing 1. We implement our scheme of concurrent handling of page-faults (*PfTouch*) by modifying the abort handler as depicted in Listing 2, while the underlying HTM system remains identical to Base except for the additional information about page-fault-induced aborts provided on the abort status returned in the RAX register.

We then compare PfTouch to the three related works discussed in Section 4. Our baseline best-effort requester-wins HTM is augmented to support suspended transactions (*Suspended*), which uses the same abort handler as

Table 2: HTM systems evaluated.

Base	Baseline, perfect read signature, SM-bits in L1 cache
PfTouch	Concurrent page-fault handling in abort handler
Suspended	Suspended transactions [17]
Prefault	Heap pre-faulting [13]
LogTM	LogTM-SE, perfect signatures, 8-entry log filter [23]

Base. The hardware model is modified so that a trap to kernel code due to interrupts or page faults does not abort an ongoing transaction. Instead, the kernel code executes non-transactionally (i.e. stores are not versioned) while the hardware keeps monitoring both coherence traffic and evictions of speculatively modified blocks from the local L1 cache. If a conflict- or capacity-induced abort signal occurs while the transaction is suspended, its processing is deferred until the CPU resumes the transaction upon return from kernel code. If an abort was signalled while the transaction was suspended, its processing begins as soon as the CPU is back executing user code. Otherwise, transactional execution is resumed successfully after the interrupt/page-fault was handled.

On its part, the heap pre-faulting scheme (*Prefault*) is implemented entirely in software, by extending the thread-local memory allocator in the STAMP library with a new function (`memory_touch`), which writes the memory location that would be returned by a subsequent call to `memory_get` (the replacement of `malloc` used inside transactions in STAMP benchmarks). If such location lies less than 128 bytes from the page boundary, the first byte of the following page is also touched. Then, a call to the `memory_touch` function is added to the `beginTransaction` wrapper in Listing 1, between lines 2 and 3.

Additionally, we compare the relative performance of the aforementioned best-effort designs, against LogTM-SE[23] (*LogTM*), a popular HTM system that provides *virtualized* transactions of any footprint or duration. LogTM-SE uses a requester-stalls policy to resolve conflicts through a timestamp-based scheme of conservative deadlock avoidance that also prevents the *starving writer* pathology [26]. *Perfect* read and write signatures are employed for book-keeping. A software abort handler walks the undo log to restore memory. To make a fair comparison against the aforementioned best-effort HTMs, no randomized exponential backoff is performed after abort in LogTM. Re-

Table 3: STAMP benchmarks and inputs.

STAMP	
bayes	-v32 -r4096 -n2 -p20 -i2 -e2
genome	-g512 -s32 -n32768
intruder	-a10 -l16 -n4096 -s1
kmeans-l	-m40 -n40 -t0.05 -i random-n16384-d24-c16
kmeans-h	-m15 -n15 -t0.05 -i random-n16384-d24-c16
labyrinth	-i random-x48-y48-z3-n64.txt
ssca2	-s14 -i1.0 -u1.0 -l9 -p9
vacation-l	-n2 -q90 -u98 -r1048576 -t4096
vacation-h	-n4 -q60 -u90 -r1048576 -t4096
yada	-a10 -i ttimeu10000.2
Modified STAMP	
intruder-v-mmap	-a10 -s1 -f20 -i packetstream-l128-n4096
ssca2-tx	-s14 -i1.0 -u1.0 -l9 -p9
vacation-ws-dr	-n2 -q10 -u100 -r4220000 -t131072

dundant writes to the log are minimized by means of a *log filter* that contains the eight most recently logged block addresses. To accommodate syscalls ocurring inside transactions in certain HTMBench benchmarks, we augment our LogTM model with the ability to run a transaction non-speculatively in mutual exclusion by means of a fallback lock which only gets acquired upon syscall-induced aborts, whose acquisition aborts all other speculative transactions and forces them to wait, similar to how irrevocability is implemented in the best-effort HTMs considered.

5.2. Benchmarks

The STAMP transactional benchmarks [10] as well as a selection of benchmarks from HTMBench[11] are used as workloads. For STAMP, the recommended medium-sized inputs shown in Table 3 are used, while in Table 4 we can see the selection of HTMBench benchmarks considered and their inputs. Figure 2 shows the scalability up to 16 threads for Base and LogTM, where we can see the poor parallel performance of bayes and labyrinth in all HTM systems. The large write-set size of their transactions in both codes, well above 32 KiB for medium inputs, leads to poor performance when executed in best-effort HTM systems due to capacity-induced aborts, as it can be seen in Figure 2. Moreover, execution time of bayes varies significantly in different

Table 4: HTMBench benchmarks and inputs.

HTMBench	
avl	1000000 1 0.8 100000
berkeleydb	-i 4096
bplus-tree	-items=100, -times=100
dedup-cp	-c -p -f -r 1 -i simsmall.dat
kyotocabinet	(kcforesttest) queue -it 1 -bnum 10000 -psiz 1000 casket 20000
leveldb	-num=4096 -benchmarks=fillseq
linkedlist	-w 30 -u 20 -i 16000 -r 32000
quaketm	50 frames
skiplist	-w 100000 -u 20 -i 16000 -r 32000
ua	Class S

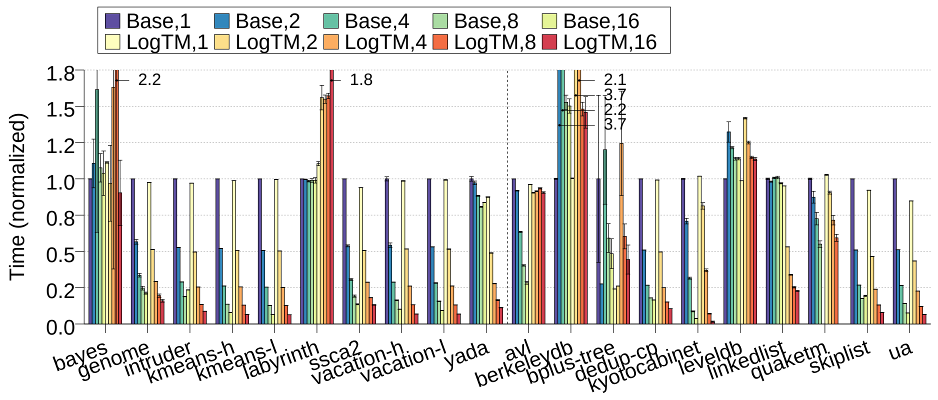

Figure 2: Scalability of baseline and LogTM HTM systems up to 16 threads.

executions, as it implements a hill climbing search algorithm that, depending on the thread interleaving, can execute a different number of transactions for the same input [27, 5], as clearly seen in Figure 2. In labyrinth, its poor scalability is partly due to the lack of hardware support for *early release* [5, 28], which prevents its main data structure from being removed from the read set after privatization, so that any thread attempting to commit its work invariably aborts all other concurrent transactions. Morover, the even more pathological performance of LogTM in labyrinth is explained by the large write set size of transactions that repeatedly abort, which not only makes the undo log compete for cache resources with program data, but also causes very expensive software rollbacks. For the aforementioned reasons, results for bayes and labyrinth are not presented in Section 6.

As for the HTMbench benchmarks considered in this work, we had to adapt them in several ways to make them suitable for our simulation infrastructure. For QuakeTM, apart from fixing several important programming bugs that caused data corruption, we transformed it from its original client-server model (where the server spends the majority of its time in the kernel waiting to receive datagrams from a socket), into a stand-alone program where the server obtains clients' commands from a trace file previously recorded by running the original client-server program in native hardware. We also adapted dedup, whose original version creates n threads for each of the three stages of its parallel pipeline, in order to isolate the effects of thread scheduling and context switching overhead when using more threads than available cores. We derived dedup-cp, a version in which the region of interest is the pipeline stage which employs coarse-grained transactions (*chunk process*), which then we run with higher thread counts to better observe the effects of contention and page faults on HTM performance without the interference from the kernel due to thread descheduling and rescheduling. To present scalability numbers similar to those presented for STAMP (Figure 2), we further modified HTMbench benchmarks with additional command line arguments to control their runtime based on work-units (e.g., iterations) to execute rather than based on duration (wall-clock time). Also, where applicable we distributed work among threads, in an attempt to keep the amount of work constant across runs with varying thread counts. Moreover, to reduce spurious aborts due to conflicts among transactions because of the data structures used by the memory allocation library, we replaced calls to libc's `malloc` by calls to the STAMP's thread-local malloc. In this way, concurrent transactions can allocate dynamic memory without experiencing conflict-induced aborts. The patch to produce the modified versions of HTMBench codes is available in [29]. Figure 2 (right) shows that all selected benchmarks from HTMBench scale up to 8/16 threads in LogTM except for berkeleydb and leveldb. In linkedlist, capacity-induced aborts lead to barely no speedup over single thread runs for the best-effort HTM system used as baseline. As for bplus-tree, at least 4 threads are required to create workloads with similar amounts of work executed, and thus numbers shown in Figure 2 for 1- and 2-thread configurations are not directly comparable.

Table 5 shows the number page-fault-induced aborts occurred in *Base* and *PfTouch* during the execution of the benchmarks mentioned in Tables 3 and 4, along with the average number of cycles spent in the *touch* function (line 13 in Listing 2) waiting for the kernel to handle the fault. Page faults have been classified in *minor* (those that do not require reading or writing any data from or to the swap file or partition or the backing file in case of mmaped I/O) and *major* (those that do). We can see that all the page faults in the STAMP benchmarks are minor. In fact, all of them are caused by the first accesses to unused heap pages which are initially zeroed.

To enrich our performance characterization, we consider three additional benchmarks, *intruder-v-mmap*, *vacation-ws-dr* and *ssca2-tx*, which attempt to better measure the applicability of each page-fault-abort mitigation technique. As shown in Table 5, all page faults experienced by transactions in the STAMP benchmarks using medium inputs are minor faults. As shown in Section 6, such faults are exclusively caused by accesses to memory dynamically allocated inside transactions. To this end, intruder-v-mmap and vacation-ws-dr attempt to fill this gap by mimicking workloads where page faults in transactions have other sources.

Intruder-v-mmap [30] is a modified version of intruder that attempts to capture the behavior of real-world workloads that perform memory-mapped I/O during transactions [24]. The changes are explained in detail in [30]. In short, threads obtain network packets from a memory-mapped input file that contains a packet stream, instead of generating it during the initialization phase. This way, page faults that occur during initialization of input data in the original version of intruder are moved to the region of interest when obtaining packets from the memory-mapped file. Furthermore, this optimized version uses a vector instead of a linked list to store packets belonging to the same flow, since flow length (i.e., vector size) is known in advance. Note that this change alone achieves nearly a 3× speedup in single-thread executions over the original benchmark.

Table 5: Number of aborts caused by page faults in *PfTouch*, and average duration (in cycles) of the *touch* in the abort handler.

	Minor page faults		Major page faults	
Benchmark	Count	Average duration	Count	Average duration
genome	55	67510	0	—
intruder	144	62643	0	—
intruder-v-mmap	391	60417	0	—
kmeans-h	0	—	0	—
kmeans-l	0	—	0	—
ssca2	0	—	0	—
ssca2-tx	0	—	0	—
vacation-h	68	65619	0	—
vacation-l	54	67583	0	—
yada	56	65510	0	—
avl	57	60609	0	—
berkeleydb	0	—	0	—
bplus-tree	126	45172	0	—
dedup-cp	2929	51597	4	661028
kyotocabinet	125	73227	0	—
leveldb	7	47303	0	—
linkedlist	0	—	0	—
quaketm	4	54075	0	—
skiplist	1207	70465	0	—
ua	0	—	0	—
vacation-ws-dr-220000	2056	44625	0	7074122
vacation-ws-dr-225000	2147	43355	4	13306006
vacation-ws-dr-230000	2171	43325	6	9791788
vacation-ws-dr-235000	2159	43679	11	16946642
vacation-ws-dr-240000	2137	44107	8	14776571
vacation-ws-dr-245000	2390	40170	15	23105994
vacation-ws-dr-250000	2185	42309	17	26510093
vacation-ws-dr-255000	3325	33654	55	21284268
vacation-ws-dr-260000	2858	36689	38	22006238
vacation-ws-dr-265000	2838	37469	70	32240634

On its part, vacation-ws-dr [31] is used to showcase the behavior of all HTM systems considered under heavy-load scenarios where major page faults arise. With that goal in mind, the input size chosen for this benchmark (-*r* option) is set slightly above four million relations, in order to push its physical memory requirements slightly above the 2 GiB limit of the simulated system. To prevent work imbalance among threads due to the appearance of such page-faults that take millions of cycles to be resolved, the static workload assignment scheme of the original benchmark is replaced with a dynamic scheme as if using work stealing, so that at the end of the parallel phase each thread may have performed a different number of database transactions depending on its particular throughput, while the total amount of work remains constant (as specified by the -*t* option). Furthermore, unlike the original vacation in which the query range (-*q* option) of objects identifiers is the same for all threads, in this version threads query disjoint ranges, so that the sum of all ranges spans all the identifiers present in the database. This change favours progress of threads whose working set remains in physical memory in the presence of major page faults affecting other threads. The patch to produce this flavour of vacation is available in [31].

As we can observe in Table 5, the only benchmark that has major page faults is vacation-ws-dr, which has been designed precisely to exhibit this behavior. In the case of intruder-v-mmap, there are no major page faults either, although we may expect them since it is accessing a file on disk which has been mmaped. We have verified that this happens because the OS reads-ahead the memory-mapped file eagerly into the page cache, as part of the mmap system call or shortly after it. This way, no major faults to read data from disk occur during the parallel phase, but accesses to the mmaped region during this phase still cause minor faults in order to set up the mappings in the page table of the process, pointing to the kernel's page cache.

Ssca2-tx simply zooms into the execution time of the only phase in ssca2 that uses transactions (inspection of adjacency arrays), in order to better characterize the fixed overhead incurred by *Prefault* in benchmarks where small, short-running transactions dominate.

6. Results

In Figures 3 and 6, we compare the relative performance of all HTM systems considered for STAMP and HTMBench, respectively, normalized to our baseline system (*Base*). We show the role that each source of overhead plays

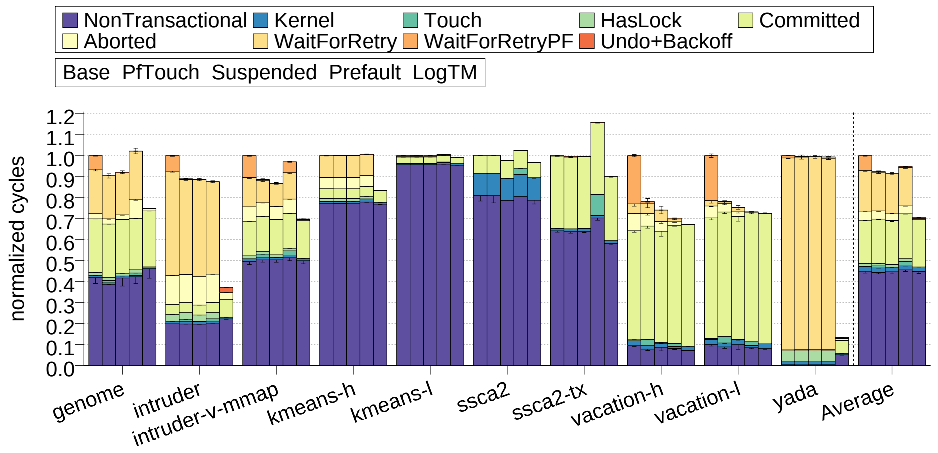

Figure 3: Execution time for STAMP benchmarks, broken down into categories.

on the performance of each configuration by breaking down execution time into disjoint components, where each execution cycle is attributed to one of the following categories: non-transactional (including time spent in barriers, *NonTransactional*); handling of interrupts/page-faults (*Kernel*); holding the fallback lock (*HasLock*); executing speculative transactions (*Committed* and *Aborted*); waiting on the fallback lock to be released before retrying a speculative transaction, either because the lock is being held by a transaction that is handling a page fault (*WaitForRetryPF*) or any other cause (*WaitForRetry*); lastly, *Touch* categorises cycles spent either handling page faults triggered by the abort handler (in *PfTouch*) or touching the heap area (in *Prefault*).

6.1. STAMP

Benchmarks such as intruder and vacation clearly show the performance gains achieved by all three page-fault-abort mitigation techniques considered. In the case of *Suspended* and *PfTouch*, the ability to handle page faults without acquiring the fallback lock enables more concurrency, as seen in the elimination of the WaitForRetryPF time. The rest of the waiting time due to irrevocable transactions that acquire the fallback lock for other causes remains almost the same in all cases. As for vacation, page faults are nearly the only reason why transactions resort to irrevocability in *Base*, and hence virtually all the cycles in HasLock, WaitForRetry and WaitForRetryPF are eliminated. In genome, intruder and yada, the HasLock and WaitForRetry components remain as threads need to break livelock situations caused by repeated conflict-induced aborts, a direct consequence of the conflict resolution policy (requester-wins).

We can see how part of the cycles that were attributed to Kernel in *Base* (page faults while in non-speculative transactions) are moved to the Touch component in *PfTouch*. This is the time spent handling the page faults in both cases and has a larger impact on performance that may be apparent at first sight because in *Base* those kernel cycles are executed while *no other* thread can make useful transactional work (because the fallback lock is taken), whereas in *PfTouch* and *Suspended* those same kernel cycles (Touch or Kernel components) are executed concurrently with other transactions. Furthermore, in *Base*, the page faults that occur in a transaction invariably trigger the irrevocability mechanism which in turn forces the abort of all other concurrent speculative transactions. On the other hand, with *PfTouch* or *Suspended*, the rest of transactions are not affected.

Comparing the performance of *PfTouch* against *Suspended*, we can see that they achieve very similar performance levels in benchmarks like genome, intruder and vacation. As said before, the reason for their advantage is the increased concurrency while handling page faults, but each one achieves this differently: *Suspended* does it by suspending the transaction to handle the page fault and then resuming it without discarding any work, while *PfTouch* aborts the transaction, handles the page fault, and then retries it. These resutls confirm that our slightly augmented RTM interface is able to capture the majority of the benefits of a more complex hardware solution such as suspended transactions.

The performance exhibited by *Prefault* approximates that of *Suspended* in all STAMP benchmarks significantly affected by page-fault-induced aborts, indicating that almost all such aborts are due to dynamic memory allocation. This is the result of STAMP's simplistic thread-local memory allocator, which does not even support freeing memory: since memory never gets reused, page faults in the heap area become recurrent. *Prefault* triggers all those page faults before the transaction and sidesteps such aborts, thus reducing the Aborted component in some cases as much as *Suspended*. This is clearly visible in Figure 4 for vacation, where page-fault-induced aborts occur close to the end of its large transactions, hence resulting in a lot of discarded work for *Base*. On the downside, the inability of *Prefault* to help performance when page-fault-induced aborts are not caused by dynamic memory allocation is visible in intruder-v-mmap, in which both *PfTouch* and *Suspended* show their key advantage over a software-based heuristic such as heap prefaulting: their general applicability. Another key shortcoming of *Prefault*, the extra overhead added to *every* transaction, becomes apparent in ssca2: to make heap pre-faulting transparent to the programmer, it is implemented inside the *beginTransaction* function of Listing 1, by touching the heap before a transaction is speculatively executed for the first time. Depending on the cost of such *blind* heap prefetch, *Prefault* may slow down programs with many small-sized transactions like ssca2, in which we see a slowdown of 14.4% in its transactional phase (4.0% overall). Note that as a result of STAMP's simple memory allocator, the heap touch operation only comprises less than 20 instructions, but more realistic allocators would likely increment its fixed cost.

To further understand the performance of the different ways of handling page faults, we can see their effects in the time spent doing work in a transaction that is finally aborted, which is shown in Figure 4. This would correspond to the cycles included in the *Aborted* category in Figure 3, split in seven finer grained categories: *Conflict* are cycles aborted due to conflicts between transactions; *L1Capacity* and *L2Capacity* are cycles wasted due to aborts caused by the limited capacity of the L1 and L2 caches respectively (capacity aborts); the cycles wasted before an abort is triggered due to a page faults are accounted for as *PageFault*; *Interrupt* are the cycles of transactions aborted due to interrupts; *FallbackLock* are the cycles that are wasted due to aborts caused by the acquisition of the fallback lock. *Syscall* are the cycles that are wasted due to aborts caused by system calls ocurring inside transactions.

Figure 4 shows that in most cases more than half of the cycles due to aborts are caused by conflicts among transactions. The only exceptions are vacation and yada. As observed by looking at Figures 3 and 4, the additional concurrency achieved by side-stepping thread serialization on the fallback lock caused by page faults results in performance gains over *Base* yet causes slightly more discarded work in certain benchmarks in all other best-effort HTM configurations. On its part, a virtualized HTM implementation like *LogTM* generally discards less work than best-effort counterparts as *LogTM* only aborts transactions because of conflicts. The exception is yada, where *LogTM* achieves a dramatic reduction in execution time (almost 90%) at the cost of discarding 5× more transactional work than best-effort HTM systems, simply because the absence of a fallback path lets threads execute speculative work without serializing for any reason other than conflicts (some of them

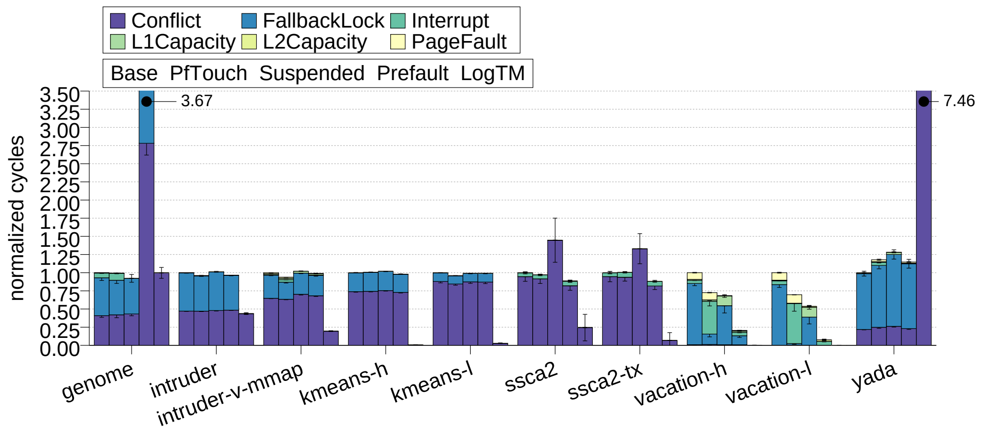

Figure 4: Cycles wasted due to aborts in STAMP benchmarks, categorized by cause of abort.

resolved by stalls in *LogTM*).

In most benchmarks, the page fault component in the aborted cycles is not visible because aborts caused by conflicts (including conflicts on the fallback lock) clearly dominate. It is interesting to see the effects that *Prefault* has in the case of genome, where it eliminates WaitForRetryPF cycles but in turn increases the amount of wasted cycles due to conflicts. The reason for this is that page-fault-induced aborts in *Base* that throttle down concurrency in a highly-contended phase of the benchmark are removed by *Prefault*, giving way to repeated conflict-induced aborts which eventually result in the fallback lock being acquired anyways, explaining the increase in the *Conflict* category shown in Figure4 for this benchmark. In *PfTouch*, this pathological scenario does not arise because contending transactions affected by page faults give way to others while the transaction aborted by the page fault handles it again in the abort handler.

The poor scalability seen in Figure 2 for yada in *Base*, and by extension in all best-effort HTMs evaluated, is mainly due to the contention experienced by its very long running main transaction (*regionRefine*). Single-threaded runs indicate that such transaction takes on average 200,000 cycles to complete in *LogTM*, while in *Base* around 85% of all executions of such transaction are non-speculative (i.e., through the fallback path), with the remaining 15% committed speculative executions taking around 100,000 cycles. With only one thread, mostly page-faults and interrupts, and to a lesser extent capacity-induced aborts, make threads resort to the fallback path. In multithreaded runs, it is the *friendly fire* [26] caused by the requester-wins policy of the evaluated best-effort HTM systems what makes all threads take the fallback path after repeated conflict-induced aborts, explaining why over 75% of all aborted cycles are the result of a conflict on the fallback lock shown in Figure 4.

In vacation, the *PfTouch* optimization removes all aborted cycles caused by the acquisition of the fallback lock when one transaction suffers a page fault. However, the amount of wasted cycles is only reduced by 25% overall, because some of those speculative transactions concurrent with the page fault processing in the abort handler are now aborted by interrupts. The Page-Fault component seen in *Base* remains unchanged in *PfTouch* configuration, and only gets removed by *Prefault*, *Suspended* and *LogTM* configurations, since in these cases either the page fault occurs before the transaction starts (*Prefault*), or there is no need to abort transactions to handle page faults (*Suspended* and *LogTM*). As shown in the figure, the performance advantage of *Prefault* over *PfTouch* comes mostly from the interrupt-induced aborts affecting the latter, and to a lesser extent the ability of the former to sidestep page-fault induced aborts. Further analysis of this matter revealed that the unexpected increase in the number of interrupts seen in *PfTouch* is caused by the `cmpxchg` instruction as simulated in Gem5 and its interaction with the operating system. In the simulated x86 processor model, such instruction is decoded into several micro-instructions, including a load to the given memory location followed by a store to it, whose result is two consecutive page faults generated by the instruction. The load micro-instruction causes a minor page fault to a zeroed virtual page (previously mmapped by the memory allocator to fulfill requests) which the operating system handles mapping it to a shared zeroed physical page which backs many virtual pages from the same or different processes. Later, the store micro-instruction causes another minor page-fault because the operating system has to allocate a new physical page for the virtual address before actually modifying it (this copy-on-write optimization is crucial to allow sharing of static code and data between processes and to allow the operating system to overcommit memory). The page table entry and the TLB entry allocated by the first page-fault have read-only permissions to allow the operating system to detect the second fault caused by the write. Unfortunately, since the TLB translation allocated by the load micro-instruction needs to be updated before executing the write micro-instruction, the operating system must ensure that the TLB translation is not present in any other TLB, and thus it must interrupt all other

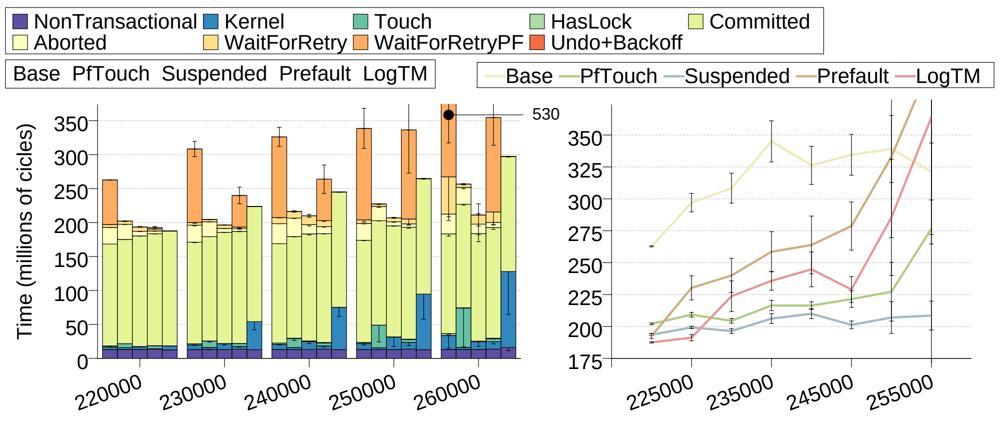

Figure 5: Execution time of vacation-ws-dr when running into physical memory limits.

cores to invalidate the appropriate TLB entry (even though it is very unlikely in this case that this mapping is actually present in any other TLB). These bursts of interrupts affecting all cores are the cause of interrupt-induced aborts seen in Figure 4 for *PfTouch*. *Prefault* does not show this component because it uses a store instruction instead of a compare-and-swap to touch the heap area: writing any value (in our case, zero) when touching the next page from the free pool is only possible because in *Prefault* it is known that the memory location touched does not contain valid data. On the contrary, *PfTouch* is a general mechanism that is capable of dealing with page-faults that occur in any memory region (including, but not limited to the heap), and thus cannot modify its contents. Note that the extra penalty of a double fault incurred by *PfTouch* could be avoided if the first access performed by `cmpxchg` was handled as a store.

On their part, both *Suspended* and *LogTM* eliminate aborted cycles due to interrupts and page faults, achieving nearly identical performance in vacation-l, while the marginal gains of *LogTM* over *Suspended* in vacation-h are attributed exclusively to its ability to resolve conflicts through stalls rather than aborts. Yada, and in less extent, genome, also show a noticeable fraction of the aborted cycles caused by interrupts when *PfTouch* is employed, for the aforementioned reason.

6.2. Major page faults

Figure 5 compares the performance of *Base*, *PfTouch*, *Suspended*, *Prefault* and *LogTM* using vacation-ws-dr. The benchmark is executed with increasing input sizes (see Table 3), starting with a value of *-r* (number of relations) which creates a database whose memory size is slightly below the physical memory available in the simulated system (see Table 1). The goal is to show HTM performance under workloads that may experience major page faults. Real-world applications can approach and surpass the physical memory available to the process, either due to large working set sizes, or because of resource contention amongst several processes in multi-programmed environments. Figure 5 shows that *Prefault* performs slightly better than *PfTouch* in the absence of major faults, for the reasons explained for the original vacation, related to the behavior of compare-and-swap. *LogTM* and *Suspended* perform best as they avoid discarding work when a page fault occurs. However, as soon as the working set of the workload begins exceeding the available physical memory (in vacation-ws-dr that happens at $4,225$ million database relations), major faults begin to arise. At this point, *PfTouch* and *Suspended* becomes the only schemes that can tolerate them without sustaining a severe slowdown: in Figure 5 we can see how, when increasing input size from $4,225$ to $4,245$ million relations, *PfTouch* gracefully handles the appearance of major faults with barely no performance loss, while *Base* and *Prefault* begin experiencing significant performance drops, more acute as more and more major faults appear. It is interesting to see that from $4,225$ million relations, *PfTouch* and *Suspended* outperform *LogTM* because of the extra memory requirements of the undo log that leading to additional major page faults suffered, for the same input size. *Prefault* is unable to trigger the fault ahead of the affected transaction, and thus cannot prevent thread serialization on the fallback lock in such scenarios where the page fault is not caused by accesses to newly allocated dynamic memory. On the contrary, *PfTouch* sidesteps thread serialization and allows the remaining threads to continue executing while the affected thread remains halted until the requested page is brought back from the swap area in disk.

6.3. HTMbench

Since most of these benchmarks contain fine-grained transactions that are the result of replacing locks by transactions, the amount of data speculatively accessed is typically small. As a result, most of the benchmarks suffer few or no page faults, as shown in Table 5. Only dedup-cp and skiplist show some potential for performance improvement by a page-fault-mitigation technique. As expected, execution time and aborted cycles shown in Figures 6 and 7 are similar in *Base*, *PfTouch* and *Suspended* in most of the benchmarks. Only dedup-cp, which consists of coarse-grained transactions that dynamically allocate and operate on memory chunks of hundreds to thousands of bytes in size, shows differences among the considered HTM systems: *Prefault* cannot perform at par with *PfTouch* and *Suspended* since the prefaulting window is not adequately sized (the heuristic is tuned for the STAMP benchmarks and only touches memory up to 128 ahead of the next available location). However, *Prefault* outperforms *Base* and *PfTouch* in skiplist thanks to a 25% reduction in the cycles wasted due to conflict-induced aborts, which in turn is a consequence of the prefetching effect of touching the next available heap before beginning the transaction: by moving cache misses ahead of the transaction, its duration is reduced and this in turn shrinks its window of exposure to aborts by a concurrent conflicting transactions. Although this positive effect on contention also occurs in berkeleydb and ua, it does not translate into performance improvement for several reasons: unlike skiplist, in both benchmarks transactions only account for a small fraction of all executed cycles. Moreover, *Prefault* suffers a slowdown in ua for the very same reason as explained for ssca2 in the previous section (many small-sized transactions in which overhead of touching the heap every time becomes notable, as shown by the Touch component). Finally, the varying performance seen for *LogTM* is worth mentioning: in avl, the moderate contention setup (80% tree lookups, 20% tree insertions) causes frequent aborts, which in LogTM are expensive since the log unroll is done in software. Furthermore, the conflict resolution policy of LogTM (requester-stalls) causes the *futile stall* pathology [26]. On the other hand, LogTM achieves very important improvements in benchmarks like kyotocabinet and linkedlist, in both cases by sidestepping the fallback to irrevocability of best-effort HTMs upon repeated conflict-induced aborts (kyotocabinet) or capacity-aborts (linkedlist).

7. Conclusions

In this work, we address one of the limitations of the Intel TSX specification: how page faults occurring inside transactions has to be managed. According to it, fault occurring within the boundaries of a transaction must be suppressed as if the faulting memory access had never occurred. The transaction aborts and the hardware informs the abort handler that the transaction may not succeed on retry. The abort handler then determines that it must take the fallback path, and proceeds to abort all other concurrent speculative transactions — ensuring that no new transactions can begin. Subsequently,

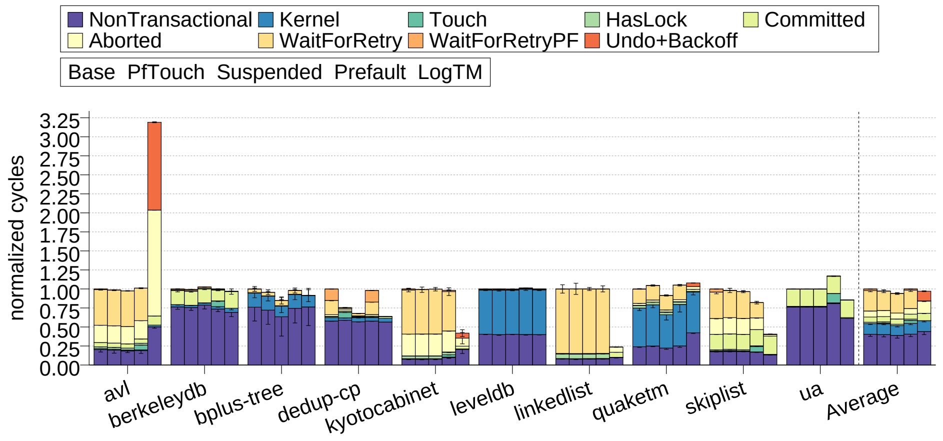

Figure 6: Execution time for HTMBench benchmarks, broken down into categories.

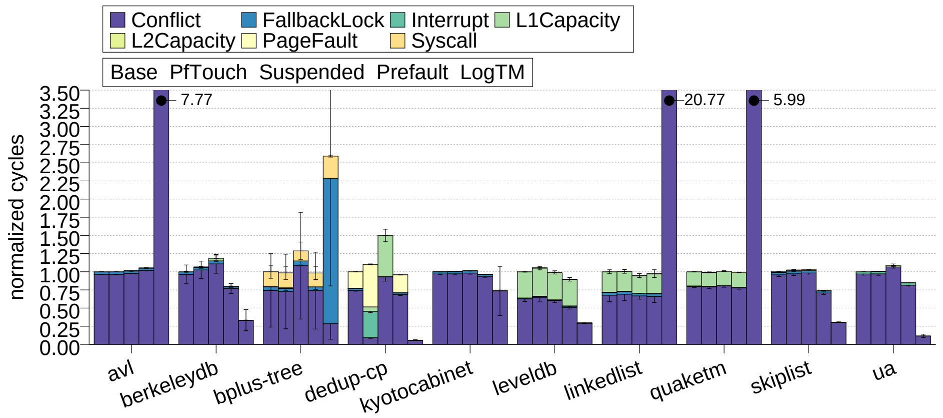

Figure 7: Cycles wasted due to aborts in HTMBench benchmarks, categorized by cause of abort.

the irrevocable transaction re-executes everything up to the point where the faulting memory access was found. This time, the transaction is not speculative and thus triggers the page fault: once handled by the operating system (OS), the transaction resumes its execution.

Although handling page faults in this simple manner avoids the need for non-trivial hardware support to pause/resume speculative transactions while the kernel executes, and also makes HTM support completely transparent to the OS, it can have significant impact on performance as page faults occurring within transactions preclude concurrency during such high-latency OS events. This way the programmer must by all means avoid page faults within transactions, something that struggles with the HTM premise of helping simplify parallel programming or that cannot always be guaranteed in some cases (i.e., programs that perform memory-mapped I/O during transactions [24] or simply whose working set exceeds physical memory).

To overcome this, we propose PfTouch, a very simple extension of the Intel RTM specification that enables page faults within transactions to be handled by the operating system concurrently with the execution of other speculative transactions. Particulary, with PfTouch, page faults informs the abort handler about the reason for the abort (a page fault) and the address causing it through the hitherto unused bits of the RAX register. In this way, the abort handler can detect that the abort was due to a page fault, and use the faulting address reported by the hardware in order to fire the page fault. The OS remains unchanged, oblivious to the HTM support. Also, exposing the faulting address to the abort handler does not open up new side channels, as the same information could be inferred with the existing interface.

Through detail simulations of a 16-core CMP architecture, we demonstrate that despite its simplicity, PfTouch achieves significant performance gains (average reductions in execution time of 7.7%) thanks to circumventing irrevocability, and thus serialization, for these costly events.

As part of our ongoing work, we are developing several additional optimizations that require minimal hardware changes and no intrusion in the operating system, with the goal of bridging the performance gap between best-effort HTM systems like Intel RTM, and more sophisticated implementations appeared in the literature so far, such as LogTM-SE [23].

Acknowledgments

This work has been supported by the Spanish MCIU and AEI, as well as European Commission FEDER funds, under grant "RTI2018-098156-B-C53".

[1] M. Herlihy, J. E. B. Moss, Transactional memory: Architectural support for lock-free data structures, in: 20st Int'l Symp. on Computer Architecture (ISCA), 1993, pp. 289–300.

[2] C. Click, Azul's experiences with hardware transactional memory, in: 2009 Transactional Memory Workshop, 2009.

[3] D. Dice, Y. Lev, M. Moir, D. Nussbaum, Early experience with a commercial hardware transactional memory implementation, in: 14th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS), 2009, pp. 157–168.

[4] C. Jacobi, T. Slegel, D. Greiner, Transactional memory architecture and implementation for IBM System z, in: 45th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO), 2012, pp. 25–36.

[5] R. Yoo, C. Hughes, K. Lai, R. Rajwar, Performance evaluation of intel transactional synchronization extensions for high performance computing, in: ACM/IEEE Conf. on Supercomputing (SC), 2013.

[6] R. Quislant, E. Gutierrez, E. L. Zapata, O. Plata, Privatizing transactions for lees algorithm in commercial hardware transactional memory, Journal of Supercomputing 74 (2018) 1676 – 1694.

[7] Intel Corporation, Intel 64 and IA-32 architectures optimization reference manual, chapter 15: Intel TSX recommendations (2019).

[8] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, D. A. Wood, Supporting nested transactional memory in LogTM, in: 12th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS), 2006, pp. 359–370.

[9] I. Calciu, J. Gottschlich, T. Shpeisman, M. Herlihy, G. Pokam, Invyswell: a hybrid transactional memory for haswell's restricted transactional memory, in: 23rd Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT), IEEE, 2014, pp. 187–199.

[10] C. Cao Minh, J. Chung, C. Kozyrakis, K. Olukotun, STAMP: Stanford transactional applications for multi-processing, in: IEEE Intl. Symposium on Workload Characterization, 2008, pp. 35–46.

[11] Q. Wang, P. Su, M. Chabbi, X. Liu, Lightweight hardware transactional memory profiling, in: 24th Int'l Symp. on Principles & Practice of Parallel Programming (PPoPP), 2019, pp. 186–200.

[12] U. Drepper, Parallel programming with transactional memory, Communications of the ACM 52 (2) (2009) 38–43.

[13] B. Goel, R. Titos-Gil, A. Negi, S. A. McKee, P. Stenstrom, Performance and energy analysis of the restricted transactional memory implementation on haswell, in: 28th Int'l Parallel and Distributed Processing Symp. (IPDPS), 2014, pp. 615–624.

[14] Z. Wang, H. Qian, J. Li, H. Chen, Using restricted transactional memory to build a scalable in-memory database, in: Proceedings of the Ninth European Conference on Computer Systems, 2014, pp. 26:1–26:15. `doi:10.1145/2592798.2592815`.

[15] V. Leis, A. Kemper, T. Neumann, Scaling htm-supported database transactions to many cores, IEEE Transactions on Knowledge and Data Engineering 28 (2) (2016) 297–310.

[16] M. M. Pereira, M. Gaudet, J. N. Amaral, G. Araújo, Study of hardware transactional memory characteristics and serialization policies on haswell, Parallel Computing 54 (2016) 46–58.

[17] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, H. Le, Robust architectural support for transactional memory in the power architecture, in: 40th Int'l Symp. on Computer Architecture (ISCA), ACM, 2013, pp. 225–236. `doi:10.1145/2485922.2485942`.

[18] I. Calciu, T. Shpeisman, G. Pokam, M. Herlihy, Improved single global lock fallback for best-effort hardware transactional memory, in: 9th ACM SIGPLAN Workshop on Transactional Computing, 2014.

[19] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, Meltdown, arXiv:1801.01207 (Jan. 2018).

[20] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, M. Costa, Strong and efficient cache side-channel protection using hardware transactional memory, in: 26th USENIX Security Symposium, 2017, pp. 217–232.

[21] S. Chen, F. Liu, Z. Mi, Y. Zhang, R. B. Lee, H. Chen, X. Wang, Leveraging hardware transactional memory for cache side-channel defenses, in: Proc. of the 2018 on Asia Conference on Computer and Communications Security, 2018, pp. 601–608.

[22] P. Kocher, J. Jaffe, B. Jun, Differential power analysis, in: Proceedings of the Advances in Cryptology – CRYPTO99, 1999, pp. 388–397.

[23] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, D. A. Wood, LogTM-SE: Decoupling hardware transactional memory from caches, in: 13th Int'l Symp. on High-Performance Computer Architecture (HPCA), 2007, pp. 261–272.

[24] M. Ghilardi, High performance concurrency in common lisp: Hybrid transactional memory with STMX, in: 7th European Lisp Symposium, 2014, p. 38.

[25] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al., The gem5 simulator, ACM SIGARCH Computer Architecture News 39 (2) (2011) 1–7.

[26] J. Bobba, K. E. Moore, L. Yen, H. Volos, M. D. Hill, M. M. Swift, D. A. Wood, Performance pathologies in hardware transactional memory, in: 34th Int'l Symp. on Computer Architecture (ISCA), 2007, pp. 81–91.

[27] A. Dragojevic, R. Guerraoui, Predicting the scalability of an STM, in: 5th ACM SIGPLAN Workshop on Transactional Computing, 2010.

[28] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, H. Tomari, Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8, in: 42nd Int'l Symp. on Computer Architecture (ISCA), 2015, pp. 144–157.

[29] htmbench-gem5, website (2020).
URL `http://ditec.um.es/~rtitos/patches/htmbench-gem5`

[30] intruder-v-mmap, website (May 2019).
URL `http://ditec.um.es/~rtitos/patches/intruder-v-mmap`

[31] vacation-ws-dr, website (May 2019).
URL `http://ditec.um.es/~rtitos/patches/vacation-ws-dr`