HeTM: Transactional Memory for Heterogeneous Systems

Daniel Castro[1], Paolo Romano[2], Aleksandar Ilic[3], and Amin M. Khan[4]

[1,2,3] *INESC-ID & Instituto Superior Técnico*, *Universidade de Lisboa*, Lisbon, Portugal.
[4] *Department of Computer Science*, *UiT The Arctic University of Norway*, Tromsø, Norway.

[1]daniel.castro@tecnico.ulisboa.pt, [2]romanop@gsd.inesc-id.pt, [3]aleksandar.ilic@inesc-id.pt, [4]amin.khan@uit.no

***Abstract*—Modern heterogeneous computing architectures, which couple multi-core CPUs with discrete many-core GPUs (or other specialized hardware accelerators), enable unprecedented peak performance and energy efficiency levels. However, developing applications that can take full advantage of the potential of heterogeneous systems is a notoriously hard task. This work takes a step towards reducing the complexity of programming heterogeneous systems by introducing the abstraction of Heterogeneous Transactional Memory (HeTM). HeTM provides programmers with the illusion of a single memory region, shared among the CPUs and the (discrete) GPU(s) of a heterogeneous system, with support for atomic transactions. Besides introducing the abstract semantics and programming model of HeTM, we present the design and evaluation of a concrete implementation of the proposed abstraction, referred herein as Speculative HeTM (SHeTM). SHeTM makes use of a novel design that leverages speculative techniques, which aims at hiding the inherently large communication latency between CPUs and discrete GPUs and at minimizing inter-device synchronization overhead. We demonstrate the efficiency of the SHeTM via an extensive quantitative study based both on synthetic benchmarks and on a popular object caching system.**

***Index Terms*—transaction, memory, CPU, GPU, heterogeneous, computing, system**

I. Introduction

Single-core performance of central processing units (CPUs) have reached a plateau in the last decade. In order to enable further increases of the processing capacity, while attaining high energy efficiency, modern computing architectures have henceforth adopted two key paradigms, namely *parallelism* and *heterogeneity*. As a result, nowadays, heterogeneous architectures that combine multi-core CPUs with many-core GPUs (or similar co-processors, e.g., TPUs [32]) have become the *de facto* standard in a broad range of domains that include HPC, servers and mobile devices.

Unfortunately, though, developing applications that take full advantage of the raw performance potential of modern massively parallel, heterogeneous architectures is a notoriously hard task. This has fostered, over the last year, intense research efforts aimed at developing new abstractions and programming paradigms for reducing the complexity of software development for modern heterogeneous platforms.

This work focuses on one key problem that arises when developing concurrent applications, whose complexity is exacerbated when considering massively parallel heterogeneous architectures, namely how to regulate access to shared data in a scalable way. We tackle this problem by proposing the abstraction of Heterogeneous Transactional Memory (HeTM). The HeTM abstraction combines two paradigms for concurrent programming, i.e., Transactional Memory (TM) [24], [48] and Shared Memory (SM) [34], [33], by providing the illusion of shared memory regions that are seamlessly accessed by CPUs and GPUs of a heterogeneous system, whose concurrent accesses can be synchronized via atomic transactions.

A large body of research has been devoted over the last years to investigate efficient implementations of both the TM and SM abstractions. Major industrial players in the heterogeneous computing landscape put a considerable amount of effort on implementing the SM abstraction, e.g., NVIDIA's Unified Memory [22], which represent a strong evidence that the industrial world perceives the benefits, in terms of ease of programming. However, existing SM implementations for heterogeneous systems provide programmers with low-level synchronization primitives, such as atomic operations and locks, exposing programmers to another well-known source of complexity: the need of designing efficient, yet provably correct, techniques to regulate concurrent access to shared data.

This is a notoriously hard problem, as designing efficient fine-grained locking protocols is a complex and error prone task [40] that can compromise one of the key principles of modern software development, i.e., software composability [23]. TM addresses exactly this problem: thanks to the abstraction of atomic transactions, programmers only need to specify *which* set of operations/code blocks have to be executed atomically, delegating to the TM implementation the problem of *how* atomicity should be achieved. The literature on TM has been very prolific over the last decade, leading to the development of a plethora of solutions in software [48], [15], hardware [24], [56], [38] and combinations thereof [6]. Existing TM solutions, however, consider homogeneous systems, in which threads

This work is funded by Fundação para a Ciência e a Tecnologia (FCT) OE component with references PTDC/EEIS-CR/1743/2014 and PTDC/EEI-COM/29271/2017 and by Programa Operacional de financiamento POR Lisboa supported by FEDER with references Lisboa-01-0145-FEDER-031901 (PTDC/CCI-COM/31901/2017) and Lisboa-01-0145-FEDER-029271 (UID/CEC/50021/2019). Amin's work is funded by the Research Council of Norway under PREAPP project (grant n° 231746/F20) and eX3 project (grant n° 270053). The authors would also like to thank the anonymous reviewers for their valuable comments and helpful suggestions.

978-1-7281-3613-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PACT.2019.00026

execute either on CPUs [56], [45] *or* on (discrete) GPUs [18], [53]. As such, existing TM systems fall short in harnessing the full potential of heterogeneous systems, failing to support execution scenarios in which CPUs and GPUs cooperate by concurrently accessing and manipulating the same state [25], [59], [37] — which is precisely the goal pursued by HeTM. Being an extension of the TM abstraction, HeTM is particularly attractive for irregular applications, such as graphs and complex data structures that make extensive use of pointers, for which designing scalable locking schemes can be complex. HeTM enables the possibility of accelerating this class of applications by offloading part of the computations, which would be executed concurrently by some CPU threads, to the GPU.

Building an efficient TM for heterogeneous systems, though, is far from being a trivial task. In fact, in homogeneous platforms, where the TM abstraction is confined within the boundaries of a single processing device, e.g., a multi-core system or a discrete GPU, conflict detection can be implemented via fast communication channels, e.g., the caches of a multi-core system. This is what allows existing TM designs to incur limited overhead, even though they trigger conflict detection multiple times during transaction execution, possibly as frequently as upon each memory access of a transaction. The HeTM abstraction, though, spans physically separated computational devices, which communicate via channels, such as PCIe [42], that are orders of magnitude slower than the ones assumed by conventional TM systems for homogeneous platforms. In these settings, thus, conventional TM approaches that impose multiple system-wide synchronizations along the critical path of execution of *each* transaction would incur prohibitive overheads that would cripple performance.

In this work, we tackle this challenge by presenting SHeTM (Speculative HeTM), the first implementation of the proposed HeTM abstraction. SHeTM leverages a set of novel techniques that operate in synergy to effectively mask the latency of the inter-device communication bus.

Hierarchical conflict detection. SHeTM's novel hierarchical approach aims at removing inter-device conflict detection from the critical path of transaction's execution, in order to amortize its cost across batches of transactions. More precisely, SHeTM detects **intra-device conflicts**, i.e., conflicts generated between transactions that execute on the same device (CPU or GPU), by relying on conventional TM implementations for homogeneous systems — an approach that we term *synchronous*, as conflicts are detected during transaction execution. **Inter-device conflicts**, conversely, are checked *asynchronously*, i.e., conflicts are detected periodically between *batches* of transactions that are concurrently executed and committed, in a speculative fashion, at different devices. In absence of conflicts, the updates of each device are merged, yielding a consistent state at both devices. If inter-device conflicts are revealed, the speculatively committed transactions are rolled back and the state of the devices whose transactions were discarded is re-aligned to that of the "winning" device.

The use of speculation and asynchronous inter-device conflict detection not only amortizes the performance toll imposed by the synchronization over a high-latency inter-connection bus across a large number of transactions, it also enables the use of embarrassingly-parallel conflict detection schemes that, by operating on large transaction batches, can be very efficiently executed by modern GPUs.

Non-blocking inter-device synchronization: Although the cost of inter-device conflict detection can be amortized over a batch of transactions, the larger the batch of transactions processed in a synchronization round, the higher the likelihood of experiencing conflicts across devices. Thus, in conflict prone workloads, where using smaller transaction batches is desirable, it is crucial for reducing the overhead of the inter-device synchronization by minimizing the period of time during which transaction processing is blocked. To this end, SHeTM introduces an innovative scheme that ensures that, even while inter-device synchronization is being performed, either the CPU or the GPU are able to process transactions. This goal is achieved by combining two mechanisms: i) overlapping the GPU-based validation of the transactions' batch with the processing of transactions on the CPU-side; ii) letting the GPU start processing the transactions of the next synchronization round, while the updates produced by the transactions it executed in the current round are being copied back to the CPU.

Conflict-aware dispatching & early validation: SHeTM exposes a programmatic interface that allows to control the assignment of transactions to either CPU or GPU. SHeTM exploits this information to implement a conflict-aware transaction dispatching scheme that aims at reducing the likelihood of inter-device contention. This is achieved by dispatching transactions that are likely to contend to the same device, where conflicts can be detected and resolved efficiently using the local TM implementation. Further, in case inter-device contention does arise, SHeTM employs an early validation scheme that aims at reducing overheads (i.e., wasted work) by detecting conflicts before the synchronization for the current round is activated.

Modular and extensible design. SHeTM is designed to ease integration with generic CPU-based and GPU-based TM implementations. To this end, SHeTM exposes a simple generic interface, which a TM needs to invoke in order to expose to SHeTM the read-sets and write-sets of the transactions it speculatively commits. The ability of SHeTM to incorporate different TM implementations is quite relevant in practice, given that the design space of TM is very wide and a number of studies have shown that no-one-size-fits-all TM implementation exists that can ensure optimal performance across all possible workloads [54], [11]. This flexibility allows therefore to easily incorporate in SHeTM additional TM implementations, and to further increase the robustness of its performance in a wide spectrum of workloads.

We evaluate SHeTM via an extensive experimental study, based on synthetic benchmarks — which we use to shape workloads aimed at quantifying the overheads and gains derived from the various mechanisms SHeTM employs — and a real world application, MemcachedGPU [25] — which allows us to assess SHeTM's performance with a realistic workload as well as to showcase the benefits, in terms of load balancing and ease of programming, stemming from the possibility of concurrently accessing common data from physically separated computational units.

II. Related Work

Existing programming models for heterogeneous systems aim to provide different abstraction levels to unify the execution among devices with different architectures, programming paradigms and memory spaces. These models span from low-level and user-managed frameworks (such as OpenCL [19]) up to the fully automated run-time systems (e.g., StarPU [1], OmpSS [14] and Cashmere [26]). Other recent efforts aimed at simplifying the accelerator programming with high-level OpenMP-like directives, as highlighted in OpenACC [20] and OpenMP 4.0 [4]. There are a number of ongoing efforts in the academia and industry aimed to automate data management and to unify memory in hybrid accelerated systems, for example, at the compiler level (CGCM [30], Spark-GPU [57] and RSVM [31]), NVIDIA CUDA Unified Memory [9] or even support at the Linux kernel level [51]. These solutions share our common high-level goal of simplifying the development of applications for heterogeneous systems. Yet, none of them tackle the challenges involved in ensuring the consistency guarantees provide by TM [21], exposing programmers to the notorious complexity of lock-based synchronization [23].

As mentioned, the literature in the area of TM has elaborated a plethora of design, exploring both hardware and software implementations. Although the majority of the existing literature focus on TM implementations for CPUs, TM is also gaining space in the GPU world [18], [27], [55], [49]. In this area, a relevant related work is the recent APUTM [52], which addressed the problem of implementing a STM for integrated GPUs. However, integrated GPUs reside in the same coherent domain as the CPU, unlike the case of discrete GPUs — which we target in our work. As such, developing a TM for integrated GPUs is a much less challenging endeavor, as, in fact, this problem can be solved by re-using existing designs for CPU TMs. To the best of our knowledge, our work is the first to present a TM system for heterogeneous systems that encompass both CPUs and discrete GPUs. It is also the first work to revisit the definition of conventional TM consistency semantics, e.g., [21], [28], to keep into account the specific architectural characteristics of heterogeneous systems.

In a broad sense, HeTM is related to the work on speculative processing in distributed systems. In particular, optimistic simulation systems [17], [43], where the state of local simulation objects is allowed to advance in a speculative fashion, i.e., skipping synchronization with remote objects and rolling back to a consistent state if *a posteriori* it is detected to have missed any relevant event from a remote object. Another related area has been investigated for speculative transaction processing techniques in distributed and replicated databases [44], [47]. Similarly, in this case the principle is similar, letting transactions commit speculatively and automatically roll-back the state of individual database replicas in case of any errors in speculation. HeTM builds on the same principles, but introduces new ad hoc designed techniques to meet the characteristics of heterogeneous systems composed of GPU and CPU.

III. Defining the HeTM abstraction

As mentioned, HeTM provides the illusion of a single transactional shared memory that is concurrently accessed by a set of physically separated devices, where devices are equipped with their own local memory and communicate over an interconnection bus like PCIe.

In the definition of the HeTM abstraction we do not consider, for the sake of generality, how transactions are generated and dispatched to the various execution devices. We leave the definition of these aspects to concrete implementation of the HeTM abstraction (see Section IV). We will simply assume that threads, in execution at any computational device attached to the HeTM platform, can access and manipulate its state exclusively by means of transactions. To this end, HeTM exposes a conventional API, through which threads can start a new transaction, submit read and write operations and request the commit or abort of the transaction. Extending the proposed HeTM abstraction to support intra-transaction parallelism [2], [58] and non-transactional accesses [35] would be possible, but it is outside of the scope of this work.

The rest of this section focuses on defining the correctness semantics that should be expected from a HeTM platform, such as the one that we will present in Section IV, that exploits speculative techniques to mask the costs of inter-device synchronization. More in detail, we intend to reason on the correctness of TM implementations that can commit transactions in a speculative fashion, i.e., without first checking for conflicts with transactions executing on remote devices, and that may therefore have to be later aborted in case an inter-device contention is eventually detected.

This speculative transaction execution model — in which transactions are first *speculatively committed* based only on local information, and only subsequently are *committed* — is desirable, in a HeTM platform, as it allows to remove intra-device conflict detection from the critical path of transactions' execution. In fact, in such a model, upon a transaction T is speculatively committed, $i)$ the thread that requested T's commit can be unblocked and process new transactions, and $ii)$ T's updates can be immediately made visible locally. On the other hand, this speculative execution model also enables a broader spectrum of concurrency anomalies with respect to conventional transaction execution models that do not contemplate the notion of speculatively committed transactions.

We start by observing that existing consistency criteria for classical TM systems, such as Opacity and Virtual World Consistency [21], [28] (see Section II), are unfit to capture the dynamics of the speculative transaction execution model that we advocate to enable efficient implementations of the HeTM abstraction. Roughly speaking, existing TM consistency criteria ensure, with various nuances, two key properties:

- P1. *The behavior of every committed transaction has to be justifiable by the same sequential execution containing only committed transactions, without contradicting real-time order.*
- P2. *The behavior of any active transaction, even if it eventually aborts, has to be justifiable by some sequential execution (possibly different) containing only committed transactions.*

We argue that property P1 remains adequate for a HeTM system. In fact, to preserve the ease of use of the TM abstraction, speculation should serve solely to enhance efficiency, while being totally hidden to applications. As such, the consistency semantics of committed transactions should remain unaltered, even if speculation is used for efficiency reasons.

Property P2, on the other hand, appears unfit to define the consistency semantics of HeTM platforms. In fact, the specification of P2 prohibits observing the updates of *any* uncommitted transaction, thus including the updates of speculatively committed transactions. Hence, if a transaction T attempted to read a data item updated by a speculatively committed transaction T', P2 would oblige any HeTM implementation to block T until the final outcome (commit/abort) of T' is determined — limiting the effectiveness of speculation to mask the costs of inter-device synchronization.

Note also that allowing active transactions to observe the effects of *any* speculatively committed transaction would not be a viable solution either. In fact, it would allow a transaction to observe the effects of two *conflicting* speculatively committed transactions. This would defeat the motivation at the basis of P2: avoiding that applications may fail in complex/unpredicted ways due to observing a state that no sequential execution could have ever produced.

Overall, we argue that consistency semantics offered by a HeTM platform should depart from classical consistency TM criteria by allowing different devices to use different sequential transaction histories to justify the execution of their local transactions. Intuitively, these transaction histories should be composed by: i) a prefix (possibly of different size) of the sequential execution history containing committed transactions (which, by P1, must be the same at each device), followed by ii) a device-dependent sequential history composed by transactions that speculatively-committed at that device.

We capture these semantics via the variant of property P2:

- P2†. *The behavior of any active or speculatively committed transaction T has to be justifiable by some sequential execution containing i) committed transactions and ii) speculatively committed transactions that executed on the same device as T.*

Properties P2† and P2 pursue the same high-level goal: guaranteeing that the state observed by any transaction T could have been produced in some sequential execution. Unlike P2, though, P2† allows to include in the sequential execution used to justify T's execution not only committed transactions, but also speculatively committed transactions that executed on the same device as T. This means that transactions that execute at different devices must observe a common history of committed transactions, but may witness the effects of different speculatively committed transactions, which are still being checked for inter-device conflicts.

Note that P2† also requires that the behavior of speculatively committed transactions (and not just that of active transactions) can be justified by a sequential execution. As active transactions can only read from committed or speculatively committed transactions, this implies that the only updates that can ever be observed are the ones produced by transactions that reflect some sequential history. Further, a transaction T can observe the effects of a (speculatively committed or committed) transaction T', only provided that T' does not conflict with any other transaction T'', whose effects T has already observed so far. In fact, if T were to observe the effects of two transactions that conflict either directly or indirectly, it would be impossible to include them both in the same sequential execution history that should be used to justify the execution of T.

IV. The SHeTM platform

This section presents SHeTM (Speculative HeTM), an implementation of the HeTM abstraction that relies on speculation to minimize the overheads of inter-device synchronization.

A. Architecture and programming model

SHeTM implements the proposed HeTM abstraction for heterogeneous platforms composed by one or more cache-coherent multi-core CPUs and a discrete GPU. SHeTM is implemented in C and relies on the CUDA API to orchestrate the execution of the GPU.

SHeTM maintains a full replica of the shared TM region, which we call STMR (Speculative Transactional Memory Region), on both the CPU and GPU. At each device, the execution of transactions is regulated by a local TM library, referred to as *guest* libraries. SHeTM adopts a modular software architecture that seeks to attain inter-operability with generic TM implementations for CPU and GPU. This feature is important, since supporting the integration of arbitrary guest TM libraries allows to adapt the choice of the TM implementation used on each device to the characteristics of the application workload and the device. In Section IV-B we discuss which mechanisms SHeTM employs to integrate third-party guest TM libraries, as well as the assumptions that these libraries need to satisfy to correctly inter-operate with SHeTM.

Programming model. SHeTM offers a conventional TM interface for demarcating (i.e., beginning, committing, aborting) transactions and declaring read/write accesses to the STMR. There are, however, relevant aspects related to the heterogeneous nature of the HeTM abstraction that programmers should take into account when developing transactional applications for SHeTM and that have influenced the design of SHeTM programming interfaces.

A first observation is that the STMR's replicas maintained by the CPU and GPU may be mapped in different positions in their address spaces. Thus, the management of pointers in SHeTM raises issues analogous to the ones that affect other implementations of the shared memory abstraction (e.g., in POSIX mmap or SystemV shmem [50]), such as: if pointers to a position within the STMR are stored in the STMR, they must be expressed as relative offsets and not as absolute addresses.

A second relevant observation is that architectural differences of CPUs and (discrete) GPUs have a great impact on their programming models and, as such, HeTM systems should keep these aspects into account to attain high efficiency. One key issue is that, differently from CPUs, where transactions are typically executed individually, in GPUs it is desirable to execute transactions in relatively large batches [49], [9], as this allows for: i) amortizing the latency of transactions' activation; ii) enhancing throughput when transferring to/from the GPU the inputs/output required/produced by transactions' execution; iii) improving resource utilization on modern GPUs.

To reconcile these differences, SHeTM abstracts over the computational model of CPU and GPU via a thread pool model in which each device exposes a number of worker threads. Worker threads are the only entities that can directly access the STMR, i.e., application threads that need to manipulate (or access) STMR should do so by submitting transactional requests to the worker threads (via SHeTM's API).

SHeTM views each instance of a transaction as an abstract operation that consumes an input and produces an output. SHeTM is opaque to the structure of transactions' inputs and outputs, requiring only information on their size in order to correctly transfer transactional requests/responses to/from the worker threads. In order to support the efficient execution of transactions on both CPU and GPU, SHeTM allows programmers to associate each transaction via: i) a "transactional function", which is meant to execute on the CPU and processes exactly one transaction; ii) a "transactional kernel", which is meant to execute on the GPU and processes a batch of transactions of a given size.

Developers of transactional kernels have the responsibility to control which and how many threads to activate, how many transactions each thread should execute, as well as how transactional inputs should be consumed. SHeTM, in turn, is responsible for activating transactional kernels, shipping to the GPU the corresponding transactions' inputs and retrieving the transactions' result to the host once the kernel ends.

Programmers are not obliged to provide two implementations for a given transaction. If they do so, though, this provides SHeTM with the flexibility to select the implementation/device to use for executing a given transaction instance in a dynamic fashion, using a work-stealing policy that aims to balance load on both CPU and GPU.

Transaction scheduling and dispatching. For each registered transaction, SHeTM allocates a number of request queues. The number of queues that SHeTM allocates for a given transaction depends on the number of implementations that were registered for it. If a single implementation was defined (either for CPU or for GPU), only a single queue is created, which is used to store all the requests for that transaction. If implementations for both the CPU and GPU are provided, instead, SHeTM allocates three request queues, noted CPU$_Q$, GPU$_Q$ and SHARED$_Q$. As their names suggest, the first two queues are meant to buffer requests which were submitted for execution on the CPU and GPU, respectively. This indication is passed to SHeTM via the programming interface used to support the submission of transactional requests, through which an optional *device-affinity* parameter can be specified.

This mechanism allows SHeTM to exploit external knowledge, e.g., provided by programmers or automatic tools (e.g., static code analysis [46] or on-line scheduling techniques [12], [13]), on the conflict patterns between different transaction instances and mitigate inter-device contention by dispatching conflict-prone transactions to the same device.

If, upon submission of a request, no *device-affinity* is indicated (and both CPU and GPU implementations exist for the corresponding transaction), then the request is routed to SHARED$_Q$, which is accessible by both devices on the basis of a work-stealing policy. Note that the enqueued requests are consumed at different granularity by the CPU and GPU. CPU worker threads process requests individually, extracting them from the CPU$_Q$ queues in a round-robin fashion, or, if CPU$_Q$ is found empty, from SHARED$_Q$. The processing of requests from the GPU$_Q$ queues, and the activation of the corresponding transactional kernel is coordinated by a management thread, which we call GPU-controller, running on the CPU. This thread monitors GPU$_Q$ and activates the corresponding transactional kernel when any GPU$_Q$ queue contains a sufficient number of requests to feed the kernel.

In many applications and standard benchmarks, transactions are naturally distributed to threads via some form of queue. This is the case of, e.g., MemcachedGPU [25], where transactions are triggered by the reception of network messages that are first stored into queues. Other examples are Intruder and Labyrinth of the STAMP benchmark suite [36]. This type of applications naturally fits the programming model of HeTM and incur no additional overheads due to the SHeTM's queuing system.

In applications that do not rely intrinsically on queues, the overhead of SHeTM varies depending on the workload characteristics: the larger the transaction execution time, the lower the overhead of the queuing system — as the lower will be the frequency of access to the queue by the worker threads and, consequently, the likelihood of contention on the queue(s). Note, though, that, on the CPU side, programmers can bypass the SHeTM queuing system and let application level threads manipulate the STMR, provided that: i) the STMR is only accessed transactionally via the same API used by the worker threads; and, ii) the conflict resolution policy never aborts speculatively committed transactions on the CPU. In these conditions, correctness is preserved at the cost of exposing additional complexity to programmers — as they become responsible for implementing the transaction scheduling and dispatching mechanisms provided by the SHeTM framework.

B. Integration with guest TM libraries.

To guarantee the HeTM's consistency semantics described in Section III, SHeTM assumes that the underlying TMs ensure opacity (or, in general, any TM consistency criterion that guarantees the properties P1 and P2 defined in Section III).

SHeTM abstracts over the internal logic of the guest TM libraries and interfaces with them by exposing a simple callback function that the guest TM should invoke, whenever a transaction commits, detecting conflicts in a hierarchical fashion, i.e., first locally in each device and then globally.

CPU instrumentation. On the CPU side, upon the commit of a transaction, a guest TM library must provide as input to SHeTM's callback function an array containing the $\langle \textit{address}, \textit{value}, \textit{timestamp} \rangle$ of each memory position *updated* by that transaction. The specified timestamp must be usable by HeTM to totally order the updates to that memory position and is easily provided both by software and hardware TM implementations. For instance, most software TM implementations, e.g., TinySTM [15] or NoREC [10], use a logical timestamp to totally order the commits of all transactions. The same can be done in hardware TM implementations, such as Intel TSX, by reading the processor cycles, i.e., using the RDTSCP instruction [7]. Gathering transactions' write-sets imposes no additional overhead to a guest STM, as STMs need anyway to track the write-sets in software. For HTM, SHeTM requires the software instrumentation of write operations to gather the transaction's write-set. It is worth noting that, in many realistic workloads, writes are largely outnumbered by reads and, as such, the resulting instrumentation overhead is small. The HeTM's callback function appends the write-sets into thread-local data-structures, referred herein as the **CPU write-set logs**, and periodically offloads them to the GPU to perform inter-device conflict detection.

GPU instrumentation. On the GPU side, a guest TM library must communicate to the HeTM's callback the set of addresses read and written by the committing transaction. Conversely, on the GPU side the read-set and write-set of a committing transactions are used to update two bitmaps, noted $\mathbf{RS}^{\mathbf{GPU}}_{\mathbf{bmp}}$ and $\mathbf{WS}^{\mathbf{GPU}}_{\mathbf{bmp}}$, that track the regions of the STMR that GPU transactions read or wrote, respectively. After those bitmaps are updated, the transaction's read-set and write-set can be immediately discarded. The necessity for this asymmetric instrumentation logic at the CPU and GPU is further detailed in Section IV-C.

Additional assumptions. SHeTM needs to manipulate the state of the STMR to merge the updates produced at both devices and to cancel the effects of speculatively committed transactions in case inter-device conflicts are detected. These updates are performed in a non-transactional way, i.e., bypassing the APIs of the guest TM library — ensuring that no transaction is executing concurrently, to preserve consistency. This design is safe under the assumption that any meta-data managed by the guest TM libraries are maintained externally to the STMR, e.g., in a disjoint memory region on the local device. This assumption is met in practice by most TM implementations, and is valid for all existing HTM implementations (which maintain their metadata in the processor's caches), and for all word-based STMs (where TM metadata must necessarily be stored in a disjoint memory region to avoid interference with the application's memory layout).

Supported libraries. Currently, SHeTM supports three TM implementations: two on the CPU side – TinySTM [15] and Intel's TSX [29], implemented respectively in software and hardware – and one on the GPU side, namely PR-STM [49].

C. Basic Algorithm

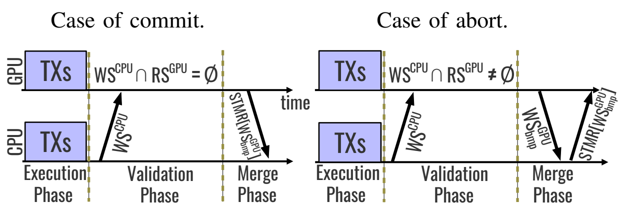
We start by describing a basic variant of the SHeTM's algorithm that serves a twofold purpose: i) it allows us to simplify presentation, by explaining the design of SHeTM in an incremental fashion; and, ii) it exposes several sources of inefficiency that we address in the following text. At this stage, we will assume a fixed policy to deal with inter-device contention that deterministically discards the transactions speculatively committed by the GPU. A discussion on how to relax this assumption and support policies that discard transactions speculatively committed by the CPU can be found in the extended technical report [8].

SHeTM orchestrates the execution of GPU and CPU in *synchronization rounds*, where each round is composed by three phases: execution, validation, and merge (see Figure 1a). In a nutshell, SHeTM adopts a *hierarchical conflict detection mechanism* that operates as follows: during the execution phase, conflicts are detected only among transactions that execute on the same device, using a local TM implementation; inter-device conflicts are then verified during the validation phase via a novel scheme that leverages the massive parallelism of modern GPUs to maximize performance.

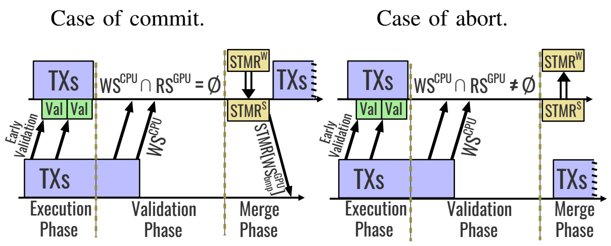
1) ***Execution Phase:*** In the execution phase, transactions are extracted from the input queues and fed to the devices during a user-tunable period. Transaction processing is executed in an independent way at both devices, starting from a consistent snapshot, i.e., an identical replica of the STMR at both devices, and executing transactions in a speculative fashion: the execution of transactions is regulated exclusively by the local TM library, which only detects conflicts between local transactions, avoiding any inter-device synchronization.

When a transaction requests to commit, the unmodified commit logic of the local TM library is used to atomically propagate the transaction's update to the local STMR replica. This local commit event coincides with the speculative commit, in the execution model assumed by the HeTM abstraction. At this point the TM library invokes the call-back functions exposed by SHeTM, as referred in Section IV-B. On the CPU, the write-set of the transaction is appended to a per-thread log. On the GPU side, the transaction's read-set and write-set are used to update $\mathrm{RS}^{\mathrm{GPU}}_{\mathrm{bmp}}$ and $\mathrm{WS}^{\mathrm{GPU}}_{\mathrm{bmp}}$ bitmaps. The bitmaps encode the set of addresses read and written by every transaction that speculatively committed during the execution phase, and they are updated concurrently by the GPU threads that are in charge of executing transactions.

As mentioned, the duration of execution phase is a user-tunable parameter that allows to explore an interesting performance trade-off, which will be studied in Section V-D. Longer periods imply less frequent synchronizations, which



(a) A basic variant of SHeTM.



(b) Illustrating the behavior of SHeTM.

Figure 1: Figure 1a illustrates a simple variant of SHeTM, while Figure 1b presents how to avoid unnecessary copies.

means lower overhead in case the synchronization is successful. However, longer period of executions mean also that a larger number of transactions are speculatively executed at both devices, increasing the probability of inter-device contention — thus leading to wasting more work (aborted transactions).

2) ***Validation Phase:*** The goal of this stage is to determine whether there was any conflict between the transactions processed by the CPU and the GPU during the execution phase. We designed the logic of this phase on the basis of the following key observations:

• As the local TM libraries are assumed to ensure opacity, the behavior of the speculatively committed transactions at each device is already guaranteed to be equivalent to a sequential execution (although defined over different sets of transactions). The set S of transactions speculatively committed during the processing phase at a device $D \in \{\text{CPU}, \text{GPU}\}$ can thus be logically subsumed by a single, equivalent transaction, noted T^D, whose read-set and write-set is the union of the read-sets and write-sets of the transactions in S. This observation allows us to reduce the problem of conflict detection among a number of speculatively committed transactions to the problem of detecting conflicts between a pair of logical equivalent transactions, which we denote as T^{CPU} and T^{GPU}.

• Detecting conflicts between a pair of transactions can be reduced to verifying intersections between their read-sets and write-sets [41], [3]. This computation can be efficiently parallelized using GPUs, especially if the sets are large, as it is the case for T^{CPU} and T^{GPU}, which subsume a (typically very large) number of transactions.

The design of the SHeTM's validation scheme, as well as its instrumentation logic, were engineered, based on this observation, so as to take full advantage of modern GPUs.

• In many realistic workloads, transactions read a much larger number of memory positions than they write to [16]. As such, the read-sets of T^{CPU} and T^{GPU} are likely to be much larger than their corresponding write-sets. Motivated by this observation, we designed the validation scheme in a way to avoid transmitting the read-sets over the inter-connection bus. Note that there are two possible orders in which T^{CPU} and T^{GPU} may be serialized, namely $T^{\text{GPU}} \rightarrow T^{\text{CPU}}$ or $T^{\text{CPU}} \rightarrow T^{\text{GPU}}$. For the former order to be valid, none of the writes generated by T^{GPU} should be "missed" by T^{CPU}, i.e., $\text{WS}^{\text{GPU}} \cap \text{RS}^{\text{CPU}} = \emptyset$ (where WS^{GPU} and RS^{CPU} denote the write-set and read-set of T^{GPU} and T^{CPU}, respectively). The latter order, conversely, requires verifying whether $\text{WS}^{\text{CPU}} \cap \text{RS}^{\text{GPU}} = \emptyset$. Keeping into account that we intend to leverage the GPU to perform validation, SHeTM opts for testing whether the **CPU transactions can be serialized before the GPU transactions, i.e., $T^{\text{CPU}} \rightarrow T^{\text{GPU}}$**. In fact, the computation of $\text{WS}^{\text{CPU}} \cap \text{RS}^{\text{GPU}} = \emptyset$ can be performed on the GPU side by shipping only the write-sets of the CPU transactions — whereas, the opposite serialization order ($T^{\text{GPU}} \rightarrow T^{\text{CPU}}$) would require shipping to the GPU the read-sets of the CPU transactions.

• In order to guarantee that the updates of the T^{CPU} and T^{GPU} can be mutually exchanged and applied at each device, yielding a state equivalent to the one produced by the schedule $T^{\text{CPU}} \rightarrow T^{\text{GPU}}$, it is necessary to exclude also the presence of write-write conflicts, i.e., whether $\text{WS}^{\text{CPU}} \cap \text{WS}^{\text{GPU}} = \emptyset$. The approach taken in SHeTM to guarantee this property is to track the writes performed by GPU transactions not only in $\text{WS}^{\text{GPU}}_{\text{bmp}}$, but also in $\text{RS}^{\text{GPU}}_{\text{bmp}}$. By guaranteeing that $\text{WS}^{\text{GPU}} \subseteq \text{RS}^{\text{GPU}}$, the verification of $\text{WS}^{\text{CPU}} \cap \text{RS}^{\text{GPU}} = \emptyset$ also ensures that $\text{WS}^{\text{CPU}} \cap \text{WS}^{\text{GPU}} = \emptyset$. Furthermore, it is worth noting that writes are typically outnumbered by reads, the overhead incurred by tracking the writes in two bitmaps is expected to be low.

Let us put all these pieces together and discuss how the validation phase operates in a systematic fashion.

The validation phase starts by transferring the write-set logs gathered by each CPU thread to the GPU. The logs are streamed in chunks to achieve high throughput and activate validation kernels that operate at sufficient granularity to achieve high utilization of GPU resources. A validation kernel on the GPU takes as input a chunk of a log and operates as follows: For each tuple $\langle \mathit{address}, \mathit{value}, \mathit{timestamp} \rangle$ in the input log it is checked whether the corresponding entry in the GPU's read-set bitmap is set — which indicates that some of the transactions speculatively committed by the GPU during the execution phase read that address.

If the CPU write is found to have invalidated the read-set of T^{GPU} the validation phase returns a negative outcome, but it continues applying the full set of write-set logs sent by the CPU. This ensures that, at the end of the validation phase, the GPU's STMR incorporates all the effects of T^{CPU}. Thus, if in the merge phase, the state of the GPU's STMR needs to be re-aligned to the current state of the CPU's STMR, it is sufficient to simply undo the effects of the T^{GPU}.

If the CPU write does not invalidate the read-set of T^{GPU}, the corresponding value stored in the CPU write-set log is applied to the GPU's STMR. This is performed non-transactionally, since the GPU is not processing transactions during the validation phase. However, since the CPU logs are validated in arbitrary order on the GPU side, before applying it is necessary to verify if the version currently present in the GPU's STMR is not more recent than one that is being applied. To this end, on the GPU, SHeTM maintains a timestamp array, denoted as TS, which has an entry per word of the STMR reserved to store the timestamps of the CPU writes applied during the validation phase. During validation, GPU threads consult the TS to determine whether the write being validated reflects a more recent state than the one already present in the STMR, applying that log tuple only if this is the case. Note that since concurrent GPU threads may be validating writes targeting the same address, the atomicity of the test for freshness and the value application is ensured via a lock implemented using the first bit of the corresponding TS entry.

*3) **Merge Phase:*** The merge phase ensures that the replicas of the STMR at the CPU and at the GPU are consistent, before starting the execution phase of the next synchronization round. The way in which the states of the CPU and GPU are realigned depends on the outcome of the validation phase.

If the validation phase is successful, i.e., no inter-device conflicts are detected, the GPU's replica of the STMR already incorporates the updates of the transactions that speculatively committed at both devices, since recall that, during the validation phase, the GPU also applies the CPU write-sets into its local STMR replica. To this end, the GPU-controller thread fetches the GPU's write-set bitmap, which identifies the memory regions updated by the transactions speculatively committed by the GPU and activates the memory transfers to update in-place the STMR's replica on the CPU.

If the validation phase fails, the state of the CPU's STMR overrides the GPU's STMR. To this end, the GPU controller thread obtains the GPU's write-set bitmap and transfers the CPU's state over the memory regions marked as updated on the GPU's write-set bitmap, thus undoing any side-effect of the execution of transactions on the GPU side.

D. Optimizations

SHeTM integrates a number of additional mechanisms that aim at tackling two main sources of inefficiency: **the blocking time** (i.e., the period during which transaction processing is blocked) due to inter-device synchronization, and the overhead imposed in case of inter-device contention. We present these techniques in the following text and illustrate them in Figure 1b.

Inter-device synchronization. As illustrated in Figure 1a, in the basic algorithm presented in Section IV-C, transaction processing is blocked throughout the validation and merge phases both at the CPU and at the GPU. This is clearly undesirable for efficiency reasons, especially if one considers that, to reduce the likelihood of inter-device contention, it is desirable to use relatively short execution phases.

SHeTM tackles this issue by integrating mechanisms aimed at reducing the blocking time both at the CPU and at the GPU.

On the CPU side, during the validation phase, SHeTM allows the worker threads to continue processing transactions concurrently with the streaming of the logs accumulated during the execution phase. The CPU blocking time on the execution phase only occurs when a very few log chunks are left to be offloaded to the GPU. In practical settings, the speed at which logs can be transferred is higher than that at which new logs can be produced by the worker threads. Thus, this mechanism effectively overlaps transaction processing at the CPU side with the log transfers to the GPU, while generating a relatively little amount of additional logs to validate for the GPU.

On the GPU side, at the end of the merge phase, the basic algorithm blocks transaction processing while transferring to the CPU the memory regions updated by the GPU. This is done to ensure that the state of the GPU's STMR is not corrupted due to the execution of transactions while the device to host transfer is ongoing. SHeTM tackles this problem by employing a double buffering approach. At the start of the execution phase, a *shadow copy* (STMR^S in Figure 1b) of the GPU's STMR (STMR^W in Figure 1b) is created, via a device to device copy. As soon as STMR^S is created, GPU transaction processing can immediately resume, since STMR^S is isolated from the updates of transactions (which operate exclusively on the STMR^W) and can be used to feed the device to host transfer.

Inter-device contention. As discussed in Section IV-A, SHeTM's API allows exploiting external knowledge on transactions' conflict patterns (via the device affinity specified at transactions' submission time) to control the dispatching of transactions and reduce inter-device contention. Besides striving to reduce the likelihood of inter-device contention, SHeTM incorporates two additional mechanisms that aim at reducing two sources of overhead when inter-device conflicts do occur:

• **Wasted work.** In the basic algorithm, conflicts are detected only at the end of the execution phase. This leads to wasting a large amount of work at the GPU, if a conflict is detected in the validation phase. We tackle this problem by introducing an early validation scheme that periodically transfers the CPU's logs to the GPU, where they are validated (but not applied) while transactions are concurrently processed on both devices. As early validations are concurrent with transaction processing, it is still necessary to validate all the write-set logs produced during the execution phase in the validation phase. Yet, by anticipating the detection of inter-device conflict, as we will see in Section V, early-validation can provide significant gains in contention prone workloads by reducing the time the GPU spends performing computations that are eventually discarded.

• **Rollback latency.** Realigning the GPU's state to that of the CPU, in case of inter-device contention, imposes significant overhead in the basic algorithm. Every memory region updated by the GPU has to be copied from the CPU and, during this transfer, transaction processing is blocked at both devices. Fortunately, the availability of the shadow copy is of great help in this case. Recall that the shadow copy reflects a consistent state of the STMR, as at the beginning of the current synchronization round. Thus, in order to align the shadow copy

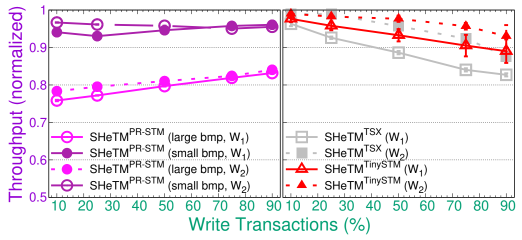


Figure 2: Cost of instrumentation of guest TM libraries.

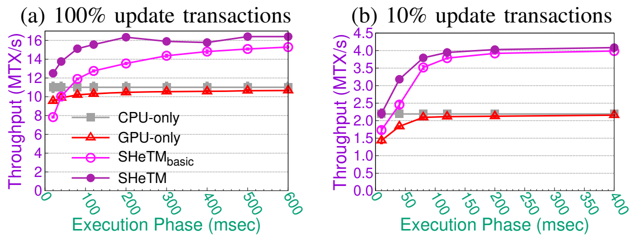


Figure 3: Efficiency in absence on contention.

to the current state of the CPU, it suffices to apply to it the CPU's write-set logs.

• **Enhancing memory transfer's throughput.** In order to exploit PCIe bandwidth while transferring information, transfers are performed in *chunks* of relatively coarse granularity. To this end, the CPU write-set logs are shipped to the GPU using a granularity of 48 KB; also, the write-set bitmap on the GPU tracks updates with a granularity of 16KB.

As a further optimization, the GPU-controller coalesces transfers of contiguous chunks from the GPU to the CPU during the merge phase (in the case of no inter-device contention), as well as when performing the device to device copy from the shadow to the working copy of the STMR (in the case of inter-device contention).

V. Evaluation

This Section presents an experimental study that aims at answering the following key questions: the costs imposed by the instrumentation of the guest TM libraries (Sec. V-A); overhead introduced to workloads whose scalability is not limited by inter-device contention (Sec. V-B); performance degradation due to inter-device contention (Sec. V-C); optimization gains over simpler designs (Sec. V-B and Sec. V-C); and finally, how effective SHeTM is with realistic applications (Sec. V-D).

Our evaluation is conducted using a machine equipped with an Intel Xeon E5-2648L v4 CPU (14 cores with support for HTM, 32GB DRAM), an Nvidia GTX 1080 GPU (8GB XDDR5, driver 387.34, CUDA 9.1), and running Ubuntu 16.04.3 LTS (kernel 4.4.0-57). Applications are manually instrumented to use the SHeTM API.

We based our evaluation on a set of synthetic benchmarks conceived to assess different aspects of SHeTM's design, and on MemcahedGPU [25].

In all the tests, we use 8 worker threads on the CPU side. As for the transactional kernels, we tuned their configuration (number of transactions per kernel activation, active threads and thread blocks) on the basis of preliminary evaluations to maximize the GPU throughput. The synthetic workloads use the same transactional logic on both the CPU and GPU and operate on a STMR of size 600MB, unless otherwise specified; the STMR size in MemcachedGPU is around 480MB.

A. Instrumentation Costs

Let us start by assessing the overhead induced by the software instrumentation that SHeTM requires for its guest TM libraries. To this end we consider two workloads, noted W_1 and W_2, that access the STMR uniformly at random. In W_1, read-only transactions issue 4 reads, whereas update transactions read and update 4 memory positions. W_2 is identical to W_1, except that both transaction types issue 40, and not 4 reads. W_1 is designed to stress the instrumentation of read and write operations. W_2 is selected as representative of many realistic workloads, in which reads outnumber the writes.

In the plot in Figure 2 we vary on the x-axis the percentage of the update transactions from 10% to 90% and report on the y-axis the throughput normalized w.r.t. un-instrumented version of PR-STM [49] for the GPU (left plot), and of TinySTM [15] and TSX for the CPU (right plot).

In the left plot (GPU), we consider using two different levels of tracking granularity for the read-set bitmap (RS^{GPU}_{bmp}), namely 4B (small bmp) and 1KB (large bmp). We can see that, independently of the considered workload, the use of the small granularity bitmaps induce, larger overheads, approx. 20%, as its larger size leads to a lower locality of reference. In contrast, the coarser granularity bitmap reduces significantly the instrumentation overhead, to approx. 5%, at the cost, though, of spurious aborts due to the risk of false positives in the conflict detection scheme. As a matter of fact, the trade-off between instrumentation overhead and access tracking granularity is well known in the literature, e.g., TM [15].

In the right plot (CPU), we observe that the instrumentation cost is on average around 5% for W_2 for both TinySTM and TSX. In all scenarios, the overhead is below 10% except for the most write intensive variants of W_1, where it remains anyway below 20% even in presence of 90% of update transactions.

B. Efficiency in absence of inter-device contention

Next, we intend to assess which overheads SHeTM incurs in workloads whose scalability is not limited by inter-device contention. Here, we consider two variants of the W_1 workload, generating 100% (W_1-100%) and 10% (W_1-10%) update transactions, respectively.

We avoid inter-device contention by partitioning the STMR in two halves and restricting CPU and GPU to access a different half. The results of this study are reported in Figure 3, in which we vary on the x-axis the duration of the execution phase from 1 msec to 600 msec and report on the y-axis the throughput of SHeTM and of the following baselines: the basic variant of SHeTM presented in Section IV-C, noted *SHeTM basic*; TSX running solo, noted *CPU-only*; PR-STM running solo and copying its STMR to the host, after executing a kernel, using double buffer (i.e., without blocking), noted *GPU-only*.

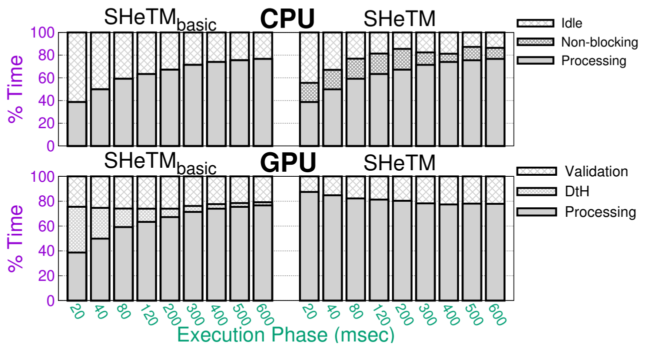


Figure 4: Break-down of exec. times (100% update transactions)

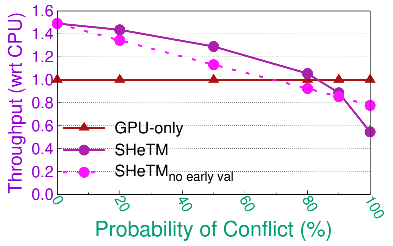


Figure 5: Sensitivity to inter-device contention.

The throughput plot on the left, which refers to W_1-100%, shows that as the execution period grows the performance of SHeTM also increases, which is as expected, since the relative amount of time spent performing the validation and merge phases reduces, amortizing their cost over larger period of useful processing (see Figure 4). The peak throughput of approx. 17M tx/sec, is reached at 200 msecs and plateaus beyond that value. SHeTM's peak throughput is about 55% higher than the peak throughput of CPU-only and GPU-only (approx. 11 M tx/sec) and only 23% lower than the throughput of an idealized system that could total the combined throughput of both uninstrumented devices.

By contrasting the performance of SHeTM with that of *basic* we can clearly appreciate the performance gains enabled by the optimizations described in Section IV-D, which are particularly significant with small execution periods (up to +56% higher throughput when the execution period lasts 1 msec). The bar plots in Figure 4, which report the breakdown of times spent by the CPU and GPU in various phases, allow us to derive additional insights on the sources of these gains. The use of double buffering on the GPU side to overlap kernel processing with the device to host transfer in the merge phase is the largest source of gains (DtH in SHeTM$_{\text{basic}}$, which is replaced by processing time in the optimized SHeTM). On the CPU side, the ability to overlap transaction processing (noted *non-blocking* in the figure) with the shipping of logs to the GPU has also a meaningful impact on reducing the blocking time, although not as strong as on the GPU side.

Finally, let us analyze the results reported in the right plot of Figure 3, which refers to the workload with 10% of update transactions. In this scenario, which considers a less extreme (and arguably more realistic) application workload, the peak throughput of SHeTM converges to 4M tx/sec, which is very close to the peak throughput achieved by an idealized solution that achieves a performance equal to that of the two device — an additional evidence of the efficiency of the proposed design.

C. Sensitivity to contention

We now consider the same workload as in the previous study, but inject with a given probability a conflicting access at random in the stream of writes generated by the CPU transactions.

We vary on the x-axis the inter-device conflict probability, fix the duration of the execution phase at 80 msecs and compare, in Figure 5, the performance of SHeTM with and without the early validation mechanism. On the y-axis we report the throughput normalized with respect to TSX (unistrumented) running solo and report, as reference, also the throughput achieved using PR-STM, running solo with double buffering.

The analysis of this plots reveals several insights. The first observation is that SHeTM consistently outperforms both TSX and PRSTM for abort rates as high as 80%. In medium contention, e.g., 50% probability of contention, SHeTM continues to deliver a 30% gain over the fastest individual device (CPU). Even when operating at the extreme 100% abort rate it incurs only a modest overhead (approx. 20% if the early validation is disabled). Overall, these results confirm the robustness of SHeTM performance even in adverse scenarios.

Early validation appears to be a powerful mechanism to mitigate overhead, especially in medium-high contention scenario (50% and 80% abort rate). Only beyond 80% inter-device contention SHeTM presents overheads w.r.t. CPU and GPU. Such scenarios are arguably non-representative of the desired operational region of any TM system. In 100% inter-device contention, early validation fails constantly, triggering the completion of the current execution phase and device transfer of the CPU logs. This is logically equivalent to operate with a much shorter execution phase, which, as seen in Figure 3, tends to induce longer blocking periods of the CPU.

D. MemcachedGPU

MemcachedGPU [25] extends Memcached, a popular in-memory object caching system, in order to use GPUs to serve lookup (GET) and update (PUT) requests for cached objects.

Its original implementation presents an ad-hoc synchronization mechanism to manipulate the contents of the cache held in GPU's memory. Besides being non-trivial, PUT operations are executed single threaded and block other concurrent GETs. Furthermore, CPU and GPU work in a pipeline fashion, i.e., they do not access concurrently the same data at the same time. SHeTM ameliorates all these issues while maintaining the cache's state synchronized both on the CPU and GPU.

A fixed number of sets compose a cache. $\langle key, value\rangle$ pairs are hashed into the target set, which has 8 slots and may contain up to that many pairs (8-way associative). We use keys/values of 16B/32B, respectively. The LRU replacement policy evicts old pairs based on per-slot timestamp.

The CPU implementation is straightforward: a worker thread computes the target set and then issues a transaction to perform the GET/PUT logic. On the GPU, the target set

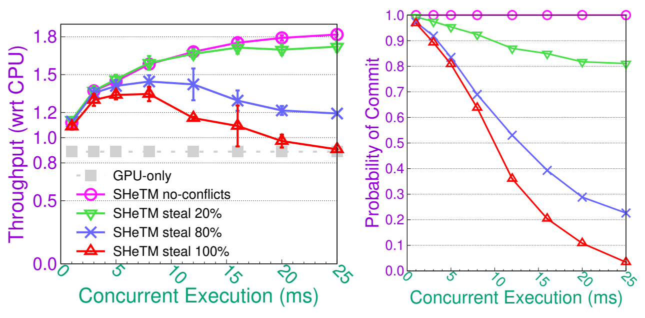


Figure 6: Throughput of HeTM for Memcached with possible conflicts.

is searched in parallel (as in the original MemcachedGPU implementation [25]) non-transactionally and a transaction is used to retrieve/set the value (for GETs/PUTs, respectively) and to update the LRU metadata of the corresponding slot.

The resulting implementation scheme introduces intra-device conflicts between any concurrent operation with the same input key, as each GET changes the per slot timestamp. To reduce the likelihood of inter-device contention CPU and GPU use distinct timestamps, which implies that the LRU policy is local to CPU or GPU, i.e., the pair freshness is only affected by device-local transactions. The key advantage of this approach is that CPU GETs do not conflict with GPU GETs (and vice-versa).

Concurrent inter-device PUT operations conflict if they access the same set, which guarantees that it is safe to use local LRU policies — as concurrent updates to the same set by different devices will trigger an abort and be rejected by SHeTM. To this end, a per set timestamp (common to both the CPU and CPU) is updated whenever a PUT operation is executed at either device.

Concurrent inter-device GETs and PUTs that access the same key conflict only if it is the CPU to issue the PUT. In fact, since SHeTM attempts to serialize the transactions executed by the CPU before the ones of the GPU, if the GPU issues a PUT it is perfectly fine for the CPU to "miss" this update.

In this experiment, we use a cache with 1,000,000 sets, the workload is composed of 99.9% of GETs and the object popularity follows a Zipfian distribution with parameter $\alpha = 0.5$ — which is a reasonable value for evaluating caches [5].

We consider 4 different workloads. In the first workload (*no-conflicts*), we balance the load (i.e., cache operations) input to the GPU and CPU by using the last bit of the key accessed by an operation. This guarantees that the input queues of the CPU and GPU can never contain operations that access a common set, thus, excluding the possibility of inter-device contention.

We then evaluate load unbalanced scenarios, which emulate application-level shifts of the popularity of accessed objects, namely a drop of the arrival rate of requests for GPU_Q and a corresponding increase for CPU_Q. This causes the GPU to start stealing from the CPU queues, i.e., processing requests for objects that can be concurrently accessed by the CPU. We consider three scenarios in which the GPU steals requests from the CPU queues with increasing probability, i.e., *steal 20%*, *steal 80%* and *steal 100%*. The last one emulates the extreme scenario where no device affinity is set to mitigate contention, so that both devices access the same keys.

Figure 6 shows that SHeTM achieves almost indistinguishable performance in the *no-conflicts* and the *steal 20%*, being in both cases less than 20% away from the ideal solution and 80% better than both GPU-only and CPU-only. We consider as an ideal solution the case that incurs no overhead and totals the equivalent normalized throughput of both CPU-only and GPU-only, which in this case is approx. 1.9.

The abort rate converges to that of the steal rate (resp. 0%, 20% 80% and 100%), since the number of potentially conflicting transactions executed at both devices increases when the duration of the execution round is increased. In fact, if its duration grows large enough, during a batch where one of the devices *steals* transactions from the other, the probability of inter-device contention unavoidably converges to 1 for this workload. With the duration of approx. 25 msecs, execution rounds where GPU steals from the CPU are very likely to fail.

The gains remain significant even in case the GPU steals operations from the CPU queue with 80% of probability (20% to 40% speed-up over CPU-only). Finally, it is worth highlighting that even when the contention-avoidance dispatching mechanisms of SHeTM are not used and contention is high (steal 100%), SHeTM achieves robust performance with speed-ups of up to approx. 30% (10 msecs rounds), and throughput on par with CPU-only even when the inter-device conflict probability (right plot) converges to 1.

VI. Conclusions and Future Work

This work introduced the abstraction of Heterogeneous Transactional Memory (HeTM). HeTM aims to facilitate programming of heterogeneous platforms, by abstracting the difficulties of data sharing across multiple physically separated units via the illusion of a single transactional memory shared among CPUs and (discrete) GPU(s).

Besides introducing the abstract semantics and programming model of HeTM, we presented an efficient, yet modular, implementation of the proposed abstraction, named Speculative HeTM (SHeTM). We demonstrated the efficiency of SHeTM via an extensive quantitative study based both on synthetic benchmarks and on a popular object caching system.

This work opens a number of research questions related to defining alternative semantics and designs for the HeTM abstraction. A specific question that we intend to investigate in the future is how to extend SHeTM to orchestrate the execution of multiple GPUs.

An other interesting question is how to leverage, e.g., static code analysis [46] or on-line scheduling techniques [12], [13] to automatically dispatch transactions to either CPU or GPU.

Finally, it would be interesting to evaluate SHeTM using a broader range of irregular applications, e.g., extending the recent work by Nelson et al. [39] that investigated the use of locking schemes to parallelize k-means on GPUs.

References

[1] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[2] W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun, "Implementing and Evaluating Nested Parallel Transactions in Software Transactional Memory," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010, pp. 253–262.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[4] O. A. R. Board *et al.*, "OpenMP Application Program Interface," *version 4.0*, 2013.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: evidence and implications," in *INFOCOM*, vol. 1. New York, NY, USA: IEEE, March 1999, pp. 126–134.

[6] I. Calciu, J. Gottschlich, T. Shpeisman, G. Pokam, and M. Herlihy, "Invyswell: A Hybrid Transactional Memory for Haswell's Restricted Transactional Memory Categories and Subject Descriptors," in *PACT*. IEEE, 2014, pp. 187–200.

[7] D. Castro, P. Romano, and J. Barreto, "Hardware Transactional Memory meets Memory Persistency," in *IPDPS*. New York, NY, USA: IEEE, 2018, pp. 368–377.

[8] D. Castro, P. Romano, A. Illic, and A. M. Khan, "HeTM: Transactional Memory for Heterogeneous Systems," *CoRR*, vol. abs/1905.00661, 2019.

[9] N. Corporation, "CUDA C Programming Guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2015.

[10] L. Dalessandro, M. F. Spear, and M. L. Scott, "NOrec: Streamlining STM by abolishing ownership records," in *PPoPP*, ACM. Bangalore, India: ACM, 2010, pp. 67–78.

[11] D. Didona, N. Diegues, A.-M. Kermarrec, R. Guerraoui, R. Neves, and P. Romano, "ProteusTM: Abstraction Meets Performance in Transactional Memory," *SIGOPS Oper. Syst. Rev.*, vol. 50, no. 2, pp. 757–771, Mar. 2016.

[12] N. Diegues, P. Romano, and S. Garbatov, "Seer: Probabilistic Scheduling for Hardware Transactional Memory," in *SPAA*, ser. SPAA '15. New York, NY, USA: ACM, 2015, pp. 224–233.

[13] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh, "Preventing Versus Curing: Avoiding Conflicts in Transactional Memories," in *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, ser. PODC '09. New York, NY, USA: ACM, 2009, pp. 7–16.

[14] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel processing letters*, vol. 21, no. 02, pp. 173–193, 2011.

[15] P. Felber, C. Fetzer, P. Marlier, and T. Riegel, "Time-Based Software Transactional Memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, pp. 1793–1807, 2010.

[16] P. Felber, S. Issa, A. Matveev, and P. Romano, "Hardware Read-write Lock Elision," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: ACM, 2016, pp. 34:1–34:15.

[17] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Commun. ACM*, vol. 33, no. 10, pp. 30–53, Oct. 1990.

[18] W. W. L. Fung, I. Singh, A. Brownsword, and T. Aamodt, "Kilo TM: Hardware transactional memory for GPU architectures," *IEEE Micro*, vol. 32, no. 3, pp. 7–16, 2012.

[19] K. O. W. Group *et al.*, "The OpenCL Specification," *version*, vol. 1, no. 29, p. 8, 2008.

[20] O. W. Group *et al.*, "OpenACC specification," *version 2.7*, 2018.

[21] R. Guerraoui and M. Kapalka, "On the Correctness of Transactional Memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 175–184.

[22] M. Harris, "Unified Memory in CUDA 6," Nov. 2013. [Online]. Available: https://devblogs.nvidia.com/unified-memory-in-cuda-6/

[23] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable Memory Transactions," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '05. New York, NY, USA: ACM, 2005, pp. 48–60.

[24] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-free Data Structures," *SIGARCH Comput. Archit. News*, vol. 21, no. 2, pp. 289–300, May 1993.

[25] T. H. Hetherington, M. O'Connor, and T. M. Aamodt, "MemcachedGPU: Scaling-up Scale-out Key-value Stores," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. New York, NY, USA: ACM, 2015, pp. 43–57.

[26] P. Hijma, C. J. Jacobs, R. V. van Nieuwpoort, and H. E. Bal, "Cashmere: Heterogeneous many-core computing," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 135–145.

[27] A. Holey and A. Zhai, "Lightweight Software Transactions on GPUs," in *ICPP*. New York, NY, USA: IEEE, 2014, pp. 461–470.

[28] D. Imbs and M. Raynal, "Virtual World Consistency: A Condition for STM Systems (with a Versatile Protocol with Invisible Read Operations)," *Theor. Comput. Sci.*, vol. 444, pp. 113–127, Jul. 2012.

[29] Intel Corporation, "Desktop 4th Generation Intel Core Processor Family (Revision 028)," Intel Corporation, Tech. Rep., 2015.

[30] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic CPU-GPU communication management and optimization," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 142–151.

[31] F. Ji, H. Lin, and X. Ma, "RSVM: A Region-based Software Virtual Memory for GPU," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13)*. IEEE, 2013, pp. 269–278.

[32] N. Jouppi, "Google supercharges machine learning tasks with TPU custom chip," May 2016. [Online]. Available: https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html

[33] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," in *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, ser. WTEC'94. Berkeley, CA, USA: USENIX Association, 1994, pp. 10–10.

[34] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. Comput. Syst.*, vol. 7, no. 4, pp. 321–359, Nov. 1989.

[35] M. Martin, C. Blundell, and E. Lewis, "Subtleties of Transactional Memory Atomicity Semantics," *IEEE Comput. Archit. Lett.*, vol. 5, no. 2, pp. 17–17, Jul. 2006.

[36] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-processing," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE, 2008, pp. 35–46.

[37] S. Mittal and J. S. Vetter, "A Survey of CPU-GPU Heterogeneous Computing Techniques," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 69:1–69:35, Jul. 2015.

[38] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari, "Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 144–157.

[39] J. Nelson and R. Palmieri, "Don't Forget About Synchronization! A Case Study of K-Means on GPU," in *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PPoPP '10. ACM, 2019, pp. 11–20.

[40] V. Pankratius and A.-R. Adl-Tabatabai, "A Study of Transactional Memory vs. Locks in Practice," in *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '11. New York, NY, USA: ACM, 2011, pp. 43–52.

[41] C. H. Papadimitriou, "The Serializability of Concurrent Database Updates," *J. ACM*, vol. 26, no. 4, pp. 631–653, Oct. 1979.

[42] PCI-SIG, "PCI Express (Peripheral Component Interconnect Express), PCIe Specification," 2019. [Online]. Available: http://pcisig.com/

[43] A. Pellegrini, R. Vitali, and F. Quaglia, "The ROme OpTimistic Simulator: Core Internals and Programming Model," in *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTools '11. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011, pp. 96–98.

[44] S. Peluso, J. Fernandes, P. Romano, F. Quaglia, and L. Rodrigues, "SPECULA: Speculative Replication of Software Transactional Memory," in *Proceedings of the 2012 IEEE 31st Symposium on Reliable Distributed*

Systems, ser. SRDS ’12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 91–100.

[45] T. Riegel, C. Fetzer, and P. Felber, “Time-based Transactional Memory with Scalable Time Bases,” in *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA ’07. New York, NY, USA: ACM, 2007, pp. 221–228.

[46] ——, “Automatic data partitioning in software transactional memories,” in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*. New York, NY, USA: ACM, 2008, pp. 152–159.

[47] P. Romano, R. Palmieri, F. Quaglia, N. Carvalho, and L. Rodrigues, “On Speculative Replication of Transactional Systems,” *J. Comput. Syst. Sci.*, vol. 80, no. 1, pp. 257–276, Feb. 2014.

[48] N. Shavit and D. Touitou, “Software Transactional Memory,” in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC ’95. New York, NY, USA: ACM, 1995, pp. 204–213.

[49] Q. Shen, C. Sharp, W. Blewitt, G. Ushaw, and G. Morgan, “PR-STM: Priority Rule Based Software Transactions for the GPU,” in *Euro-Par 2015: Parallel Processing*. Springer, 2015, pp. 361–372.

[50] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 9th ed. Wiley Publishing, 2012.

[51] The Linux kernel development community, “Heterogeneous Memory Management (HMM),” 2019. [Online]. Available: https://www.kernel.org/doc/html/latest/vm/hmm.html

[52] A. Villegas, A. Navarro, R. Asenjo, and O. Plata, “Lightweight Software Transactions on GPUs,” *Supercomputing*, 2018.

[53] A. Villegas and R. Ubal, “Stretching transactional memory,” in *TRANSACT 2016 - 11th ACM SIGPLAN Workshop on Transactional Computing*. ACM, 2015.

[54] Q. Wang, S. Kulkarni, J. Cavazos, and M. Spear, “A Transactional Memory with Automatic Performance Tuning,” *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 54:1–54:23, Jan. 2012.

[55] Y. Xu, R. Wang, N. Goswami, T. Li, L. Gao, and D. Qian, “Software transactional memory for gpu architectures,” in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. New York, NY, USA: ACM, 2014, p. 1.

[56] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, “Performance Evaluation of Intel® Transactional Synchronization Extensions for High-Performance Computing,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC ’13. New York, NY, USA: ACM, 2013, pp. 19:1–19:11.

[57] Y. Yuan, M. F. Salmi, Y. Huai, K. Wang, R. Lee, and X. Zhang, “Spark-GPU: An accelerated in-memory data processing engine on clusters,” in *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 2016, pp. 273–283.

[58] J. Zeng, J. Barreto, S. Haridi, L. Rodrigues, and P. Romano, “The Future(s) of Transactional Memory,” in *2016 45th International Conference on Parallel Processing (ICPP)*, Aug 2016, pp. 442–451.

[59] Z. Zhong, V. Rychkov, and A. Lastovetsky, “Data partitioning on multicore and multi-GPU platforms using functional performance models,” *IEEE Transactions on Computers*, vol. 64, no. 9, pp. 2506–2518, 2015.