Abstract—Over the course of the last decade, concerns with power consumption and demand for increasing processing power have lead micro-architectures to a highly parallel paradigm. Notably the GPU, now uses a versatile manycore architecture, which, along with its raw computation power, led researchers to use them for general purpose computing.

Whilst these parallel architectures allow for great peak performances, they also raise complex programming problems, especially in regards to synchronizing accesses to shared data. As a result, applications that strive to exploit the parallelism potential of heterogeneous systems equipped with both GPU and CPUs are notoriously difficult to develop.

In this context, Transactional Memory (TM) has been proposed to facilitate concurrent programming. TM is an abstraction that moves the complexity of synchronizing shared data access in multi-threaded systems away to a library.

This work proposes Heterogeneous Transactional Memory (HeterosTM) the first (to the best of the author’s knowledge) TM system capable of supporting concurrent execution of applications that exploit both CPUs and GPUs. In order to minimize synchronization overheads, HeterosTM relies on speculative techniques, which aim to amortize the costs of enforcing consistency among transactions executing on the different units. This system’s architecture is presented and the tested.

Keywords — Transactional Memory; Graphical Processor Units; General Purpose Computing on GPUs; Heterogeneous System;

I. INTRODUCTION

Single-core performance of current Central Processing Unit (CPU)s has reached a plateau, making further improvements very difficult to achieve. Hence, the microprocessor industry moved to multicore architectures as a way to increase raw instruction throughput and allowing greater performances. Nowadays multicore and manycore architectures are a staple in everything from research to consumer grade products.

These new highly parallel architectures have a high potential in terms of performance, theoretically. However, they bring the cost of increased complexity in programming, due to the need for synchronization between multiple threads or machines. Traditionally, for multi-threaded software, locks have been the predominant mechanism used for synchronizing access to shared data among multiple threads. The usage of locks is notoriously challenging, as the programmer needs to keep in mind problems like deadlocks and livelocks, which can be very difficult to avoid for software with unpredictable memory access patterns. On Graphics Processing Unit (GPU)s, the Single Instruction Multiple Thread (SIMT) architecture exacerbates this problem by increasing the number of ways in which a deadlock/livelock can manifest.

Synchronization between CPU and GPU is challenging to implement, as their nature as physically separated devices - in most cases - makes communication between these units costly. Despite these difficulties, as heterogeneous systems are ubiquitous, research on this area is very prolific [1], with some applications already leveraging these systems [2], [3].

As these are non trivial challenges, even for experienced coders, a wide body of research has investigated the idea of using software/hardware libraries to abstract some of the difficulties in developing concurrent applications for these systems. One such example is Transactional Memory (TM) [4], which facilitates shared memory access in multi-threaded systems, by abstracting them to a software library or dedicated hardware unit.

Multiple TM implementations exist for CPUs, but research in this area regarding GPUs is scarce, in spite of the emergence of General Purpose Computation on Graphics Processing Units (GPGPU). However, these solutions do not support sharing of data among threads running on heterogeneous CPU/GPU systems. Thus, they fail to fully explore the computing capability of this class of computing systems.

This work presents Heterogeneous Transactional Memory (HeterosTM), the first (to the best of the author’s knowledge) TM system capable of supporting concurrent execution of applications that exploit both CPUs and GPUs. The intended goal is to facilitate programming of these systems by abstracting the difficulties of data sharing over multiple, physically separated, units via the TM abstraction. Applications running this system are able to share computational power and access to the dataset, transparently, between the CPU and the GPU, through the running of concurrent transactions.

HeterosTM is evaluated using both a synthetic benchmark, Bank, and a real application, MemcachedGPU [5]. Experimental results for Bank show that HeterosTM offers better performance than Nvidia’s transparent transfers with Zero-copy, whilst the results for MemcachedGPU show that it can
outperform single unit performance by about 50%.

This report is structured as follows: on Section II, the research done on the topics TM and GPGPU is presented. The architecture and usage of the HeterosTM system is detailed on Section III, and its experimental evaluation is presented in IV. Finally, some concluding remarks are presented in Section V.

II. RELATED WORK

The switch to multi/manycore processors is a milestone in the history of computing. Whilst these architectures provide amazing potential for performance, they also come at the cost of an increased programming challenge. The biggest difficulty is ensuring synchronization amongst shared data, as in a situation where multiple threads are running, accesses can be made by any of the threads, at any given time. Concurrent accesses to the same memory locations can lead different threads to observe or produce inconsistent results.

An example of these high performance architectures is the GPU. The huge market demand for high-definition 3D graphics has led to a steady increase in processing power. This demand also shaped their architectures, whilst GPUs were originally special purpose accelerators with a very rigid pipeline, they now feature a more versatile manycore architecture.

GPGPU is a relatively recent field of research, which focuses on leveraging GPU’s processing power for non-graphics related tasks. GPUs have proven to be very efficient, especially for highly data-parallel problems where their manycore architecture can be fully exploited. However, the GPU’s highly parallel architecture, exacerbates the challenges of shared data synchronization.

Locks have been the predominant mechanism used to synchronize shared data access in multi threaded systems. Coarse-grained locks provide bad performance but are easy to implement. On the other hand, fine-grained locks provide the best possible performance, but they are difficult to implement correctly, requiring extensive testing.

A. Transactional Memory

TM has been proposed as an alternative to locks [4]. The base premise of this concurrency control mechanism is that accesses to the shared memory can be grouped into a transaction, which is executed as if it was a single atomic operation. The objective is to hide the complexity of synchronization in a TM library that implements this simple abstraction of atomic blocks. As such, the programmer defines his intent - i.e., marking the blocks of code that need to be synchronized, much like marking critical sections of code - instead of having to define the implementation of the synchronization as with traditional locking.

In TM contexts, a transaction is defined as a set of operations that are executed in order and satisfy two properties: atomicity - i.e., if one of the transactional operations fail, all others follow suit - and serializability - i.e., the results produced by a batch of transactions are can be produced by executing them serially, in a given order [4].

Opacity [9] is used as a correctness criterion for TM. Opacity states that all live transaction must see consistent memory states, and no modification made by aborted transactions can be visible to other transactions.

When using TM the programmer marks a set of instructions with an atomic construct or a transaction start/end marker. It is then up to the underlying TM system to ensure correct synchronization among concurrent threads.

When a transaction executes the marked instructions it generates both a read-set (list of addresses it read data from) and a write-set (list of addresses it has written data to). When all operations are concluded, the TM validates the read and write sets, which can lead to either a commit - saving the write set to the main memory - or an abort - reverting all the changes made by the transaction. Validation exists to guarantee correctness between transactions, as they may be modifying the same addresses, causing what is referred to as a conflict.

Despite being initially proposed as a Hardware based solution, TM has seen implementations in Software and in combination of both - named Hybrid Transactional Memory (HyTM). TM has recently gained relevance with hardware vendors, such as Intel and IBM, who have included support for Hardware Transactional Memory (HTM).

To develop HeterosTM, both CPU and GPU implementations of TM were used as the basis for the system. For the CPU implementation, TinySTM [10] was the system of choice. TinySTM is a state-of-the-art Software Transactional Memory (STM), which is highly configurable, allowing it to be very versatile.

B. General Purpose Computing on GPUs

Nvidia’s Compute Unified Device Architecture (CUDA) is a scalable programming model, based on the C programming language [6]. It is the most commonly used programming model for GPGPU, as it features great performance and robust libraries.

A CUDA program starts with a single-threaded function, which is the base unit of execution. This function is called a kernel and it is the code that the GPU runs. Since GPUs use a SIMT execution model, a single thread can not be launched alone, which is why threads are grouped into warps. A warp is a group of 32 threads, which run simultaneously. Conditional branches may cause threads within the same warp to execute different instructions, a phenomenon called thread divergence.

It is important to note that GPUs are hardware accelerators and physically separated devices, and therefore communication between them and the CPU is performed over the PCI Express busses, usually with very limited bandwidth and relatively high latencies.
To counteract this, Nvidia introduced zero-copy. Using zero-copy, the host (CPU) can pin a region of its own memory to be used for the GPU. Using this approach, the programmers see the system as if the GPU’s memory is replaced by the host memory, and all data is transferred to and from the GPU transparently. Similarly, to the system presented in this work, Zero-copy allows both systems to operate on the data concurrently. However, race conditions may happen between devices when using Zero-copy, which hampers its usage.

Regarding the GPU’s memory, some considerations need to be taken by the programmer. As with any multi-thread system, memory races are a problem. To avoid the interference of other threads when executing an instructions coders can use Atomic operations. These operations are conflict-free between multiple threads. However interference from other threads may still occur when accessing the Global Memory, as accesses to this memory’s are weakly-ordered. To guarantee a consistent ordering, coders must make use of Nvidia’s memory fences, which make all writes visible to other threads.

C. Transactional Memory on GPUs

Despite their high theoretical performance, GPU programming is a difficult task. However achieving high efficiency on GPUs is a non-trivial task, as it necessary to adopt specific programming models and extract parallelism at a much finer grade level than that of multi-threaded CPU code. This motivates the usage TM solutions on GPU.

GPU TM implementations already exist, although they are scarce when compared to the number of CPU implementations, as the unordered memory access model of the GPU creates complex challenges for new TM solutions.

For this work the selected GPU TM is PR-STM [11]. In PR-STM each thread is assigned a unique priority, which essentially functions as a static contention manager, and committing is a two stage process, involving acquiring two different locks: the write-lock and the pre-lock. Threads with higher priorities can “steal” pre-locks, from lower priority threads. This means that, on conflict detection when acquiring a lock, at least one thread will continue, avoiding livelocks and deadlocks.

III. HETEROGENEOUS TRANSACTIONAL MEMORY

Building on the previous research on the field of TM, a new TM system, named Heterogeneous Transactional Memory (HeterosTM), is presented in this work. The goal of this system is to be able to transparently schedule a transactional workload across CPU and GPU, maintaining correctness, and achieving a superior performance to a CPU-only or GPU-only solution. All of this, whilst using a programming interface that abstracts the inter-unit synchronization problems away from the programmer, hence greatly reducing the complexity of development.

TM systems are already used to abstract the difficulties of synchronising multiple threads within a single processing unit, however, to the best of the author’s knowledge, no solution exists that extends the convenient TM abstraction beyond the boundaries of CPUs or GPUs. Nonetheless, with the popularity in GPGPU research, the interest in applications that make effective use both CPU and GPU grows [1], [2], [3].

A. Architecture Overview

A large variety of heterogeneous systems exist, however, for the purposes of this work the system’s considered were those comprised of a singe CPU and a single dedicated GPU. Nonetheless, HeterosTM’s design was extended to support multiple-GPUs. To start, the basic, single GPU/CPU design is introduced.

When using HeterosTM, the programmer is responsible only for providing the transactional code and inserting inputs into the system. In turn, the system is responsible for scheduling transactions across the different processing units and ensuring correctness. To improve performance, HeterosTM allows programmers to provide hints on which computational unit (CPU vs GPU) should be used to process transactions.

The likelihood of successfully executing concurrent transactions on heterogeneous units can be strongly affected by the effectiveness of the transactional partitioning scheme. HeterosTM currently adopts a simple hint-based approach, which allows programmers to exploit application/domain knowledge to take informed decisions on which computational unit to use for executing their transaction.

HeterosTM’s execution model involves switching between two states: Execution and Synchronization. The system alternates between the two, either running a transactional workload in the Execution state, or running system error checking, in the Synchronization state.

Synchronization amongst threads running on the CPU and threads running on the GPU occurs in a lazy fashion, when the set of transactions to be executed on the GPU is completed. Once synchronization triggers the HeterosTM system uses transaction logs, which contain the cumulative CPU write-sets, to compare the memory accesses amongst units to ensure no inter-unit conflict has occurred. Comparison is done by searching for intersections between a unit’s write set, and the others read-sets, which serializes the former before the latter.

In case of successful synchronization, with no conflicts detected, both units’ datasets are updated with each other’s produced results. When any conflict is detected among the sets of transactions, HeterosTM decides which batch of transactions to commit, causing the other unit to drop its results and rerun with new ones.

The system’s initial design goal was to use the GPU as an accelerator. Based on this vision, only the CPU’s commits
are preserved in case of conflict. This design decision was motivated by CPU’s capability to expose results to the user/outside world, in which case late aborts due to unit synchronization are undesirable.

However, when transactional workloads are highly parallel, the GPU’s performance can dwarf the CPU’s. Hence, always giving priority to the CPU in case of inter-unit conflicts, may be largely suboptimal performance wise and may expose the transactions scheduled for processing GPU to the risk of starvation. To cope with this issue, HeterosTM supports different Conflict Resolution (CR) policies, which allow optimizing the system’s behaviour to maximize different objective functions (e.g., throughput vs fairness).

Multi-GPU: As mentioned, the focus of this work is on investigating and evaluating heterogeneous CPU-GPU systems hosting a single GPU. Yet, the base design of HeterosTM has been extended also to support concurrent execution on a set of GPU(s).

When using multiple GPUs, each GPU receives its own batch of transactions, which are launched concurrently. When the units conclude execution of all the transactions in their batches, they notify the CPU, which starts to send its write-set for conflict detection. Each GPU does this comparison, so that can commit independently.

If comparison with CPU’s logs succeeds, the units lock access to the CPU temporarily, to commit their write sets in sequence. The GPU commit order is established dynamically, according to which GPU finishes its comparison with the CPU first. Committing GPUs must also validate against the successfully committed preceding GPUs’ write-sets.

Scheduling: When programmers submit transactions for execution in the HeterosTM system, these are dispatched towards a set of queues, from where the processing units fetch their inputs. HeterosTM maintains a set of queues per each transaction type that can be executed in the system. The system keeps these sets of queues per-unit, from where each processing unit fetches its transactions, and also keeps an additional shared queue for non-attributed transactions.

For each transaction, the queue stores, the inputs, the results produced, and if they have already been run successfully or not. Ideally the queues for each unit are partitioned perfectly from the others, transactions may be moved to another unit’s queue for load balancing amongst units.

B. Programming Interface

To use HeterosTM the programmer needs to first define the transactional inputs, and then the transactional functions and kernels to be ran. HeterosTM’s Application Programming Interface (API) is composed by 4 core primitives:

- \texttt{REGISTER\_TRANSACTION( )}: This function is used to register a transactional code block in the HeterosTM system. The programmer has to provide pointers to the functions to be executed, for both CPU and GPU. This primitive also requires a unique id, as an input to identify each CPU/GPU transaction pair, variable types inputs and outputs of these transactions.
- \texttt{START( )}: Called to start the execution of the HeterosTM system. The arguments for this primitive are the thread counts for the various units, the number of GPUs to use and the memory to be shared by the CPU and GPU. This primitive is also responsible for allocating the datasets’ memory and the queues for the registered transactions.
- \texttt{SUBMIT( )}: Used to request the execution of a transaction of the chosen type, and to specify its input arguments (if any), as well as the user provided hint, of which computational unit to use to execute the transaction. The hint may also be empty, placing the transaction in the shared queue. This call returns the unique ID of the scheduled transaction.
- \texttt{GET\_RESULT( )}: This primitive is used to fetch the produced results of the provided transaction ID, if any exist.

When using HeterosTM, these primitives are essentially called in order. The programmer inserts his transactions into the system with \texttt{REGISTER\_TRANSACTION( )} primitive, and then starts execution by calling \texttt{START( )}. \texttt{START( )} launches the CPU threads, initializes the GPU(s) with the data needed for execution and builds the necessary queues.

When the \texttt{START( )} has been successfully called, the user is free to submit transactions via \texttt{SUBMIT( )}. Finally, to retrieve these transactions’ return values the user call \texttt{GET\_RESULT( )}.

Every time a new transaction is submitted, the system dispatches it to the computational unit specified by the programmer-provided hint. This helps mitigating inter-unit contention, but may lead to load unbalancing scenarios, where one unit keeps getting all the inputs while others starve. To cope with this issue, in case all the queues of a unit are empty, the unit can consume transactions from the shared queue, or from the queues of other units.

C. Integrating TinySTM and PR-STM in HeterosTM

To implement TM for each of the units, HeterosTM includes two state-of-the-art STM implementations: TinySTM for the CPU, and PR-STM for the GPU. In order to support HeterosTM speculative execution model, both Tiny and PR STM had to undergo some lightweight and unobtrusive modifications.

Like most TM systems, both Tiny and PR maintain read and write sets of transaction for commit time validation. However, these logs are only temporary and are discarded after the transaction’s completion.

In the proposed system, though, these logs must be kept until a inter-unit synchronization is triggered, due to the need of detecting conflicts among transactions executing in different computational units.
Changes to TinySTM: The CPU is always synchronized before the GPU, therefore, it only needs to save its own write-set.

To minimize impact on performance it was decided that the system maintains distinct per CPU thread logs. Additionally, space for the logs is preallocated, which decreases the time needed to store the write/read sets. With this base design two variations of a CPU logging systems were developed:

- **Address Log**: In this case, each entry of the log stores only the memory address of the position accessed. This saves the minimum amount of information possible for a comparison, and therefore has the smallest possible footprint, facilitating its transfer to the GPU.
- **Versioned Log**: In this case, beside address, the log also stores the value written and the current value of TinySTM’s global time-stamp.

Changes to PR-STM: The GPU TM also requires modifications, with the goal of storing transactional logs.

An important consideration when developing for the GPU is that dynamic memory allocation inside a kernel has a huge negative impact on performance, and is considered a bad practice. Therefore all GPU logs should be designed using some form of statically allocated log.

GPU log designs expect that some memory locality exists for these reads and writes, which will improve performance by using caching and coalesced accesses more efficiently.

On the GPU side, HeterosTM gathers logs for the read-sets and write-sets of committed transactions, supporting two different approaches:

- **Explicit log**: A per-thread explicit log of all the memory positions accessed. Threads store only the address of the accessed memory positions, in the GPU’s global memory and logs are kept per-thread.
- **Compressed log**: A global data structure, functioning as bitmap of the accessed memory positions. This log’s structure uses bytes, instead of bits like in regular bitmaps, so that it avoids reading it to update it. The logs for read-sets and write-sets are laid out in memory as two contiguous memory words, which indicate if that address was read or written, respectively.

D. Synchronization

Synchronization is one of the two systems states for HeterosTM, where the transaction batches processed by CPU and GPU are validated in order to detect whether they developed any conflict. The former case requires aborting one of the batches. In the latter case, instead, the write-sets produced by the two computation units must be merged in order to ensure that both units reach a common state.

This state is split into 4 consecutive phases. Setup, is the first phase and is used to lock the CPU and prepare for the log comparison. The Validation phase happens next. During this phase the CPU sends its logs to the GPU, which compares them with its own logs, to detect if the CPU’s writes intersect with its reads, transferring back the outcome of the validations to the CPU. The Validation phase ends when either all comparisons are finished or one of them detects a conflict, and the system enters the Resolution phase. When an conflict is detected this phase is used to decide on which unit should get priority, otherwise when no conflict is detected both units’ transactional batches can be committed, and have to be applied to both units. When the Resolution phase finishes, the units chosen to do inter-unit commits do so during the Commit phase.

Logs are compared on the GPU, as this is a fully parallel workload. Since the comparison is also a workload that does not need to be executed in order, it also allows the usage of CUDA Streams [6], which do not ensure any sort of ordering, to run overlapping comparison kernels. Since HeterosTM’s CPU logs are per-thread, each thread is responsible for launching comparison with its logs on the GPU, when notified by the CPU’s control thread.

To avoid excessive non-log related memory transfers between units, the Validation phase is also used to update the GPU with the CPU’s write-set, which is done in parallel CPU’s log comparison.

Using the same type of fine-grained memory transfers to update the CPU’s dataset with the GPU’s produced results is not efficient in the CUDA programming model, as it requires dynamically allocated memory. To cut unnecessary transfers, HeterosTM divides its dataset into chunks of 128KB and transfers only the updated chunks by consulting the GPU’s write-set.

When using the Address log, the approach of comparing and applying requires that the CPU’s dataset is transferred to the GPU before Validation. Transferring the whole dataset is very costly, and scales poorly with increased dataset size. This can be avoided by using the Versioned Log as it already contains all the necessary information to run comparison and apply the write sets, and does not require the initial memory image transfer.

If a conflict between units is detected, the Conflict Resolution phase is used to decide on which unit should be allowed to commit. HeterosTM implements three different Conflict Resolution techniques, which are presented in Section III-F. The unit which maintains its results after the Conflict Resolution overrides the others dataset with its own during the Commit phase, therefore deleting results produced by the other.

E. Validation

The validation is the most resource intensive phase of Synchronization, as it is comprised multiple memory transfers, both to and from the GPU. It is also very important for HeterosTM that conflicts are properly detected, so that it can ensure opacity and correctness. Below are presented
the algorithms used by HeterosTM for the Validation phase, starting with the CPU’s.

CPU Side: HeterosTM uses two types of CPU threads. The Worker Threads are the ones executing the transactional workload. On the other hand, the Control Thread is the one responsible for interfacing between the GPU and CPU Worker Threads.

When the Synchronization phase is triggered, the CPU Worker threads block, while the Control Thread is carrying out the memory transfers in the Setup phase. After the setup HeterosTM enters the Validation phase. During this phase the Control Thread is idle, whilst the Worker Threads are queueing comparisons.

Worker Threads queue Comparison Streams, and then wait until either they all conclude or one of them detects an error. Upon error detection, or full log comparison conclusion, the Worker Threads are blocked, whilst the Controller Thread runs through the remaining phases for synchronization.

On the other hand, Non-Blocking Synchronization, only blocks the worker threads at the end of validation phase, i.e., more precisely during the validation of the last comparison stream. In this algorithm after queueing a comparison stream, the Worker Threads return to executing transactions, instead of idling while waiting for the GPU to finish the batch, only locking when the remaining logs are small.

GPU Side: Log comparison is an ideal workload for the GPU’s architecture, since it is comprised of a large amount of tasks, which may be executed fully in parallel and in any order. The design of HeterosTM, with logs being inherently partitioned both by being per thread and by being a linked list, also favours the usage of Streams, as comparison can be done in any order, unaffected by the GPU’s weak ordering.

Comparison for the both logs is very straightforward. The kernel, is launched with a number of threads equal to the size of the log it is comparing. Each thread reads from one element of the CPU log, and then searches the GPU’s log for that very same memory address. If it detects an conflict, the unit sets a flag in global memory to 1.

The kernel for the Versioned Log, adds version checking, which was added to allow the log entries to the GPU dataset. The version checking is what allows the Versioned Log comparison to be executed out-of-order while still ensuring that by the end of the execution we have the most updated values applied to the GPU’s dataset.

These extra operations mean that comparing the Versioned log is marginally slower than the Address log. However for large datasets, transferring the full memory image is a very large overhead, which may benefit Versioned over Address log.

As each logging solution favours a certain workload, HeterosTM implements both, leaving it to the user to decide which is more appropriate for his application.

F. Conflict Resolution

When a conflict is detected the system must decide how to proceed, by selecting which units get to keep their commits and which will be overridden. HeterosTM implements three different Conflict Resolution (CR) policies:

- GPU Invalidation: In this configuration the CPU is seen as the master copy, and always keeps its commits no matter what. This may lead to GPU starvation, however, it allows CPU commits to be externalized immediately.
- Favour Fastest: A throughput oriented CR. With this configuration, the system compares the commits produced by the units since the last Synchronization, and the unit with the most commits wins.
- Balanced CR: Focused on avoiding starvation, this CR ensures that both the GPU and CPU both get to commit a configurable ratio of batches of transactions.

IV. EXPERIMENTAL EVALUATION

To evaluate the performance of HeterosTM, a prototype was developed and tested. The results of these tests are presented in this chapter. Due to time constraints, the full HeterosTM API was not implemented, although all the underlying synchronization and scheduling algorithms which power this API were developed, with two exceptions: load balancing between units, and load balancing inside the units, between the different queues.

HeterosTM was two tested with two benchmarks:

- Bank: a synthetic benchmark provided by TinySTM, which simulates the bank transactions by transferring money between accounts.
- MemcachedGPU: part of GPU version of Memcached [5], a popular distributed memory object caching system adopted by many popular web-services, such as Facebook or Twitter.

All tests on this chapter, unless specifically mentioned, were run on a machine with an Intel Core i7-5960X processor, and two Nvidia Geforce GTX980 GPUs (GM204 micro architecture), connected to the CPU with a PCI-Express 3.0 bus.

A. Bank

For the Bank benchmark only the synchronization algorithms were studied, as the transaction queues were not implemented. Instead each unit generated its own transactional inputs locally.

HeterosTM’ Bank tests analysed the following parameters: size of the dataset, synchronization frequency and inter-unit conflicts. These tests were run with both types of CPU logs, the Favour Faster CR and using the Blocking Synchronization Algorithm.
Dataset Size: The dataset size influences the intra contention and the cost of synchronization when using the Address Log. To evaluate how performance is affected by this factor, a scenario with no inter contention was used. Fig 1 presents the results of this test.

Peak performance for the combined system is hit when the GPU hits its peak performance point, as the CPU’s performance is mostly independent of dataset size, and this dataset is large enough that contention is not influential.

As expected for small datasets, transferring the whole memory image is not costly, the Address log out performs the heavier memory transfers of the Versioned Log. On the other, for the large datasets, the Address log’s performance decay is noticeably higher and the Versioned proves to be the better solution.

Synchronization Frequency: When using a speculative model, the frequency of error checking is a key tuning parameter. If synchronization is too frequent, the overhead incurred with the constant stopping becomes too great and performance decays. On the other hand, synchronizing too late, may lead to a lot of wasted work.

To establish an ideal frequency of synchronization, Bank was tested with varying GPU work batch sizes as well different levels of overlapping between the two unit’s partitions. Fig. 2 presents the results of the test.

Analysing these results, it is observable that HeterosTM outperforms CPU/GPU only solutions for any inter contention levels, as long as the batch sizes are small. Its peak performance is close to the combined performance’s of the two involved units. As expected, when batches start to get larger both the probability and the cost of conflicts increase as the units produce more results between each synchronization.

In ideal scenarios, this means that the developed system is competitive even with fine-grained solutions, despite its generic design.

Since for this test the CR of choice was the Favour Faster CR, in the worst scenarios were synchronization fails repeatedly, performance degrades to a level slightly inferior than the one obtained with the GPU only solution, as this is the unit with the highest throughput.

However the this CR assumes CPU invalidation, which may not be desirable for some workloads. Running the same test using the static GPU invalidation the results will instead degrade to CPU only performance.

Both these CR algorithms may lead to starvation of one of the units. To counter this HeterosTM presents the Balanced CR, which tries to balance the commits between the two units according to factors set by the programmer. For testing purposes, Balanced CR is configured to block CPU commits for every failed GPU commit, and repeating the test for the medium inter-contention scenario the results in Fig. 3 are obtained.

Observing these results, it is clear that whilst performance for this configuration of Balanced CR falls short for higher contention scenarios. However, looking at Fig. 3b), it can be concluded that the goal of balancing commits between both units is achieved.

Comparison with Zero-copy: Zero-copy was presented as having similar functionality to the system herein proposed. Zero-copy functions in a very different way to HeterosTM. When the CPU accesses memory which is currently being accessed on the GPU, the Nvidia API generates a page-fault and transfers the value. This imposes a much higher overhead than HeterosTM’s speculative scheme, as it constantly saturates the PCI-Express connection between the CPU and GPU with small, inefficient, memory transfers.

To compare them HeterosTM was converted to use Zero-copy, for Synchronization only, where transfers are bigger and more efficient. The test ran shows that using Zero-copy decreases performance by up 20% when compared HeterosTM, as expected, since lots of coarse-grained transfers are not as efficient as large burst data transfers.

B. MemcachedGPU

The second benchmark ran to profile HeterosTM is an adaptation of the MemcachedGPU algorithm, by Hetherington et al [5]. This algorithm is part of a GPU conversion of Memcached, a distributed, in-memory, object caching system, used in several web applications to reduce database loads. Memcached has also been converted to use TM [12].

To refer the objects its caching, Memcached uses a key-value store schema. In the original CPU algorithm key-
value pairs are stored in hash map. This approach requires dynamic memory allocation, therefore to adapt it to GPU it was necessary to make some compromises. Instead of a hash map, MemcachedGPU uses a structure similar to an hash table, with a fixed number of entries per hash value, in a model that is more akin to that of a fully-associative cache with multiple ways, with evictions used when all entries are full.

MemcachedGPU supports two types of requests: GET requests, which look up values in the hash table and are read-only transactions, and SET requests, which update the hash table with new key-value pairs.

For the purposes of testing HeterosTM, the algorithms concerning both types of requests were implemented for both units, each using their respective TMs. Requests were partitioned between the two units using hash algorithms.

For these tests, a more complete version, when compared to Bank, of HeterosTM was implemented, making use of unit queues and transactional inputs. As the network components of Memcached were considered to be out of scope, the system is initialized with a large number of already queued transactional inputs. Deciding between the different transaction types is done by a con-flip at every queue fetch.

To ensure that request processing was meeting latency constraints, the system was first tested as a single unit solution. The results of this test are represented on Fig. 4.

From the observation of Fig. 4a) the behaviour of the GET kernel stands out as being the most unexpected result. This effect happens when the thread usage of the kernel exceeds that of the CPU’s registers, forcing them to be spilled to global memory. Additionally, it can also be seen that for this workload CPU and GPU are much closer in terms of performance, as these are very small batches which do not favour the GPU, and the CPU does not require memory transfers for its work.

**Evaluating HeTM:** To test HeterosTM, the GPU was configured to use roughly 16k threads (128 blocks of 128 threads each), in order to hit a processing latency of around 1 ms. The workload was configured using the data from [13] as the basis, with 0.1% of SET requests, and the rest of the requests as GET requests. To simulate Load Balancing or errors in partitioning, a varying percentage of accesses to the shared queue was made. As this is a workload were writes are very sparse, and the dataset very large, the Versioned Log along with the Non-blocking algorithm were used.

The results of these test, observable in Fig. 5 a), show that HeterosTM is able to outperform single unit solutions in scenarios were the partitions do not overlap significantly. However, unlike Bank in this scenario performance does not degrade so gracefully, and high contention scenarios have roughly half of the throughput of the single-unit performance. This due to two factors: necessity to transfer results, and unfavourable workload configuration for the GPU.

Unlike Bank, Memcached requires transactions to have a return value: the response packet. This increases the number of memory transfers significantly, which in turn increases the overhead of synchronization.

Adding to this, Memcached’s workload, despite being highly parallel, is not very well catered to run on GPUs, due to the time requirements forcing packets to be processed in small batches were the overheads of the transfers are rather large. In Fig.5 b), a much run with doubled batch
size presents much better peak performance, whilst halving the workload increases resistance to contention.

**Multi GPU:** Following the tests with a single GPU, HeteroSTM was tested using the extended multi-GPU design. This test was run using the same configurations as the previous Memcached HeteroSTM tests: 128 blocks of 128 threads and 0.1% of SET requests, using two GPUs. The results of this test are presented in Fig. 6.

These results point to HeteroSTM being more favourable when using multiple GPU instead of a single one. The reality is that, the multi-GPU algorithm causes much longer synchronizations, and the CPU spends much more time locked. This essentially causes faster per-unit comparison despite the overall comparison time being longer. The brunt of the results are then produced by GPUs, which are rarely aborted by the CPU, and compete only amongst themselves.

Whist it is intuitive to expect that inter-GPU conflicts should have a higher impact on performance, the use of GPU direct to transfer logs from unit to unit, along with the smaller size of these logs, makes GPU comparison more efficient, with conflicts being more readily tossed aside instead of necessitating to parse several logs as is the case with the CPU comparison.

**V. Conclusion**

This work presents Heterogeneous Transactional Memory (HeteroSTM), a system which simplifies programming for heterogeneous by allowing concurrent accesses to same memory, by multiple, physically split, units. To achieve this the system applies a speculative model, were units work separately on the data, and periodically check to see if they’ve committed any overlapping accesses.

Users of the system, code for each of the units using transactional code, and establish inputs and outputs for these functions. The user registers these functions into HeteroSTM, and then starts it with the desired characteristics. Having started the system, the user inserts transactional inputs into the HeteroSTM system, along with the an id of the unit where it expected to run. The system uses this id, along with internal considerations, such as load balancing, to
schedule and run the transaction in one of the available units. Transactions run speculatively, and the system ensures correctness.

In order to develop a more flexible system, the specific algorithms for partitioning are left to the programmer, as a generic solution is very difficult to design, and was considered to be out of scope. This harms the system’s usability by requiring more effort from the programmer. [14], [15] have developed systems partitioning of transactional code, which would be an interesting solution to avoid user input on partitioning.

Finally, for the sake of development, heterogeneous systems for this system were considered as CPU plus dedicated GPU, inter-linked through a PCIe connection. This a reduction view, and HeterosTM could be expanded to cover more types of heterogeneous system, such as those with integrated GPUs.

Still, the results obtained when testing the system, prove that the proposed design achieves the goals set for it, after some application specific tuning. This means that the goal of facilitating programming was not fully achieved, however, abstracting the difficulties of synchronizing the different units is still a significant gain. In contrast, the performance and correctness goal were entirely achieved, proving that this system can succeed.

REFERENCES


