



UNIVERSIDAD
DE MÁLAGA

Departamento de Arquitectura de Computadores

TESIS DOCTORAL

Transactional Memory on Heterogeneous Architectures

Alejandro Villegas Fernández

Marzo de 2018


Dirigida por:
Dr. Óscar Plata González
Dr. Rafael Asenjo Plaza





UNIVERSIDAD
DE MÁLAGA

AUTOR: Alejandro Villegas Fernández

 <http://orcid.org/0000-0002-0269-2640>

EDITA: Publicaciones y Divulgación Científica. Universidad de Málaga



Esta obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial-SinObraDerivada 4.0 Internacional:

<http://creativecommons.org/licenses/by-nc-nd/4.0/legalcode>

Cualquier parte de esta obra se puede reproducir sin autorización pero con el reconocimiento y atribución de los autores.

No se puede hacer uso comercial de la obra y no se puede alterar, transformar o hacer obras derivadas.

Esta Tesis Doctoral está depositada en el Repositorio Institucional de la Universidad de Málaga (RIUMA): riuma.uma.es



Dr. D. Óscar Plata González.
Catedrático del Departamento de Ar-
quitectura de Computadores de la Uni-
versidad de Málaga.

Dr. D. Rafael Asenjo Plaza.
Catedrático del Departamento de Ar-
quitectura de Computadores de la Uni-
versidad de Málaga.

CERTIFICAN:

Que la memoria titulada “Transactional Memory on Heterogeneous Archi-
tectures”, ha sido realizada por D. Alejandro Villegas Fernández bajo nuestra
dirección en el Departamento de Arquitectura de Computadores de la Universi-
dad de Málaga y constituye la Tesis que presenta para optar al grado de Doctor
en Tecnologías Informáticas.

Málaga, Marzo de 2018

Dr. D. Óscar Plata González.
Codirector de la tesis.

Dr. D. Rafael Asenjo Plaza.
Codirector de la tesis.



UNIVERSIDAD
DE MÁLAGA

A Ana, a mi familia, amigos y compañeros



UNIVERSIDAD
DE MÁLAGA

Agradecimientos

Durante estos años de trabajo he tenido el placer de convivir con muchas personas que, de una forma u otra, han formado parte de esta tesis y a las que me gustaría agradecer que hayan estado junto a mí en este periodo.

En primer lugar, agradecer a mis directores Óscar y Rafa, y a mi tutora M. Ángeles todo su apoyo y trabajo durante este tiempo. Sin duda han sido un modelo a seguir en cuanto a dedicación, capacidad de trabajo, confianza y otras muchas cualidades y, sin su aportación y sabiduría, esta memoria y mi trabajo no habrían sido posibles. Gracias.

Este agradecimiento me gustaría extenderlo a mis compañeros de laboratorio y fatigas (sin ningún orden específico): Cervilla, Pedrero, Herruzo, Dr. Q, Denisa, Vilches, Mangel, Alex, Óscar, Arjona, Esteban, Sergio, Fran, A. Lázaro, Carlos, Bernabé, Andrés, Sergio, José Carlos, Andrés e Iván... y espero no olvidarme de ninguno. También quiero agradecer su ayuda, soporte y compañía al resto de profesores (especialmente a aquellos con los que he colaborado en docencia y durante la organización de las Jornadas): gracias por cada una de las cosas que he podido aprender trabajando junto a vosotros. Indudablemente tengo que dedicar unas palabras a los técnicos de laboratorio y a Carmen. Tampoco habría podido llegar hasta este punto sin vuestra inestimable ayuda.

I would also like to thank Prof. David Kaeli and the people I met while in Northeastern: Rafa, Xiangyu, Yash, Julian, Fanny, Chen, Fritz, Xian, Shi, Kabi, Charu, Leiming, Saoni, Xun, Yifan, Amir, Paula... I am really happy we met and I really want to see you soon again. Also I want to thank Angela and Nevada for making me feel like home during my stay in Boston.

Quiero agradecer a Ana su apoyo durante todo este tiempo, especialmente por soportar mis largas jornadas de trabajo, mis estancias y mi mal humor :). Igualmente, agradezco todo el apoyo que he recibido por parte de familiares y amigos que, sin duda, me han dado fuerza para continuar con mi trabajo.

Finalmente, mencionar las fuentes de financiación que han permitido que pueda llevar a cabo esta tesis y las estancias: los proyectos TIN2013-42253-P y TIN2016-80920-R del Gobierno de España, P11-TIC8144 y P12-TIC1470 de la Junta de Andalucía, y al plan propio de la Universidad de Málaga. Agradecer a la red HiPEAC el apoyo financiero para realizar una de mis estancias.

Abstract

Nowadays, heterogeneous architectures are populating the world in different forms: from heterogeneous CPUs able to focus on performance or efficiency, to GPU accelerators joining multi-core CPUs within the same chip, to Systems on Chip that integrate DSPs, FPGAs and many other types of processor in the same area. These architectures enable programs to be decomposed and deployed on different pieces of hardware, seeking for energy efficiency and performance. This is possible thanks to technologies like OpenCL, that allow programmers to write code to be deployed across different platforms. However, the synchronization primitives available in these languages are very simple: they are restricted to kernel launches or, at most, the use of atomic operations or signals. The construction of more advanced synchronization mechanisms is a complex task delegated to programmers.

In the multi-core CPU world, Transactional Memory (TM) has emerged as an effective technique to provide synchronization in form of mutual exclusion to implement critical sections. In contrast to other lock-based mutual exclusion mechanisms, TM encloses critical sections in transactions that are executed in parallel in a speculative way. Conflict detection and version management mechanisms ensure that different transactions do not modify the same memory objects, ensuring mutual exclusion. If two transactions access the same memory object and, at least, one of the accesses is a write, one of the transactions signals a conflict and aborts, discarding its speculative changes and restoring memory to a consistent state. The goal of TM is to provide a simple programming interface to determine the bounds of the transaction and, thanks to its speculative nature, provide high performance.

TM is becoming popular in multi-core CPUs (in fact, CPU vendors such as Intel and IBM incorporate TM in their latest processors) and the research community is investigating on how to integrate this technology in GPUs. As the industry is already creating heterogeneous processors it is important to under-

stand the possibilities, impact, and tradeoffs that TM can have on such architectures. This is not a simple task: TM can be deployed as a software, hardware or hybrid solution, and scheduling transactions on the proper device can have an important impact in performance and energy consumption. In addition, as mentioned before, currently there exists multiple heterogeneous devices and the number and variety is increasing. Thus, TM and the heterogeneous architectures can be analyzed from different points of view.

In this thesis we tackle this problem from different perspectives. Firstly, we analyze an existing software TM solution on an heterogeneous CPU of the ARM big.LITTLE architecture, featuring separate performance-oriented and efficiency-oriented cores. In addition, we propose a scheduling technique to analyze the impact the scheduling can have in TM-instrumented applications. Our experiments have been performed using the ODROID XU3 device which integrates a big.LITTLE processor and grants access to power monitors able to provide precise energy measurements. Secondly, we propose a novel software TM design targeting heterogeneous CPU+GPU processors. The goal is to create a TM system adapted to the architectural particularities of each device, while providing an efficient communication mechanism among them. This proposal has been tested in an AMD Kaveri heterogeneous processor implementing advanced HSA (Heterogeneous Systems Architecture) features. Lastly, we target GPUs as accelerator for heterogeneous systems, proposing a hardware TM design to execute transactions on such device with high efficiency and at a low hardware cost. This design has been evaluated via simulation using the Multi2Sim Simulation Framework.

Contents

Agradecimientos	i
Abstract	iii
Contents	viii
List of Figures	x
List of Tables	xi
1.- Introduction	1
1.1 Heterogeneous Architectures	2
1.2 Complexities of parallel and heterogeneous programming	4
1.3 Transactional Memory	8
1.4 Thesis motivation and research questions	8
2.- Background and Related Work	11
2.1 GPU accelerators	11
2.1.1 GPGPU programming	11
2.1.2 GPU architecture	13
2.1.2.1 Memory architecture	15
2.1.2.2 Execution model	16



2.2	Heterogeneous CPU+GPU processors	16
2.3	Heterogeneous CPUs	17
2.4	Transactional Memory	18
2.5	TM on GPUs	20
2.5.1	Software TM on GPUs	20
2.5.2	Hardware TM on GPUs	22
2.6	TM on low-power CPUs	24
3.-	Transactional Memory on Heterogeneous CPUs	25
3.1	Background: TinySTM and STAMP	26
3.2	The ODROID XU3 Platform	27
3.3	Isolated Energy/Performance evaluation	28
3.3.1	Little cluster analysis	29
3.3.2	Big cluster analysis	29
3.3.3	Full system analysis	30
3.3.4	Little cluster and big cluster comparison	33
3.3.5	Conclusions	33
3.4	Concurrent execution of TM applications on heterogeneous CPUs .	33
3.5	Scheduling TM applications on heterogeneous CPUs	37
3.5.1	Scheduling on heterogeneous CPUs	37
3.5.2	ScHeTM: A TM-aware scheduler for heterogeneous CPUs .	38
3.5.2.1	Design phase: suitability functions	40
3.5.2.2	Scheduling phase: application execution	42
3.5.2.3	Evaluation of ScHeTM	43
	Baseline scheduler.	44
	Balanced ScHeTM.	44
	Performance-oriented ScHeTM.	45
	Efficiency-oriented ScHeTM.	46

Transaction-oriented ScHeTM.	46
3.5.2.4 Conclusions	47
3.5.3 TM on heterogeneous CPUs: conclusions and future work	48
4.- Transactional Memory on Heterogeneous CPU+GPU processors	51
4.1 Background: NOrec and GPU-STM	52
4.2 APUTM	53
4.2.1 Transactional Metadata	53
4.2.2 Version Management	56
4.2.3 Conflict Detection	56
4.2.4 Misellanea	58
4.2.5 Execution example	58
4.2.6 Read-Modify-Write transactions	61
4.3 Evaluation	63
4.3.1 Experimental setup	63
4.3.2 APUTM characterization	63
4.3.3 Application evaluation	67
4.4 APUTM: conclusions and future work	70
5.- Transactional Memory in GPU Local Memory	73
5.1 Motivating example	74
5.2 GPU-LocalTM Design	75
5.2.1 Transactional SIMT Execution	77
5.2.2 Forward Progress	79
5.2.3 Version Management	81
Register checkpointing.	84
5.2.4 Conflict Detection	85
5.2.4.1 Directory-based Conflict Detection (DCD)	85

5.2.4.2	Shared-Modified DCD (SMDCD)	86
5.2.4.3	DCD and Private Read/Write Signatures (pRWsig)	87
5.2.4.4	DCD/pRWsig and Shared Write-Only Signatures (sWOSig)	89
5.3	GPU-LocalTM Modeling	91
5.4	Evaluation	94
5.4.1	Performance Evaluation	95
5.5	Improving the serialization mechanism	100
5.5.1	Work-item selection mechanism evaluation	103
5.6	GPU-LocalTM: conclusions and future work	106
6.-	Conclusions	109
6.1	Future work	113
	Appendices	115
A.-	Resumen en español	115
A.1	Motivación y cuestiones de investigación	116
A.2	Memoria transaccional en CPUs heterogéneas	118
A.3	Memoria transaccional en procesadores APU	121
A.4	Memoria transaccional en memoria local de GPU	123
A.5	Conclusiones	130
	Bibliography	133

List of Figures

1.1	Coarse-grained locks in CPU and GPU.	6
2.1	Block diagram of the AMD HD Radeon 7970 GPU.	14
2.2	Conditionals in the SIMT execution model.	16
2.3	Samsung Exynos 5422.	18
3.1	Exynos 5422 processor featured in the ODROID platform.	28
3.2	Little cluster evaluation.	30
3.3	Big cluster evaluation.	31
3.4	Big + little evaluation.	32
3.5	Big cluster vs little cluster evaluation.	34
3.6	Diagram of the proposed scheduler ScHeTM.	39
3.7	Evaluation of the greedy scheduler.	45
3.8	Evaluation of ScHeTM using a balanced configuration.	45
3.9	Evaluation of ScHeTM using a performance-oriented configuration.	46
3.10	Evaluation of ScHeTM using an efficiency-oriented configuration.	47
3.11	Evaluation of ScHeTM using a transaction-oriented configuration.	47
4.1	Metadata required to implement APUTM.	53
4.2	Functions provided by the APUTM interface.	54
4.3	Auxiliary functions in APUTM.	55
4.4	Program example using APUTM	59
4.5	Unified log implementation of APUTM.	62



4.6	Characterization of APUTM when varying the number of accessed memory positions.	64
4.7	Characterization of APUTM for different probabilities of conflict.	65
4.8	CPU and GPU abort analysis of APUTM	66
4.9	Evaluation of a hash table implemented using APUTM.	67
4.10	Evaluation of Bank implemented using APUTM.	68
4.11	Evaluation of Genetic implemented using APUTM.	69
5.1	Baseline GPU architecture: AMD's Southern Islands.	77
5.2	Transactional and serialization execution modes	80
5.3	Version management and register checkpointing.	81
5.4	Shadow memory organization.	83
5.5	Signature design (local memory size is per work-group).	88
5.6	Speedup of TM and FGL benchmarks.	96
5.7	Normalized execution breakdown.	97
5.8	Instructions executed within transactions.	98
5.9	Commit ratio.	98
5.10	False positives.	99
5.11	Transactional and serialization execution modes.	99
5.12	Multiple work-item serialization mechanism.	101
5.13	Descending work-item serialization mechanism.	102
5.14	Speedup for the different selection schemes.	104
5.15	Execution breakdown for the selection mechanisms.	105
5.16	Transactional execution modes for the selection mechanisms.	105
A.1	Samsung Exynos 5422.	119
A.2	Diagrama del planificador ScHeTM.	120
A.3	Metadatos necesarios para implementar APUTM.	122
A.4	Recursos de memoria alojados por GPU-LocalTM.	125
A.5	Modos de ejecución transaccional en GPU-LocalTM	127

List of Tables

3.1	STAMP applications.	27
3.2	Execution time for the concurrent evaluation on big.LITTLE . . .	36
3.3	Application behavior on big.LITTLE.	36
3.4	Calculation of $F_c(N)$ for a sample application.	41
3.5	Calculation of the suitability function	42
4.1	Example of an execution using APUTM.	60
5.1	Example of transactional SIMT execution.	79
5.2	Example of a transactional execution.	82
5.3	DCD and SMDCD mechanisms.	87
5.4	pRWsig conflict detection mechanism.	88
5.5	pRWsig + sWOSig conflict detection mechanism.	90
5.6	AMD's Southern Islands GPU implementation on Multi2Sim 4.2. .	92
5.7	Benchmarks used to evaluate GPU-LocalTM.	95
5.8	Workload features and the best performing GPU-LocalTM version.	100
5.9	Transactional execution using multiple-ascending WfS.	104
A.1	Application behavior on big.LITTLE.	119
A.2	Características de la familia de GPUs Southern Islands	129





UNIVERSIDAD
DE MÁLAGA

1 Introduction

If we observe the computing needs of today and try to predict those that will arise in the future, we can conclude that heterogeneous processing will be present in many devices and applications: from cell phones to supercomputers, going through cars and smart cities. The reason is that different data and different algorithms fit better the characteristics of some devices than others. To provide an example of state-of-the-art technology, self-driving cars are a clear scenario where heterogeneous computing is not an option but a requirement. In such vehicles, images must be collected and analyzed, and graphics processing units (GPUs) are very efficient performing image processing and deep neural network (DNN) tasks. In order to meet real-time requirements of such complex system, some parts of the algorithm can be implemented in hardware using field-programmable gate arrays (FPGAs). And, of course, multi-core CPUs play an important role, both orchestrating the operation of other devices and performing those tasks in which they are more efficient than other processors. However, multi-core CPUs themselves are no longer an homogeneous device: CPU cores may share the same ISA, but have different capabilities to offer energy efficiency or to provide processing power when required. Programming this set of devices is a challenging task and technologies have been developed to exploit their capabilities. For instance, OpenCL [38] provides an interface to program and communicate different devices maintaining portability (but not performance-portability). In most part of these languages and technologies, synchronization primitives are still very basic: atomics, synchronization points (in form of barriers and kernel launches), and signals. With these basic primitives programmers can construct complex synchronization techniques to ensure mutual exclusion in the access to shared data, but this is a difficult and error-prone task that can harm performance. Transactional Memory (TM) [35] has proven to be an efficient synchronization method in multi-core



CPUs. TM provides a simple interface to define the bounds of a critical section and is able to obtain good performance due to the optimistic execution model. Given this popularity and the potential performance benefits, TM is considered to be part of future C++ standards.

TM is being incorporated to commercial multi-core CPUs to complement the other basic synchronization techniques: Intel provides ISA extensions (TSX) to support basic hardware TM [72], and IBM has released two different systems with built-in hardware TM support, IBM BlueGene/Q [67] and System z [37], and more recently Power 8 [3]. Recently, TM solutions for GPUs have started to appear in the literature, both software [12, 70, 36] and hardware [29, 28, 17]. Provided that this technology is being adopted by CPU vendors and its applicability on GPUs is being investigated, it is important to understand the impact that it can have in heterogeneous devices. Additionally, it is interesting to cover TM from different perspectives: hardware implementations (HTM), software implementations (STM), and from the scheduling point of view.

This thesis is organized as follows. In the rest of Chapter 1 we describe the heterogeneous architectures, basic synchronization primitives, Transactional Memory, and provide a motivation for this thesis. In Chapter 3 we analyze a popular software TM library on an heterogeneous CPU, and we include a simple scheduler to assess the impact that scheduling can have in TM-instrumented applications. Chapter 4 presents a novel software TM design on top of CPU+GPU heterogeneous processors. Chapter 5 describes our hardware TM proposal for GPU architectures, which focus on transactions taking advantage of the scratchpad memory. Lastly, Chapter 6 presents the conclusions of this thesis.

1.1 Heterogeneous Architectures

In the past, traditional single-core CPU designs aimed to improve performance by increasing clock speeds. As a result, the gap between processor and memory speeds increased hitting the *memory wall*. Large cache memories were incorporated in the CPU to mask the latency of memory. Another way of increasing performance is to keep the processor busy by implementing instruction-level parallelism (ILP) techniques. However, this increases the complexity of CPU designs and is hard to find programs that fully benefit from ILP with a single instruction stream. Soon, single-core CPUs reached the *ILP wall*. The large cache sizes, the complexity of implementing ILP, and the increment in CPU clocks resulted in more power consumption and needs of heat dissipation beyond current technology. This effect, known as *power wall* is one of the limiting factors in CPU design.

With the goal of producing more powerful processors, CPU vendors shifted from single-core to multi-core designs. Multi-core CPUs are able to deliver higher performance with lower energy requirements, but with the overhead of writing parallel code (we discuss such overhead in Section 1.2).

In parallel to this evolution, hardware accelerators started to gain importance. Hardware accelerators are specialized pieces of hardware that perform some functions more efficiently than traditional CPUs. The most popular hardware accelerator is the graphics processing unit (GPU). Initially, GPUs were merely video chips that pushed images from the frame buffer out to the monitor. Later, in the 1980s and 1990s, GPUs were able to manipulate 2D and 3D images using fixed-function primitives implemented in hardware. In the first years of the decade of the 2000s the first programmable GPUs appeared, allowing programmers to write small programs to modify the attributes of pixels or vertices. At that point, researchers started to use GPUs as accelerators for general-purpose programs by mapping their data structures into the pixel and vertex processors integrated in the GPU [56]. This leads to the creation of technologies as CUDA [47] and OpenCL [38] that allow programmers to use the GPU as a general-purpose processor. Since then, GPUs have become the *de facto* accelerator for data-parallel applications.

In recent years, the CPUs and GPUs have become closer and now are part of the same chip. Currently, these heterogeneous CPU+GPU architectures (also known as accelerated processing units (APUs)) are the most popular and are widely available in platforms such as cell phones, desktop and laptop computers, game consoles, and servers, among others. Intel Haswell, AMD Kaveri, Samsung Exynos and NVIDIA Tegra K1 are examples of processors that integrate CPU and GPU in the same chip. Usually, the GPUs integrated with CPU cores are less powerful than discrete ones due to space and power constraints. However, they benefit from lower latency in the data transfers (as compared to discrete GPUs that communicate via a PCI interface) and, in some cases, have coherent caches to ease CPU-GPU communication. With these architectures available, programmers can access a large amount of computing resources at a low power consumption. However, the burden in this case is to map and orchestrate the execution of different parts of the application onto different parts of the hardware.

Heterogeneous architectures do not only refer to CPU+GPU processors. Cores in multi-core CPUs do not necessarily have to be homogeneous. In the particular case of the ARM big.LITTLE architecture, CPU cores are grouped in two clusters: the big cluster and the little cluster. Cores in the big cluster are focused on high performance and offer large caches and fast clock speeds. Cores in the little cluster have a simple in-order design, smaller caches and operate at less

frequency, with the goal of minimizing power consumption. Both groups of cores share the same ISA, and applications can be executed and migrated from one cluster to another depending on the performance/energy needs.

Other heterogeneous architectures beyond CPU+GPU and big.LITTLE architectures are being developed integrating more kinds of processors. To provide an example of an state-of-the-art heterogeneous device, we cite the Xilinx Zynq UltraSCALE+ CG¹. This device integrates a quad-core ARM Cortex-A53 processor, a dual-core ARM Cortex-R5 processor (specialized in real-time applications), a Mali-400 GPU, a 16nm FinFET+ FPGA, as well as dedicated units for security, I/O, and performance and energy monitoring, among others.

1.2 Complexities of parallel and heterogeneous programming

Typically, to take advantage of the resources available in multi-core CPUs, applications are divided into separate computing threads that run on different cores. Programmers have to decompose their program into tasks that are assigned to the different threads which have to be properly managed to successfully execute the application. As the threads are mapped to independent CPU cores, their execution is non-deterministic in terms of ordering. Thus, when synchronization is required, it has to be explicitly defined by the programmer.

The most common scenario in which synchronization is required is the access to shared data. To ensure the correctness of the program, the portion of the code accessing shared data (also known as critical section) must be protected by a mutual exclusion mechanism. Multi-core CPUs come with a set of atomic instructions to allow programmers implementing this mechanism. In particular, the compare-and-swap (CAS) atomic instruction performs load, comparison, and store operations with no other memory instruction affecting the same memory position interleaved. This instruction operates on only one memory position at the time. If the critical section requires accessing more than one memory location, a lock mechanism can be implemented using a CAS operation.

Lock-based techniques can be implemented in two different ways, each with its own advantages and disadvantages. On one hand, coarse-grained lock techniques are easy to implement: a single lock, acquired using a CAS operation, manages the access to the critical section. Mutual exclusion is guaranteed as

¹<https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html>

only one thread is allowed to enter the critical section at a time. The simplicity of this implementation comes with the drawback of poor scalability. When using a coarse-grained lock solution, the execution of the critical section is sequential, affecting performance and leaving computing resources underutilized. On the other hand, expert programmers can implement a fine-grained lock solution. This is a problem-dependent implementation. Commonly, a set of locks is defined to protect the access to individual objects or memory positions. Before proceeding to the critical section, each thread has to properly acquire as many locks as needed to ensure that no other thread is accessing the same memory positions. If the thread succeeds, then it may progress into the critical section. This way, thread modifying different memory positions execute the critical section in parallel, potentially improving performance. However, ensuring the correctness of this implementation is challenging: threads may intend to acquire the same locks in different order and, if not properly managed, may create a deadlock.

These problems extend from homogeneous multi-core CPUs to heterogeneous platforms. In the following, we will consider APU (CPU+GPU) processors as reference. In particular, we focus on HSA-enabled² APUs. These APUs provide an unified memory space in which pointers can be seen by both the CPU and GPU. A cache coherence protocol ensures that coherence is kept across platforms. Similarly to multi-core CPUs, a set of platform-atomic instructions are used to synchronize both devices. Therefore, the same locking techniques as in multi-core CPUs can be implemented on GPUs and across both platforms within the APU processor. However, even in the case of coarse-grained locks, the implementation becomes more difficult. While each thread on the CPU has a private program counter, threads on the GPU are grouped so a subset of the threads (*wavefront* or *warp*) share the same program counter. Thus, threads within a wavefront execute in lockstep (Single Instruction - Multiple Data (SIMD) execution). Traditional CPU implementations are prone to deadlocks when executing in SIMD architectures. This leads to increased programming complexity as the mutual exclusion mechanism has to be implemented differently in each platform.

Figure 1.1 provides an example of a coarse-grained lock implementation on CPUs and GPUs. The goal is to offer a clear and simple example of the challenges programmers face when implementing mutual exclusion in an heterogeneous platform. Before proceeding to explain this implementation, we discuss two important concepts that help understanding this example: the atomic CAS (compare-and-swap) operation and the control flow in the SIMT execution model.

- The **atomic CAS operation**, available in most languages with atomics

²Heterogeneous Systems Architecture: <http://www.hsafoundation.com/>

```

1 while (CAS(lock,0,1)){;}
2 //Critical Section
3 atomicStore(lock,0);

1 bool done = false;
2 while (!done){
3   if (!CAS(lock,0,1)){
4     //Critical Section
5     done = true;
6     atomicStore(lock,0);
7   }
8 }

```

Figure 1.1: Coarse-grained lock implementation in CPUs (left), and the transformation required to avoid deadlocks in the SIMT programming model (right).

support, works as follows³. `CAS(obj, expected, desired)` compares `obj` and `expected`. If they are equal, then the value of `obj` is replaced by `expected`. Otherwise, the value of `obj` remains the same. In both cases, the function returns the old value of `obj` (i.e., the value stored by `obj` before being replaced by `expected`, if required).

- **Control flow** in the SIMT execution model deserves special attention as it is crucial to understand how locking mechanisms can be implemented on GPUs. We use as example an *if* conditional statement as in Figure 1.1 (lines 3 to 7). As threads execute in lockstep, conditionals are implemented as follows. Firstly, when a conditional is reached a *convergence point* is set at the end of the conditional (line 7 in the example of Figure 1.1). This convergence point acts as an implicit barrier for the threads executing in lockstep. Then, the condition is evaluated. The threads whose condition is evaluated as false are disabled while those whose condition is evaluated as true remain active. Active threads proceed to execute the block of code of the conditional statement in lockstep, until the convergence point is reached. When the convergence point is reached, the threads that were disabled during the evaluation of the condition are enabled again. This way, all the threads that were active before evaluating the condition are enabled again, and only those whose condition was true executed the block of code of the conditional statement. Loops work in a similar manner as if statements: the convergence point is set at the end of the loop for these threads that do not fulfill the condition of the loop.

The left-hand side of Figure 1.1 shows the traditional CPU implementation of a coarse-grained lock using an atomic CAS (compare-and-swap) operation,

³The CAS operation is explained as in C++11. See http://en.cppreference.com/w/cpp/atomic/atomic_compare_exchange for more details

present in many languages. In Figure 1.1, we assume that *lock* is a variable shared among all threads initialized to 0. The first thread arriving to the while loop (line 1) executes the atomic CAS operation. As result of the execution of the instruction `CAS(lock,0,1)` the condition in the while loop is false (as the CAS operation returns 0). This way, the thread leaves the while loop (proceeding to line 2) and the variable *lock* is set to 1. This means that such thread was able to acquire the lock. The remaining threads that reach the while loop are not allowed to leave the loop as the condition will remain true (the CAS operation returns 1). After executing its critical section, the thread that acquired the lock releases it (line 3). Then the lock is available to be acquired by another thread that intends to execute the critical section. Mutual exclusion is guaranteed by this mechanism, but at the cost of serializing the access to the critical section. However, when executed in the GPU, this code creates a deadlock. In the SIMT programming model, as threads execute in lockstep, only one of them is able to get the lock and leave the while loop (line 1). As there is a divergence in the execution of the program, the SIMT programming model sets a convergence point at the end of the while loop (line 2) before executing the critical section, creating an implicit barrier. This implicit barrier forces the thread who acquired the lock to wait for the rest to finish the execution of the while loop. This will never happen, as the lock is held by the waiting thread and the remaining threads will not leave the while loop until the lock is released. Thus, the classic spinlock creates a deadlock in the SIMT programming model. The right-hand side of Figure 1.1 shows the required transformation for this spinlock to work in the SIMT programming model. In this case, all the active threads enter the loop (line 2). The convergence point (i.e., implicit barrier) for this loop is set in line 8, which will be eventually reached by all the threads. Then, only one of the threads acquires the lock (line 3) inside an if statement. In this case, the convergence point is set at line 7. This way, the threads that did not acquire the lock perform the implicit barrier at line 7, while the thread that acquired the lock executes its critical section (line 4), sets itself to go to the convergence point of the while loop (line 5), and releases the lock (line 6). With this code transformation we can safely implement coarse-grained locks in the SIMT programming model.

The goal of this example is to help the reader to understand the difficulties that programmers face when deploying their applications across heterogeneous platforms and, in particular, when mutual exclusion is required to protect the access to shared data. Of course, these implementations can be even harder for fine-grained locks or *ad-hoc* solutions for specific problems.

1.3 Transactional Memory

Transactional Memory (TM) [35] provides a higher level of abstraction for implementing mutual exclusion in multi-threaded programming. TM uses the concept of *transaction* to enclose a block of code to be executed atomically and in isolation (this is, with mutual exclusion guarantees). In contrast to pessimistic lock-based solutions, transactions are allowed to execute in parallel. Proper implementation of conflict detection and version management mechanisms ensure mutual exclusion: if two or more transactions access the same memory position and, at least, one of the accesses is a write (transactions conflict), then only one of the transaction progresses while the remaining ones abort execution, undo their speculative changes to memory and retry.

The use of TM implies several advantages over traditional locking mechanism. Firstly, TM offers a simple interface to define the bounds of the transaction and, optionally, instrumentation of the memory read/write operations. In contrast to lock-based mechanism, TM does not require the definition of extra data structures or variables to deal with the implementation of mutual exclusion. Secondly, the TM system deals with the correctness of the implementation. Thus, programmers that use TM to implement mutual exclusion do not have to deal with deadlock-/livelock situation as the TM implementation takes into account these situations. Lastly, as TM is optimistic by nature, performance is expected to be superior to lock-based implementations as conflict-free transactions can execute in parallel. Ideally, the TM interface provides a programming model with a complexity similar or lower than the use of coarse-grained locks, but with the performance of fine-grained locks.

1.4 Thesis motivation and research questions

The increasing demand for processing power with low energy consumption encourages hardware vendors to bet for heterogeneous processors. These heterogeneous processors are quickly becoming more complex, which translates in extra burden put on the programmers' shoulders. Both the industry and the research community are trying to reduce such burden by introducing new programming models and scheduling techniques [5, 46, 50], among others. As TM has proven to be an effective technique to ensure mutual exclusion in traditional multi-core CPUs, it is important to understand the impact it can have in heterogeneous processing.

The design space space for TM in heterogeneous architectures is immense. TM itself can be implemented as STM or HTM (or even considering a hybrid solution). Furthermore, transaction scheduling is an active field of research [48, 53, 73] which tries to maximize the benefits of TM while reducing its overheads by properly assigning transactions to computing resources. Once established the scope in the TM field, it can be applied to heterogeneous architectures in different ways. TM can be applied to heterogeneous CPUs (i.e., the big.LITTLE architecture). Additionally, TM can be integrated in each one of the accelerators in an heterogeneous device. Also, provided the unified memory space in HSA-enabled heterogeneous platforms, TM can be used to implement mutual exclusion for data shared across multiple devices.

Given the design space in the intersection between TM and heterogeneous architectures, in this thesis we analyze this confluence from different perspectives. Mainly, the goal of this thesis is to answer the following research question: ***Can TM be applied on different heterogeneous architectures?*** More specifically, other questions that we intend to answer are: *Can scheduling to be used to improve performance of TM in heterogeneous processors?*, *Can we implement a TM system to operate on APU processors?*, and *Can we implement a hardware TM for GPUs without requiring expensive microarchitectural changes?*

Our main contributions in this thesis and the related publications, which intend to answer the previous research questions, are the following:

- We analyze the behavior of current state-of-the-art STM implementations when executing on big.LITTLE CPUs [65, 66] and the role that scheduling can have in terms of performance, energy consumption and the efficiency of the TM implementation [64].
- We propose a STM implementation for heterogeneous CPU+GPU processors with unified coherent memory [62]. The main design decision is to reduce the overhead of communicating GPU and CPU transactional meta-data.
- We propose a HTM implementation for a GPU accelerator [63, 61, 59, 60, 58]. In particular, we focus on the GPU scratchpad memory, which is used to optimize application performance.

The goal of this thesis is to understand the benefits of different TM implementations in different parts of heterogeneous platforms.



UNIVERSIDAD
DE MÁLAGA

2 Background and Related Work

In this chapter we present some background on heterogeneous processors, transactional memory (TM) and the related work applicable to this thesis. Although there exists a wide variety of heterogeneous processors and TM implementations, this chapter is restricted to these works related to our contributions. Section 2.1, Section 2.2, and Section 2.3 provide background on GPUs and heterogeneous processors, discussing architectural and microarchitectural details needed to understand the rest of the thesis. Section 2.4 describes the basics of TM and the different ways it can be implemented. The related work is organized in two different sections: Section 2.5 contains a review on the existing TM designs for GPUs and Section 2.6 briefly discusses recent advancements in TM for low-power processors.

2.1 GPU accelerators

GPUs have been widely adopted as accelerators for parallel general-purpose computing thanks to its massive multi-threading capability. In this section, we tackle the description of GPUs from two different perspective: the programming perspective and the architecture perspective.

2.1.1 GPGPU programming

The GPGPU (general-purpose GPU) programming has undergone numerous advances in the latest years. Initially, technologies such as CUDA and OpenCL

supported programming of GPUs as general-purpose processors. These languages are dual-source: programmers have to write a host program (usually in C++) that gets access to the accelerator, and then offload the *kernel* code (written in the aforementioned technologies) to the GPU. Recently, single-source technologies have arisen to ease the use to the computing capabilities of the GPU. Languages like Sycl and C++AMP, where CPU code can be annotated to be offloaded to the GPU, are example of technologies trying to ease GPU programming. In this thesis we use OpenCL naming as it highlights the key aspects to take into account in GPGPU programming. In particular, we focus on the definition of kernels. Additionally, most of the concepts in OpenCL have an equivalent in other languages as CUDA, so this explanation can be applied to other scenarios.

The key aspects of OpenCL are the execution model and the memory model. In the OpenCL **execution model**, programmers write a kernel to be executed by a *NDRange*. The *NDRange* is a multi-dimensional arrangement of *work-groups*. A work-group is a set of computing threads called *work-items*, which are also arranged in a multi-dimensional way and execute the kernel code. Work-items within a work-group are grouped into *wavefronts*. Work-items within a wavefront execute in *lockstep*. At this point, the programmer is responsible of defining the number of work-groups and the number of work-items per work-group to execute the kernel. Note that the underlying GPU architecture may impose some constraints in the number of work-items supported per work-group as well as the number of work-groups supported by the GPU but, in the end, the programmer is free to choose the most appropriate values for the kernel. However, the wavefront size is a value defined by the execution device (i.e., the GPU). Programmers have access to this value which can be used to optimize execution, but this value can not be chosen nor changed.

The **memory model** offered by the OpenCL interface consists of four different memory spaces.

1. Global memory, which contains global buffers. Information stored in global memory is accessible by all the work-groups (and, hence, all the work-items) executing the kernel, as well as the host CPU.
2. Constant memory, which contains information that is constant during the execution of a kernel. As global memory, is accessible by the host CPU and all the work-groups within the GPU.
3. Local memory, whose visibility is restricted to the work-items belonging to the same work-group. It is not accessible by the host CPU and stored data can not be shared accross different work-groups. However, depending

on the hardware platform, it provides performance advantages over the use of global memory. Programmers are encouraged to use local memory as scratchpad to speed up their applications.

4. Private memory, which store values private to each work-item. Information stored in private memory can not be shared by different work-items and is not accessible by the host CPU. Typically, private memory is stored in registers or in a per-work-item private partition of global memory.

In addition to these memory spaces, modern OpenCL implementations include atomic operations in global memory and local memory for synchronization. Barriers and memory fences are also available as synchronization methods. Memory fences can be used to enforce consistency, avoiding memory reordering optimizations performed by the compiler or the microarchitecture. Barriers synchronize the execution of work-items within the same work-group. There is no barrier operation that synchronizes work-items from different work-groups. This kind of synchronization can be performed by using global memory atomic operations or by terminating the kernel and launching a new one (kernel-level synchronization).

Hardware vendors are in charge of mapping the OpenCL execution model and memory model in their architectures. In Section 2.1.2 we describe how this is done for our baseline architecture.

2.1.2 GPU architecture

Currently, there exist a wide variety of GPU architectures coming from different designers and vendors. At the moment, the most important are NVIDIA, AMD, Intel, and ARM. Without loss of generality, we use the AMD's Southern Islands family of GPUs [4, 57] to explain the architecture, as it is the baseline GPU employed in part of this thesis.

In this architecture, the GPU is organized as a set of heavily multi-threaded processors called Compute Units (CU). For instance, the AMD HD Radeon 7970 (see Figure 2.1) consists of 32 CUs. In this GPU, the Ultra-Threaded Dispatcher maps work-groups to the CUs. Each CU is able to handle multiple work-groups, but each work-group is assigned to a single CU. The CU contains the following functional and control flow units:

- The wavefront scheduler fetches instructions from memory for a given wavefront and issues its execution to the appropriate functional unit. Note that all work-items within the same wavefront execute the same instruction. In

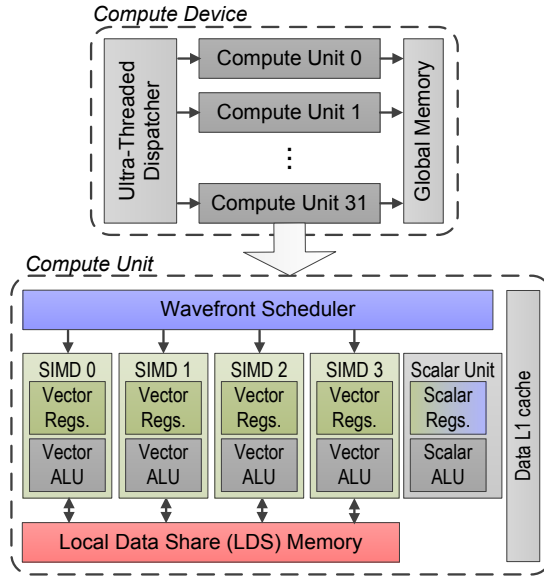


Figure 2.1: Block diagram of the AMD HD Radeon 7970 GPU.

this architecture, a wavefront contains up to 64 work-items and a work-group contains up to 4 wavefronts (this is, the maximum number of work-items per work-group is 256).

- The SIMD units are the main execution unit within the CU. They act as arithmetic-logic units capable of executing integer, floating-point and other kind of operations for all active work-items within the wavefront. SIMD units also contain vector registers, which are used to store the private variables for each work-item. Note that, in the case of having a large amount of private variables, the compiler can opt for register spilling, storing private variables in global memory.
- The vector memory unit is in charge of executing memory instructions that require access to global memory.
- The scalar unit executes scalar instructions and also has access to data stored in global memory. In addition to that, the scalar unit contains a set of scalar registers. The use of the scalar unit and scalar registers is an architectural optimization for private variables whose value remains the same for all work-items within the wavefront. For instance, the index in

a for-loop (which is private for each work-item) normally has the same value for all work-items within the wavefront. If the compiler detects such situation, it can promote the use of a single scalar register and use the scalar unit to update its value, instead of reserving 64 vector registers (one per work-item within the wavefront) to hold the same value.

- The LDS (local data share) handles the storage of local memory variables (i.e., shared among work-items within the same work-group). In this particular architecture, local memory is organized in 32 banks with interleaved access, where consecutive addresses map to different consecutive banks.

2.1.2.1 Memory architecture

In this architecture there are dedicated units to access global memory and local memory (vector memory unit and LDS unit, respectively). Both units share some characteristics. These units are designed to serve multiple memory requests at a time. If the simultaneous memory request map to different banks (i.e., *coalesced* memory accesses), then the memory petitions are served in parallel. Otherwise, a coalescing unit is in charge of serializing memory requests to the same bank. This way, programmers are encouraged to write their program using coalesced memory accesses for both memory spaces in order to obtain substantial speedups. In addition to that, both units implement atomic operations to allow for atomic memory accesses in their respective memory space.

Among the differences in both memory spaces, the most noticeable is size. Global memory can store hundreds of megabytes or gigabytes of information, depending on the GPU model. In this architecture, the local memory space contains 64 kilobytes of storage. However, each work-group is allowed to access only up to 32 kilobytes of local memory; the space remaining is reserved to allow for multiple work-groups running simultaneously minimizing context switching penalties. This difference in space is due to the location: global memory is located *offchip* and accessed through a cache hierarchy, while local memory is located *onchip* and uncached. This implies that accesses to local memory present lower latency as compared to global memory (in particular, memory latency for local memory variables is about two orders of magnitude lower as compared to global memory). This way, local memory is used as scratchpad to accelerate some portions of the kernel code, provided that accessed variables fit in local memory and that programmers deal with the explicit management of local memory.

2.1.2.2 Execution model

The GPU SIMT execution model relies on the existence of execution masks. There are two execution masks in this architecture: the EXEC mask and the VCC mask. The EXEC execution mask contains a bit for each work-item within the wavefront, indicating if it is active (or inactive). The VCC mask stores the results of a vector comparison instruction. In a simplified way, the VCC masks acts as a “vectorized Z flag”. By combining the EXEC and VCC masks, compilers can implement conditionals, loops and other features using predication techniques [24]. Figure 2.2 contains a simplified example of the evaluation of a condition using the predication technique. The comparison instruction `v_cmp_gt_i32` updates the state of VCC, storing the value 1 for those work-items that fulfill the condition. Then, the current value of EXEC is backed up in the scalar registers `s8` and `s9`, and is updated with the values of VCC using the `s_and_b64` instruction. At the end of the conditional, the previous state of EXEC has to be restored, by loading the backup stored in the scalar registers `s8` and `s9`.

```

1 if (id < 1)
2 {
3 ...
4 }
1 v_cmp_gt_i32 vcc, s8, v1
2 s_mov_b64 s[8:9], exec
3 s_and_b64 exec, s[8:9], vcc
4 ...
5 s_mov_b64 exec, s[8:9]
```

Figure 2.2: Example of the implementation of a conditinal using the EXEC and VCC execution masks.

There exist one EXEC mask as well as one VCC mask per wavefront. Thus, the size of these masks is 64-bit. These 64-bit masks are stored using two consecutive 32-bit general purpose scalar registers.

2.2 Heterogeneous CPU+GPU processors

In this section we provide background on the CPU+GPU processors. These processors integrate within the same chip a multi-core CPU and a GPU. In particular, we focus on those defined by the Heterogeneous System Architecture (HSA) Foundation and taking as example the HSA-compliant AMD Kaveri processors.

The HSA Foundation proposes a series of standards to improve the programmability of heterogeneous computing devices. It defines a vendor-independent virtual ISA, called HSAIL, which allows the same source code to run on differ-

ent HSA-compliant devices. Tools are provided to create code in most common high-level programming languages such as C++, C++AMP, and OpenCL. In addition, the programming interface is designed to provide easy access to the different computing devices available in the heterogeneous processor, as well as a shared virtual memory space.

To describe the APU processors we use as example the HSA-compliant AMD Kaveri A10-7850K processor. This processor integrates 4 CPU cores as well as 8 GPU compute units (CUs). CPU threads are mapped in the CPU cores. The NDRange, work-groups, wavefronts and work-items work in the GPU as described in Section 2.1.2. Threads within the wavefront execute in lockstep. The most remarkable HSA features offered by this processor are the pageable shared virtual memory, and memory coherence between CPU and GPU. This way, the same pointer can be accessed by both the CPU cores and the GPU CUs. In addition, platform atomics and memory ordering operations are available.

2.3 Heterogeneous CPUs

Multi-core heterogeneous CPUs are those designed with asymmetric cores. In particular, we focus in the ARM big.LITTLE architecture [1]. This architecture features a set of high-performance *big* CPU cores and a set of high-efficiency *little* CPU cores. We refer to these sets as big cluster and little cluster, respectively. Both clusters share the same ISA and have access to the same main memory. Thus, an application can run in any of them without requiring two different binaries. Applications with high performance requirements are intended to be attached to the big cluster, while those without these requirements are to be processed using the little cluster.

By default, these architectures support three types of scheduling.

1. The clustering switching model assigns applications either to the big or the little cluster, but not both simultaneously. This way, the processor can be seen as if it operates in two different modes: high-performance mode or power-saving mode.
2. The in-kernel switching mode pairs a big and a little core into a virtual core. Each application running on a virtual core can only use one of them. With this configuration, each application can decide to run in the big or little core depending on the computational or energy demands.
3. The global task scheduling, which permits to observe all cores within the

processor so applications can run in any of them without any restrictions.

The Samsung Exynos 5422 processor is an example of CPU based in the ARM big.LITTLE architecture (see Figure 2.3). The big cluster is a 4-core Cortex-A15 CPU while the little cluster is a 4-core Cortex-A7 CPU featuring 2 Mbyte and 512 Kbyte L2 caches, respectively. The processor also integrates a Mali-T628 GPU and 2 Gbyte of low-power DDR3 RAM. We tested this processor with Linux odroid 3.10.59+ operating system, which incorporates the global task scheduling policy.

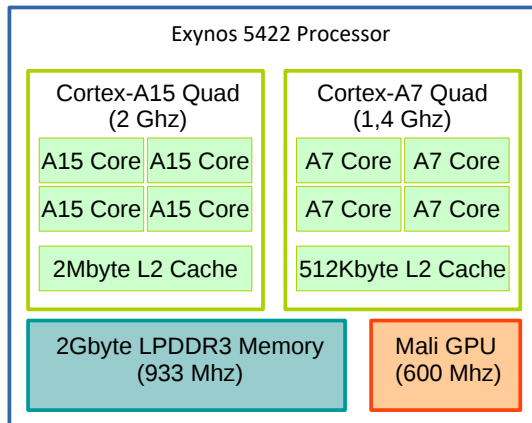


Figure 2.3: Samsung Exynos 5422 based in the ARM big.LITTLE architecture.

2.4 Transactional Memory

Transactional Memory (TM) [35] has emerged as a promising alternative to locking mechanisms to coordinate concurrent threads. TM provides the concept of transaction to determine the bounds of a critical section, decreasing the programming effort required to define such kind of synchronization. Normally, the transaction is enclosed by *TX_Begin* and *TX_Commit* instructions. Depending on the implementation, read and write memory operations should be instrumented by using the *TX_Read* and *TX_Write* instructions, respectively. This can be either a requirement of the TM implementation, or used to discriminate transactional memory accesses from non-transactional memory accesses in order to improve performance.

A transaction enforces the properties of atomicity and isolation during the execution of a critical section. Atomicity implies that a whole transaction is executed as if it is a single, indivisible instruction. The isolation property implies that speculative changes made inside transactions do not affect the state of the program until they have successfully committed. If these properties are respected, transactions are allowed to run in parallel, but the results are the same as if they were executed serially. A conflict occurs if two or more transactions intend to access to the same shared memory location and, at least, one of the accesses is a write. In such situation, one of the transaction can continue execution, while the rest of the transactions must discard its updates to memory (the transaction *aborts*) and restart their execution. This is achieved by implementing appropriate conflict detection/resolution and version management mechanisms.

Conflict detection can be performed following eager and lazy strategies:

- In *eager conflict detection* conflicts are detected just before issuing memory read/write operations. Detecting conflicts eagerly prevents the transaction from running as soon as it is known it should abort. However, this requires heavy instrumentation of the read and write memory operations that can harm performance needlessly if no conflict occurs.
- With *lazy conflict detection* transactions do not detect any conflict until the end of the transaction. This conflict detection mechanism assumes that conflicts rarely occur. It requires less instrumentation of the read and write memory operations, as conflict detection is performed just once at the end of the transaction. Thus, in the case that there is no conflict, the overhead introduced by the TM system is lower as compared to eager conflict detection. However, if the transaction has some conflicts, they are not detected until the end of the transaction, increasing the amount of wasted work (i.e., the transaction continues executing until its end, even if data is stale). Another implication of lazy conflict detection is that they may continue executing on stale data if there is a conflict that has not been detected yet. Thus, ensuring the correctness of the implementation is more complex compared to eager conflict detection.

As in conflict detection, version management can be implemented eagerly or lazily.

- *Eager version management* stores speculative data in place (i.e., in its final memory location) and performs a backup of the old values in an *undo log*. In case the transaction finds a conflict, then the values stored in the undo

log have to be restored to memory, discarding speculative values. Note that, if multiple transactions access the same memory position, they can be accessing speculative data, harming the isolation property. In this case, the version management and conflict detection mechanism should be properly coupled to avoid such situation.

- *Lazy version management* stores speculative values in a transaction-private *write buffer*. If the transaction ends with no conflict, then memory should be updated with the speculative values stored in the write buffer. In such case, committing the transaction is slower as compared to eager version management. In the case of conflicting transaction, then the write buffer must be discarded, incurring in less overhead during the commit operation.

TM can be implemented in software (STM) and hardware (HTM). HTM solutions are efficient, as they are implemented in hardware at core level. Nevertheless, due to the limited resources available in hardware, they are more constrained in terms of memory usage and forward progress guarantees. Due to these limitations, some hardware TM solutions are known as *best effort* TM systems. To deal with these restrictions and to ensure forward progress, hardware TM systems provide the programmers with tools to program a software fallback path, which adds programming complexity to the TM paradigm. These restrictions do not apply to STMs. Software solutions impose less limitations and allow for more complex implementations. Furthermore, STM can be implemented and used in existing and legacy hardware. On the other hand, the overhead of the software implementation is higher as compared to HTM, affecting scalability and performance [11].

2.5 TM on GPUs

In recent years, some TM proposals for GPU architectures have been developed. Note that all of them consider only global memory, leaving the low-latency local memory on the side.

2.5.1 Software TM on GPUs

On the software side of the design space we find different proposals, working at different scopes and granularities.

Cederman *et al.* propose two STM systems [12] focused on the conflicts occurring among different work-groups. They do not consider potential interactions between single work-items. Both implementations follow a lazy-lazy scheme: lazy conflict detection and lazy version management. Their first implementation, called *blocking* uses a set of irrevocable locks to protect memory positions accessed within the transaction (this is, only transactions able to get all the required locks are able to commit, but serially). They discuss that the correctness of this implementation is highly dependent on the fairness of the GPU scheduler: work-groups waiting for a lock can swap the work-group holding a lock repeatedly and, if the later is not scheduled again, the program may deadlock. Their second proposal is *obstruction-free*. In this approach, committing transactions try to acquire locks that protect the locations to be updated and, at the same time, announces the values to be written. Conflicting transactions (i.e., those that try to acquire the same locks) now have two options. In case a conflicting transaction is able to get all the locks, but has not updated memory, value to be written can be forwarded from one transaction to the other. Another option is, if the conflicting transaction did not get all the locks or the values in memory mismatch, to abort the transaction. In this implementation transactions either commit or abort without waiting for other transactions to finish their commit. In average, the obstruction-free algorithm produces better performance and less amount of aborted transactions compared to the blocking implementation.

Xu *et al.* propose GPU-STM [70], a lazy-lazy software TM which works at the granularity of a work-item. Conflict detection is performed in two phases: timestamp-based validation (TBV) and value-based validation (VBV). TBV uses a set of versioned locks, which store the state locked/unlocked as well as the last time it was modified. Transactions try to acquire all the locks that protect the access to the memory locations to be updated. If the timestamp stored in all the locks coincides with the last time the transaction was consistent with memory, then the transaction can commit directly. Otherwise, the transaction needs the VBV phase, in which all accessed locations within the transaction need to be checked against the current values in memory. In case the values mismatch, the transaction aborts; otherwise it commits. A successfully committing transaction must update all the versioned locks that were acquired in order to inform other transactions that memory has been updated. This proposal is able to outperform the use of coarse-grained locks on GPUs.

Holey *et al.* also propose three STM enabled at a work-item granularity [36]. Their first design is an Eager STM (ESTM) which detects conflicts eagerly. To do that, they use versioned locks to prevent the access to shared data. These versioned locks include, in this case, shared and modified bits, as well as the iden-

tifier of the transaction that first accessed the memory location. This approach is able to detect RAW, WAR, and WAW violations, signaling them as conflicts. Their second approach, a Pessimistic STM (PSTM), simplifies the model treating reads and writes the same. This implementation is more efficient, but RAR accesses are inaccurately signaled as conflicts. However, as many transactions perform read-modify-write operations on the same memory positions, most of these conflicts would be detected later. Their third implementation is an Invisible Read STM (ISTM) that validates reads during commits. Read operations do not modify any entry in the versioned locks, which reduces the overhead of non-conflicting transactions. In their experiments, ISTM and PSTM outperform the ESTM implementation, also offering a better abort/commit ratio in most of the cases.

Shen *et al.* propose PR-STM [55], which uses versioned locks and priority-based lock stealing techniques to improve conflict detection. Briefly, transactions pre-lock memory locations that are accessed during the transaction. Pre-locks can be stolen by transactions with higher priority. If a transaction successfully pre-locks all the memory positions required, then it will try to make these locks definitive. In case they can not be locked, it means that a transaction with higher priority has stolen the pre-lock. This way, progress is guaranteed as the transaction with higher priority is able to succeed. Additionally, this pre-locking and locking mechanism based in priorities ensure that transactions executing in lock-step successfully commit with no deadlock. They compare their proposal against GPU-STM and the CPU implementation called TinySTM [26, 25], outperforming both.

In this thesis we contribute with a software TM design that focuses on APU architectures [62]. Our contribution, based in previous software TM for GPUs and CPUs [70, 20], helps to understand the main challenges found when implementing TM on a heterogeneous APU processor.

2.5.2 Hardware TM on GPUs

Regarding hardware solutions, we can highlight Kilo TM [29] and two performance improvements, one on top of Kilo TM, called WarpTM [28], and other on top of WarpTM [17].

Kilo TM [29] is a hardware TM system operating on global memory which implements a lazy-lazy (i.e., at commit time) approach using specific commit units. Each commit unit works on a partition of global memory. In Kilo TM, transactions store a read-set and a write-log that is sent to the commit units

at the end of the transaction. The read-set is validated against the values in global memory (more specifically, in the L2 cache). Also, a hazard detection mechanism is used to detect conflict among committing transactions which have not updated memory yet. This process is performed in parallel in several commit units. Note that each commit unit works on a portion of global memory and, thus, a transaction must be validated by all commit units to be considered successful. The outcome of the validation (i.e., conflict/no-conflict) is communicated among all commit units. In case there is no conflict, transactions commit their values to memory; otherwise the transaction restarts.

Kilo TM was improved by WarpTM [28]. Firstly, intra-warp (a *warp* is the equivalent to wavefront in CUDA jargon) conflict detection optimizes the use of resources as conflicting transactions among a warp are stalled before sending their read-set and write-log to the commit units. Secondly, conflict detection is optimized to speed up conflicts in read-only transactions. Specifically, a set of global timers is used to register the last time memory positions were modified. Read-only transaction can use these timers to commit directly without checking their read-set. With these optimizations enabled, performance of Kilo TM is improved by 65%.

WarpTM was extended to detect conflicts before sending transactions to the commit units and to stall transactions that are likely to conflict [17]. The first optimization is achieved by maintaining a *conflicting address table* per SIMT core to indicate if a word is read or written by committing transactions. When a transaction is running, the SIMT core can detect conflicts early by accessing this table. This table is also used by the second optimization: transactions that detect a conflict early can pause execution until the conflicting transaction successfully commits or aborts. After that, the transaction can continue execution, provided that the read-log is still consistent with memory.

In this thesis we contribute to this research area presenting our own hardware TM design. Our proposal focuses on transactions using local memory to store shared data [63, 61, 59, 60, 58]. In addition, our contribution focuses on minimizing the hardware resources required to implement TM and implements a serialization mechanism to ensure forward progress without requiring further programming (i.e., no software fallback code is required).

2.6 TM on low-power CPUs

In this thesis we discuss the impact that TM can have when executing applications on heterogeneous CPUs following the big.LITTLE desing. Besides our analysis and proposal, researchers have studied the impact TM can have on efficiency-oriented homogeneous CPUs. Note that most of the research carried out in this direction involves simulation for both power and performance estimation.

Gaona *et al.* [31] characterize the energy consumption of two hardware TM systems. In addition, they propose the use of dynamic serialization for HTM in order to reduce energy consumption by minimizing the amount of wasted work [30]. Moreshet *et al.* [44] and Ferri *et al.* [27] perform an energy analysis of hardware TM using simulations. Their results show an improvement in energy consumption when using TM as compared to lock-based approaches. Baldassin *et al.* use simulations to characterize the software TM library TL2 [21] using low-power ARMv7 processors. They propose a dynamic voltage and frequency scaling approach [7, 6] as well as a strategy based in the use of scratchpad memory [39] with the objective of reducing energy consumption. Sanyal *et al.* [54] propose clock-gating techniques to be used in hardware TM in order to reduce energy consumption and improve performance.

To contribute expanding the knowledge in this area, in this thesis we provide an analysis of software TM on a low-power heterogeneous processor [65, 66]. Results of this analysis are used to schedule TM-based applications on such processor and understand their behavior [64].

3 Transactional Memory on Heterogeneous CPUs

As TM becomes popular in the homogeneous multi-core CPU world, the scientific community is starting to study its potential in low-power multi-core CPUs. As we expose in Section 2.6, most of the research effort focuses on analyzing and improving TM (either STM or HTM) for low-power homogeneous CPUs by using simulation tools. At the moment of writing this thesis, heterogeneous CPUs are not yet a target of TM applications.

In contrast to prior work on TM on low-power CPUs, we follow a different approach when considering heterogeneous CPUs. Instead of directly proposing architectural and microarchitectural changes to existing big.LITTLE designs to add TM support, we analyze how existing STM libraries designed for homogeneous CPUs adapt to this kind of architectures. In particular, we characterize the applications from the benchmark suite STAMP [43] using the STM library TinySTM [25, 26]. Being able to characterize a set of well-known applications and a widely used TM library provides meaningful information to programmers who already developed their applications with homogeneous CPUs in mind. For instance, it helps to predict the application behavior when the OS decides to schedule it whether on the big cluster or the little cluster.

Once both the TM system and application behavior is known, we can propose improvements to allow the TM implementation to take advantage of the heterogeneous CPU. These improvements can be done at the architectural level, compiler level, library level, and/or runtime level. In our case, we decide to investigate the role that scheduling can have in improving execution time and energy efficiency of applications that use TM. By applying our desing to the highest possible level,

we propose a solution that can be used directly in off-the-shelf hardware and using proven and well-known libraries.

In summary, the contributions of this chapter are:

- An analysis of the benchmark suite STAMP using TinySTM as TM implementation on a big.LITTLE platform.
- A simple scheduler for heterogeneous CPUs aimed to highlight the benefits that scheduling can deliver when improving performance and energy efficiency of TM applications on heterogeneous CPUs.

3.1 Background: TinySTM and STAMP

Before going forward in this chapter, we introduce the set of applications and TM implementation analyzed.

TinySTM [25, 26] is a time-based, word-level software TM available from <http://tmware.org/tinystm>. TinySTM includes three implementations for transaction management: write-back (updates are buffered until commit time), write-through (updates are directly written to memory), and commit-time locking (locks are only acquired upon commit). By default, unless the contrary is stated, we use the write-back policy and not commit-time locking (i.e., encounter-time locking).

It is worth noting that TinySTM requires the C++ *atomic_ops* library. By default, TinySTM includes a stripped-down version of such library implemented for several architectures. The reason is that, internally, the library uses platform-dependent atomic operations via intrinsics. Unfortunately, the ARM architecture is not included among the implementations. Nonetheless, the OS installed in the platform used for evaluation incorporates an implementation of such library for ARM architectures that could be linked with TinySTM.

STAMP [43] is a benchmark suite widely used to evaluate TM systems. It includes 8 applications and a number of input parameters and data sets for evaluation. Its applications come from different domains: from science and engineering to machine learning and security. The 8 applications present different characteristics designed to stress different capabilities of the TM under evaluation: length of transactions, size of the read and write sets, transaction execution time, and amount of contention.

Table 3.1 summarizes the characteristics of the STAMP applications relative

Application	TX Length	R/W Set size	TX Execution Time	Contention
Bayes++	Long	Large	High	High
Genome++	Medium	Medium	High	Low
Intruder++	Short	Medium	Medium	High
Kmeans++	Short	Small	Low	Low
Labyrinth++	Long	Large	High	High
Ssca2++	Short	Small	Low	Low
Vacation++	Medium	Medium	High	Low
Yada++	Long	Large	High	Medium

Table 3.1: Applications included in the STAMP benchmark suite. Qualitative characterization is relative to each other.

to each other. For each application, STAMP provides a set of data inputs to be used. In our experiments we use the input marked as ++ (see [43] for more details) as it is the one providing more contention and requiring more computational resources.

3.2 The ODROID XU3 Platform

Once the TM implementation and the applications are known, we introduce our evaluation platform. The Odroid XU3 [2] is a low-power computation platform integrating a Samsung Exynos 5422 processor, based on the ARM big.LITTLE architecture (see Figure 3.1). In this processor, the big cluster is a 4-core Cortex-A15 CPU while the little cluster is a 4-core Cortex-A7 CPU featuring 2 Mbyte and 512 Kbyte L2 caches, respectively. The system integrates a Mali-T628 GPU and 2 Gbyte of low-power DDR3 RAM. As operating system, the ODROID comes with the Linux odroid 3.10.59+. INA231 power monitors¹ are included in such device and are used to measure the power consumption of memory, GPU and both clusters separately. The values stored in these monitors are available by reading the contents of the /sys directory in the filesystem. Note that the power monitors record instant values, but do not provide the power consumption during a specified time. A library has been developed to provide such functionality. This library starts a separate computing thread that reads the values of the power monitors and integrates them over time, providing power consumption in Joules. Sampling time for the measurements is 100 milliseconds. Once this sampling thread is running, a set of functions is provided to be used to measure the power consumption of different pieces of code.

¹<http://www.ti.com/product/INA231>

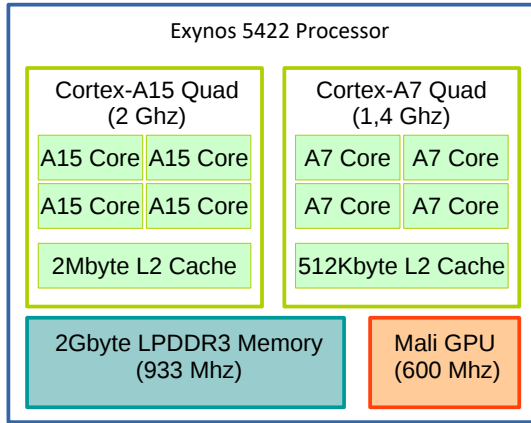


Figure 3.1: Diagram of the Exynos 5422 processor featured in the ODROID platform.

3.3 Isolated Energy/Performance evaluation

In this section we evaluate the behavior of the STAMP applications using TinySTM on top of the aforementioned Odroid platform. Applications were instrumented to access the INA231 power monitors available in the hardware to measure energy. Energy is measured for the whole chip by accumulating the values obtained from the four power monitors.

From the 8 applications available in STAMP, we selected 5, excluding Bayes, Genome, and Yada for different reasons. Bayes presents unexpected behavior, resulting in inconsistent performance results from one execution to another. Note that this issue has been previously documented in [51]. Genome exhibits synchronization errors which results in deadlocks when executing some multi-threaded experiments. In the case of Yada, the most part of the executions result in out-of-memory errors notified by the OS.

With respect to TinySTM we use the write-back policy and encounter-time locking mechanisms implemented. As a remainder, the write-back policy stores writes in a transaction-private write-set, which is updated to memory if the transaction reaches the commit instruction with no conflicts. The encounter-time locking technique locks memory positions accessed by the transaction when accessing to memory, in contrast to the commit-time locking mechanism which uses these locks when executing the commit operation.

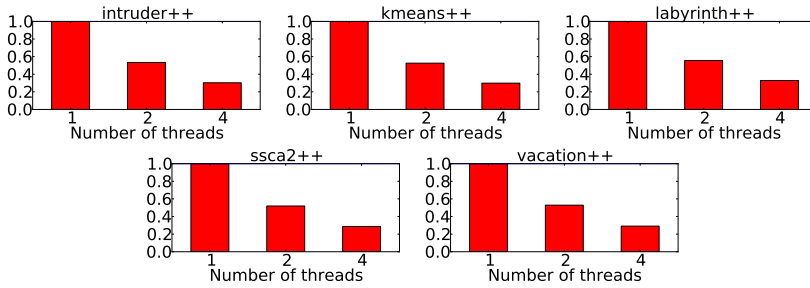
We examine 3 metrics for evaluation. Firstly, we evaluate the normalized execution time with respect to a single-threaded execution on the cluster that is being evaluated. Secondly, we measure the energy consumption, again, normalized to the energy consumption of a single-threaded execution. Lastly, using these metrics, we calculate the Energy-Delay Product (EDP). All the experiments were carried out comparing their results with respect to a single thread running TinySTM. For each application, we run 10 tests and calculate the average of its output. The experiments show consistent results for every execution. We do not use a sequential version of the code as we intend to provide scalability metrics for TinySTM instead of comparing TinySTM against other implementations.

3.3.1 Little cluster analysis

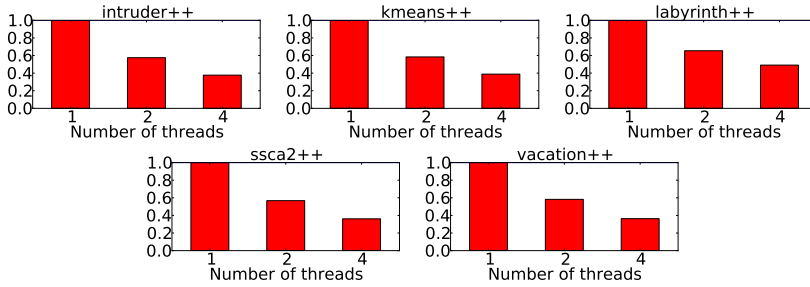
Figure 3.2 shows the results of the evaluation of the little cluster. In this case, execution is normalized to a single thread running on the little cluster. The threads are pinned to the little cores by using the `taskset` command provided by the OS. TinySTM achieves good scalability for the three parameters (execution time, energy consumption, and EDP). The performance scalability of TinySTM added to the power efficiency of the Cortex-A7 processor, allows us to observe a reduction of EDP between 80% and 90% with respect to 1 thread on a little core when running the application using 4 cores/threads.

3.3.2 Big cluster analysis

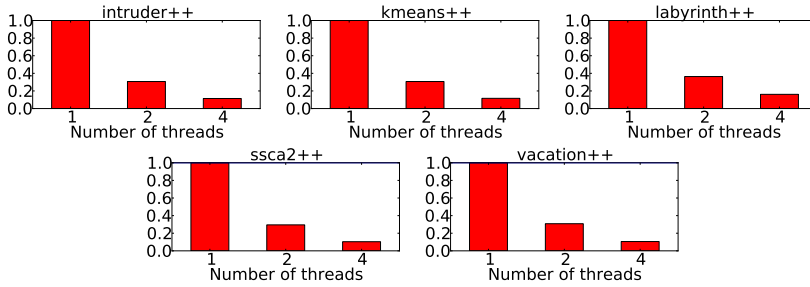
Figure 3.3 shows the evaluation of the big cluster. The picture is different as compared to the little cluster. In this case, we observe scalable results for the execution time, but not as scalable as in the little cluster. In addition, as the Cortex-A15 processor is not as power efficient as the Cortex-A7, we observe that the energy consumption is higher when using 4 threads in Intruder, Kmeans, and Vacation. Despite TinySTM shows to be scalable in terms of execution time, it is not in energy when running on high-performance processors. The scalability obtained in terms of execution time is not enough to compensate the energy increment when using 4 threads. As result, the EDP is not always optimal when using 4 threads, achieving better results when running on 2 threads.



a. Normalized execution time w.r.t 1 thread running TinySTM on a little core.



b. Normalized energy consumption w.r.t 1 thread running TinySTM on a little core.

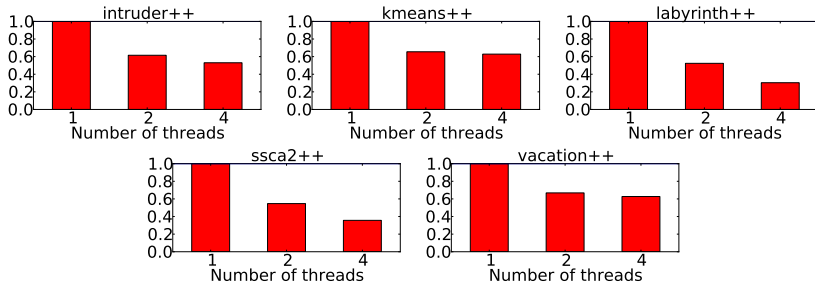


c. Normalized EDP w.r.t 1 thread running TinySTM on a little core.

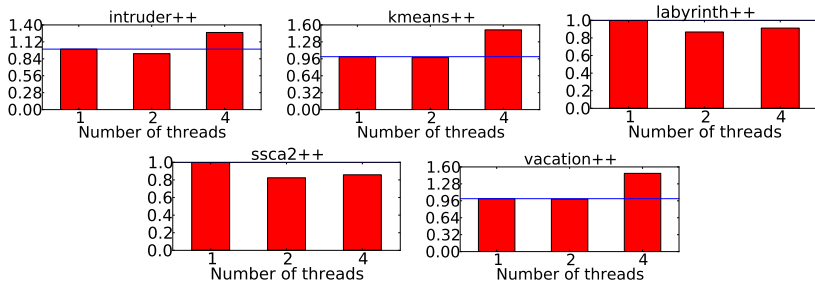
Figure 3.2: Little cluster evaluation.

3.3.3 Full system analysis

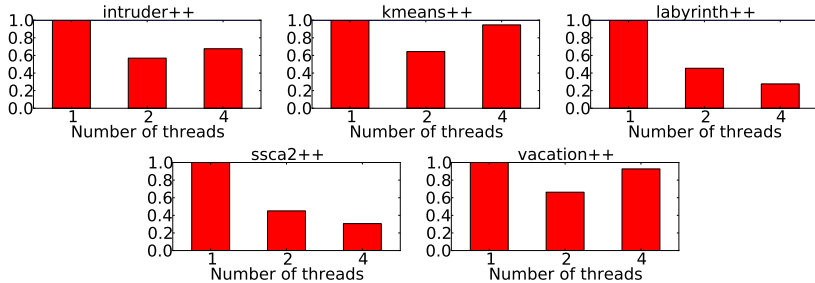
Figure 3.3 shows the evaluation for the full system. Before running these experiments, we observed that the OS scheduler tries to use the big cluster as soon as it detects any overload. We decided to keep the default behavior of the OS scheduler in order to analyze if some changes should be required for an optimal



a. Normalized execution time w.r.t 1 thread running TinySTM on a big core.



b. Normalized energy consumption w.r.t 1 thread running TinySTM on a big core.

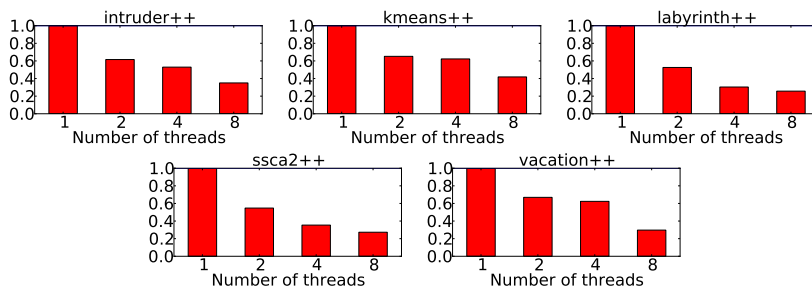


c. Normalized EDP w.r.t 1 thread running TinySTM on a big core.

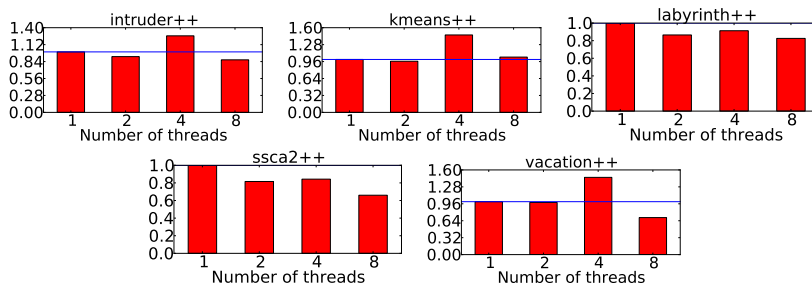
Figure 3.3: Big cluster evaluation.

scheduling. Results show that, using up to 4 threads, they are scheduled on the big cluster and the performance and energy results are similar. When adding 4 more threads, they are scheduled on the little cluster. The applications achieve good performance scalability when adding the Cortex-A7 multi-core processor. In addition, its power-efficiency results in a reduction of the energy consumed by

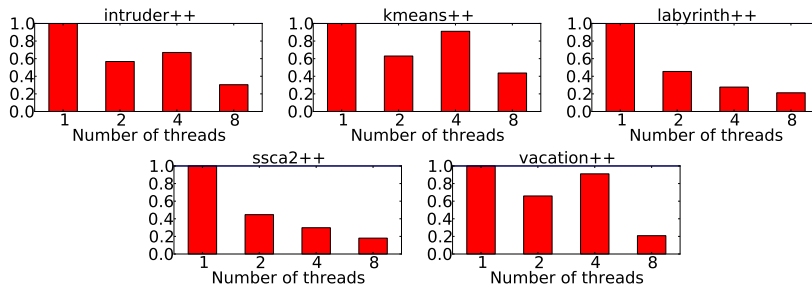
the applications when using 8 cores. The only exception is Kmeans. However, despite the small increment of the energy consumption as compared to the use of one thread, it is balanced by the execution time reduction achieved when using the little cluster. This results in an improvement of the EDP in all the cases.



a. Normalized execution time w.r.t 1 thread running TinySTM on a big core.



b. Normalized energy consumption w.r.t 1 thread running TinySTM on a big core.



c. Normalized EDP w.r.t 1 thread running TinySTM on a big core.

Figure 3.4: Full system (big + little) evaluation.

3.3.4 Little cluster and big cluster comparison

In Figure 3.5 we evaluate the performance of the little cluster as compared to the big cluster. For execution time, (Fig. 3.5.a) we calculate $ExTime_{A7}/ExTime_{A15}$ for 1, 2, and 4 cores. Values higher than one reveal that the big cluster performs better than the little cluster. Values smaller than one represent the opposite. Running the applications with a single thread results in better performance for the big cluster. However, we observe that (with the exception of Labyrinth), as we keep adding more threads and memory conflicts appear, it is not as efficient as the little cluster. The application Labyrinth, which features long transactions with many accesses to memory and where most of the code is inside a critical section, requires of a performance-oriented multicore CPU and is able to get full advantage of the big cluster. We perform the same comparison for the energy consumption. In this case, the low-power Cortex-A7 processor performs better than the Cortex-A15 in terms of energy consumed. The same applies for the EDP, except for Labyrinth. For this application, as the big cluster performed (in terms of execution time) much better than the little cluster, the EDP shows improvements between 1.5X and 2.5X despite the higher energy requirements.

3.3.5 Conclusions

The main conclusion we can draw out of this analysis is that both clusters behave differently when executing applications that use TM. In terms of performance and energy consumption, the little cluster presents more scalability as compared to the big cluster. However, applications such as Labyrinth that require more computational power and has high memory imprint are able to benefit from the performance-oriented architecture and the larger caches offered by the big cluster. Applications also benefit from using both clusters simultaneously, providing better performance at a lower energy consumption when using the whole chip.

3.4 Concurrent execution of TM applications on heterogeneous CPUs

The previous study considers a single application running on the heterogeneous CPU, even if only one of the clusters is occupied. As our goal is to schedule multiple applications to run simultaneously, it is important to understand if the

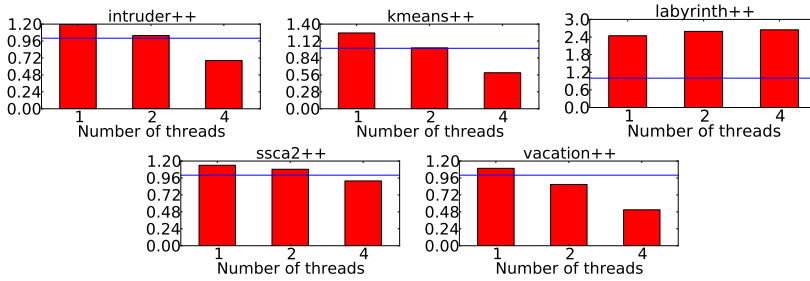
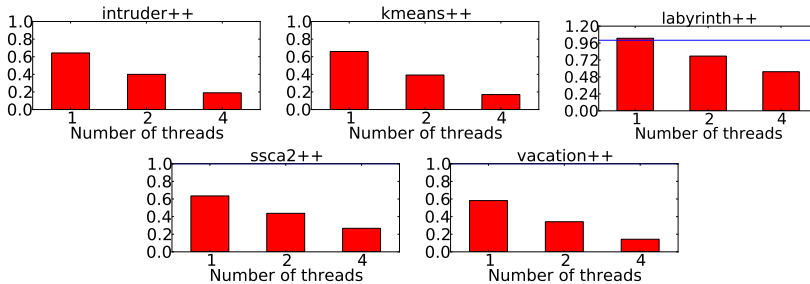
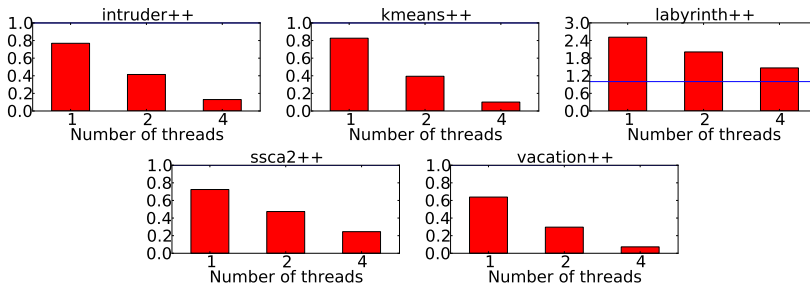
a. $ExTime_{A7}/ExTime_{A15}$.b. $Energy_{A7}/Energy_{A15}$.c. EDP_{A7}/EDP_{A15} .

Figure 3.5: Big cluster vs little cluster evaluation. Values higher than 1 indicate an advantage of the big cluster over the little cluster; values lower than 1 indicate the opposite.

conclusions drawn when executing a single application in the device stand true when executing two or more concurrently.

To do that, we paired the 5 applications evaluated with each other (one of the applications is set to run on the big cluster while the other is scheduled

on the little cluster). The main objective is to check if application behavior dramatically changes when a second application is executed in the other cluster. Note that this is not exhaustive testing, as there are some effects such a context switching that are not measured. However, this test provides some insight on the performance degradation that can be expected when executing two applications simultaneously. This way, we have 25 pairs of applications to be evaluated.

Table 3.2 shows the execution time of the 25 pairs of applications when running concurrently. Results are the average of 10 executions. For these experiments we consider that each application uses the whole cluster where it is scheduled (i.e., applications use 4 threads). In this table we can observe that the behavior of a given application executing on a given cluster does not change when executing concurrently another application on the other cluster; but performance degrades in different ways. For instance, the column in bold highlights the experiments where Kmeans is scheduled in the big cluster. Depending on the application scheduled on the little cluster, its execution time varies between 37.8 and 44 seconds. In the row in bold we observe the execution time of Kmeans, but this time scheduled on the little cluster. In this case, the execution time varies between 23.7 and 27.3 seconds. As conclusion, Kmeans always performs better running on the little cluster as compared to the big cluster. The same applies to the rest of the applications, with the exception of Ssca2 which present similar execution time for both clusters. This experiment was repeated for the number of transactions aborted, as well as the energy consumption, obtaining similar results. The conclusion of these experiments is that, if an application presents better behavior in one of the clusters with respect to the other, this behavior remains the same (but with some performance degradation) if another application runs on the other cluster.

In Table 3.3 we summarize the behavior of the applications when running concurrently with another application. All the applications show better energy consumption when running on the little cluster, due to its high efficiency. With respect to the number of aborted transactions, each application has different behavior depending on the cluster on which it is executed. Execution time is, in general, more favorable to the little cluster. There are several reasons that explain that behavior. For Kmeans, the number of aborted transactions is smaller in the little cluster. That means that the high performance offered by the big cluster is wasted executing transactions that abort. Another reason is that the eventual improved peak performance offered by the big cluster only benefit CPU-bounded and cache-hungry applications such as Labyrinth. Lastly, the heavy instrumentation introduced by TinySTM increases the pressure on the memory subsystem of the processor. The higher frequency at which the big cluster operates not only

Big	Little	Big	Little	Big	Little	Big	Little	Big	Little
Intr.	Intr.	Lab.	Intr.	Ssca2	Intr.	KM	Intr.	Vac.	Intr.
149.0s	103.9s	34.4s	88.0s	45.8s	98.9s	44.0s	91.4s	286.0s	96.3s
Intr.	Lab.	Lab.	Lab.	Ssca2	Lab.	KM	Lab.	Vac.	Lab.
129.3s	91.9s	37.2s	93.5s	36.4s	93.6s	39.9s	91.2s	273.5s	92.2s
Intr.	Ssca2	Lab.	Ssca2	Ssca2	Ssca2	KM	Ssca2	Vac.	Ssca2
138.2s	38.6s	35.2s	33.0s	33.1s	35.0s	37.8s	32.8s	277.9s	34.8s
Intr.	KM	Lab.	KM	Ssca2	KM	KM	KM	Vac.	KM
131.2s	26.3s	34.2s	23.7s	38.3s	26.1s	41.8s	27.3s	274.1s	24.2s
Intr.	Vac.	Lab.	Vac.	Ssca2	Vac.	KM	Vac.	Vac.	Vac.
143.3s	162.8s	35.8s	140.1s	39.1s	146.2s	41.3s	142.8s	292.3s	164.2s

Table 3.2: Execution time (in seconds) of the applications when running concurrently one on the big cluster and another on the little cluster. Intr, Lab, KM, and Vac stand for Intruder, Labyrinth, Kmeans, and, Vacation, respectively. To help understand this table, the row in bold marks the experiments where Kmeans is executed on the little cluster, while the column in bold marks those where Kmeans is executed on the big cluster. The remaining rows and columns are organized in the same way for the rest of the applications.

does not help to improve the execution time, but introduces higher contention on the memory, harming performance in a larger ratio as compared to the little cluster. This behavior has been observed in non-TM memory-bound applications, in which the latency in the accesses to memory and contention harms performance on the big cluster. Novel techniques are used to schedule the compute-bound part of the applications on the big cluster, while the memory-bound sections are scheduled on the little cluster [68].

	Execution time	Energy consumption	Aborted transactions
Intruder	little	little	similar
Kmeans	little	little	little
Labyrinth	big	little	similar
Ssca2	similar	little	big
Vacation	little	little	similar

Table 3.3: Summary of the behavior of the applications when executing concurrently with other application. For each metric, the values “big” and “little” indicate in which cluster the application performs better; “similar” means that differences are close to 10% or less.

To finish with this comparison, we want to highlight that the conclusions of this study are similar to those obtained by the isolated execution of the applica-

tions. If we compare Table 3.3 with the results in Figure 3.5, we observe some similarities. The comparison of both clusters when executing applications in isolation is similar to the obtained when executing concurrently two applications on the heterogeneous processor. This empiric evaluation allows us to conclude that, for this set of applications, the analysis can be done either in isolation or concurrently, and results are not affected in a significant way. Note that some degradation can be observed in the concurrent execution, but application behavior does not change.

3.5 Scheduling TM applications on heterogeneous CPUs

In this section we present our scheduling model for TM applications on the heterogeneous CPU. Our goal is to provide a prototype in which TM applications can be scheduled across both clusters to improve performance, energy efficiency, and reduce the overhead produced by aborted transactions.

3.5.1 Scheduling on heterogeneous CPUs

Besides the 3 basic scheduling modes supported by default in big.LITTLE CPUs (cluster switching, in-kernel switching, and global task scheduling) as described in Section 2.3, research has been carried out to provide more advanced techniques that improve energy efficiency and performance. Note that TM is not considered as part of these models, they consider multi-threaded applications with no special attention on the mutual exclusion mechanism implemented (if any is required). Normally, these techniques focus on two kinds of scheduling: *thread-to-cluster* and *thread-to-core*. The thread-to-cluster scheduling process is in charge of deciding if the application is suitable for running on the big cluster or, on the contrary, it is preferred to be scheduled on the little cluster. The thread-to-core scheduling, based on the application requirements and the availability of resources, decides on the number of cores to use on the given cluster. Recently, some techniques have been proposed scheduling the applications with higher computing requirements on the big cluster [9], scheduling applications based on time and progress fairness [42], and by tracking the behavior of the application in previous executions [16].

Our proposal is inspired in the recent work of Libutti *et al.* [41]. Their scheduler performs thread-to-cluster scheduling and gives some hints to the OS to

perform thread-to-core scheduling. Briefly, this scheduler works in two separate stages:

1. In the first phase, called analysis phase, applications are executed and analyzed, and the *stake functions* are calculated. These functions are calculated for each application and each cluster, and depend on the CPU time required by the application and the contention in computing and memory resources expected when running concurrently with other applications.
2. In the second phase, scheduling phase, information from the previous phase is used to schedule the applications. In particular, the values calculated by the stake functions are used to reduce execution time and minimize contention in the access to resources. The thread-to-cluster scheduling is performed attending to the expected resource contention. Applications that experiment higher contention are scheduled on the big cluster, provided that it is available for execution. The little cluster is used only for applications with lower contention or when the big cluster has no resources available. Once the cluster is selected, the scheduler observes the resources available and the stake functions of the application for the given cluster. If it is possible to allocate all the resources needed for the application (this information is provided by the stake function), then the application runs using all the resources needed and in isolation. In case that there are not enough resources (i.e., the cluster is partially occupied running other application), then the scheduler informs the OS on the resources needed by the application. At this point, the execution of the application is in charge of the OS scheduler.

3.5.2 ScHeTM: A TM-aware scheduler for heterogeneous CPUs

Inspired by the previous work on schedulers for heterogeneous CPUs, we propose a model that includes the overhead produced by aborted transactions as well as energy and performance metrics. We call our proposal ScHeTM (**S**cheduling - **H**eterogeneous CPUs - **T**M applications).

Our model shares some similarities with the presented by Libutti *et al.* [41]. Firstly, our scheduler also operates in two stages: a first stage in which applications are analyzed, and a second stage to perform the scheduling process. Secondly, the analysis is performed per-application and per-cluster, as in the previous proposal. Thirdly, this analysis involves the calculation of some functions

to characterize the application (in our case, we name this function as *suitability* function). Lastly, thread-to-cluster scheduling is also performed in our model.

In contrast to these similarities, ScHeTM have some key differences with prior work. In our case, we do not perform thread-to-core scheduling. We consider that the applications use all the resources available in the cluster, once the thread-to-cluster scheduling has been performed. Thread-to-core scheduling techniques are proposed as future work. Furthermore, in contrast to the stake function that only consider performance-related measurements, in our suitability function we take into consideration three metrics: execution time, energy consumption, and the amount of aborted transactions. Another difference is the addition of 3 parameters to adjust the importance of each one of these 3 metrics. This way, the behavior of the scheduler can vary depending on the status of the device (i.e., the scheduler can focus on minimizing energy consumption if the device is set in power-saving mode).

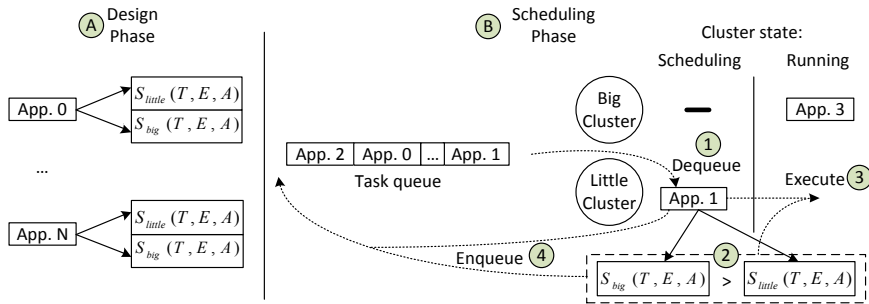


Figure 3.6: Diagram of the proposed scheduler ScHeTM.

Figure 3.6 shows an overview of the design of ScHeTM, which consists of two phases. During the first phase (A: *design phase*) applications are analyzed, calculating the suitability functions $S_{big}(T, E, A)$ and $S_{little}(T, E, A)$. These functions are used to score the behavior of the application in the given cluster. The parameters T , E , and A are metrics depending on execution time, energy consumption, and aborted transactions, respectively. Thus, applications should be properly instrumented during the execution of this phase to get the T , E , and A metrics, but this instrumentation can be omitted in the next phase. The second phase (B: *scheduling phase*) is in charge of assigning applications to clusters (thread-to-cluster scheduling). For simplicity, we assume that applications come from a task queue and that they should be executed in no particular or-

der. During this phase, a cluster can be either scheduling an application (if the cluster was idle) or executing an application that has been scheduled on it. This phase is executed each time a cluster is idle. In the example of Figure 3.6, we assume that the little cluster is idle while the big cluster is running an application (App. 3). The little cluster then dequeues an application from the input queue ① and proceeds to compare both suitability functions of this application ($S_{big}(T, E, A)$ and $S_{little}(T, E, A)$) ②. If the result is favorable to the little cluster ($S_{big}(T, E, A) \leq S_{little}(T, E, A)$), then the application is set to run on such cluster ③. Otherwise, the application is more suitable for executing on the big cluster. As the big cluster is busy, then the application returns to the task queue ④. The cluster, still idle, dequeues a new application from the task queue and repeats the process until it finds one more suitable for its execution. Note that, if one of the clusters is much better than the other, then the cluster can remain idle indefinitely. Thus, a mechanism is required to ensure that both clusters eventually receive an application to execute and avoid resource underutilization.

3.5.2.1 Design phase: suitability functions

In general, the suitability function is defined as $S_c(T, E, A) = tF_c(T) + eF_c(E) + aF_c(A)$, where c is one of the two clusters (big or little), $F_c(T)$, $F_c(E)$, and $F_c(A)$ are functions calculated from the execution time, energy consumption, and aborted transactions (respectively), and t , e , and a are parameters that weigh these three characteristics. Note that, in this approach, and T , E , and A follow "the lower the better" rule, so the suitability function is designed with that in mind.

$S_c(T, E, A)$ is designed to return values in the range $[0,1]$. The value 0 means that the cluster c is not suitable for executing the current application, while the value 1 means that this cluster is much better than the other. This value is calculated from the weighted sum of 3 functions ($F_c(T)$, $F_c(E)$, and $F_c(A)$) and 3 weights (t , e , and a). The three functions (represented in general as $F_c(N)$, where c is the current cluster and N is any of the parameters T , E , or A) are also designed to return values in the range $[0,1]$. This means that the weights must fulfill that $0 \leq t \leq 1$, $0 \leq e \leq 1$, $0 \leq a \leq 1$ and $t + e + a = 1$. The 3 weights are those parameters used to give more importance to one metric than the others, if the scheduler is set to meet performance, energy, or aborted transactions constraints. In the evaluation section of this chapter we assess the impact that tuning up these values can have in the behavior of the system.

The calculation of the $F_c(N)$ is very similar, but different, for each cluster. In the rest of the section we provide details for the calculation of $F_{little}(N)$,

but a similar process is done for the big cluster. The calculation follows the formula $F_{little}(N) = 1 - N(little)/N(big)$ where $N(little)$ and $N(big)$ indicate the values measured for the parameter N in the little cluster and big cluster, respectively. The explanation of this formula is that it returns higher values if the measurement in the big cluster is higher than in the little cluster. For instance, for the calculation of energy consumption, the value of $F_{little}(N)$ gets closer to 1 as better is the little cluster executing the application. In the case $F_{little}(N) = 0$ it means that both clusters perform the same for the metric N . The opposite calculation has to be performed on the big cluster (i.e., $F_{big}(N) = 1 - N(big)/N(little)$). The difference in performance, energy consumption, or number of aborted transactions in both clusters can make such formula return a negative value as the numerator is higher than the denominator. In these cases, the value is rounded up to 0, as this values already represents that the other cluster is more suitable for executing the application. Table 3.4 shows an example of an application, where the three metrics are measured and then $F_c(N)$ is calculated for all of them in both clusters.

Metric	Measured value	$F_{big}(N) = 1 - N(big)/N(little)$	$F_{little}(N) = 1 - N(little)/N(big)$
Execution time	big: 10 s. little: 20 s.	$F_{big}(T) = 0.5$	$F_{little}(T) = 0.0$
Energy consumption	big: 100 J. little: 50 J.	$F_{big}(E) = 0.0$	$F_{little}(E) = 0.5$
Aborted transactions	big: 500 little: 200	$F_{big}(A) = 0.0$	$F_{little}(A) = 0.6$

Table 3.4: Calculation of $F_c(N)$ for a sample application in both the big and little cluster. Negative values mean that the cluster is not suitable for executing the application and are rounded up to 0.

Once these 3 values are calculated for each cluster (6 values in total), then they can be stored in a table or a similar data structure. The suitability function for each cluster can be calculated from these values using the formula $S_c(T, E, A) = tF_c(T) + eF_c(E) + aF_c(A)$. In a static system, where the parameters t , e , and a remain unchanged, the suitability function can be calculated before execution and its results can be used through the scheduling process. In a dynamic system, where the preferences of the system can change over time, then $S_c(T, E, A)$ can be recalculated when required. Table 3.5 shows the calculation of $S_c(T, E, A)$ in a system where the performance, energy consumption and abort rate is balanced. Data used in this calculation comes from table 3.4. Note that the application performed better in 2 of the 3 metrics analyzed. As we can observe, the value

of $S_c(T, E, A)$ is higher in the cluster little, meaning that this cluster is more suitable for running the application.

$S_c(T, E, A) = tF_c(T) + eF_c(E) + aF_c(A)$
$S_{big}(T, E, A) = 0.17$
$S_{little}(T, E, A) = 0.37$

Table 3.5: Calculation of $S_c(T, E, A)$ using the data from Table 3.4 and using the parameters $t = e = a = 1/3$.

3.5.2.2 Scheduling phase: application execution

In this model we assume that the applications, which have been already analyzed to calculate the suitability functions, enter the system from a task queue. This assumption is similar to that made in HPC schedulers, where a queue management system is in charge of deploying the applications to the computing resources. In addition, is not unusual that users of these systems have to specify some requirements of the application, such as memory and disk usage, and expected computation time.

At the beginning, both clusters are idle and dequeue different tasks from the task queue. Each cluster tests $S_{big}(T, E, A) > S_{little}(T, E, A)$ for the application that has been assigned. If the comparison is favorable to the current cluster (i.e., true for the big cluster and false for the little cluster), then the application executes on such cluster. On the contrary, if the comparison is favorable to the other cluster, then the application is sent back to the task queue. In that case, a new application is dequeued from the task queue, repeating the previous evaluation. This way, each cluster executes the applications where execution time (or energy consumption, or the number of aborted transactions) is predicted to be lower as compared to the other cluster.

The system described previously presents an important disadvantage, which is resource underutilization: if a cluster is always worst than the other, then no application is scheduled there. In some systems this can be tolerable and even desirable, but in many other cases it is interesting to exploit the full potential of the heterogeneous CPU. To correct this issue, we define the parameter B as the maximum number of applications discarded per cluster. A per-cluster counter, r is set to 0 each time an application is scheduled on such cluster. If the application is discarded by the cluster, then ScHeTM performs $r = r + 1$. In the case $r = B$, then this application is forced to execute on such cluster,

regardless of the value of $S_c(T, E, A)$. To soften this change in the behavior of ScHeTM, instead of comparing the values of $S_c(T, E, A)$, the values compared come from this formula: $S_c(T, E, A) + r * (1 - S_c(T, E, A)) / B$. The intuition of this formula is to leave the value of $S_c(T, E, A)$ as it is in the first retry ($r = 0$) and scale it linearly until $r = B$. In that case, the value obtained is the constant 1, which is the maximum value returned by $S_c(T, E, A)$. This way, in a system with more than B applications, each cluster eventually executes at least one. This simple approach solves the problem of resource underutilization, but has some disadvantages that we discuss during the evaluation.

3.5.2.3 Evaluation of ScHeTM

ScHeTM is evaluated on the Odroid XU3 platform which is described in Section 3.2. The input task queue is composed by 25 applications, which is a random combination of the 5 applications previously evaluated, using the same configuration and input parameters. By checking back Table 3.3, we observe that Labyrinth is the only application with better performance in the big cluster. To evaluate a more balanced scenario, we increased the probability of such application to appear in the system. In total, 10 out of the 25 applications in the queue are instances of Labyrinth. In our test we also set the maximum number of rejected applications $B = 5$. All results presented in this section are the average of 10 executions.

In this section we analyze a baseline scheduler, and 4 different configurations of ScHeTM. We analyze 3 metrics, which correspond to the configurable parameters provided in ScHeTM: execution time, energy consumption, and number of aborted transactions. Each of the metrics is represented by a figure with 3 columns. The two first columns represent the value measured in the little cluster and the big cluster. The column labeled as “Total SoC” represents the value observed in the whole system-on-chip as is calculated in a different way for each metric.

- In the case of execution time this column is calculated as the maximum value of the other two, as the time taken to execute all the applications corresponds to the time employed by the slowest cluster.
- The case of energy consumption is harder to analyze. In our instrumentation, each cluster measures the energy consumption for the whole chip, as there are some parts (for instance, the memory subsystem) that are hard to isolate when two applications are executed simultaneously each in one

cluster. This means that some of the measurements are taken twice (once per cluster) as the applications execute concurrently. For this reason, the column Total SoC represents the maximum value of energy consumed by any of the clusters (i.e., to avoid adding the same measurements twice, each cluster records the energy consumption of the whole chip during the execution and Total SoC is the highest of these values). In practice, usually corresponds to the energy consumed by the big cluster. We consider this a valid approximation as we are interested in the energy consumption of the whole system, not the individual applications.

- With regards to the number of transactions aborted, we obtain the column Total SoC by adding the values of the other columns.

Baseline scheduler. Before evaluating ScHeTM, we designed a greedy scheduler to be utilized as baseline. The goal is to observe if ScHeTM is able to improve the results provided by a simple scheduling technique. As in ScHeTM, we consider that the greedy scheduler receives the applications to be scheduled from a task queue. Whenever any of the clusters is idle, the greedy scheduler dequeues an application from the task queue and assigns it for execution on said cluster. Thus, the greedy scheduler never returns an application back to the task queue, as every cluster executes the application it receives without considering its expected performance. Although the scheduling may not be optimal, this scheduler intends to keep the clusters busy as much as possible.

Figure 3.7 shows the execution time, energy consumption, and aborted transactions when using the greedy algorithm to schedule the task queue of 25 applications. In this case, execution time is balanced, being the little cluster a bit behind the performance offered by the big cluster. A similar scenario occurs for the energy consumption: both clusters require the similar amount of energy to execute the applications but, in this case, the big cluster consumes a bit more energy as compared to the little cluster. A very different situation happens when measuring the number of aborted transactions. In this case, the big cluster aborts more transactions than the little cluster. Note that the little cluster does not abort 0 transactions, but a very small number of them as compared to the big cluster. The reason is that the applications that produce more aborted transactions are scheduled in the big cluster. In particular, Intruder and Kmeans produce millions of aborted transactions, as compared to the tens, hundreds, or thousands signaled by the other applications.

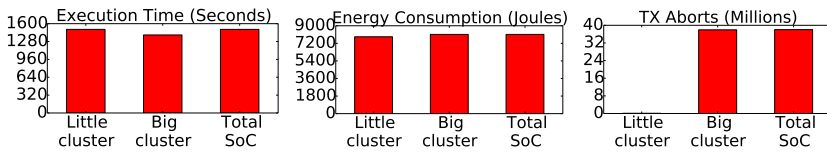


Figure 3.7: Evaluation of the greedy scheduler. Execution time is provided in seconds, energy consumption in Joules, and aborted transactions in millions.

Balanced ScHeTM. In this scenario, we use ScHeTM to schedule the same set of applications using a balanced configuration: $t = e = a = 1/3$. This way, the system does not have preference for any metric above the others.

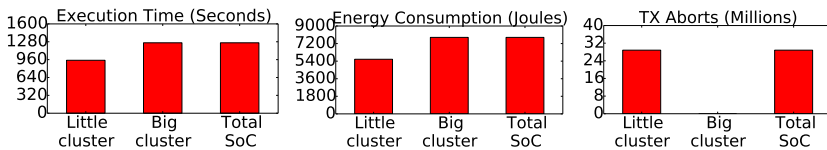


Figure 3.8: Evaluation of ScHeTM using a balanced configuration ($t = e = a = 1/3$). Execution time is provided in seconds, energy consumption in Joules, and aborted transactions in millions.

In Figure 3.8 we observe the results obtained by this configuration of ScHeTM. In this scenario, 17 out of the 25 applications have been scheduled on the big cluster and the remaining ones on the little cluster. ScHeTM was able to identify the computing demands of Labyrinth, and all the 10 instances were executed on the big cluster in every experiment. In the greedy scheduler, Labyrinth used equally both clusters, depending on the execution. For this reason, execution time is reduced by 20% when using ScHeTM. In addition, ScHeTM was able to identify that, for the applications that produce more aborted transactions, execute more efficiently on the little cluster. By scheduling them on such cluster, the amount of aborted transactions is reduced by 33% in comparison to the greedy scheduler. However, as the power-hungry big cluster executes more applications than the little cluster, energy consumption increases by 7%.

Performance-oriented ScHeTM. As ScHeTM can be configured via its parameters to set a preference for execution time, energy consumption, or aborted transactions. In order to emphasize in execution time, we set the parameters $t = 0.7$ and $e = a = 0.15$. This way, execution time has a higher weight in the calculation of $S_c(T, E, A)$ than the other parameters.

Figure 3.9 shows the evaluation of ScHeTM using this configuration. Although the number of applications executed on each cluster is the same as in the balanced configuration, their distribution is different. Only Labyrinth and Ssca2, which are able to benefit from the extra computational power provided by the big cluster, are always scheduled on such cluster. This lead to a better workload distribution, as the little cluster executes only those applications whose performance does not benefit from the big cluster. As result, we get a reduction in the total execution time of the system of about 40% compared to the greedy scheduler. Such reduction in execution time goes along with a reduction in energy consumption of 15%. The number of aborted transactions remains in an intermediate value between the greedy scheduler and the balanced version of ScHeTM, but the current configuration does not focus on reducing this overhead.

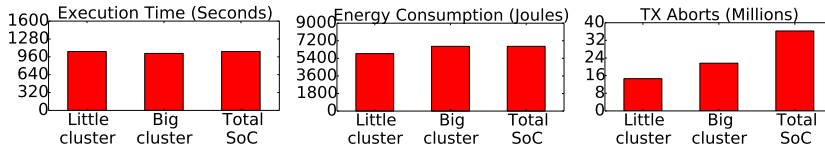


Figure 3.9: Evaluation of ScHeTM using a performance-oriented configuration ($t = 0.7$ and $e = a = 0.15$). Execution time is provided in seconds, energy consumption in Joules, and aborted transactions in millions.

Efficiency-oriented ScHeTM. This second configuration is designed with the goal of reducing the energy consumption of the system. Thus, the parameters are set as $e = 0.7$ and $t = a = 0.15$.

In Figure 3.10 we observe the results of this evaluation. From these results we can deduce that such configuration has a negative effect in the parameter that was intended to be optimized: energy consumption has increased by 6% compared to the balanced configuration of ScHeTM. The reason for this behavior is simple: all the applications are more efficient in terms of energy consumption when executing on the little cluster (see Table 3.3 in Section 3.4). When evaluating the suitability functions, in every case the result of the comparison $S_{little}(T, E, A) > S_{big}(T, E, A)$ is true, and the little cluster accepts all the applications for execution. On the contrary, the big cluster rejects the execution of the applications and resorts to the mechanism to avoid resource underutilization. As result, the big cluster rejects executing 5 applications and is forced to accept one in spite of its energy efficiency. For this reason, the behavior of the system is similar to the greedy scheduler: applications get executed equally by both clusters. Later we discuss how the effects of this problem can be mitigated.

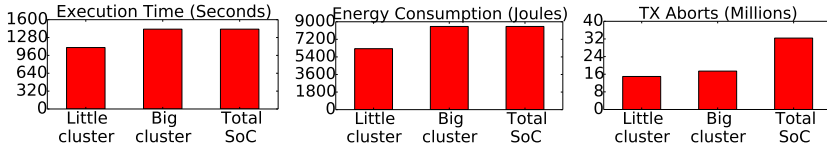


Figure 3.10: Evaluation of ScHeTM using an efficiency-oriented configuration ($e = 0.7$ and $t = a = 0.15$). Execution time is provided in seconds, energy consumption in Joules, and aborted transactions in millions.

Transaction-oriented ScHeTM. The last configuration of ScHeTM that we are evaluating focuses on reducing the overhead produced by aborted transactions. To set this configuration we use the parameters $a = 0.7$ and $t = e = 0.15$.

Figure 3.11 shows the results of the evaluation of such configuration. The results are quite similar to the obtained by the balanced configuration of ScHeTM. The main reason is that in 3 out of the 5 applications the difference in the number of aborted transactions is not significant. A second reason is that, even if an application has a lower number of aborted transactions on one of the clusters, the difference in the other parameters (i.e., time and energy) is still more noticeable. For instance, Ssca2, has 13% more aborted transactions on the big cluster compared to the little cluster, but is 4 times more energy-efficient in the later. On the contrary, scheduling Kmeans on the big cluster reduces the number of aborted transactions by 40%, but it consumes 5 times the energy consumed by the little cluster. For that reason, the a parameter is less relevant in the application behavior as compared to the others, and no substantial benefit can be obtained from this configuration compared to a balanced execution.

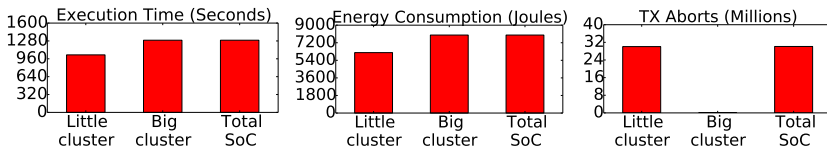


Figure 3.11: Evaluation of ScHeTM using a transaction-oriented configuration ($a = 0.7$ and $t = e = 0.15$). Execution time is provided in seconds, energy consumption in Joules, and aborted transactions in millions.

3.5.2.4 Conclusions

With ScHeTM we explore the possibilities and impact that scheduling TM applications on heterogeneous CPUs can have in terms of performance, energy consumption, and reducing transaction overhead. With a simple and configurable model, we are able to obtain up to 40% reduction in execution time and up to 15% reduction in energy consumption, and in both cases reducing or keeping similar the amount of aborted transactions.

However, we have two concerns about our proposal that should be addressed by more complex scheduling systems. The first problem is that, if one of the parameters is always better in one of the cluster, then ScHeTM is not able to find a schedule better than a greedy scheduler. This is due to our implementation of the mechanism that avoids resource underutilization, which is designed keeping simplicity in mind. However, more advanced solutions can consider, instead of returning applications to the queue when they are not suitable for execution, keeping them some time on the side. Once the cluster has rejected a number of applications, then it can have access to the last N rejected applications and have the opportunity of choosing the one whose execution produces lower degradation in performance, energy consumption, or aborted transactions. For instance, if the big cluster rejects 5 applications, then it should have the opportunity of choosing the one with better $S_{big}(T, E, A)$; or that one whose difference between $S_{big}(T, E, A)$ and $S_{little}(T, E, A)$ is smaller. The second problem relates to these metrics whose difference is relatively small compared to the others. We have observed this when trying to optimize the number of aborted transactions: the difference in aborted transactions among clusters is smaller than the difference in energy consumption. Thus, the number of aborted transactions has lower impact in the overall performance and trying to optimize such parameter does not lead to a more efficient schedule.

3.5.3 TM on heterogeneous CPUs: conclusions and future work

In this chapter of the thesis we explore the behavior of an existing STM library when executed on a big.LITTLE platform. In particular, we test a set of applications from the STAMP benchmark suite which have been linked with the STM library TinySTM. These applications are analyzed in both clusters separately, using both at the same time, and running concurrently an application on each cluster. Furthermore, we have designed ScHeTM, a scheduler for heterogeneous CPUs that performs the schedule taking into consideration a TM-related metric

(aborted transactions).

The main conclusions and contributions of this chapter are the following:

- For the set of applications analyzed, the little cluster presents better scalability and lower energy consumption than the big cluster.
- Applications with higher computing demands still benefit from the power offered by the big cluster.
- For this set of applications, if an application performs better in cluster A when executes in isolation, it still executes better (but with degraded performance) in such cluster when an application is executing concurrently on cluster B.
- We propose a simple scheduling model that permits to select which parameter to optimize: execution time, energy consumption, or aborted transactions. A proper configuration of such parameters results speeds up execution time in up to 40% and reduces energy consumption in up to 15%.
- We observe that optimizing a well-differentiated parameter (excecution time, in our case) improves the overall behavior of the application.
- The scheduling scheme is not able to improve a greedy scheduler if the applications behave always better in one of the cluster, or if the clusters behave similarly for the selected parameter.

We want to propose some directions for future work. As mentioned in Section 3.5.2.4, the mechanism included in our scheduler used to avoid resource underutilization can be improved to select more appropriate applications. In addition, the desing phase (training) of the scheduler could be omitted - applications can be assigned to idle clusters (as in the greedy scheduler) the first time they are executed and information can be gathered from such execution. In subsequent execution, if enough information has been obtained, the application can be executed followed the policy established by ScHeTM. Additionally, non-TM applications have to be considered in the model, as many applications do not require of synchronization or use other synchronization techniques. An immediate way to include these applications is to consider that they present 0 aborted transactions (i.e., if no TM is used, then the application does not abort any transaction). Also, the model should consider implementing a thread-to-core mechanism to complement the existing thread-to-cluster proposal. This way, applications that do not require the full cluster, or even single-threaded applications, can be managed by the scheduler.



UNIVERSIDAD
DE MÁLAGA

4 Transactional Memory on Heterogeneous CPU+GPU processors

In multi-core CPUs, TM has emerged as a promising alternative to the use of locks. As consequence, CPU vendors are including hardware TM as part of their commercial CPUs [72, 67, 37, 3]. At the same time, GPUs have become default accelerators for graphics and data-parallel algorithms. For that reason, CPU vendors are also including integrated GPUs as part of their processors, creating the so-called Accelerated Processing Units (APUs). However, both worlds (APU processors and TM) are still separated.

In this chapter we prototype a Software TM (STM) library targeting APU processors. The goal is to allow for transactions in both the CPU and the GPU simultaneously. The implementation of TM in heterogeneous CPU-GPU processors is a challenging task, as both CPU and GPU work under different programming models. Multi-core CPUs follow the MIMD model, where multiple cores may operate on different data or a shared memory, while GPUs follow the SIMD programming model, where multiple threads execute the same instruction (lock-step execution) on different memory positions. Another challenge is the memory space. CPUs have access to a main memory via a coherent cache hierarchy. On the GPU, main memory is accessed via a cache hierarchy where, in most cases, the L1 data cache is not coherent. In addition, GPUs feature a low-latency scratch-pad memory (shared memory in CUDA, local memory in OpenCL, or tiled memory in C++AMP) that can be used to accelerate the management of GPU-private transactional metadata. Lastly, communication between CPU and



GPU is an important problem to solve. Platform atomics ensure that values communicate effectively between both devices, but these operations are expensive in terms of memory latency.

To better understand and overcome these challenges, we propose APUTM, a software TM designed to work on APU processors. APUTM can be configured to run separately on the CPU cores or the GPU, or simultaneously in both devices, ensuring mutual exclusion in any case. APUTM is inspired by NOrec [20], combining a fast timestamp-based conflict detection mechanism with a precise value-based validation. For the GPU, APUTM implements a mechanism to allow for parallel commits of transactions while updating the timestamp information to communicate with the CPU cores. In our evaluation, we use a synthetic workload and 3 microbenchmarks to analyze different configurations of APUTM. We provide a discussion on the impact that our design decisions have on the performance of APUTM on both devices and provide hints for future improvements.

4.1 Background: NOrec and GPU-STM

The design of APUTM is inspired by two different STM proposals. On the CPU side, our design takes out some ideas from NOrec, while the GPU side is inspired by GPU-STM.

NOrec [20] implements conflict detection by comparing the values read from memory during the transaction with the values in memory at commit time. Mismatching values lead to a transaction abort. To speed up this process, NOrec uses a single global sequence lock to serialize writer transactions. If this lock has not changed since the last validation, then the transaction commits its changes to memory immediately and updates the global sequence lock. This way, read-only transactions are linearized before writer transactions, allowing these to commit faster.

On the GPU side, GPU-STM [70] offers many features desirable for APU architectures. In a similar way as NOrec, GPU-STM uses the concept of global sequence lock, making feasible the integration of both solutions. In addition, GPU-STM allows for parallel commits of transactions that do not conflict. However, this is implemented by using a set of locks, which requires executing a high number of atomic operations. In our design, instead of that, we synchronize transactions using the global sequential lock and use other techniques to permit parallel commits.

4.2 APUTM

In this section we present APUTM, a software TM designed specifically for APU architectures. APUTM is aimed at minimizing the number of platform atomic operations required for proper CPU-GPU communication. This is achieved by sharing a single global sequence lock (inspired by NOrec) among CPU and GPU transactions. APUTM implements lazy version management: each transaction keeps track of its speculative writes to memory in a private write-set, which has to be committed to memory if the transaction ends with no conflict. Conflict detection is performed lazily using the aforementioned global sequence lock and a value-based validation (if required).

In the following subsections we discuss the transactional metadata required to implement APUTM, the conflict detection and version management mechanism on both computing devices (CPU and GPU), and some notes on the correctness of the algorithm.

```

1 //Global metadata
2 atomic_int * gclock;

1 //Private metadata
2 struct pr_descr{
3   int snapshot;
4   int status; //RUNNING or ABORTED
5   <address,value> * reads;
6   <address,value> * writes;
7 };

1 //Wavefront metadata
2 struct wf_descr{
3   atomic_long * commit_mask;
4   atomic_int * leader_id;
5   <address,owner> * wf_writes;
6   atomic_int * next_write;
7 };

```

Figure 4.1: Global, private, and wavefront metadata required to implement APUTM.

4.2.1 Transactional Metadata

As seen in Figure 4.1, we classify the metadata required to implement APUTM in three types. The first type is *global metadata*, which is shared by CPU and GPU transactions and, thus, must be accessed and modified using platform atomics. The only variable of this kind is a global sequence lock, which we will refer to as *gclock*. The second type of metadata is *private metadata*, which is individually owned by each CPU transaction and GPU transaction, and consists of 4 items: a snapshot containing the value of *gclock* the last time the transaction was proven consistent, the status of the transaction (either RUNNING or

ABORTED), a read-set containing $\langle \text{address}, \text{value} \rangle$ pairs of the memory reads performed by the transaction, and, lastly, a write-set with the same structure containing the speculative writes to memory. The third type of metadata, called *wavefront metadata*, is GPU-specific. This metadata is shared among the GPU threads that execute in lockstep and is allocated in the scratch-pad memory to accelerate its utilization. The first element is a bit mask with one bit per thread within the wavefront called *commit-mask* used to mark conflicting threads. The variable *leader_id* contains the identifier of the thread within the wavefront in charge of accessing global metadata. In addition, it contains a shared write-set with $\langle \text{address}, \text{owner} \rangle$ pairs, which is used to note down the thread within the wavefront that intends to write to an specified memory address. We call this shared write-set *wf-write-set*, and *next_write* is the variable pointing to the next empty position in such set.

```

1  TMBegin(){
2  clear(reads)
3  clear(writes)
4  status=RUNNING
5  snapshot=gclock.atomic_load()
6  if (GPU){
7    next_write.atomic_store(0)
8    clear(commit_mask)
9  }
10 }

11 TMCommit()
12 {
13  if(GPU)
14    checkWFConflicts()
15  acquire(global_lock)
16  consistent= checkConsistency()
17  if(consistent){
18    if(!empty(writes)){
19      for(<addr, val> in writes)
20        mem[addr]=val
21        atomic_add(gclock, 1)
22    }
23  }else{
24    abort
25  }
26  release(global_lock)
27 }

28 TMWrite(addr, val)
29 {
30  if(contains(writes, addr){
31    update(writes, addr, val)
32  }else{
33    add(writes, addr, val)
34    if(GPU) {
35      p=atomic_add(next_write,1)
36      wf_writes[p]=<addr, tx_id>
37    }
38  }
39 }

40 TMRead(addr)
41 {
42  if(contains(writes, addr){
43    val=get(writes, addr)
44  }else{
45    consistent=
46      checkConsistency()
47    if(consistent){
48      val=mem[addr]
49      add(reads, addr, val)
50    }
51  }
52  return val
53 }

```

Figure 4.2: Functions provided by the APUTM interface.


```

54 checkConsistency(){
55   if(snapshot==gclock.atm_load())
56     return TRUE
57   do{
58     time=gclock.atm_load()
59     for(<addr, val> in reads){
60       if(mem[addr]!=val)
61         return FALSE
62     }
63   }while(time!=gclock.atm_load())
64   snapshot=time
65   return TRUE
66 }

67 acquire(lck){
68   if(GPU){
69     atomic_store(leader_id,th_id)
70     if(th_id==leader_id)
71       while(!atomic_cas(lck,0,1)){}
72   }
73   if(CPU){
74     while(!atomic_cas(lck,0,1)){}
75   }
76 }

77 checkWFConflicts(){
78   for(<addr1,val> in writes{
79     for(<addr2,ownr> in wf_writes{
80       if(addr1==addr2 &
81         ownr!=th_id &
82         commit_mask(ownr)==0){
83         commit_mask(th_id)=1
84         abort
85       }
86     }
87   }
88   for(<addr1,val> in reads{
89     for(<addr2,ownr> in wf_writes{
90       if(addr1==addr2 &
91         owner!=th_id &
92         commit_mask(ownr)==0){
93         commit_mask(th_id)=1
94         abort
95       }
96     }
97   }
98 }

asd

```

Figure 4.3: Auxiliary functions in APUTM.

Discussion: automatic / semi-automatic metadata generation: The metadata required by APUTM is implemented as part of the library and its details are hidden to the users of the library (programmers). However, many of its data structures can be defined either statically (with pre-defined size) or dynamically. At the moment, dynamic memory allocation is not supported by most GPUs and, thus, we consider a static approach. However, this decision comes with the disadvantage that the size of the read-set, write-set, and the wf-write-set must be known at compile time. Currently, we expect that programmers provide an upper bound of the size of these structures to allocate enough room for transactional metadata. Compilers can also help to determine this value by performing a static analysis of the code: if the number of read and write accesses can be determined before execution, then the transactional metadata can be pre-allocated by the compiler. Otherwise, if the static analysis of the code is not able to determine the amount of metadata required, a different approach must be considered. For instance, a pool of memory can be pre-allocated to store all transactional metadata, and APUTM can be adapted to manage the access to

such memory space. In case the memory pool reaches its maximum capacity, and space is still required to store transactional metadata, transactions should abort (capacity aborts). In that case, APUTM should be replaced by another mutual exclusion mechanism such as locks. This is a corner case which is unlikely to appear, as transactions are not intended to operate on data whose size is close to the amount of main memory installed. This proposal is future work beyond the scope of this thesis.

4.2.2 Version Management

APUTM implements lazy version management. Figures 4.2 and 4.3 show the pseudo-code implementing APUTM, which is used to describe version management. In the `TMBegin` operation, both CPU and GPU transactions reset their write-sets (line 3). GPU transactions also reset their `wf-write-set` by setting the next write position to 0 (line 7). `TMWrite` operations register speculative writes in both the private write-set (line 31) and the `wf-write-set` (in case of GPU, lines 34-37), but only if the memory location has not been previously accessed. In the case of successive writes to the same memory location, only the private write-set should be updated (line 31, the `wf-write-set` still keeps the proper ownership information and does not need to be updated). Values in the private write-set should be accessed during the `TMRead` operation in case the memory value of a previously written location is requested (line 43). During the `TMCommit` operation, if there is no conflict, the private write-sets are committed to memory (lines 19-20). Note that GPU transactions executing within the same wavefront commit their write-sets in lockstep. Thus, the conflict detection mechanism should ensure that this step is only reached by threads that have no conflicts with other GPU threads within the same wavefront.

4.2.3 Conflict Detection

Conflict detection is performed lazily (i.e., implemented as part of the `TMCommit` operation). As the CPU and GPU offer different programming models (MIMD and SIMD, respectively), conflict detection is implemented differently in each device.

CPU transactions acquire a global lock to perform the commit operation (line 15). Once acquired, no changes are permitted to memory by other transactions and the transaction starts the consistency checking (line 16). If `gclock` has not changed since the last consistency checking, then the transaction may

commit (line 55). Otherwise, the read-set must be validated against the current values in memory to detect a conflict if one of the values has been modified (lines 60-61). If validation succeeds, the transaction is able to commit its write-set to memory (lines 19-20), update the value of `gclock` (line 21) and release the global lock (line 26). Note that read-only transactions do not update `gclock` (line 18) and are linearized at the moment indicated by the current value of `gclock`.

GPU transactions commit as CPU transactions, but before acquiring the global lock, APUTM filters out the transactions that conflict with other transactions within the same wavefront (line 14). To do that, they check if their private write-set contains a value that another transaction within the wavefront intends to write by checking the `wf-write-set` (lines 78-87). If the owner of such entry is a different transaction, then the transaction aborts (lines 80-84). To avoid being aborted by a *doomed* (i.e., previously aborted) transaction, the bit corresponding to the owner transaction is checked from the `commit_mask` (line 82). To inform other transactions within the wavefront that the current transaction is aborted, it sets its corresponding bit in the `commit_mask` to 1 (line 83). Modifications and reads to such mask (lines 82-83, 92-93) should be implemented atomically as multiple transactions access the mask simultaneously. Note that all transactions within the wavefront are analyzing the same `wf-write-set` entry simultaneously. Following this process, only one of the writers to a given memory location is able to remain active while the rest of them abort. In addition, if there are multiple writers to the same memory location, there exist multiple entries in `wf-write-set` corresponding to such location. The first of these entries determines which of the writes will be committed to memory if there are no other conflicts. Once we have filtered out the transactions that conflict because they intend to write the same locations, then we have to detect read-write conflicts (i.e., transactions that have read a memory location that another transaction intends to write). The process is performed in a similar way as explained before, but comparing the private read-set of each transaction against the `wf-write-set`. Transactions that did read a memory position that another non-aborted transaction intends to write are marked as conflicting (lines 88-97). At this point, only non-conflicting transactions within the wavefront are still active. These proceed to the second stage of the conflict detection mechanism, which is detecting conflicts against transactions in other wavefronts and CPU transactions. This is achieved by acquiring a global lock as in CPU (line 15). Note that all the non-conflicting transactions will intend to acquire the global lock in parallel with two negative side effects. Firstly, it can create a deadlock if this is not implemented properly in the SIMT programming model (see [36] for some examples of this issue). Secondly, even if implemented correctly, only one of the threads within the wavefront is able to

acquire the lock creating unnecessary serialization within the wavefront. To overcome this problem, we select a leader transaction among the active ones within the wavefront to be in charge of acquiring the global lock in order to allow the whole wavefront to proceed to conflict detection in parallel (lines 68-72). Usually, in GPU programming, this is achieved by selecting the thread with lower ID. However, due to speculative execution, the thread with lower ID might have conflicted earlier and, thus, would not be active at this moment. Instead of that, we use a variable shared by the whole wavefront called *leader_id*. Executing in lockstep, each non-conflicting transaction atomically writes its own ID in such variable (line 69). Once all writes are performed, the transaction in charge of acquiring (and later releasing) the global lock is the one whose ID coincides with the value of *leader_id* (the release lock function is not depicted in Figure 4.3, but its implementation is similar to the acquire lock function). Note that lockstep execution combined with atomic operations ensures that all writes to *leader_id* have been performed before proceeding to acquire the global lock. When the global lock is acquired by the leader transaction, the whole wavefront proceeds to conflict detection as in CPU (lines 16-25).

4.2.4 Misellanea

Correctness. Changes in memory performed by committed transactions may leave the private read-sets of running transactions inconsistent with respect to the updated memory values, harming the opacity property [32]. As in NOrec [20], the solution to ensure this property is to check for consistency in each TMRead operation, if access to memory is required (lines 45-46).

Isolation. APUTM supports weak isolation, in the sense that transactions are isolated only from other transactions, but not from non-transactional memory accesses. Dalessandro *et al.* provide further discussion on weak isolation, which is observed to be strong enough to provide transactional data-race-free semantics [19].

Nesting. Nesting is currently not supported by APUTM. However, a *flattened nesting* approach [34] can be considered. Note that in the GPU side, transactions executing in lockstep may have different nesting levels and implementing wavefront-level optimizations for nesting may not be trivial.

4.2.5 Execution example

Figure 4.4 depicts an example of a program managing transfers between different bank accounts used in our evaluation.

```

1 //Bank account transfer example
2 transfer = getTransfer()
3 while(transfer != NULL)
4 {
5     TMBegin()
6     AccFrom = transfer.from
7     AccTo = transfer.to
8     Amount = transfer.amount
9     if(TMRead(accounts[AccFrom]) > Amount)
10    {
11        TMWrite(accounts[AccFrom],
12            TMRead(accounts[AccFrom]) - Amount)
13        TMWrite(accounts[AccTo],
14            TMRead(accounts[AccTo]) + Amount)
15    }
16    TMCommit()
17    transfer = getTransfer()
18 }

```

Figure 4.4: Example of a bank transfer program implemented using APUTM

In this example, a computing thread (either a CPU thread or a GPU work-item) performs a number of transfers between different bank accounts until no more transfers are available. We assume that a function called `getTransfer()` (lines 2 and 17) provides a new transfer to be performed by the thread, or `NULL` in case no transfer is available. As the transfer is thread-private, accesses to its information (origin, destination, and amount to transfer - lines 6 to 8) do not need to be instrumented. However, accesses to the `accounts` array, containing information on the amount of money stored in each account, need to be instrumented using `TMRead` and `TMWrite` operations. Information on the amount stored in the origin account is accessed using a `TMRead` operation and, if funds are enough to perform the transfer (line 9), then the transfer can be done. This is achieved by using `TMRead` to access the amounts stored in the origin and destination accounts (lines 12 and 14, respectively), and using the `TMWrite` operation to update the accounts with the updated amounts (lines 11 and 13). The `TMCommit` operation makes these speculative changes definitive in case no conflict is detected.

Table 4.1 shows a short example of the execution of the code in Figure 4.4 when executed in an APU processor. The example is simplified to 1 CPU thread that performs a transfer from account 14 to account 15, and two GPU transactions executing within the same wavefront (thus, executing in lockstep) that perform transfers from accounts 17 to 14 and from accounts 40 to 19, respec-

gclock	CPU Thread acc14 to acc25	GPU WI-0 acc17 to acc14	GPU WI-1 acc40 to acc19	Comments
7	snapshot = 7 reads = {14} writes = {14} line = 13	snapshot = 7 reads = {17,14} writes = {17,14} line = 16	snapshot = 7 reads = {40,19} writes = {40,19} line = 16	CPU TX registered the speculative read and write of account 14. GPU TXs registered their accesses to their accounts and are able to commit.
9	snapshot = 7 reads = {14,25} writes = {14,25} line = 16	-	-	GPU TX successfully commit and update gclock. CPU TX aborts as 1) gclock != snapshot and 2) acc14 has been modified by the GPU.
9	snapshot = 9 reads = {} writes = {} line = 5	-	-	CPU TX restarts with an updated snapshot.
9	snapshot = 9 reads = {14,25} writes = {14,25} line = 16	-	-	CPU TX successfully commits without checking its reads as gclock == snapshot

Table 4.1: Example of an execution of the program in Figure 4.4 by a CPU transaction and 2 GPU transactions executing in lockstep. Both GPU transactions are able to commit, while the CPU transaction needs to restart once, as it conflicts with a GPU transaction. TX stands for transaction, accX stands for account X, and line is the next line in Figure 4.4 to be executed.

tively. Note that there is no conflict between both GPU transactions, but the CPU transaction conflicts with one of the GPU transactions.

In the first row of Table 4.1 the CPU thread is still executing the transaction (line 13 of the code in Figure 4.4) while the GPU transactions have reached the TMCommit operation (line 16). Note that all the snapshots of all the transactions are still consistent with gclock. In this scenario, the GPU transactions are able to commit in parallel, updating the value of gclock. In the second row, the GPU transactions have committed, and the CPU transaction reaches the TMCommit operation. Now the CPU transaction needs to check the entire read set as its snapshot is not consistent with the value of gclock. As the account 14 has been modified by one of the GPU transaction (we assume that the value in memory is different from the one registered in the read set of the CPU transaction), this transaction aborts. In the third row we observe how the CPU transaction restarts execution. By executing the TMBegin operation (line 5), the transaction reads the current value of gclock into its snapshot and clears its read set and write set.

In the last row we observe that the CPU transaction is able to commit as there are no conflicts with other transactions. Note that, as the snapshot is consistent with gclock, the transaction can commit immediately without checking its read set, speeding up conflict detection.

4.2.6 Read-Modify-Write transactions

Many applications require mutual exclusion to manage the access to a single shared data object (for instance, modifying a single entry of an in-memory database), which is read, modified, and written without interaction with other objects. Normally, these applications are said to have *transactional read-modify-write* accesses. These type of transactions are common and optimizations have been proposed both in CPUs [52] and GPUs [36]. In this work we include two optimizations influenced by the work of Holey *et al.* [36]. Firstly, if two different transactions read the same memory position, a conflict can be signaled as that memory position is expected to be written in the future. This way, such conflict can be detected earlier in the commit phase. Secondly, storage of speculative reads and writes (i.e., the read-set and the write-set) can be reduced. As every position being read is expected to be written, the read-set and the write-set can be unified in a single access log. We call this log, and the APUTM implementation making use of it, *unified log*.

Figure 4.5 shows the more relevant changes required in APUTM to implement the unified log version. In this case, a single unified log is used to store the address, value read from memory, speculative value to be written to memory, and some flags indicating if the position is read or written (lines 5-6). Now, in the GPU implementation, wavefront conflicts can be detected if two transactions access the same position (lines 17-28) regardless of the type of access (read or write). Note that now the GPU transactions only examine the unified log, which is expected to be shorter than the read-set and the write-set separately. In this case, consistency checking has to be done for memory position that are marked as read (line 36). In the same manner, the commit operation only writes to memory the speculative writes (line 52).

The *rw* flag contains information on the access to memory (read, write, or read-and-write) that is used in these operations. In the first access to a given memory location, the flags are set to read or write, depending on the kind of access. A second access can be classified as RAR, WAR, RAW, or WAW (read-after-read, write-after-read, read-after-write, and write-after-write, respectively). RAR accesses are marked as read, but not written. This way, the memory lo-

```

28 checkConsistencyU(){
29   if(snapshot==gclock.atm_load())
30     return TRUE
31   do{
32     time=gclock.atm_load()
33     for(<addr,val_read,-,rw> in
34       unif_log){
35       if(rw == READ & mem[addr]!=
36         val_read)
37         return FALSE
38     }
39   }while(time!=gclock.atm_load())
40   snapshot=time
41   return TRUE
42 }
43 TMCommit()
44 {
45   if(GPU)
46     checkWFConflictsU()
47   acquire(global_lock)
48   consistent=checkConsistencyU()
49   if(consistent){
50     for(<addr,-,val_write,rw> in
51       unif_log)
52       if (rw == WRITE) mem[addr]=
53         val
54     atomic_add(gclock, 1)
55   }else{
56     abort
57   }
58   release(global_lock)
59 }

```

```

1 //Private metadata
2 struct pr_descr{
3   int snapshot;
4   int status;
5   <address, read_val,
6     write_val, rw> * unif_log;
7 };
8
9 //Wavefront metadata
10 struct wf_descr{
11   atomic_long * commit_mask;
12   atomic_int * leader_id;
13   <address,owner> * wf_accesses;
14   atomic_int * next_access;
15 };
16 checkWFConflictsU(){
17   for(<addr1,-,-,-> in unif_log{
18     for(<addr2,owner> in
19       wf_accesses{
20       if(addr1==addr2 &
21         ownr!=th_id &
22         commit_mask(owner)==0){
23         commit_mask(th_id)=1
24         abort
25       }
26     }
27 }

```

Figure 4.5: Unified log implementation of APUTM.

cation is checked for consistency, but not updated to memory at the end of the transaction. WAW and RAW accesses are set as written but not read, as consistency checking is not needed. Note that, in a RAW access, the value that is read corresponds to the speculative value, which does not reside in memory yet. WAR accesses are set as read-and-write as they need to be checked for consistency and update memory, if the transaction successfully commits. The value of these flags is set during the TMRead and TMWrite operations as described above. Furthermore, these operations are modified to register the accesses into the unified log instead of the separate read-set and write-set.

4.3 Evaluation

4.3.1 Experimental setup

To perform our experiments we utilize the AMD Kaveri APU 7850K as described in Section 2.2. We use the maximum number of threads supported by the system without oversubscription, which is 4 for CPU and 2048 (calculated as $4(\text{wavefronts}/\text{CU}) \times 64(\text{wavefront size}) \times 8(\text{CUs})$) for GPU. The system is equipped with 8 GB of DDR3 memory operating at 1.6 Ghz. To access the capabilities of the APU processor (i.e., unified memory space and platform atomics) we use C++AMP with the default options of the compiler integrated in ROCM 1.2¹. Results of the experiments are the average of 10 executions.

4.3.2 APUTM characterization

To characterize APUTM, we use a synthetic workload that allows us to model the number of accesses to memory per transaction and the probability of conflict among transactions. We test different conflict detection and version management implementations. The experiment called *Gclock-rw* refers to the implementation that uses *gclock* to speed up conflict detection and that maintains separate read and write sets. *Gclock-u* also utilizes *gclock* for the same purpose but, in this case, an unified log is used to register both reads and writes. Note that, in the GPU side, RAR accesses by different transactions within the same wavefronts are considered conflicts during the *checkWFConflicts* operation. These two implementations are also tested without using *gclock* to speed up conflict detection or checking for consistency, but directly performing a value-based comparison of the read-set (or the reads stored in the unified log) against memory. These variants are called *R-set-rw* (when the read-set is utilized) and *R-set-u* (when the reads in the unified log are utilized). In addition, we test alternative implementations of these four versions that do not consider opacity guarantees. The goal is to evaluate a lightweight version of APUTM in which correctness of the program is delegated to programmers or to a static analysis performed by the compiler. Note that in all the experiments, the results are correct as the tests are tolerant to non-opaque TM implementations.

To compare the efficiency of each implementation in each computing device (i.e., CPU and GPU), we analyze three scenarios that execute 100000 transactions. First, we evaluate two scenarios in which CPU and GPU (separately)

¹<https://github.com/RadeonOpenCompute/ROCM>

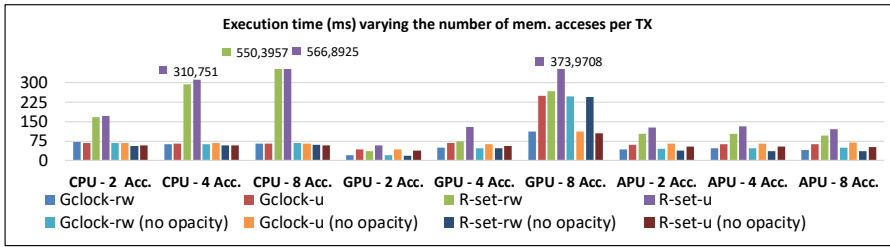


Figure 4.6: Characterization of APUTM when varying the number of accessed memory positions.

execute all the transactions. In the third scenario, called APU, both the CPU and GPU collaborate to execute the same amount of transactions. Specifically, we establish a static partition of 50000 transactions on the CPU and 50000 transactions on the GPU (note that such partition is not a requirement of APUTM but is useful to compare the efficiency of both devices).

Figure 4.6 shows the performance of the different APUTM implementations when executing transactions with different number of accesses to memory. In general, using gclock improves performance on both devices, as it avoids many validations of the read-set. However, in the CPU the differences are more noticeable as compared to the GPU. The reason is that a read-only CPU transaction does not modify gclock and, thus, the next transaction may commit without requiring of checking its read-set. In contrast, even if the number of read-only transactions running of the GPU is the same, is highly unlikely that all transactions within a wavefront are read-only. When committing 64 transactions concurrently, if at least one performs a write operation, gclock is updated. Then, the following transactions need to check their read-set against memory. Furthermore, the use of the unified log instead of separate write-set and read-set has some impact in the performance of the GPU, which is also translated to the APU implementation. Note that read-only transactions conflict in the unified log scenarios, harming performance. Lastly, we can observe the cost of implementing opacity. Normally, performance of opaque implementations is lower as the consistency checking introduces several overheads in the TM instrumentation. It is worth mentioning that for 8 memory accesses within a transaction the performance of the GPU decreases substantially. The reason is that the high number of concurrent transactions combined with the size of the transactional metadata (i.e., read-set, write-set and unified log) introduces high pressure in the memory subsystem harming performance.

In Figure 4.7 we observe the execution time when increasing the probability of conflict. This increase does not have an important impact in the performance for different reasons. Firstly, many conflicts are detected when the read-set is validated during read operations allowing transactions to restart quickly. Secondly, conflicting transactions do not modify gclock, allowing future transactions to commit rapidly. Lastly, in GPU transactions, even with lower values of the probability of conflict there is, at least, one conflicting transaction per wavefront. Thus, the performance of the whole wavefront is affected in a similar way in spite of the number of conflicting transactions.

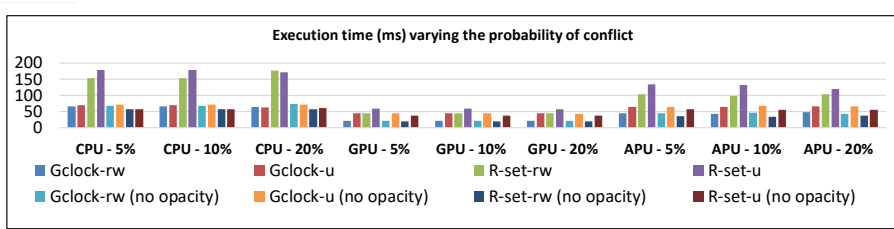


Figure 4.7: Characterization of APUTM for different probabilities of conflict.

Comparing the use of the unified log with the separate read-set and write-set, we observe that the latter implementation is more efficient as permits read-only transactions to commit with no conflicts (in case no other transaction intends to write any of the read locations). As in previous experiments, non-opaque experiments produce slightly improved performance.

We also compare the number of transactions aborted in CPU and GPU when executing APUTM across both devices (Figure 4.8.a). In this figure we represent the quotient of CPU aborted transactions over GPU aborted transactions for the base Gclock-rw and R-set-rw solutions. The difference is of 3 orders of magnitude less in the CPU. The reason is that the GPU executes more concurrent transactions than the CPU and, thus, the interaction among different transaction results in a higher probability of conflict. Note that the behavior is different depending if gclock is used to speed up conflict detection or not. In both experiments we observed that there are no noticeable differences in the number of aborted transactions in the GPU side, so most of the differences come from the CPU transactions. In the case of the CPU, the Gclock-rw solution speeds up the commit phase as compared to R-set-rw. This means that, in a given time frame, more transactions commit their values to memory causing other transactions to abort. This way, as the probability of conflict increases, aborted transactions in the CPU also increases. The opposite happens in the R-set-rw solution: as trans-

actions stall for a longer period in the commit phase, the number of concurrent CPU transactions is smaller and the amount of aborted transactions is reduced as compared to the GPU.

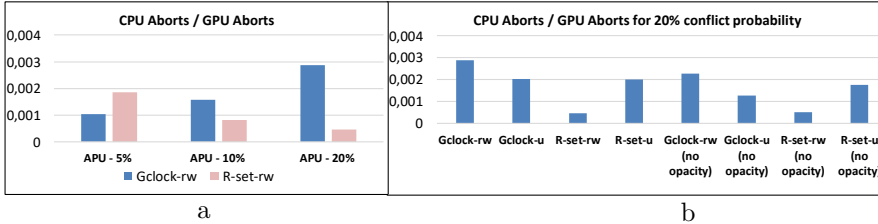


Figure 4.8: CPU and GPU abort analysis of APUTM

In Figure 4.8.b we observe the impact that the use of the unified log and the opacity checking have in the number of aborted transactions for the high contention scenario (20% of aborted transactions). Using the unified log (Gclock-u and R-set-u) has no significant impact in the number of aborted transactions per device. However, when using separate read-set and write-set we observe that more transactions abort in the CPU if we use gclock to speed up conflict detection. The reason is the same described above: more transactions commit in the CPU in the same amount of time when using gclock and, thus, the probability of conflict increase. When avoiding the opacity checks, the number of aborted transactions is reduced in the CPU compared to the GPU. The main reason is that this opacity checking creates more overhead in the CPU (note that this can be executed in parallel as long as gclock has not changed, and that the GPU can benefit from coalesced memory accesses when checking the unified log or the read-set). This extra overhead in the CPU implies that transactions commit at a lower rate than the GPU and, thus, the amount of conflicts is, porcentually, smaller.

In addition to the synthetic workload, we create a hash table as example of concurrent data structure that can be implemented using APUTM. The Hash application implements a hash table where entry is managed by an index that points to the next empty slot in such entry. Upon selecting the appropriate entry to insert a value, each thread must read the corresponding index (obtaining an empty position to insert its value) and increase its value (leaving it pointing to the next empty position). This process is enclosed within a transaction. We have designed two variants of the algorithm. The first one, called *Hash (isolated)*, works as described above. The second implementation, *Hash (computation)*,

intends to simulate an application that uses the hash table by performing some computation and inserting its result in the designated slot. We added a random amount of computation for each transaction, with an average of 1000 arithmetic operations per memory access. In both cases, we run one transaction per available computing resource (i.e., 4 CPU transactions and 2048 GPU transactions) which intends to insert a value in a random entry of a hash table with 50000 entries available.

To evaluate the hash table we use the basic versions that use `gclock` and the `read-set` for validation, both implementing `opacity` (see Figure 4.9). In this experiment, the use of `gclock` outperforms the implementation using only the `read-set` to check for conflicts. However, when executed in isolation (with no computation), APUTM does not deliver better execution time over than a sequential implementation. The reason is that the benchmark presents such a low computation time that cannot amortize the overhead of the TM library. However, as we include some computation in the equation, it tips the scales in favor of APUTM thanks to the parallelism offered by the APU processor. Additionally, commit ratio (i.e., $\text{committedTX}/(\text{committedTX} + \text{startedTX})$) stays around 70% for both APUTM implementations. It is worth mentioning that most of the conflicts come from GPU transactions, for two reasons: 1) they are more numerous than CPU transactions and 2) lockstep execution of wavefront transactions increase the probability of conflict as they reach the commit stage simultaneously.

4.3.3 Application evaluation

To assess APUTM we designed two applications: Bank and Genetic. Figure 4.10 and Figure 4.11 show the evaluation conducted using both benchmarks.

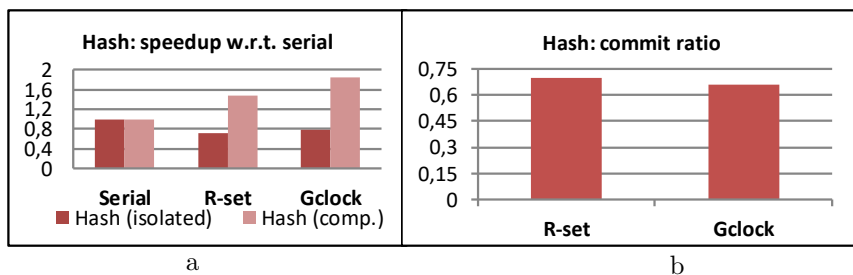


Figure 4.9: Evaluation of a hash table implemented using APUTM.

The first application is based in the Bank kernel provided with TinySTM [26, 25], where a number of money transfers are performed on a set of bank accounts. Each transfer withdraws money from one account and deposits the same amount in another account, provided that the original account has enough funds. In order to avoid inconsistencies, this process is enclosed inside a transaction. For our experiments, we consider 10000 transfers over a set of a million accounts, and the origin and destination accounts are chosen randomly. In this case, we do not set a static partition of the amount of transfers to be executed on each device. CPU and GPU dynamically acquire and execute transactions until the total number of transfers is 10000.

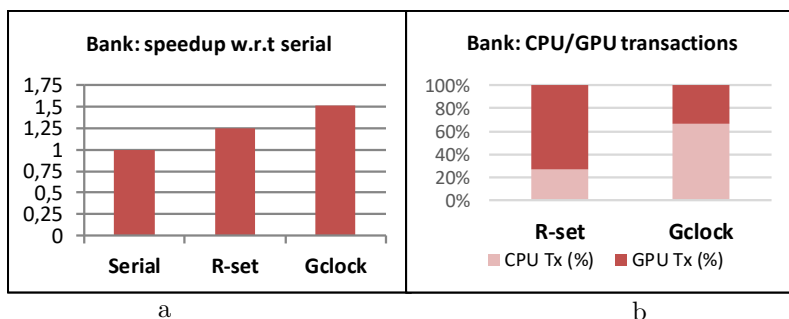


Figure 4.10: Evaluation of Bank implemented using APUTM.

Both APUTM implementations (i.e., using a gclock to accelerate conflict detection, or performing a value-based validation) outperform serial execution and, as expected, the use of gclock improves performance. The performance provided by the use of APUTM is not as good as expected compared to serial execution. However, our goal is to propose a simple design to evaluate our design decisions. Future optimizations can improve performance of APUTM in different scenarios. Concerning the percentage of CPU transactions, we can observe that about 27% of the transactions run on the CPU (and, thus, 73% of them run on the GPU) in the implementation not using gclock. The use of this lock gives us a different scenario, where 64% of the transactions execute on the CPU and 36% execute on the GPU. The reason of such difference is related to the benefits of using gclock in CPU, which permits individual transactions to commit quickly. This way, the CPU can execute more transactions (as compared to the GPU) if

gclock is employed. Note that the aforementioned workload partition is not statically performed by the programmer or APUTM: transactions are dynamically scheduled on the GPU and the CPU when they are idle and there exist pending transactions to be executed.

The application Genetic implements a genetic algorithm that solves the knapsack problem using genetic operations. We are assuming that we have a set of objects of different weights, and the goal is to fill a bag with objects maximizing the total weight of the bag up to an upper bound (capacity). Initially, the algorithm creates a pool of random solutions, which are represented using a binary string. The genetic algorithm works iteratively. In each iteration two random solutions are picked from the pool. Both are evaluated, and the fittest (i.e., closest to the maximum capacity) is kept, while the other suffers a mutation in some of its bits. These new solutions replace the old ones. We execute 10000 iterations of this algorithm on a set of 10 million solutions. Given the low probability of conflict, TM is a good candidate to implement these iterations in parallel. In our parallel implementation, CPU and GPU threads enclose this process in transactions, which are used to ensure that two threads do not modify the same items simultaneously.

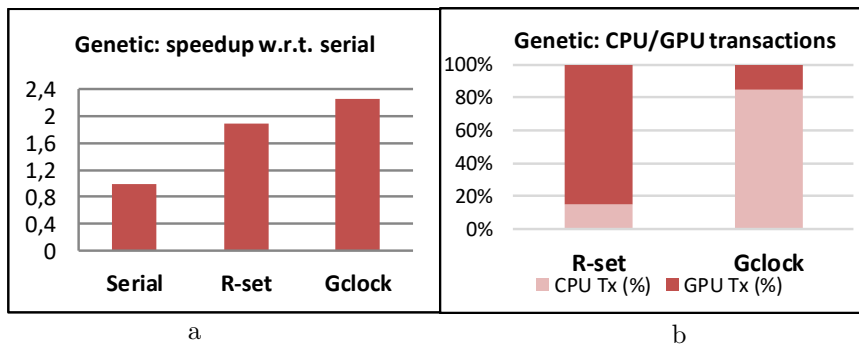


Figure 4.11: Evaluation of Genetic implemented using APUTM.

Figure 4.11 shows the evaluation of the genetic algorithm. The speedup obtained is in the same order of magnitude as in the Bank application. Performance is slightly better as the probability of conflict is lower, but also due to the higher computing demands of this algorithm that hide the overhead of the TM library more effectively as compared to the Bank application. The amount of transactions (iterations of the genetic algorithm) executed by each device be-

has the same as in Bank for the same reason: gclock permits CPU transactions to commit faster and, thus, more transactions can be scheduled on such device.

4.4 APUTM: conclusions and future work

In this chapter we present APUTM, a software TM designed specifically for APU architectures. Multiple implementations are presented: one using a global sequence lock to commit transactions quickly, another checking for conflicts using a transaction-private read-set, and two variants of these two using a unified log. The main goal of these APUTM implementations is to understand the behavior of transactions on APU processors. Besides that, APUTM is able to outperform sequential execution in some scenarios.

The main contributions of APUTM are the following:

- It offers a common interface based on TM to implement mutual exclusion in algorithms deployed on both the GPU and CPU of an heterogeneous processor.
- It presents different implementation of the TM operations adjusted to the architectural differences of the CPU and the GPU.
- It provides fine-grained synchronization (i.e., with no kernel termination requirements) using APU-specific platform atomics.
- It focuses on reducing costly cross-platform communication by minimizing the use of such platform atomics. Additionally, in the case of the GPU side, conflicts are detected before accessing main memory for further validation.

We plan to improve APUTM in several ways: coalescing GPU memory accesses for a better use of memory bandwidth, creating a scheduler for a better distribution of transactions across devices, and increasing the granularity of GPU commits from wavefront to work-group to improve performance. In addition, we plan to extend the evaluation of APUTM. We are currently implementing more applications to assess APUTM, as well as studying the overhead of the library with respect to the total execution time and other TM-related metrics.

Discussion: architectural support for transactions on the GPU. In the CPU world, software TM solutions offer more flexibility but lack the performance achieved by hardware TM. As hardware TM is commercially available, research is focusing on providing hybrid solutions that combine the best of both

worlds. For instance, a recent work by Diegues *et al.* [22] compares the performance of several software and hardware TM systems, and considers hybrid solutions. However, the scope of this work is restricted to multi-core CPUs.

In the same way, future TM proposals for GPUs and heterogeneous systems will try to have the advantages of both software and hardware approaches by combining them into a hybrid solution. However, more research is needed before hardware TM for GPU and heterogeneous architectures become a commercial product. Thus, given the current state of the art, it is important to study different hardware TM approaches for GPUs, which in the future can lead to more powerful hybrid solutions and that can be integrated in APU architectures. By observing the code in Listings 4.3 and 4.2, we can identify some functions that can be accelerated by hardware. For instance, checking a conflict between transactions within a wavefront is a task that can be accelerated if implemented in hardware. In addition, with the SIMT execution and transactional metadata managed by hardware, programmability and performance improvements can be expected. In Chapter 5 we present a hardware TM solution for GPUs focusing on the use of scratchpad memory and simplifying the programming model by providing correctness and forward progress guarantees. As we discuss later, some of its characteristics can be used to implement a hybrid TM for global memory and (potentially) a solution including the CPU cores.



UNIVERSIDAD
DE MÁLAGA

5 Transactional Memory in GPU Local Memory

Traditionally, synchronization in GPUs is done by using barriers (which synchronize work-items within a work-group, but not work-items in different work-groups), memory fences and, more recently, atomic operations in both global memory and local memory. Having these basic synchronization primitives to support mutual exclusion usually results in inefficient use of resources, poor performance and high programming complexity. These techniques are normally used to obtain efficient *ad-hoc* implementations [45] of non-trivial parallel algorithms. Furthermore, recent research has proposed new lock-based synchronization mechanisms to improve resource utilization and increase programming flexibility [69, 71, 40]. Given the benefits of TM in the multi-core CPU world, it has been integrated in GPU in software [12, 70, 36, 55] and hardware [29, 28, 17]. In contrast to other synchronization techniques (e.g., atomics, barriers, locks), prior work on TM has only targeted global memory. Therefore, applications that require synchronization at the local memory level have to be developed using synchronization techniques that come with higher programming costs.

This chapter presents *GPU-LocalTM*, lightweight hardware TM support for work-items accessing data at the local memory level. One of the main goals is to offer TM as an alternative to existing methods (e.g., locks) to the programmer. Then the programmer can decide which method to use, depending on the needs of the application. In this sense, GPU-LocalTM is designed in a way that minimizes overhead in case TM is not used. To achieve this goal, GPU-LocalTM reuses existing storage resources for version management and includes a lightweight conflict detection mechanism based on tables and signatures [10, 13]. Alternative conflict detection mechanisms can be enabled, based on the resources available

and the performance requirements. Additionally, previous hardware implementations [29, 28, 17] follow a lazy-lazy scheme (i.e., performing conflict detection and updates to memory at commit time). Our proposal is, however, eager-eager (i.e., performs conflict detection and updates to memory at access time). One of the advantages of eager-eager implementations is that there only exist a speculative value for each memory location, as backups are performed after conflict detection, reducing the storage requirements. In lazy-lazy approaches, speculative values are stored until the end of the transaction, resulting in a larger memory overhead as compared to eager-eager implementations.

5.1 Motivating example

Provided that prior work focused on global memory instead of local memory when implementing TM solutions, in this section we discuss the benefits of the use of local memory and how TM can improve programmability and performance when designing applications that use such memory space.

Most OpenCL and CUDA programming reference manuals recommend the use of local memory in GPU applications [47, 38] (in fact, about 52% of the sample applications in the AMD APP SDK for OpenCL and about 60% of the OpenCL Rodinia [14, 15] applications use local memory). In addition, open source libraries have been developed that support efficient movement of data between memory spaces [8]. Programmers are encouraged to use local memory in their kernels to improve application performance.

As an example, two applications, Hash Table (HT) and Genetic Algorithm (GA), were implemented on a GPU to be executed by a work-group consisting of 256 work-items (the details of the applications are discussed later in Section 5.4). The execution is performed via simulation on the Multi2Sim simulation framework [57]. Both applications access shared data allocated in local memory. Thus, a mutual exclusion mechanism is needed to ensure the correctness of the programs. We designed 3 versions mutual exclusion mechanism: a coarse-grained locks (CGL) implementation, a fine-grained locks (FGL) implementation, and, lastly, a TM implementation based in our proposal. Figure 1.1 (see Section 1.2) shows the use of a coarse-grained lock in the SIMT programming model, and the equivalent code using TM (as in our proposal). Fine-grained locking is usually much harder to use and the specific implementation depends on each kind of problem. Thus, programmers have to develop a different solution for each problem, as a generic template for fine-grained locks can not be provided.

Firstly, we analyze the programming effort and performance of the FGL solutions compared to CGL. Our simulation results show that the HT application is 90 times faster when using FGL in comparison with CGL. This FGL implementation relies on the use of atomics, and is highly efficient due to the extra programming effort devoted to define and manage locks properly. The FGL version of the GA application, in contrast, is slower than the CGL version (0.7X). Lock management becomes much more complicated than in HT, requiring a huge programming effort. Specifically, each work-item in the FGL version of GA tries to acquire a set of locks to ensure mutual exclusion. Lock acquisition must be serialized in order to avoid deadlocks and livelocks. In fact, 17% of the code deals with lock management operations. Given the extra lock management, the FGL version of GA is an inefficient solution. This problem occurs in many other kernels, and as a result, FGL is not a common idiom in GPUs. To support fine-grain synchronization, programmers adopt other techniques that require detailed and error-prone programming [69, 45]. The simplicity of the TM interface helps to reduce overall programming complexity.

GPU-LocalTM is intended to be an alternative to locking techniques, with the aim of improving programmer productivity and efficiency. For instance, the TM versions of the above applications reduce the number of lines of code by 15% (HT) and 17% (GA), while our simulations results indicate that we can achieve up to a 40 times (HT) and a 2 times (GA) speedup versus CGL versions. It should be noted that GA combined with TM experience some speedup, while when using FGL it did not (this is discussed in Section 5.4). In order to measure the programming effort, we calculated the Halstead complexity [33] metric for GA. The programming effort of the FGL version results in 2.12 times the programming effort of the CGL implementation. The TM version has a programming effort of 0.81 as compared to the CGL version, which means programming is simplified since lock management code is avoided.

5.2 GPU-LocalTM Design

In this section we present GPU-LocalTM, our hardware TM design focused on transactions at local memory level. GPU-LocalTM extends the GPU ISA with two new instructions, *TX_Begin* and *TX_Commit*. They work at a wavefront granularity, since wavefronts are the schedulable unit of work on a compute unit. However, work-items can commit individually. While a wavefront is executing a transaction (all work-items are running transactionally), other wavefronts can execute different work-items transactionally or non-transactionally. The sys-

tem is implicitly transactional, as all local memory accesses performed within transaction boundaries are considered to be transactional. Due to the lack of a cache coherence protocol at the local memory level, GPU-LocalTM provides *weak isolation*, as transactions are serializable only against other transactions (non-transactional local memory accesses are silent). Barriers are not allowed in a transaction, as transactional execution causes wavefront divergence (thus, not all the work-items may reach the barrier). Finally, a *flattened nesting* [34] approach for nested transactions is considered.

Briefly, GPU-LocalTM operates as follows: Once a wavefront executes the TX.Begin instruction, the succeeding memory instructions are considered transactional. Prior to issuing a local memory operation by a work-item, a check for a possible conflict with a previous memory operation within the work-group is carried out. If two work-items request access to the same memory location, the later arriving work-item identifies a conflict (i.e., requester loses). When the conflict is detected, the conflicting work-item remains disabled until the wavefront reaches the end of the transaction (TX.Commit). At commit time, those work-items within the wavefront that were not able to complete their memory accesses due to conflicts, the transaction rolls back and re-starts from the TX.Begin instruction. Otherwise, if all the work-items were able to complete, the wavefront continues execution from the next instruction after the TX.Commit.

GPU-LocalTM is designed on top of the baseline GPU architecture as explained in Section 2.1.2. Figure 5.1 describes such architecture. The implementation of GPU-LocalTM requires minor logic modifications in the wavefront scheduler, SIMD and LDS units, as well as space reserved in the vector and scalar register files and in local memory.

The wavefront scheduler is modified to include simple logic to manage the transactional execution of wavefronts. Subsection 5.2.1 describes how this is accomplished. Version management is eager and it is implemented in the LDS unit. New transactional values are stored in place, while old values are moved to an area called *shadow memory*. This region is allocated in local memory, and tracks ownership information required for conflict detection. Subsection 5.2.3 describes version management in detail. Conflict detection is also eager and implemented in the SIMD units, in coordination with the wavefront scheduler (in charge of registering transactional conflicts). Our eager conflict detection implementation relies on the LDS unit to obtain ownership information about transactional local memory accesses. In addition, to speed up conflict detection, signatures may be added to the system. Subsection 5.2.4 describes in detail the conflict detection mechanism.

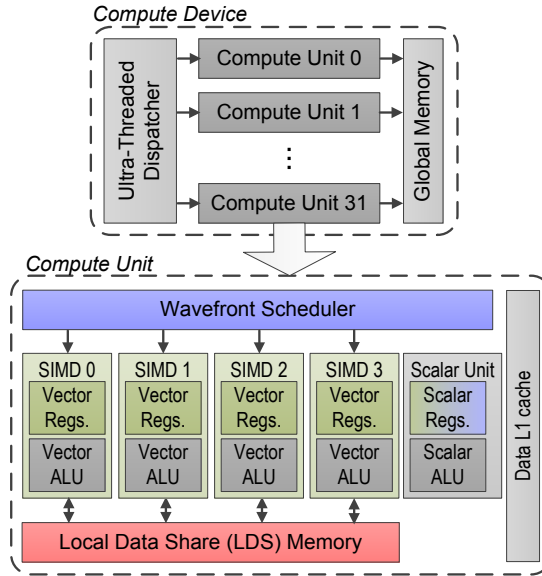


Figure 5.1: Baseline GPU architecture: AMD’s Southern Islands.

5.2.1 Transactional SIMT Execution

As explained in Section 2.1.2.2, the SIMT execution model in our baseline architecture is implemented using predication and execution masks. In particular, a per-wavefront 64-bit execution mask called EXEC contains one bit per work-item. If the bit associated to a given work-item is set to 1, it means that such work-item is active and executing instructions. Otherwise, if value of the bit is 0, it means that the work-item remains quiesced.

Whenever a wavefront is executing a transaction, some of its work-items may find conflicts in their memory accesses. Those work-items that detected a conflict must be disabled until the end of the transaction (commit time), and then they are rolled back and restarted. A simple method to mark transactional conflicts is using the EXEC mask to halt the execution of the work-items that detected conflicts. However, allowing the conflict manager to modify EXEC can lead to inconsistent situations as it is explicitly handled by the compiler [4] (i.e., it can be modified by an ISA instruction). For instance, EXEC is used to implement loops and conditionals, enabling and disabling work-items that fulfill a condition (represented in VCC). If a transaction conflicts in a conditional statement, and the EXEC mask is modified by the GPU-LocalTM in order to disable the work-

item, it can be re-enabled again by an ISA instruction when the wavefront finishes the conditional statement (and not when the transaction is re-started). Allowing GPU-LocalTM to implicitly modify EXEC requires reengineering the compiler to allow for conditionals and loops inside transactions.

To deal with this situation, we add a new 64-bit *transaction conflict mask* (TCM) per wavefront (one bit per work-item) entirely managed by hardware. TCM is used to mark conflicting work-items running transactionally: when a work-item detects a conflict it (eagerly) sets its bit in the TCM to 1. Just as with the EXEC and VCC masks, the TCM is mapped to two consecutive scalar registers.

With this new mask, transactional SIMT execution proceeds as follows: a work-item (in a wavefront) is enabled if its bit in EXEC is set (1) and its bit in TCM is reset (0). Otherwise, the work-item is disabled. Note that TCM will not be in use when executing non-transactional code. When the work-items within a wavefront execute TX_Begin (see Figure 1.1), the TCM is reset (all bits to 0). If a conflict is detected by a work-item, its bit in the TCM is set to 1 (the work-item is disabled, i.e., the requester loses). At commit time (TX_Commit), all work-items with their bits in TCM set to 0 will commit (i.e., those that did not detect conflicts). If TCM contains at least one bit set to 1, the wavefront re-starts the transaction for those work-items that detected conflicts. This is accomplished by copying the TCM to EXEC and returning to TX_Begin (where the TCM is cleared). Finally, when all work-items in the wavefront commit successfully, the transaction ends, and execution continues after the TX_Commit.

As a work-group may have various wavefronts, each one has its own set of TCM, EXEC and VCC masks operating independently (e.g., some wavefronts execute transactionally, while others not). Since the TCM is handled by hardware, no modification is needed in the compiler to manage the TCM or to include loops/conditionals inside transactions.

Table 5.1 shows a simple example of transactional execution using a TCM to mark conflicting work-items. In this example, during the execution of the transaction, work-item 1 detects a conflict and it is selected to be aborted and re-started. In order to do that, this work-item is marked by setting its bit in the TCM. By performing this action, the work-item is automatically disabled until the end of the transaction. At commit time, the rest of work-items successfully complete the transaction (making visible their updates to local memory) and wait while work-item 1 is re-started. This second time, work-item 1 also commits successfully.

Instruction	EXEC	TCM	Mode	Comments
...	111...1	-	n-TX	Non-transactional execution
TX_Begin	111...1	000...0	TX	Wavefront start transaction
Mem Access	111...1	010...0	TX	Conflict detected by WI 1
...	111...1	010...0	TX	From here, WI 1 is disabled
TX_Commit	111...1	010...0	TX	Transaction end
				WIs 0, 2, 3 ... commit successfully
				Wavefront rollbacks and restarts
				WI 1 (TCM is copied to EXEC)
TX_Begin	010...0	000...0	TX	Only WI 1 retries transaction
...	010...0	000...0	TX	No new conflicts detected
TX_Commit	010...0	000...0	TX	Transaction ends
				WI 1 commits successfully
...	111...1	-	n-TX	Non-transactional execution

Table 5.1: Example of transactional SIMT execution. A work-item (WI) is enabled if “EXEC[WI] & not(TCM[WI])”. Single lines separate transaction executions. TX means transactional execution and n-TX means non-transactional execution.

5.2.2 Forward Progress

The transactional SIMT execution model described above does not guarantee forward progress. The reason is that some of the work-items may conflict an unlimited number of times, causing the transaction to restart indefinitely for these work-items (livelock). This situation can be detected at the end of the transaction if the TCM remains the same after two consecutive transaction re-executions. Traditionally, hardware TM implementations for CPUs do not deal with this situation, but provide tools for programmers to solve the problem. For instance, Intel’s TSX does not guarantee that a given transaction, even when running in isolation, will eventually finish. However, TSX allows programmers to detect if a transaction is started for the first time or if it has been restarted. Programmers can use this feature to implement an alternative *fallback* code that guarantees forward progress when the hardware TM is not able to do so. Typically, when this situation is detected, programmers explicitly abort all the running transactions and resort on a coarse-grained lock to guarantee mutual exclusion and forward progress. Thus, two versions of the algorithm (or the mutual exclusion library) have to be provided by programmers: one using TM and another one using a different implementation.

In the case of a GPU, this burden is even more noticeable as mutual exclusion is hard to implement in the SIMT programming model. In addition, one of our

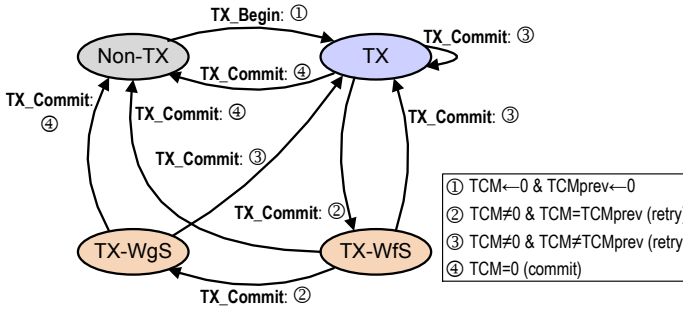


Figure 5.2: State diagram showing transactional and serialization execution modes. TX means transaction or transactional. TCMprev represents the contents of the TCM at the end of the previous transaction retry.

goals is to offer TM as a simple interface to implement mutual exclusion. In order to avoid having the programmer to provide fallback code, GPU-LocalTM includes a two-level automatic serialization mechanism that provides forward progress guarantees: *wavefront serialization (WfS)* and *work-group serialization (WgS)*. These modes are automatically activated by hardware and without requiring the programmer intervention. Figure 5.2 provides a state diagram showing the events that trigger transitions between execution states.

The WfS mode is enabled when a livelock situation is first detected (two consecutive transaction retries end with the same TCM with some bit/s set to 1). Once in this mode, the transaction is retried a third time but, instead of clearing the TCM at the beginning of transaction’s execution, only one of the active bits is reset. In particular, the bit of the conflicting work-item with the lowest identifier is selected to be set to 0. This action results in the execution of only the selected work-item within the complete wavefront during the third transaction retry. If the execution ends with no new conflicts, apart from already detected in the previous retry (this is determined by testing if the bit of the selected work-item remains 0), the transaction is again retried, but this time in normal mode (i.e., transactionally and not using WfS; (TX in Figure 5.2). Thus, the WfS mode serializes the execution of those work-items that hinder forward progress.

However, even executing only one work-item per wavefront, a new conflict can be observed with a work-item from a different wavefront in the same work-group, leading to the same livelock situation. This occurs when the bit of the selected work-item remains set to 1 at the end of the WfS execution. In such

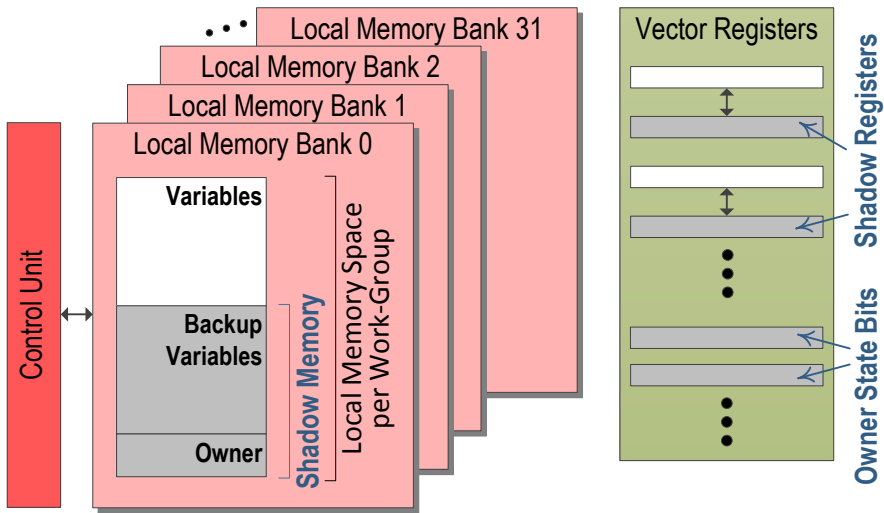


Figure 5.3: Version management (LDS unit, SIMD units) and register checkpointing (SIMD units) in GPU-LocalTM.

situations, the WfS mode transitions to the WgS mode. In this mode, only the current wavefront re-executes transactionally. Other transactional wavefronts in the work-group are aborted and stalled at the *TX_Begin* instruction until the selected work-item ends execution. Now there is no possibility of a conflict, so forward progress is assured. After this execution, the transaction is retried again in normal mode and the stalled wavefronts are allowed to continue execution. Table 5.2 shows a simple example of a transaction that requires entering the WfS mode to assure committing all work-items.

Note that, in order to ensure forward progress, the conflict detection mechanism must guarantee that executing a single work-item within the work-group does not abort due to false conflicts or capacity issues. The different conflict detection mechanisms described in Section 5.2.4 fulfil this property.

5.2.3 Version Management

GPU-LocalTM uses eager version management, where local memory updates are stored in place, while old values are saved in shadow memory. These values are used to restore the original state of the local memory in the case of a transaction abort. As the local memory is multi-banked (32 banks in our baseline GPU

		WI 0	WI 1	WI 2	WI 3	
		Read A	Read B	Read C	Read D	
		Write B	Write A	Write D	Write C	
Instruction	EXEC	TCM	TCM	prev	Mode	Comments
TX_Begin	1111	0000	-	-	TX	Transaction starts
...	1111	1111	-	-	TX	Normal transactional execution
TX_Commit	1111	1111	-	-	TX	Conflict detected by all WIs Transaction ends All WIs must re-execute
TX_Begin	1111	0000	1111	1111	TX	Transaction retries for all WIs
...	1111	1111	1111	1111	TX	Again, all WIs detect conflicts
TX_Commit	1111	1111	1111	1111	TX	Transaction ends No progress detected (livelock)
TX_Begin	1111	0111	1111	1111	WfS	Transaction retries for WI 0
...	1111	0111	1111	1111	WfS	No new conflicts detected
TX_Commit	1111	0111	1111	1111	WfS	Transaction ends WI 0 successfully commits
TX_Begin	0111	0000	0111	0111	TX	Transact. retries for WIs 1, 2, 3
...	0111	0011	0111	0111	TX	WIs 2 and 3 detect conflicts
TX_Commit	0111	0011	0111	0111	TX	Transaction ends WI 1 successfully commit WIs 2 and 3 must re-execute
TX_Begin	0011	0000	0011	0011	TX	Transaction retries for WIs 2, 3
...	0011	0011	0011	0011	TX	WIs 2 and 3 detect conflicts
TX_Commit	0011	0011	0011	0011	TX	Transaction ends No progress detected (livelock)
TX_Begin	0011	0001	0011	0011	WfS	Transaction retries for WI 2
...	0011	0001	0011	0011	WfS	No new conflicts detected
TX_Commit	0011	0001	0011	0011	WfS	Transaction ends WI 2 successfully commits
TX_Begin	0001	0000	0001	0001	TX	Transaction retries for WI 3
...	0001	0000	0001	0001	TX	No conflicts detected by WI 3
TX_Commit	0001	0000	0001	0001	TX	Transaction ends WI 3 successfully commits

Table 5.2: Example of a transactional execution requiring the WfS mode to assure forward progress, for the code shown above for four work-items within a wavefront. Single lines separate transactional retries.

architecture), version management is bank-aware (i.e., old values are logged in the same bank as stored originally, see Figure 5.3). Thus, there are 32 shadow memory areas. This design provides scalable performance, as GPUs may increase LDS storage capacity by adding extra banks. Besides, both version management and conflict detection can be carried out concurrently in different banks.

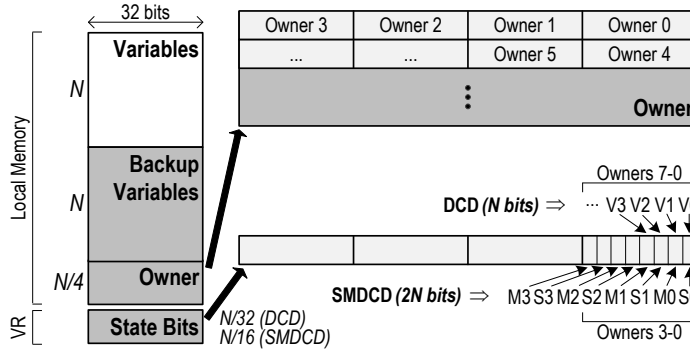


Figure 5.4: Shadow memory organization for version management (VR refers to vector registers).

Allocating shadow memory areas in local memory saves specific storage resources for version management, but at the cost of reducing the amount of local memory space available for the application. Each shadow memory area has enough room to store backups for all of the local memory variables declared within the kernel allocated in each bank. These variables are statically allocated by the compiler in consecutive addresses [4, 57]. The compiler groups all the variables declared with the `_local` modifier in a single memory chunk. In GPU-LocalTM, shadow memory is also allocated by the compiler, reserving space in local memory to backup all the local variables and to store the identifier of the accessing work-items (owners). The advantage of this implementation is that it involves simple management.

The main drawback is that it reserves space for every variable declared in local memory. The consequences of this overhead are: 1) programmers have access to a smaller amount of local memory, and 2) the maximum number of work-groups that can run concurrently within the same CU may be reduced. These space and runtime constraints are discussed in Section 5.3. Note that, after analyzing the kernel code, a sufficiently smart compiler may allocate shadow memory space only for the local memory variables accessed within the transaction. That would reduce pressure on the LDS, allowing for improved concurrency. In this thesis we assume a worst-case scenario, where the maximum amount of local memory is required by the kernel. An optimizing compiler should be able to reduce this memory overhead – we plan to pursue this question in future work.

The organization of the shadow memory is as follows (see Figure 5.4): consider a set of N words (1 word = 32 bits) declared by the application in local memory,

a contiguous section of N words is allocated to backup the values, and after this section, $N/4$ additional words are reserved to store the owners. Each word in the ownership region stores 4 owner identifiers, each one in one byte (there are 256 work-items in a work-group). In addition, there are state bits (1 or 2 per owner) required for conflict detection (see Section 5.2.4). These bits are stored in vector registers. Given this layout, when a memory access is issued to a location k , a backup value is stored at word position $N + k$ (if the access is a write), and the work-item identifier (owner) is stored at word position $2N + k/4$, byte $k\%4$. By adopting this scheme, the hardware required to backup a memory value and store its owner is minimal, as only an integer add of an offset and simple bit manipulation must be performed. The use of an offset, on the other hand, avoids the overhead of saving the address of the accessed word (this is calculated using the offset). Besides, capacity conflicts are avoided, as each memory location is ensured to have space for its backup. In contrast, the amount of local memory available to the application is reduced and could prevent kernels with high local memory requirements from executing. Note, however, that for those kernels not using TM, the compiler does not allocate any space for shadow memory.

Register checkpointing. When a transaction starts, the user-visible non-memory work-item state must be saved (and restored on transaction abort). This includes vector and scalar registers. Vector registers are checkpointed to a *shadow register* file. This is implemented by splitting the vector register file in each SIMD unit into two equally-sized parts. Every two registers, one for each part, are paired together so as one of them acts as the backup (shadow) register of the other (see Figure 5.3). Logically, registers in a pair are connected and a signal is used to copy the contents of the first register to the second register when needed. Similar to the shadow memory, the shadow register file is pre-allocated only for those kernels that use TM. Otherwise, no backup registers are defined.

Scalar registers, on the other hand, are used to store scalar shared data for an entire wavefront, such as a loop index. In such cases, the use of scalar registers develops inconsistencies when a transaction aborts inside a loop. For instance, the loop index of a for-loop might be updated inside a transaction. As this index is shared by committed and aborted transactions running on different work-items within the same wavefront, the contents of this shared scalar register is inconsistent (i.e., the aborted transactions must roll back to the previous value, while the committed transactions must keep the new value). To solve this problem, the compiler must avoid the optimizations that promote the use of scalar registers (shared by an entire wavefront) and consider the use of vector registers (work-item private). By disabling this optimization, loops and conditionals are allowed

within a transaction. For this reason, scalar registers are not checkpointed when starting a transaction.

5.2.4 Conflict Detection

GPU-LocalTM performs eager conflict detection at a work-item level. During transactional execution of a wavefront, the LDS unit serializes all local memory accesses so that, at any time, a memory bank is accessed by at most a single work-item. Parallel accesses to different banks do not present conflicts, as the banks have different address ranges.

Given this multi-bank organization, conflict detection proceeds in two steps:

(1) *Intra-bank conflict detection*: conflicts are detected for memory accesses within a bank. The conflict detection mechanism works in parallel for all memory banks (there can be no inter-bank conflicts). This step is responsible for updating the TCM, assigning a 1 to the bits for those work-items encountering a conflict.

(2) *Inter-bank conflict communication*: once a conflict is detected in a memory bank, it is communicated to the rest of banks in order to remove the shadow memory entries allocated for the conflicting work-items. This is accomplished through the TCM, avoiding an expensive broadcast operation. TCM informs each memory bank which work-items detected conflicts (bits set to 1). For each one of these work-items, all the backups are restored and the associated owner and state bits information are cleared.

Intra-bank conflicts are detected using a directory composed of the ownership and state bits information, which is stored in the shadow area (see Figure 5.4). We have designed two types of directories, which differ in the number of state bits per owner. In addition, we improve upon these designs with the help of two types of signatures.

5.2.4.1 Directory-based Conflict Detection (DCD)

For each transactional local memory access, the DCD mechanism checks the ownership and the state bit (V) associated with the memory location. Depending on the results of this check, i three actions may follow (see Table 5.3 (a)):

(1) *First (new) access*: the shadow memory entry has no owner associated (valid bit V is 0). A copy of the current value of the memory location is backed up in the corresponding shadow memory entry, and its owner is set to the work-item that made the access (and setting V to 1).

(2) *Repeated access*: the owner of the shadow memory entry is the accessing work-item. If the access is a read, the value in memory is returned. If it is a write, the memory is updated (the value was already backed up in shadow memory in the first access).

(3) *Conflict*: the owner of the shadow memory entry is a different work-item than the work-item that issued the access. The TCM is updated to mark this conflict, setting the bit of the work-item accessing to memory (WI) to 1. In addition, the backup values of WI are restored and all ownership entries in the shadow memory for WI are deleted (the work-item is disabled and ready to be re-started).

Note that, since there is no information on the type of memory access (i.e., read or write) stored in shadow memory, the DCD cannot filter out read-read conflicts. We designed two alternatives to deal with this situation: (i) adding an extra state bit per owner (see Section 5.2.4.2), (ii) adding per-wavefront signatures (see Section 5.2.4.4).

DCD is a simple and precise approach for detecting conflicts, but at the cost of an additional local memory access (to the same bank) in order to check the ownership records. In a later design, this extra access is mostly avoided by using per-work-item signatures (see Section 5.2.4.3).

5.2.4.2 Shared-Modified DCD (SMDCD)

The DCD mechanism can be improved by having two state bits per owner (instead of V): the S bit, set to 1 if the location has been accessed by multiple work-items (being the first one the owner), and the M bit, set to 1 if the location has been written. These extra bits permit us to filter out read-read conflicts. The new mechanism is called *Shared-Modified Directory-based Conflict Detection* (SMDCD).

The case of both state bits set to 1 at the same time cannot occur because a conflict has been detected previously. We use this pattern to encode the "not set" owner state (see Table 5.3 (b)). When a transaction begins execution, both S and M are set to 1. For each transactional access to local memory, the SMDCD mechanism carries out the actions specified in Table 5.3 (b). If a memory location is being accessed for the first time, SMDCD saves the identity of the owner in shadow memory and performs a backup of the current memory value if the access is a write, enabling us to distinguish between reads and writes. A read access to a memory location owned by a different work-item is allowed, as long as M is 0. These accesses set S to 1. But if M is 1, a conflict is detected (read after write).

State		Mem. Operat.	Next State		Action
Owner	V		Owner	V	
Not set	0	Read	WI	1	back up value; read memory
		Write	WI	1	back up value; write memory
WI	1	Read	WI	1	read memory
		Write	WI	1	write memory
o-WI	1	Read	o-WI	1	conflict (R/W→R); abort
		Write	o-WI	1	conflict (R/W→W); abort

(a)

State			Mem. Operat.	Next State			Action
Owner	S	M		Owner	S	M	
Not set	1	1	Read	WI	0	0	read memory
			Write	WI	0	1	back up value; write memory
WI	0	0/1	Read	WI	0	0/1	read memory
			Write	WI	0	1	write memory
WI	1	0	Read	WI	1	0	read memory
			Write	WI	1	0	conflict (R→W); abort
o-WI	0/1	0	Read	o-WI	1	0	read memory
			Write	o-WI	0/1	0	conflict (R→W); abort
o-WI	0	1	Read	o-WI	0	1	conflict (W→R); abort
			Write	o-WI	0	1	conflict (W→W); abort

(b)

Table 5.3: Conflict detection using the DCD (a) and SMDCD (b) mechanisms. WI is the label for the accessing work-item, o-WI is the other work-item, "0/1" means 0 or 1. "Abort" means the following actions: restore backup for WI, delete WI ownership entries and set TCM[WI] bit to 1.

A write access is allowed only if the owner is the requesting work-item and the memory location was not accessed by other work-items (S is 0). These accesses set M to 1. Otherwise, a conflict is flagged (write after read or write after write).

5.2.4.3 DCD and Private Read/Write Signatures (pRWsig)

DCD/SMDCD mechanisms for detecting conflicts require an extra access to a local memory bank for checking ownership information. To speed up conflict detection, avoiding most of these extra accesses, a set of private (per work-item) unified signatures [10] ($pRWsig$) are defined. There is one signature per local memory bank and per work-item, which records both reads and writes (unified).

Since there are a large number of signatures, 8-bit Bloom filters are used to limit the amount of storage resources required. Only one hash function is defined (see Figure 5.5). For the workloads evaluated, we found that a simple

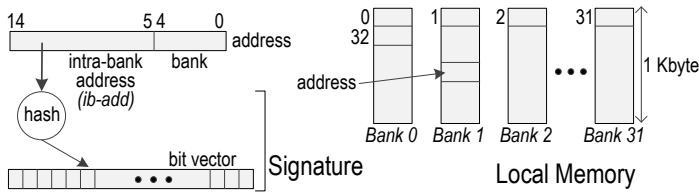


Figure 5.5: Signature design (local memory size is per work-group).

pRWsig Test		Mem. Operat.	Action
WI	o-WI		
0	0	Read/Write	back up value; insert(pRWsig); read/write memory
1	0	Read/Write	if owner is not set then back up value; read/write memory
0/1	1	Read	conflict (R/W→R); abort
		Write	conflict (R/W→W); abort

Table 5.4: Speeding up conflict detection by using private unified signatures (pRWsig). WI, o-WI, "0/1" and abort are as in Table 5.3. insert(pRWsig) inserts intra-bank address in the same-bank signature of WI.

($ib-add \bmod 8$) hash operation works well at the local memory level, where *ib-add* refers to the intra-bank position of the accessed memory reference. This hash function works well because GPU programmers commonly allocate variables in a coalesced way: in a given bank, a memory access by a work-item is usually followed by an access to the next memory location by the same or a different work-item. The above hash function minimizes the number of false conflicts when consecutive addresses are referenced.

To avoid defining new storage resources, each of these signatures is stored in a vector register so that it can be accessed efficiently for each transactional reference to a local memory location. This design reduces the number of vector registers available to the application by a small amount (see discussion in Section 5.3). However, if TM is not used, the compiler does not allocate space in the vector register file for signatures.

When a work-item issues a local memory access, the address is checked for hits in any of the signatures assigned to the current bank. Possible outcomes and consequences are described in Table 5.4. An intra-bank conflict is detected if a signature of a different work-item (o-WI) returns a positive. In this case, the work-item (WI) issuing the request signals a conflict, restoring its backed-up

values and clearing its entries in the shadow memory of the accessed bank.

Next, its bit in the TCM is set to 1, disabling the execution of WI until the end of the transaction. The positive signature test may be false, with a risk of increasing the abort rate. However, conflict detection is faster as the extra access to local memory for checking the owner is avoided. The access to pRWsig is done in parallel to minimize the checking overhead. In fact, as a work-group consists of 256 work-items, a set of 256 3-to-8 decoders are enough to check pRWsig fully in parallel. As there are 32 local memory banks, a total of 8K decoders are required to check, in parallel, all private unified signatures per CU. The cost of this hardware is negligible compared to the complete CU.

If no signature returns a positive, that means that the memory location is accessed for the first time. In this case, WI performs a backup of the value stored in this location, records its ownership in shadow memory and inserts the memory address in its signature. If, on the other hand, the signature for WI returns a positive, then we need to determine if the hit was true or false by examining the ownership information (as with DCD), since the action depends on whether the owner is WI (no action) or it is not (the backup value). This is the only case where an extra access to local memory cannot be avoided.

5.2.4.4 DCD/pRWsig and Shared Write-Only Signatures (sWOSig)

As signatures in pRWsig are unified read-read conflicts cannot be filtered out. To avoid transaction aborts due to these conflicts, a new set of per-wavefront write-only signatures (*sWOSig*) are added to GPU-LocalTM in order to register only the write set (for the entire wavefront). These signatures work in a similar way as Choi's helper signatures [18]. A conflict is detected during a memory read operation if some signatures in both sets, pRWsig and sWOSig, return a positive. A false read-read conflict would return a positive in the unified signature, but not in the write-only signature. This allows us to filter out read-read conflicts, providing a more precise conflict detection mechanism.

The use of sWOSig makes sense if the signature size is much smaller than those in pRWsig. Otherwise, the implementation of separate read and write signatures would be more efficient. As each signature in pRWsig is small (8 bits), we lower the storage requirements of sWOSig by drastically reducing the number of signatures in the set. A way is used to define each signature in sWOSig as shared by all work-items, belonging to the same wavefront (64 work-items). Then only 4 signatures in sWOSig exist (one per wavefront within the work-group) per memory bank, with a total of 128 signatures (32 memory banks). However, as

pRWsig Test		sWOSig Test		Mem. Operat.	Action
WI	o-WI	WF	o-WF		
0	0	0/1	0/1	Read	insert(pRWsig); read memory
0	0	0/1	0/1	Write	back up value; insert(pRWsig); insert(sWOSig); write memory
1	0	0/1	0/1	Read	read memory
1	0	0	0	Write	back up value; insert(sWOSig); write memory
1	0	1	0	Write	back up value; write memory
0	1	0	0	Read	insert(pRWsig); read memory
0	1	0	0	Write	conflict (R→W); abort
0	1	0	1	Read	conflict (W→R); abort
0	1	0	1	Write	conflict (W→W); abort
0	1	1	0	Read	conflict (W→R); abort
0	1	1	0	Write	conflict (W→W); abort
1	1	0	0	Read	read memory
1	1	0	0	Write	conflict (R→W); abort
1	1	0	1	Read	conflict (W→R); abort
1	1	0	1	Write	conflict (W→W); abort
1	1	1	0/1	Read	if owner is WI then read memory else conflict (W→R); abort
1	1	1	0/1	Write	if owner is WI then write memory else conflict (W→W); abort

Table 5.5: Combining private unified signatures (pRWsig) and shared write-only signatures (sWOSig) to speed up conflict detection and filter out read-read conflicts. WI (WF) is the accessing work-item (wavefront). o-WI (o-WF) is other work-item (wavefront). "0/1" and abort are as in Table 5.3. insert(pRWsig) is as in Table 5.4. insert(sWOSig) means insert accessed memory address in the sWOSig of the wavefront for WI.

many work-items share the same signature, its size is slightly larger (32 bits) than in pRWsig, in order to avoid fast saturation (many bits of the filter set to 1) [49]. In our design, we use a scalar register to store each one of the shared signatures.

Apart from the size, a key issue to avoid fast signature saturation is a suitable design of the hash function. Assuming that programmers write code with coalesced memory accesses, and similar to the design of pRWsig, a hash function ($ib-add \bmod 32$) is used to minimize false positives when accessing nearby memory addresses (shared signatures have 32 bits) (see Figure 5.5).

The use of pRWsig and sWOSig together for conflict detection is illustrated in Table 5.5, which is an extension of Table 5.4. Using sWOSig has two benefits.

First, it allows us to save unnecessary backups of old values when the memory access is a read. Second, it permits us to avoid transaction aborts due to read-read conflicts (e.g., rows 6 and 12 in Table 5.5). The last two rows in Table 5.5 represent a special case where two or more private signatures show a positive (WI and other/s) and the shared signature for the same wavefront also is a positive. We can simply signal a conflict, as some work-item in the same wavefront has written the same memory location before. A more precise response would be to check if the owner is WI, in which case there would be no conflict (that is, the positive in pRWsig for other work-items different from WI, o-WI, is false). This permits us to reduce transaction aborts, but at the cost of an extra access to local memory bank. We simulated both designs and found no significant differences in the results.

DCD-based conflict detection can be improved either by adding extra state bits (SMDCD) or by adding signatures (pRWsig, sWOSig). A directory is a precise solution because it does not suffer from false positives, as do signatures (especially if they are near saturation), but the latency for conflict detection is higher due to an extra access to local memory. In the end, the performance of the adopted solution depends on the local memory access pattern of the application.

5.3 GPU-LocalTM Modeling

GPU-LocalTM requires slight changes to the GPU microarchitecture. We have implemented these changes using the Multi2Sim 4.2 simulation framework [57]. This framework supports the Southern Islands family of AMD’s GPUs, used as the baseline microarchitecture for GPU-LocalTM. Table 5.6 provides the key features of this microarchitecture.

Transactional SIMT execution overhead. Two instructions, TX_Begin and TX_Commit, are added to the GPU ISA to use TM. They are modeled as scalar instructions, as they affect the entire wavefront. The transactional SIMT execution is managed by combining the EXEC mask, the TCM and the MCM. The time spent accessing these masks is modeled by adding an extra cycle per mask access to the latency of the TX_Begin and TX_Commit instructions.

Memory latency. Depending on the implementation, GPU-LocalTM accesses signatures and/or shadow memory for each memory operation in order to perform conflict detection and version management. These accesses to the signatures in pRWsig/sWOSig and to the shadow area add some overhead. As the hardware required to access signatures is simple (briefly, it can be implemented

Feature	Value	GPU-LocalTM
Compute Units (CU)	32	-
Vector Registers per CU	65536	2276 (2504) ($\sim 3.6\%$)
Scalar Registers per CU	2048	136 ($\sim 7\%$)
SIMD Units per CU	4	-
SIMD Lanes	16	-
LDS Size per CU	65536 bytes	37446 bytes ($\sim 57\%$)
LDS Banks	32	-
Minimum LDS Latency	2 cycles	5 cycles

Table 5.6: Relevant features of the AMD’s Southern Islands GPU implementation on Multi2Sim 4.2 and the amount of resources required by one work-group using GPU-LocalTM.

as a set of decoders), we consider that accesses to signatures can be performed in one cycle. This cycle is added to the latency of memory operations when executed inside a transaction. Accesses to shadow memory are modeled as regular accesses to local memory. Thus, when modeling memory accesses, extra latency is added depending on the number of accesses to shadow memory. As see in Table 5.6, the latency of a LDS access is 2 cycles. Any access to shadow memory results in 2 extra cycles to be added to the memory access latency. Note that coalesced memory accesses allow for parallel access to different memory banks. Thus, the latency of the memory access is calculated as the latency of the slowest of the 32 banks that can be accessed in parallel.

Storage overhead. Storage resources required in GPU-LocalTM are taken from those available in the baseline architecture, as described in Table 5.6. The vector register file is used to hold signatures from the pRWsig set, which reduces the number of available vector registers by 2048 (256 signatures per bank, and 32 banks per CU) per work-group. Signatures in the sWOSig set are stored in the scalar register file, using a total of 128 registers (4 shared signatures per bank, and 32 banks per CU). In addition, each wavefront requires its own TCM to manage the transactional execution, which is also mapped onto scalar registers. As each work-group is composed of 4 wavefronts, the 4 TCMs use 8 scalar registers (two registers per TCM). In the general case, 136 scalar registers must be reserved, which represents about 7% of the total of 2048 available scalar registers.

The amount of local memory available per work-group depends on the size of the shadow memory. If the user requests N words to store local variables, the shadow memory allocates another N words for backups and $N/4$ words for the ownership records (see Figure 5.4). The total physical amount of local memory is

64 KB, partitioned in 32 banks. With this size, a total of 29,127 bytes is available per work-group to store local variables, as the same amount and an additional 7,282 bytes are reserved for the shadow area. This represents an overhead of about 56% of the total space. However, the AMD architecture exposes half of the total physical space to each work-group, that is, 32 KB. The other half is reserved for the system to allow execution of an additional work-group at the same time. The shadow area could be allocated at runtime in this second half of the local memory. In such cases, a work-group would have 29,127 out of 32,768 bytes for user variables, but at the cost of disallowing the execution of two work-groups in parallel (only if they use more than half of the 32 KB available per work-group). Otherwise, if the complete shadow memory is stored in the same half of local memory, the storage available per work-group is reduced to 14,563 bytes. On the other hand, the state bits for owners are stored in vector registers (see Figure 5.4). There is 1 bit (DCD) or 2 bits (SMDCD) per owner. Given a total of 7,282 owners, 228 (DCD) or 456 (SMDCD) vector registers are required for storing state bits. Adding the 2,048 vector registers for signatures, 2,276 (DCD) or 2,504 (SMDCD) vector registers must be reserved, which represents about 3.5% (3.8%) of the total of 65,536 vector registers.

This implementation of GPU-LocalTM puts pressure on storage resources that can limit the types of kernels that can benefit from this TM support. Note, however, that one of the main aims in the design of GPU-LocalTM is to minimize the amount of extra hardware required, in particular, storage resources. These storage resources are allocated by the compiler (together with the runtime). Hence, kernels that do not require TM do not suffer from this storage overhead.

Programmability challenges. On traditional GPUs (i.e., with no TM support) the resources needed by a work-group on a given CU must be preallocated before starting execution. This preallocation limits the execution of work-groups whose requirements exceed the available resources. These limitations are: 1) the number of work-groups assigned to the same CU, 2) the number of vector and scalar registers needed, and 3) the amount of local memory allocated. Our implementation of GPU-LocalTM, which makes use of existing memory resources, places constraints on GPU execution that must be analyzed before TM is used. Programmers are only exposed to the changes in the amount of local memory available. Despite that the total amount of LDS per CU is 64Kb, programmers are only allowed to use 32Kb per work-group. Thus, the amount of local memory addressable by programmers when using GPU-LocalTM changes from 32,768 bytes to 28,084 bytes. Algorithms that plan to benefit from GPU-LocalTM should be adapted to assume this amount of local memory. The runtime and the com-

piler can play an important role hiding other restrictions from the programmer. Traditional compiler techniques, such as register renaming and spilling, can be implemented to relieve the pressure on the register file. Selecting the number of concurrent work-groups, when the amount of registers and local memory required are more than are available in hardware, is made by the runtime and is transparent to programmers. Lastly, the use of TM can be a decision made by the compiler. In case of a scarcity of resources, the compiler can choose the most convenient algorithm or apply a transformation similar to the one shown in the introduction of this thesis (Figure 1.1), and decide to use a coarse-grained lock implementation for the critical section.

5.4 Evaluation

Eight benchmarks were designed to evaluate GPU-LocalTM in specific scenarios, defined for two contention levels (see Table 5.7). All the benchmarks are implemented in three different versions: a version serializing the critical sections, a TM version where the critical sections are executed as transactions, an atomics version (only for HT, IT, DB and QU), and a FGL version (only for GC, GA, VA and KM). The third column in Table 5.7 describes the definition of the critical sections (CS) for each benchmark, and the last column summarizes the atomics and FGL versions. Note that the FGL implementations require lock management, adding 17%, 10%, 42% and 22% extra lines of code to GA, KM, GC, and VA, respectively.

These workloads fully exercise the different features present in GPU-LocalTM (see Table 5.8 for more details on the features of each benchmark). HT and IT feature read-modify-write transactions, focusing on short transactions. While transactions in IT perform the same number of reads and writes, HT is designed to have many read-only transactions. Transactions in DB also perform the same number of reads and writes, as in IT, but in multiple memory locations. QU restricts access to only a few memory addresses, but since it is a shared queue, with a high probability of conflict. VA and GA, on the other hand, stress GPU-LocalTM with long transactions. In GA, the read and write sets are the same, and the probability of conflict is high as most of the data is accessed within concurrent transactions. In addition, memory accesses are not coalesced. VA has a large data set and, thus, the probability of conflict is lower, since the read set is larger than the write set. GC also features a larger read set as compared to the write set but, in contrast to VA, its data set is smaller. KM accesses multiple coalesced memory locations performing read-modify-write operations.

Benchmark	Description	CS serialized / CS as transaction	FGL / Atomics
HT	In a N-bucket hash table, each WI inserts elements in a bucket traversing all entries until locating an empty position	CS comprises reading a location within the bucket and, if empty, inserting a value	<i>Atomics</i> : a CAS operation is used to check the status of the position and inserting the value
<i>Hash Table</i>	HT-LC: 256 buckets; HT-HC: 4 buckets		
IT	As HT, but each bucket has an index pointing to the empty position	CS comprises modifying the index	<i>Atomics</i> : an atomic INC operation increments the index
<i>Indexed Table</i>	IT-LC: 256 buckets; IT-HC: 4 buckets		
VA	From STAMP [43], with three groups of 1K elements. 90% of WIs reserve or cancel items and 10% insert or delete items	CS comprises the modification of a number of items depending on the contention level	<i>FGL</i> : a number of locks have to be acquired to allow access to the items. Lock acquisition is serialized to avoid deadlocks
<i>Vacation</i>	VA-LC: 2 items; VA-HC: 4 items		
GC	Decentralized algorithm adapted from[23]. Each WI changes the color of a node depending on its 2 neighbors	CS comprises modifying the color of a node after reading the color of its neighbors	<i>FGL</i> : similarly to VA, locks protects accesses to nodes. Lock acquisition is serialized
<i>Graph Coloring</i>	GC-LC and GC-HC using different graphs		
GA	Uses a genetic algorithm to solve the knapsack problem. From a set of solutions, each WI picks 2 and modifies them using genetic operators, replacing the old ones	CS comprises selecting two solutions, modifying them, and inserting them back to the set avoiding that different WIs pick the same elements	<i>FGL</i> : each solution is protected by a lock. WIs have to acquire two locks. Lock acquisition is serialized to avoid deadlocks
<i>Genetic Algorithm</i>	GA-LC: 64 solutions; GA-HC: 8 solutions		
KM	Clustering algorithm ported from OpenCL (Rodinia 3.0[14, 15]). All code is executed in the GPU, storing clusters in local memory	CS comprises updating the cluster positions depending on the closest points	<i>FGL</i> : each cluster is protected by a lock, which is acquired by using a CAS operation
<i>K-Means</i>	KM-LC: 64 clusters; KM-HC: 8 clusters		
DB	Simulates an in-memory database where each WI inserts 2 values in different IT tables	CS comprises modifying the two indices that manage the tables	<i>Atomics</i> : atomic INC operations increment the two indices
<i>DataBase</i>	DB-LC: 16 tables; DB-HC: 8 tables		
QU	Each WI performs 4 enqueue/dequeue operations on a shared queue managed by two pointers	CS comprises modifying the pointers that manage the head and tail of the queue and inserting/deleting values	<i>Atomics</i> : pointers are managed by using atomic ADD operations
<i>Queue</i>	QU-LC: 12,5% enqueues; QU-HC: 25% enqueues		

Table 5.7: Benchmarks used to evaluate GPU-LocalTM (LC and HC stand for low contention and high contention, respectively; CS stands for critical section).

5.4.1 Performance Evaluation

We perform cycle-based simulation utilizing the Multi2Sim 4.2 simulation framework [57]. All transactional data is stored in local memory.

Speedup. Four TM versions of each benchmark are executed on our simulator, each one using a different conflict detection mechanism: DCD, SMDCD, DCD+pRWsig and DCD+pRWsig+sWosig. Figure 5.6 presents the speedup achieved for those versions relative to the version that serializes the critical sections.

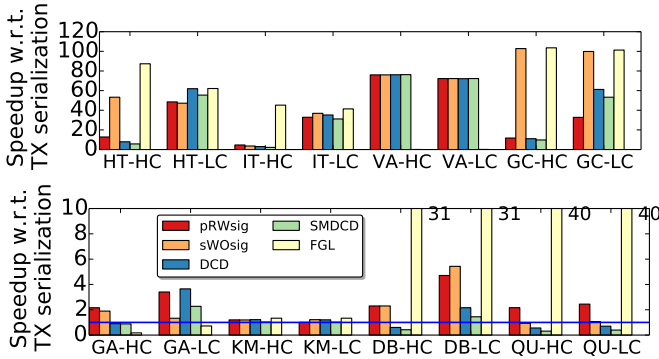


Figure 5.6: Speedup of TM and FGL benchmark versions with respect to the serialized version (pRWsig is DCD+pRWsig; sWosig is DCD+pRWsig+sWosig).

Our first observation is that, for all benchmarks, TM performs similar to, or better than, the serial version, except for the highly contended versions of QU and DB. QU has a very high conflict probability and only enjoys benefits for the lightweight and early conflict detection mechanism that DCD+pRWsig provides. In DB, both versions using signatures improve the performance over the serial code.

Regarding atomics, the FGL versions of HT, IT, DB and QU outperform the TM versions, as they are highly optimized. However, the transactional versions neither involve declaration of additional data structures nor the burden of atomic operation management by the programmer. On the other hand, our simulations show that TM outperforms FGL for VA and GA. The reason is that the use of atomics results in significant overhead for these algorithms due to lock acquisition-release and the mechanisms needed to avoid deadlocks. In addition, FGL requires much greater programming effort than TM.

TM and FGL do not scale well for KM and GA. In GA, the SIMT execution of long transactions hurts performance since work-items that finish the transaction have to wait for the complete wavefront. In KM, the critical section is small compared to the rest of the code, such that the advantages of using TM cannot amortize the associated overhead. The main source of overhead is due

to a significant number of memory accesses that suffer from high contention, as multiple work items try to access the same clusters. This situation results in the serialization of lock acquisition/release operations in FGL, and a high number of retries in TM, which represents more than half of the execution time.

The use of both types of signatures (pRWsig and sWOSig) benefits benchmarks such as HT and GC, where many read-only transactions avoid aborting unnecessarily.

Execution breakdown. Figure 5.7 shows the execution breakdown for all TM and FGL versions of the benchmarks. Our simulations show that most of the overhead occurs during memory operations, as many cycles are spent during conflict detection. However, the use of signatures avoids extra accesses to local memory. For this reason, lower memory overhead is observed in the pRWsig and sWOSig TM versions, as compared to those based only on DCD and SMDCD. The VA benchmark experiences a lower rate of conflicts in both scenarios, and due to its larger critical section, the overhead of transaction management is negligible.

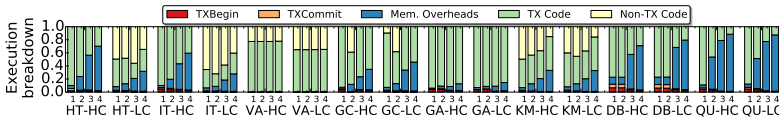


Figure 5.7: Normalized execution breakdown. TXBegin, TXCommit, and Mem. Overheads represent the overheads introduced by the TM system. TX Code represents code executed within a transaction, while Non-TX Code represents code executed outside transactions. Columns 1 to 4 stand for pRWsig, sWOSig, DCD and SMDCD TM versions, respectively.

Transactional instructions. Figure 5.8 shows the percentage of instructions executed within transactions over the total number of instructions executed for each workload. From our simulations we find that HT-LC and IT-LC execute less than 30% of their instructions transactionally, as transactions are short and the probability of conflict is small. KM also has a small fraction of its code running within a transaction. GC and HT-HC reduces their fraction of transactional instructions for sWOSig versus pRWsig. The fast conflict detection provided by shared signatures and the ability to complete read-only transactions without conflicts results in this behaviour.

Commit ratio. Figure 5.9 shows the ratio of transactions committed over the number of transactions started for each workload. As work-items within a wavefront execute in lockstep, the commit ratio is calculated per wavefront, as

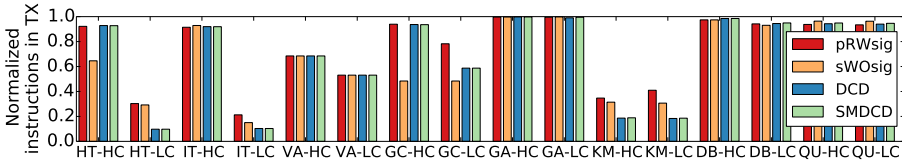


Figure 5.8: Instructions executed within transactions normalized to the total number of instructions executed.

this metric is more directly related to workload throughput. In general, the TM versions based on DCD and SMDCD result in a higher commit ratio, as they are not affected by false positives introduced by the signatures. In some applications, such as DB, HT-HC and GC, the use of signatures improves the commit ratio. Comparing these results with the speedup (see Figure 5.6), we can observe some correlation. The reason is that these applications benefit from the layout of the signatures, and fewer transactions experience re-executions.

False positives. Signatures may return false positives (which are considered conflicts) if the bit to be checked from the signature in the current memory access coincides with the bit set by a previous access to a different memory position (i.e., a signature alias). Figure 5.10 shows the ratio of false positives that occur in the TM versions based on signatures, with respect to the total number of positives. In many scenarios, this ratio is high due to the small size of the signatures. In most cases, false positives result from read-read conflicts that can be filtered out when using shared signatures (sWOSig). DB, however, does not benefit from the use of shared signatures because they saturate.

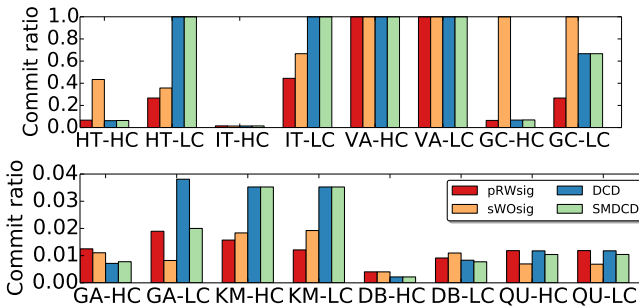


Figure 5.9: Commit ratio.

Forward progress. Figure 5.11 shows the percentage of transactions that execute in transactional, wavefront serialization and work-group serialization

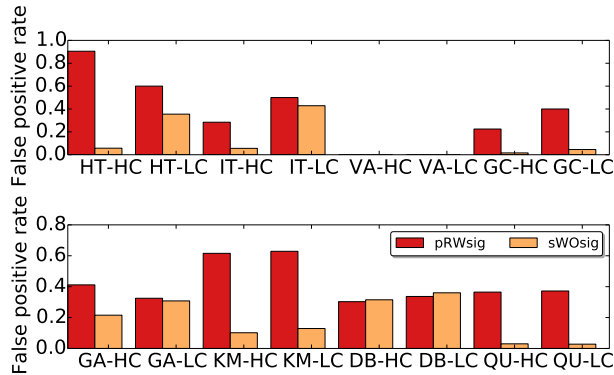


Figure 5.10: False positives.

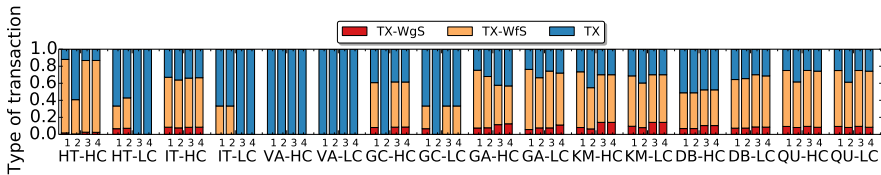


Figure 5.11: Transactional and serialization execution modes. Columns 1 to 4 stand for pRWsig, sWOSig, DCD and SMDCD TM versions, respectively.

modes. In most cases, many transactions (up to 90% in HT-HC) resort to serialization mode (especially, WfS) to assure forward progress. The reason is that, as work-items within a wavefront execute in SIMT fashion, most of the conflicts remain after a transaction retry. We analyzed that scenario (HT-HC), and on average, 48 out of the 64 work-items belonging to the same wavefront conflict when the serialization mode is required.

Discussion. As GPU-LocalTM is configurable, this simulation-based evaluation can serve as guide to programmers or a hint to the compiler to select the most suitable conflict detection algorithm, or to predict the performance when storage resources are not available. The mechanism that exhibits the highest memory overhead is DCD+pRWsig+sWOSig, as it uses vector registers, scalar registers and shadow memory. This method works well for applications with many read-only transactions, such as HT and GC, as conflicts can be detected quickly with the use of signatures and read-only transactions do not conflict. DCD+pRWsig does not use shared signatures, reducing pressure on the scalar register file. This method is more effective for applications that perform read-modify-write opera-

tions. QU, IT, VA and DB are examples that perform similar (or better) when using only pRWsig. Since DCD and SMDCD do not use vector registers, they are well suited for applications that require a large number of those registers. The effectiveness of these methods is limited to applications exhibiting a rather random access pattern (as GA), where the false positive rate can harm performance if using signatures (as HT-LC, IT-LC, and KM).

Table 5.8 summarizes the main transactional features of the benchmarks and the configuration of GPU-LocalTM to obtain best performance according to the evaluation.

Bench.	Features	Best performing GPU-LocalTM configuration
HT	Short transactions Read-only transactions	sWOSig (HT-HC), DCD (HT-LC)
IT	Short transactions Read-modify-write	pRWsig (IT-HC), sWOSig (IT-LC)
VA	Long transactions Few conflicts	Any
GC	Read-only transactions	sWOSig
GA	Long transactions Read-modify-write	DCD
KM	Long transactions Multiple accesses	DCD (KM-HC), sWOSig (KM-LC)
DB	Short transactions Multiple accesses	pRWsig,sWOSig (DB-HC), sWOSig (DB-LC)
QU	Short transactions Many conflicts	pRWsig

Table 5.8: Workload features and the best performing GPU-LocalTM version.

5.5 Improving the serialization mechanism

The wavefront serialization mode implemented in GPU-LocalTM selects a single work-item within the wavefront for execution. Specifically, whenever two consecutive retries of the transaction finish with the same TCM mask, the work-item with lower identifier is the one selected to retry in wavefront serialization mode. This choice has two important consequences.

The first consequence is that by choosing only one work-item for re-execution may not be optimal and multiple work-items could be selected at the same time. Figure 5.12 shows an example of this scenario, simplified to represent wavefronts

of 4 work-items. Initially, the 4 work-items read different memory positions. We assume that memory positions A, B, C and D are located in different memory banks, and reading these memory positions can be done in parallel ①. Then, work-items 0 and 1 try to write to memory positions that make them conflict with each other. The same applies to work-items 2 and 3. As memory positions A, B, C and D are located in different memory banks, the conflict detection mechanism executes in parallel in each memory bank and the 4 transactions abort ②. In this case, the TCM mask holds the value 1111 and the transaction retries for the 4 work-items ③. In the second retry the same scenario will happen activating the wavefront serialization mode. ④. In the GPU-LocalTM wavefront serialization mode, work-item 0 is the only one active to retry the transaction ⑤. After work-item 0 is done, work-items 1, 2 and 3 are be available for execution. Work-item 1 completes in the next retry, but work-items 2 and 3 find the same deadlock issue and the wavefront serialization mode is required again ⑥. In summary, in order to finish this transaction, the wavefront serialization mode is required to be active twice ⑦. However, when entering the wavefront serialization mode for the first time, we could execute work-items 0 and 2 in parallel with no conflicts, as they access different data ⑧. After that work-items 1 and 3 have no conflict in their memory access and they finish without requiring serialization ⑨. Concluding, the serialization mechanism proposed in GPU-LocalTM could be improved by detecting data dependencies and choosing multiple work-items for re-execution. We propose an extension to the serialization mechanism that allows for choosing more than one work-item when running the WfS mode. We call this new mechanism the *multiple* selection mechanism, while selecting only one work-item is called the *single* selection.

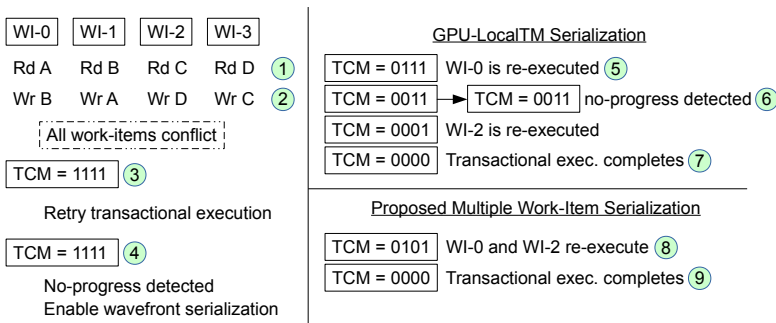


Figure 5.12: Multiple work-item serialization mechanism compared to GPU-LocalTM.

The second consequence of choosing for re-execution the work-item with lower identifier can affect performance in some cases. Figure 5.13 shows an scenario where work-items 0, 1 and 2 write to memory addresses accessed by work-item 3 while work-item 3 writes to a memory address accessed by work-item 2 ①. Assuming coalesced memory accesses, writes are performed simultaneously and conflict. In this case, TCM holds the value 1111 ② and, after re-executing the transaction and reaching the same situation, GPU-LocalTM activates the serialization mechanism ③. This guarantees forward progress for work-item 0 ④, but the remaining work-items need another 2 re-executions in serial mode to be able to complete (⑤, ⑥, ⑦). This could be solved quicker if, after detecting the deadlock, the serialization mechanism chooses work-item 3 for re-execution in the first place ⑧. In that case, work-item 3 finishes its transaction after the first iteration in serial mode ⑨. The remaining work-items have no conflicts and would finish without requiring the use of serialization. We also implement two modes, called *ascending* and *descending* which allow for selecting the order of execution of work-items when in WfS or WgS.

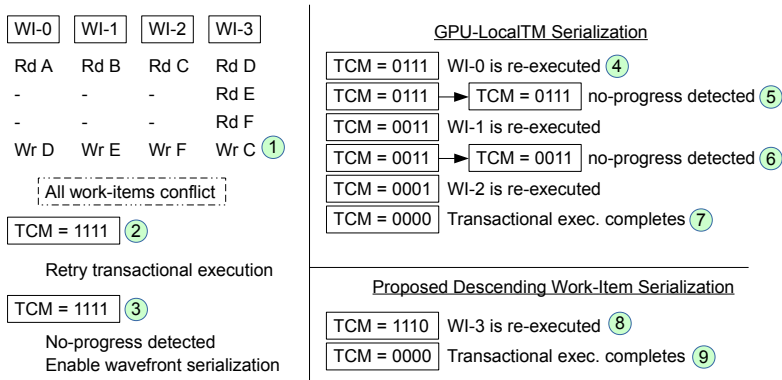


Figure 5.13: Descending work-item serialization mechanism compared to GPU-LocalTM.

Considering the order (ascending or descending) and the choice of work-items (single or multiple) gives us 4 implementations of the serialization mechanism (ascending-single, descending-single, ascending-multiple, and descending-multiple).

The ascending or descending modes are easy to design from the base GPU-LocalTM implementation, as they only differ in the direction the TCM is processed when entering the WfS or WgS modes. To implement the multiple mode,

a new 64-bit *multiple conflict mask* (MCM) is defined per wavefront. Similarly to the TCM, the MCM holds one bit per work-item within a wavefront and is also mapped to two consecutive scalar registers. The MCM is used to mark the subset of multiple work-items to re-execute in WfS mode: those work-items with its MCM bit set to 1 belong to such subset. Initially, when the transaction starts for the first time, every bit of the MCM is set to 1. To build the subset, the MCM is updated whenever a conflict is detected. The offending work-item compares its identifier with that of the owner of the accessed memory location (this information is obtained from the ownership records stored in the shadow area). The owner is always the work-item that made the first transactional access to the memory location (see DCD mechanism in Table 5.3). Thus, the owner is the work-item with which the offending work-item has conflicted to. However, when using the SMDCD mechanism, there is one case where the owner is the same work-item as the offending one (row six in the SMDCD mechanism in Table 5.3). So, if SMDCD is used for conflict detection, the policy is as follows: if the conflicting work-item is also the owner, its bit in the MCM is reset to 0 and it is not re-executed in the next retry.

Assuming we use ascending WfS mode, if the identifier of the work-item that detects the conflict is lower than that of the owner of the memory location, its bit in the MCM is left as 1. That is, this work-item is a candidate for re-execution. Otherwise, if the identifier is higher, its bit in the MCM is reset to 0. The opposite happens when the descending mode is used. Once the transaction ends and the WfS mode is required, the MCM is used to update the TCM before retrying the transaction: $TCM \leftarrow TCM \& \text{not}(MCM)$. Table 5.9 shows the transactional execution of the same code in Table 5.2, but considering a multiple-ascending WfS mode. Now, the transaction is retried three times before committing, instead of six times as in the basic (single-ascending) WfS mode.

5.5.1 Work-item selection mechanism evaluation

The four work-item selection mechanisms (single-ascending, single-descending, multiple-ascending, and multiple-descending), used in WfS mode, are evaluated for HT, IT, GC and KM (VA, GA, DB, and QU do not show significant differences in the results). When WgS mode is enabled, only single-ascending or single-descending work depending on the option selected in WfS.

Figure 5.14 shows the speedup obtained in our simulations for all work-item selection mechanisms, with respect to DCD+pRWsig using the basic single-ascending option. It is observed that selecting multiple work-items in WfS mode

Instruction	EXEC	TCM	TCM prev	MCM	Mode	Comments
TX_Begin	1111	0000	-	1111	TX	Transaction starts
...	1111	1111	-	1010	TX	Transactional execution All WIs detect conflicts
TX_Commit	1111	1111	-	1010	TX	Transaction ends All WIs must re-execute
TX_Begin	1111	0000	1111	1111	TX	Transac. retries for WIs
...	1111	1111	1111	1010	TX	All WIs detect conflicts
TX_Commit	1111	1111	1111	1010	TX	Transaction ends No progress detected
TX_Begin	1111	0101	1111	-	WfS	Transac. retries WIs 0, 2
...	1111	0101	1111	-	WfS	No new conflicts
TX_Commit	1111	0101	1111	-	WfS	Transaction ends WIs 0, 2 commit
TX_Begin	0101	0000	0101	1111	TX	Transac. retries for 1, 3
...	0101	0000	0101	1111	TX	No conflicts detected
TX_Commit	0101	0000	0101	1111	TX	Transaction ends WIs 1, 3 commit

Table 5.9: Example of a transactional execution using the multiple-ascending WfS mode, taking the same code as in Table 5.2. WfS mode uses the MCM. In WfS mode, the TCM is updated as "TCM & not(MCM)" before retrying the transaction.

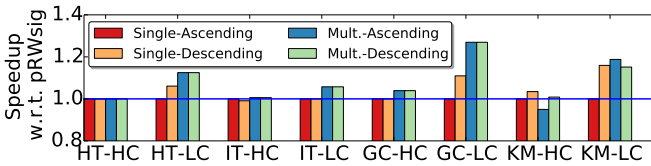


Figure 5.14: Speedup for the different work-item selection schemes used in WfS mode.

improves performance by up to 30%. The use of the descending option exhibits better or similar performance than their ascending counterpart, depending on the memory access pattern. KM-HC is the only one that performs worse when using multiple-ascending due to the high rate of false positives caused in the signatures

used for conflict detection.

Figure 5.15 shows the normalized transactional execution breakdown for all work-item selection schemes. In general, selecting multiple work-items improves performance, reducing the execution time in transactions. In addition, memory overhead due to updates of MCM have no noticeable impact in performance. The reason is that choosing multiple work-items for execution pays off is that memory overhead can be hidden.

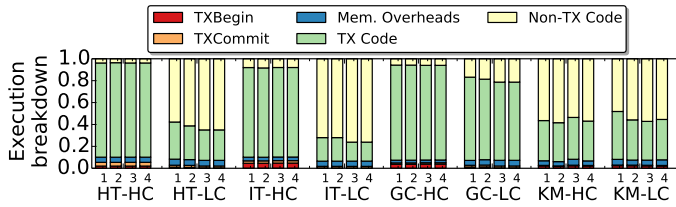


Figure 5.15: Normalized execution breakdown. Labels 1 to 4 stand for single-ascending, single-descending, multiple-ascending and multiple-descending, respectively.

Figure 5.16 shows the fraction of transactions that require the use of serialization for each benchmark. In some cases, there is no difference in the number of transactions in each execution mode. In HT-LC, GC-HC and GC-LC the number of transactions entering the WgS mode is slightly higher for the multiple work-item serialization option. The reason is the relatively high rate of false conflicts in signatures while in the WfS mode, that requires entering WgS mode to assure forward progress.

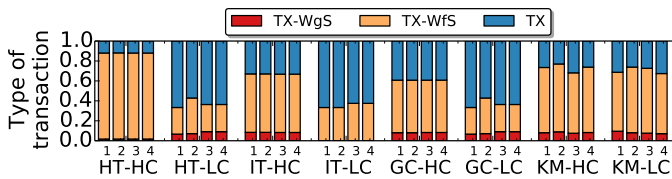


Figure 5.16: Normalized transactional execution mode. Labels 1 to 4 stand for single-ascending, single-descending, multiple-ascending and multiple-descending, respectively.

5.6 GPU-LocalTM: conclusions and future work

In this chapter we present GPU-LocalTM as a hardware TM proposal for GPUs that focuses on the use of local memory. GPU-LocalTM is designed to limit the amount of extra hardware required to support TM, as well as to minimize transaction overhead. Different proposals for conflict detection and forward progress are presented, which could be selected by the compiler/programmer, depending on the transactional memory access patterns or the availability of storage resources. Our experiments show that GPU-LocalTM outperforms the execution of kernels that rely on serialization to solve local memory conflicts. In addition, the programming effort to develop the application is, in general, much lower as compared to using fine-grained locking.

The key features and contributions of GPU-LocalTM are:

- Presents a novel hardware TM design for local memory, focusing on reducing the memory imprint required for transaction management. With regards to this aspect, GPU-LocalTM only allocates memory resources from existing storage if TM is required by the kernel.
- Offers a simple interface to reduce programming complexity when implementing mutual exclusion on GPUs. This interface improves programmability over both fine-grained locks and coarse-grained locks solutions.
- Additionally, integrates an automatic serialization mechanism that prevents programmers from having to implement alternative non-TM code to ensure forward progress. This mechanism can be improved to allow for parallel execution of transactions that are unlikely to conflict.
- Implements multiple conflict detection and version management mechanisms which can be selected by the compiler or the programmer depending on their expected effectiveness or resource limitations.
- Outperforms coarse-grained locks and fine-grained locks implementations for algorithms with low contention (i.e., close to data-parallel programs, which is the main focus of GPUs). In high-contention scenarios, programmers can opt for a sequential implementation which can be offloaded to the CPU or executed by the GPU with a simple code transformation performed by the compiler.

GPU-LocalTM can be improved in different ways. These improvements are proposed as future work.

Discussion: reducing the memory footprint. The memory resources allocated by GPU-LocalTM can be considered high in scenarios where many work-groups are executed concurrently and/or each work-group makes intensive use of local memory. In the current design, GPU-LocalTM aims to minimize the memory resources to be added to the GPU compute units. We achieved this goal by reusing existing allocation. In future work, and if vendors consider implementing a similar approach, dedicated memory structures can be added to allocate transactional metadata such as backup variables and signatures. Another solution can be the dynamic allocation of these structures. For instance, space to back up memory positions can be dynamically allocated when required instead of being pre-allocated by the compiler as in our proposal. This new approach, with the advantage of a better use of the memory resources, has two important drawbacks. Firstly, the latency of memory operation increases, as memory allocation becomes more complex as compared to our simple index-managed approach. Secondly, dynamic allocation can fail in the cases where register or memory usage is close to the hardware limits. Usually, this results in aborting the current transactions due to capacity issues. However, the serialization mechanism of GPU-LocalTM should be re-engineered to deal with that situation, as it fundamentally considers that there are no conflicts due to capacity or false positive issues.

Discussion: integrating the global memory space.

Global memory space can be integrated with GPU-LocalTM in different ways. In this thesis, we discuss two options: pure hardware TM and hybrid TM.

Our proposed eager-eager solution can be extended to use global memory. Following the same principle as in GPU-LocalTM, existing memory resources should be used to store transactional metadata. The L1 data cache can serve the purpose of storing speculative values accessed within a compute unit. Note that in current GPU architectures L1 caches are not coherent and, thus, a simplified version of a cache coherent protocol has to be implemented. The use of signatures as in GPU-LocalTM is more complex in a global memory scenario for several reasons. Firstly, the number of signatures required is very high as the number of work-items to be supported by a GPU is higher as compared to an individual compute unit. Secondly, as these signatures should be evaluated for each memory access, such evaluation is either too slow (if performed sequentially with simple hardware) or too complex to implement in hardware. Lastly, allocation of the signatures is restricted to global memory, as it is visible by all work-items within the GPU, but the latency of such memory space is too high and using signatures would not provide benefits over a value-based comparison. In that sense, dedicated fast storage and hardware should be added to hold and manipulate signatures.

A lightweight solution for the global and local memory integration is to consider a hybrid approach. The hardware proposed in GPU-LocalTM can deal with the SIMT execution. The conflict detection mechanisms implemented in GPU-LocalTM can be extended to detect conflicts in global memory for a given work-group by carving out space from the L1 cache in the same way as GPU-LocalTM uses space from the local memory. Once the hardware TM has detected local and global memory conflicts for all the work-items within a given work-group, then a software technique can be used to detect conflicts coming from work-items belonging to different work-groups. Although software techniques can be slower than hardware TM, we do believe that it can be an effective solution. As GPU programmers usually then to exploit locality within the work-group, conflicts taking place within the a work-group can be detected quickly using dedicated hardware such as GPU-LocalTM. In general, conflicts between different work-groups should be the less common case, which can be solved by software without requiring dedicated hardware.

6 Conclusions

In this thesis we face the intersection of TM and heterogeneous architectures from different perspectives.

Chapter 3 presents an analysis of existing software TM and a set of widely used benchmarks running on an heterogeneous big.LITTLE CPU. Generally, the little cores present lower energy consumption when running the set of applications evaluated; an expected result as it is the intended behavior of the big.LITTLE processor. Nonetheless, not every application is able to benefit from the extra computing power offered by the big cores. The instrumentation that the TM library adds to memory operations produce that the more powerful cores introduce more pressure on the memory system, which in some cases is detrimental. Additionally, the applications with higher abort rate increase the amount of wasted work when executed on the big cluster as compared to the little cluster. For this reason, only applications with high computing requirements compared to the overhead of the TM system are able to benefit from the big cores.

Provided that application performance is different depending on the cluster, we designed a simple scheduler for heterogeneous processors called ScHeTM. ScHeTM is a thread-to-cluster scheduler that assigns TM-instrumented applications to the different clusters of the heterogeneous processors. ScHeTM is configurable to focus on 1) improving performance, 2) reducing energy consumption, or 3) minimizing the amount of work wasted by aborted transactions. When focusing on performance, ScHeTM is able to effectively assign applications to the most suitable cluster. As the little cluster is always more energy efficient than the big cluster, the use of ScHeTM is not able to improve the results obtained by naive scheduling techniques. The difference between both clusters in the number of aborted transactions is not as noticeable as the difference in energy consumption and performance. For this reason, the impact in the schedule performed by

ScHeTM is smaller compared to the other metrics. In the end, given the proper configuration of ScHeTM, execution time of the tested applications is reduced in 40% and their energy consumption in 15%.

Results of Chapter 3 are available in the following publications:

Energy Efficiency of Software Transactional Memory in a Heterogeneous Architecture. Emilio Villegas, Alejandro Villegas, Angeles Navarro, Rafael Asenjo, Yash Ukidave, and Oscar Plata.

In *8th Workshop on the Theory of Transactional Memory (WTTM 2016 co-located with PODC 2016)*.

Chicago (IL), USA, July 2016

Evaluación del Consumo Energético de la Memoria Transaccional Software en Procesadores Heterogéneos. Emilio Villegas, Alejandro Villegas, Angeles Navarro, Rafael Asenjo and Oscar Plata.

In *XXVII Jornadas de Paralelismo, JP'16*.

Salamanca, Spain, September 2016

Planificación thread-to-cluster de aplicaciones que utilizan memoria transaccional sobre un procesador heterogéneo. Alejandro Villegas, Ernesto Rittwagen, Angeles Navarro, Rafael Asenjo and Oscar Plata.

In *XXVIII Jornadas de Paralelismo, JP'17*.

Málaga, Spain, September 2017

Chapter 4 explores TM and heterogeneous CPU+GPU processors (also known as APU processors). In this case, the focus is implementing software TM on top of such architecture. The goals are 1) providing an unified and simple interface to implement mutual exclusion using TM on both sides (CPU and GPU), 2) explore an implementation adapted to the particular characteristics of each application, and 3) design a lightweight communication method to perform cross-platform conflict detection with minimum overhead.

Our proposal, called APUTM defines an unified TM interface with different implementations. In the case of the GPU, whose architecture can support thousands of parallel transactions, APUTM solves per-wavefront conflicts prior to accessing to main memory in order to reduce pressure on the memory subsystem. Additionally, a timestamp-based solution inspired in prior CPU and GPU

software TM [20, 70] is used to minimize the communication overhead. This communication is effectively done using the platform-atomic operations available in HSA-compliant heterogeneous processors.

Results of Chapter 4 are available in the following publications:

Towards a Software Transactional Memory for heterogeneous CPU-GPU processors. Alejandro Villegas, Angeles Navarro, Rafael Asenjo and Oscar Plata. In *3rd IEEE International Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms (Repara 2017, part of ParCo2017)*. Bologna, Italy, September 2017

Lastly, Chapter 5 proposes the use of TM in GPU architectures focusing on transactions that make use of the local (scratchpad) memory. The main challenges found in implementing TM on top of said memory space are the lack of a cache that can be used for both storage and for conflict detection support as in commercial CPUs, the limited amount of memory resources available, and coupling the transactional execution with the SIMT execution model.

Our proposal, GPU-LocalTM, is intended to minimize the amount of extra hardware required to manage transactions. In this sense, transactional metadata is allocated in existing storage such as registers and local memory. This allocation of resources is performed by the compiler and, thus, these resources are freely available if TM is not required. Additionally, it offers different conflict detection mechanism that can be selected by the user or the compiler attending to the resources available and program requirements. Furthermore, the SIMT execution is modified to consider transactions. The implementation of the transactional SIMT execution presented in GPU-LocalTM has two main advantages: 1) it is hardware-managed and does not require complex changes in the program or the compiler, and 2) it integrates an automatic serialization mechanism that ensures forward progress without requiring programmers to implement a fallback code. Our simulation-based evaluation shows that our GPU-LocalTM is able to outperform GPU coarse-grained locks implementations of mutual exclusion. In some programs, fine-grained locks implementations are slower than GPU-LocalTM as the lock management overhead can be prohibitive in SIMT execution. If the program permits a simple atomic-based implementation, then performance is better than using GPU-LocalTM, but at the cost of a higher programming effort. With regards to programming effort, the simple GPU-LocalTM interface simplifies the implementation of mutual exclusion when compared to coarse-grained locks, fine-grained locks, atomics, and *ad-hoc* implementations.

Results of Chapter 5 are available in the following publications:

Hardware support for Local Memory Transactions on GPU Architectures. Alejandro Villegas, Angeles Navarro, Rafael Asenjo, Oscar Plata, Rafael Ubal and David Kaeli.

In *10th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2015, part of FCRC 2015)*.

Portland (OR), USA. June 2015

Memoria Transaccional Hardware en Memoria Local de GPU. Alejandro Villegas, Angeles Navarro, Rafael Asenjo and Oscar Plata.

In *XXVI Jornadas de Paralelismo, JP'15*.

Córdoba, Spain, September 2015

Improvements in Hardware Transactional Memory for GPU Architectures. Alejandro Villegas, Rafael Asenjo, Angeles Navarro and Oscar Plata.

In *19th Workshop on Compilers for Parallel Computing, (CPC'16)*.

Valladolid, Spain, July 2016

Hardware support for scratchpad memory transactions on GPU architectures. Alejandro Villegas, Angeles Navarro, Rafael Asenjo, Oscar Plata, Rafael Ubal and David Kaeli.

In *23rd International European Conference on Parallel and Distributed Computing, EuroPar'17*

Santiago de Compostela, Spain, September 2017

Lightweight Hardware Transactional Memory for GPU Scratchpad Memory. Alejandro Villegas, Rafael Asenjo, Angeles Navarro, Oscar Plata, and David Kaeli.

In *IEEE Transactions on Computers no. 99* doi:10.1109/TC.2017.2776908

November 2017

6.1 Future work

To conclude this thesis, we want to propose future research lines related with our work.

As mentioned in Chapter 3, ScHeTM can be improved in different ways: the mechanism included in our scheduler used to avoid resource underutilization can be improved, the training phase can be reduced (or omitted), and applications not using TM have to be included in our model. Currently, we are improving this work using a different approach. We are embedding the behavior of ScHeTM into the TM library itself instead of considering it a separate scheduling mechanism. The goal is that the applications can dynamically select the most appropriate cluster for execution on-the-fly (i.e., the application can start executing on one cluster, and switch to a different cluster if it is required). With this approach, irregular applications can switch from one cluster to another in case the behavior changes during the execution. In addition we are studying better metrics (such as transactions executed per watt and per second) to improve the scheduling mechanism.

APUTM, presented in Chapter 4, opens a promising research line. As applications start to require more advanced CPU-GPU synchronization methods, APUTM can be used as baseline for future heterogeneous software, hardware, and hybrid TM implementations. The hybrid solution is especially relevant, as current CPUs support hardware TM, but synchronization with the integrated GPU has to be done via software. Hybrid solutions can also consider implementing some of the TM-related operations (such as metadata management) in hardware to speed up our current software proposal. For instance, GPUs can provide hardware to deal with the transactional SIMT execution as in GPU-LocalTM. Besides these proposals, future work can explore different TM implementations for the APUTM interface and improve its performance.

Lastly, in Chapter 5 we present GPU-LocalTM as a hardware TM for GPUs. Future work can extend our approach to consider transactions sharing both global memory and local memory addresses. In particular, our proposal can be used as hardware support for a more flexible and advances hybrid TM solution. Briefly, transactions within the same work-group can detect their conflict via hardware, while synchronization with different work-groups (or even a multi-core CPU) can be done using software techniques.



UNIVERSIDAD
DE MÁLAGA

Apéndice A

Resumen en español

Si observamos las necesidades computacionales de hoy, y tratamos de predecir las necesidades del mañana, podemos concluir que el procesamiento heterogéneo estará presente en muchos dispositivos y aplicaciones: desde teléfonos móviles hasta supercomputadores, pasando por coches y ciudades inteligentes. El motivo es lógico: algoritmos diferentes y datos de naturaleza diferente encajan mejor en unos dispositivos de cómputo que en otros. Pongamos como ejemplo una tecnología de vanguardia como son los vehículos inteligentes. En este tipo de aplicaciones la computación heterogénea no es una opción, sino una obligación. En este tipo de vehículos se recolectan y analizan imágenes, tarea para la cual los procesadores gráficos (GPUs) son muy eficientes tanto energéticamente como computacionalmente. Muchos de estos vehículos deben utilizar algoritmos sencillos, pero con grandes requerimientos de tiempo real. Por tanto, las partes críticas de estos algoritmos pueden implementarse directamente en hardware utilizando FPGAs. Y, por supuesto, los tradicionales procesadores multinúcleo tienen un papel fundamental en estos sistemas, tanto organizando el trabajo de otros coprocesadores o aceleradores como ejecutando tareas en las que ningún otro procesador es más eficiente. No obstante, los procesadores tampoco siguen siendo dispositivos homogéneos. Los diferentes núcleos de un procesador pueden compartir un ISA, pero ofrecer diferentes características en términos de potencia y consumo energético que se adaptan a las necesidades de cómputo de la aplicación.

Programar este conjunto de dispositivos es una tarea compleja. Para acceder a estos dispositivos se han creado diferentes tecnologías. Como ejemplo, OpenCL [38] proporciona una interfaz que permite comunicar distintos elementos de cómputo manteniendo la portabilidad (aunque no garantiza que el rendimiento se mantenga de un dispositivo a otro). En este y otros lenguajes similares existen

mecanismos para sincronizar los distintos dispositivos, aunque son muy básicos. Habitualmente, esta sincronización se basa en operaciones atómicas, ejecución y terminación de kernels, barreras y señales. Con estas primitivas de sincronización los programadores pueden construir otras estructuras más complejas. El objetivo de los programadores es obtener mejores soluciones para garantizar la exclusión mutua en el acceso a datos compartidos. Sin embargo, la programación de estos mecanismos normalmente es tediosa y propensa a fallos. La memoria transaccional (TM por sus siglas en inglés) [35] se ha propuesto como un mecanismo avanzado a la vez que simple para garantizar la exclusión mutua. TM proporciona una interfaz sencilla para definir las secciones de código en las que se debe garantizar la exclusión mutua (sección crítica). TM emplea un mecanismo de sincronización optimista capaz de mejorar otros mecanismos de sincronización basados en cerrojos. La popularidad actual de TM es tal que está siendo incluido como parte de los futuros estándares de C++.

Por su parte, los fabricantes de procesadores están incluyendo TM en sus CPUs para complementar otras opciones de sincronización más básicas. Intel proporciona una extensión a su ISA llamada TSX para soportar TM [72]. Por su parte, IBM ha implementado distintos sistemas de TM en sus procesadores IBM BlueGene/Q [67], System z [37] y Power 8 [3]. En cuanto a las GPUs, aún no existen productos comerciales, pero TM es un campo de investigación activo tanto en implementaciones software [12, 70, 36] como hardware [29, 28, 17]. Dada la importancia que está cobrando TM en distintos tipos de procesadores, es importante comprender cómo TM puede adaptarse a los procesadores heterogéneos. Desde nuestro punto de vista, es interesante comprobar cómo funcionan las implementaciones software (STM), hardware (HTM) en estos dispositivos y qué papel puede jugar la planificación en ellos.

A.1 Motivación y cuestiones de investigación

Dada la creciente demanda de poder de procesamiento y de una alta eficiencia energética, los procesadores heterogéneos están siendo un producto por el que la industria está apostando muy fuerte. Sin embargo, la complejidad de estos sistemas pone una carga de trabajo importante en los programadores. Tanto la industria como la comunidad investigadora están tratando de reducir este esfuerzo de programación proporcionando nuevos modelos de programación y técnicas de planificación [5, 46, 50]. Dado que TM es una técnica efectiva en CPUs homogéneas, es importante entender su aplicabilidad en arquitecturas heterogéneas.

El área de investigación de TM en dispositivos heterogéneos es enorme. En primer lugar, TM puede ser implementado en software, hardware o como una solución híbrida. Además, las técnicas de planificación de transacciones son un área de investigación activa [48, 53, 73] que trata de maximizar los beneficios proporcionados por TM. Una vez seleccionada una de estas áreas de investigación, ésta puede aplicarse a diferentes dispositivos heterogéneos. Por ejemplo, puede aplicarse a CPUs heterogéneas de la familia big.LITTLE diseñada por ARM. También puede aplicarse a los procesadores tipo APU, que integran una CPU multinúcleo y una GPU dentro del mismo chip. Por último, TM puede implementarse como una característica especial de uno de los aceleradores de un sistema heterogéneo.

Dada esta intersección entre diferentes áreas de investigación en TM y diferentes dispositivos heterogéneos, la principal pregunta que trata de responder esta tesis es: *¿Podemos aplicar TM en dispositivos heterogéneos?* Y, más concretamente, tratamos de responder a otras preguntas: *¿Puede la planificación mejorar el rendimiento de TM en una CPU heterogénea?* *¿Es posible, con la tecnología actual, implementar TM en un procesador APU?* y *¿Es posible realizar un diseño de TM en una GPU utilizando los recursos hardware disponibles en ella?*

Las principales contribuciones de esta tesis, y las publicaciones asociadas, que tratan de responder a estas preguntas son:

- Un análisis de una popular librería de TM por software ejecutada sobre una CPU heterogénea [65, 66] y una propuesta de planificación que tenga en cuenta las características de TM [64].
- Una implementación de TM por software para procesadores APU [62]. El objetivo principal es minimizar la comunicación entre CPU y GPU, y comprender las ventajas y desventajas de TM en estos dispositivos.
- Una diseño de TM por hardware para un acelerador de tipo GPU [63, 61, 59, 60, 58]. Más concretamente, tratamos de proporcionar soporte TM en la memoria tipo *scratchpad* ofrecida por estos aceleradores.

Esta tesis se organiza de la siguiente forma. En el capítulo 1 se describen distintos tipos de arquitecturas heterogéneas, TM, y se proporciona una motivación para esta tesis. El capítulo 3 proporciona el análisis de una librería de TM por software ejecutándose sobre un procesador heterogéneo y proporciona una propuesta de planificación de aplicaciones instrumentadas con TM. El capítulo 4 presenta un diseño novedoso de una librería TM por software para procesadores

APU. El capítulo 5 describe nuestra propuesta de diseño hardware para proporcionar TM en GPUs. Por último, el capítulo 6 presenta las conclusiones de esta tesis.

A.2 Memoria transaccional en CPUs heterogéneas

En este capítulo analizamos la utilización de memoria transaccional software en una CPU heterogénea de la familia big.LITTLE y proponemos un mecanismo de planificación para mejorar su eficiencia energética y rendimiento.

Análisis de las aplicaciones con TM sobre una CPU heterogénea. En primer lugar, realizamos un estudio del conjunto de aplicaciones STAMP [43] utilizando la librería de TM por software TinySTM [25, 26]. A diferencia del trabajo previo [31, 30, 44, 27, 7, 6, 54], que utiliza simuladores para estimar rendimiento y consumo energético, nuestro análisis se realiza sobre hardware real utilizando monitores de energía. El hardware utilizado es el sistema ODROID XU3 [2] que incorpora un procesador de la familia big.LITTLE de ARM fabricado por Samsung (ver Figura A.1) y sensores de energía INA231 de Texas Instruments¹ El procesador incorpora, además, una GPU MALI-T628 y 2 Gbytes de memoria DDR3 de bajo consumo. El sistema operativo instalado sobre el dispositivo es el Linux odroid 3.10.59+.

Del análisis de estas aplicaciones concluimos que las aplicaciones escalan mejor en los núcleos orientados al bajo consumo. Todas las aplicaciones presentan un menor consumo energético en dichos núcleos. En cuanto al rendimiento, la mayoría de las aplicaciones se ejecutan más rápidamente en los procesadores de bajo consumo. El principal motivo es que son aplicaciones *memory bound* que no solo no aprovechan la potencia de cómputo de los núcleos orientados al rendimiento, sino que introducen mayor sobrecarga en el subsistema de memoria. Por contra, aquellas aplicaciones con mayores requerimientos de cómputo sí pueden aprovechar la mayor potencia de los núcleos orientados al rendimiento.

Dicho análisis se ha realizado de forma aislada para cada aplicación. Sin embargo, en un escenario real, varias aplicaciones pueden ejecutarse de forma simultánea en el mismo sistema. En este escenario, unas aplicaciones ocuparán los núcleos little, orientados al bajo consumo, mientras que otras ocuparán los núcleos big, orientados al rendimiento. Para considerar esta situación, hemos realizado un estudio de las aplicaciones cuando ambos tipos de núcleos se encuentran ocupados simultáneamente, cada uno ejecutando una aplicación. El resultado de

¹<http://www.ti.com/product/INA231>

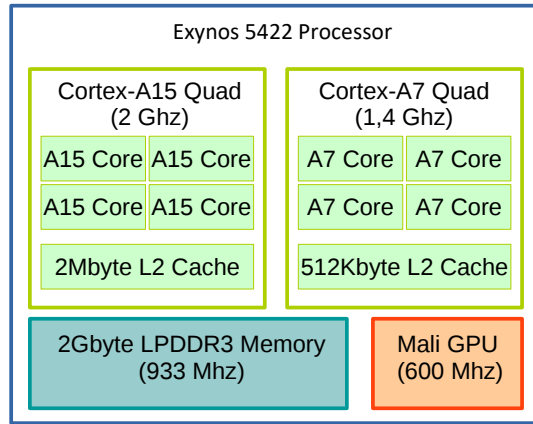


Figure A.1: Esquema del procesador big.LITTLE Exynos 5422 de Samsung integrado en la plataforma ODROID XU3.

este análisis es que ofrecen un rendimiento similar a cuando se ejecutan de forma aislada, pero con una pequeña degradación.

La tabla A.1 resume el resultado de este análisis. En ella se compara el consumo energético, el tiempo de cómputo y el número de transacciones abortadas en ambos tipos de núcleos. Señalamos que, si las diferencias en los resultados son menores al 10%, indicamos que ambos tipos de núcleos son similares. En ella observamos que los núcleos little siempre son energéticamente más eficientes que los big. En cuanto a tiempo de cómputo, únicamente la aplicación *labyrinth*, que tiene mayor carga computacional, es capaz de aprovechar la potencia de los núcleos big. El número de transacciones que abortan es un factor que depende de cada aplicación y de cada tipo de núcleo.

	Tiempo de ejecución	Consumo energético	Transacciones abortadas
Intruder	little	little	similar
Kmeans	little	little	little
Labyrinth	big	little	similar
Ssca2	similar	little	big
Vacation	little	little	similar

Table A.1: Resumen del rendimiento, consumo energético y número de transacciones abortadas de cada aplicación. Se indica qué tipo de núcleos es el más eficiente, o similar si las diferencias son menores al 10%.

Planificación de aplicaciones con TM sobre una CPU heterogénea.

Dado el análisis previo, se ha diseñado un planificador llamado ScHeTM (Scheduling - Heterogeneous CPUs - TM applications) para CPUs heterogéneas que tiene en cuenta los resultados obtenidos en tiempo de ejecución, consumo energético y número de transacciones abortadas para realizar la planificación.

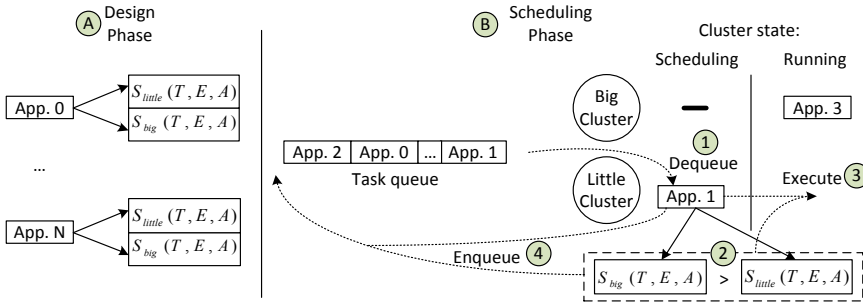


Figure A.2: Diagrama del planificador ScHeTM.

La figura A.2 muestra el diseño de ScHeTM. Este diseño se compone de 2 fases. En una primera fase (A: *design phase*), las aplicaciones son analizadas, calculando una función de idoneidad $S_c(T, E, A)$, siendo los parámetros T , E , y A medidas del tiempo de cómputo, consumo de energía y transacciones abortadas, y c se refiere bien al cluster big o al cluster little. Por tanto, para cada aplicación, ScHeTM debe calcular las funciones $S_{big}(T, E, A)$ e $S_{little}(T, E, A)$. La función de idoneidad, acotada entre 0 y 1, indica cómo de bien se adapta la aplicación a cada uno de los clusters. Valores cercanos al 0 indican que la aplicación tendrá un mal comportamiento en dicho cluster, mientras que valores cercanos al 1 indican que la aplicación se adapta mejor a dicho cluster. Para el cálculo de estas funciones, las aplicaciones deben estar debidamente instrumentadas, pero dicha instrumentación puede omitirse para pasar a la siguiente fase.

Una vez calculadas estas funciones, ScHeTM está listo para recibir aplicaciones (por simplicidad, asumimos que las aplicaciones vienen proporcionadas en una cola) y comienza la segunda fase (B: *scheduling phase*). En esta fase, un cluster se puede encontrar en 2 estados diferentes: o bien planificando una aplicación (si está ocioso), o bien ejecutando una aplicación que le haya sido asignada. La segunda fase se ejecuta cada vez que un cluster queda ocioso y existen aplicaciones pendientes de planificar. En nuestro ejemplo estamos suponiendo el cluster big ocupado ejecutando una aplicación y, por tanto, no necesita planificar

ninguna, y el cluster little ocioso. En este caso, es el cluster little el que recibe una aplicación de la cola de aplicaciones ①. A continuación se verifica si dicho cluster es idóneo para ejecutar la aplicación. Para ello, se comparan los valores $S_{big}(T, E, A)$ e $S_{little}(T, E, A)$ ②. Si esta función tiene un valor mayor en el cluster que se encuentra ocioso, entonces la aplicación se planifica a ejecutar en dicho cluster. En nuestro ejemplo, si $S_{big}(T, E, A) \leq S_{little}(T, E, A)$ (esto es, la condición ② es falsa), la aplicación pasa a ejecución en el cluster little ③. Por el contrario, si esta función tiene un valor menor en el cluster ocioso, entonces la ejecución de la aplicación se pospone. En nuestra implementación, la aplicación es devuelta a la cola ④. En ese caso, el cluster ocioso recibe una nueva aplicación a ejecutar, repitiendo este proceso. Para evitar que el cluster quede ocioso demasiado tiempo, cada vez una aplicación es rechazada se relajan las condiciones para aceptar la ejecución de nuevas aplicaciones. Esto se consigue incrementando la función $S_c(T, E, A)$ del cluster ocioso para permitirle ser más cercana a $S_c(T, E, A)$ del otro cluster.

En nuestras pruebas, y dependiendo de su configuración, ScHeTM consigue reducir en hasta un 40% el tiempo de cómputo y un 15% la energía necesaria para ejecutar un conjunto de aplicaciones del benchmark suite STAMP en comparación con un planificador ávido. Además, ScHeTM también logra reducir el número de transacciones que abortan ya que planifica las aplicaciones en el cluster más adecuado para su ejecución.

A.3 Memoria transaccional en procesadores APU

En los procesadores multinúcleo, TM ha aparecido como una alternativa prometedora a las técnicas basadas en cerrojos para garantizar exclusión mutua y está siendo incluida como parte de procesadores comerciales [72, 67, 37, 3]. De igual forma, dado que las GPUs se están convirtiendo en el acelerador más popular de la actualidad, los fabricantes están integrándolas dentro del mismo chip, creando las llamadas APUs (Accelerated Processing Units). Sin embargo, TM y las APUs son todavía mundos separados.

En este capítulo proponemos una librería de TM software enfocada a su uso en procesadores APU. El objetivo es que las transacciones puedan ejecutarse tanto en CPU como en GPU simultáneamente y que se permita la sincronización en forma de exclusión mutua entre ambos dispositivos. Nuestra propuesta, llamada APUTM, se enfoca en minimizar la comunicación entre la CPU y la GPU de los metadatos requeridos para manejar TM.

APUTM se basa en NÖrec [20] y GPU-STM [70]. En nuestra implementación, todas las transacciones (tanto de CPU como de GPU) comparten un reloj global. Este reloj es registrado por cada transacción al inicio y es actualizado cada vez que una transacción termina con éxito. De esta forma, las transacciones pueden verificar rápidamente si han tenido conflictos con otras. Si el reloj tiene el mismo valor que el registrado al inicio de la transacción, ésta puede actualizar la memoria directamente sin señalar ningún conflicto. En caso de que el reloj haya cambiado, se requieren otras verificaciones. Para ello, cada transacción de CPU y GPU almacena, de forma privada, el conjunto de valores leídos de memoria y los valores especulativos a escribir en ella. En caso de que los valores leídos no hayan sido alterados, la transacción puede hacer definitivos en memoria sus valores especulativos. En otro caso, la transacción debe reiniciar descartando los valores especulativos. Puesto que dentro de la GPU varias transacciones pueden compartir el mismo contador de programa (ejecución en *lockstep*), detectar conflictos entre dichas transacciones en un proceso complicado. Para ello se ha considerado una estructura que almacena los datos modificados por un *wavefront* (hilos que se ejecutan en *lockstep*). Comprobando esta estructura en paralelo, dichas transacciones pueden verificar si tienen conflictos con alguna otra, y detener su ejecución antes de comprobar conflictos con otros wavefronts y con las transacciones en CPU. La figura A.3 muestra los metadatos necesarios para implementar la funcionalidad previamente descrita. Remitimos al lector a las figuras 4.2 y 4.3 del capítulo 4 para verificar el pseudocódigo que describe la funcionalidad previamente expuesta.

```

1 //Global metadata
2 atomic_int * gclock;

1 //Private metadata
2 struct pr_descr{
3   int snapshot;
4   int status; //RUNNING or ABORTED
5   <address,value> * reads;
6   <address,value> * writes;
7 };

1 //Wavefront metadata
2 struct wf_descr{
3   atomic_long * commit_mask;
4   atomic_int * leader_id;
5   <address,owner> * wf_writes;
6   atomic_int * next_write;
7 };

```

Figure A.3: Metadatos globales, privados y de wavefront necesarios para implementar APUTM.

APUTM ha sido diseñado para respetar la propiedad de opacidad [32] verificando que la memoria es consistente con la transacción en cada acceso. Además, se ha considerado un modelo en el que los conjuntos de lectura y escritura privados a cada transacción son unificados. De esta forma 1) se ahorra espacio de

almacenamiento de metadatos, 2) se acelera la detección de conflictos en transacciones que realizan operaciones de lectura-modificación-escritura sobre la misma posición de memoria, a cambio de 3) perder precisión en la detección de conflictos de transacciones de solo lectura o que modifican posiciones diferentes a las leídas.

Para la evaluación de APUTM se ha diseñado una carga de trabajo sintética que estresa distintas características del sistema con el fin de caracterizarlo. Además, se han diseñado 3 *microbenchmarks* para evaluar nuestra propuesta en escenarios más realistas.

De nuestras pruebas se extraen varias conclusiones. En primer lugar, la escalabilidad en CPU es mayor dado que aprovecha mejor la detección rápida de conflictos utilizando el reloj global. Además, la ejecución lockstep en la GPU no favorece al modelo transaccional ya que un conflicto por parte de una única transacción afecta al rendimiento de todo un wavefront. En segundo lugar, con un reparto de carga dinámico, la CPU tiende a ejecutar más transacciones siempre que el reloj global esté en uso. En caso de omitir su utilización, el reparto de carga dinámico asigna más transacciones a la GPU. En tercer lugar, la mayor parte de los conflictos ocurren en la GPU debido 1) a la ejecución en lockstep que no permite desacoplar la ejecución de transacciones que tienen conflicto y 2) al mayor número de transacciones simultáneas en ejecución. Por último, APUTM consigue mejorar el rendimiento de la ejecución secuencial en aquellos escenarios con suficiente carga de trabajo para amortizar la sobrecarga introducida por la instrumentación software de la transacción.

A.4 Memoria transaccional en memoria local de GPU

Los procesadores gráficos (GPUs) han sido adoptados como aceleradores muy populares en aplicaciones que presentan un gran paralelismo de datos gracias a su modelo de ejecución *Single Instruction - Multiple Thread* (SIMT), su jerarquía de memoria y la disponibilidad de cientos o miles de núcleos de ejecución. Tecnologías como CUDA [47] y OpenCL [38] permiten el acceso a este hardware para el cómputo de propósito general. En este paper utilizamos la nomenclatura de OpenCL. Una GPU está compuesta de varios núcleos SIMT llamados *compute units* (CU). Los hilos de ejecución se denominan *work-items* y se agrupan en *work-groups*. Un programa a ejecutar se denomina *kernel*, y está compuesto por varios *work-groups*. El número de *work-groups* y *work-items* es definido por el programador. Un *work-group* es siempre planificado a una misma CU, y

una CU puede ejecutar varios work-groups. Los recursos hardware de la CU son repartidos estáticamente entre todos los work-groups, propiciando cambios de contexto muy ligeros. Dentro de la CU, los work-items de un mismo work-group son agrupados en *wavefronts* de tamaño fijo. Dentro de la CU, el wavefront es la unidad planificable. Una CU posee dos espacios de memoria direccionables por los work-items: memoria global y memoria local. La memoria global es accedida por todos los work-items que se encuentran en ejecución. Esta memoria, de gran tamaño, tiene una gran latencia (en parte aliviada por una jerarquía de caches no coherentes) y se puede utilizar para comunicar work-items de work-groups planificados en distintas CUs, además de ser la que comunica la GPU con la CPU anfitrión. La memoria local tiene una latencia y tamaño menores. Existe un espacio de memoria local en cada una de las CU. Los work-items de un work-group planificado en una CU tienen acceso a esta memoria de forma compartida. Debido a su baja latencia, la memoria local es utilizada como *scratchpad* por los work-items de un mismo work-group. Work-items pertenecientes a diferentes work-groups planificados sobre la misma CU no comparten la memoria local de forma lógica (esto es, no pueden utilizarla para comunicarse entre ellos), pero sí de forma física. Además de estos dos espacios direccionables, cada work-item posee su propio espacio de memoria privado, típicamente mapeado en registros.

En general, las aplicaciones paralelas con múltiples hilos de ejecución deben utilizar mecanismos explícitos para la sincronización. Esta sincronización puede deberse a la necesidad de establecer secciones críticas en las que la exclusión mutua esté garantizada. Sin embargo, en el modelo de ejecución SIMT esto supone un reto mayor que en otras arquitecturas. Una solución típica es serializar la ejecución. En este caso, sólo un hilo ejecuta la sección crítica de forma secuencial, limitando el paralelismo de la aplicación. Otra solución consiste en delegar la ejecución de la sección crítica a la CPU anfitrión. Sin embargo, si los datos protegidos por la sección crítica se encuentran en memoria local, habría que copiarlos a memoria global, y luego al espacio direccionable por la CPU, lo que requiere una gran cantidad de ciclos de reloj. Otra posible solución consiste en implementar un mecanismo de sincronización basado en cerrojos utilizando operaciones atómicas. Implementar cerrojos de grano grueso es una solución fácilmente adoptable por los programadores. Sin embargo, el acceso a la sección crítica se hace en serie, perjudicando el rendimiento. Los cerrojos de grano fino, a priori, pueden ser una solución más eficiente. Sin embargo, es más difícil de implementar y propenso a *deadlocks* y *livelocks*.

La memoria transaccional (TM) [35] se ha propuesto como una alternativa prometedora al uso de cerrojos. En las arquitecturas GPU están empezando a aparecer los primeros trabajos de TM, tanto software [12, 70, 36, 55] como hard-

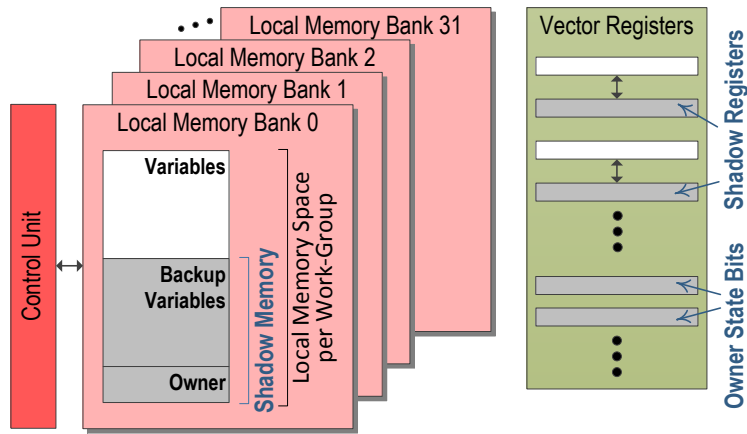


Figure A.4: Recursos de memoria alojados por GPU-LocalTM.

ware [29, 28, 17]. En este capítulo de la tesis nos centramos en las soluciones hardware. Los trabajos de investigación en TM por hardware para arquitecturas GPU únicamente consideran el espacio de memoria global. Además, estas propuestas requieren cambios significativos en el hardware y organización de la GPU, motivos por los que los fabricantes pueden estar menos motivados para su implantación. Además, el espacio de memoria local no ha sido tenido en consideración. Este espacio de memoria es importante para los programadores, puesto que es utilizado para mejorar de forma significativa el rendimiento de sus aplicaciones. Por estos motivos, en esta tesis proponemos un soporte TM hardware en GPU que sea ligero, eficiente, y que cubra el espacio de memoria local.

Nuestra propuesta, GPU-LocalTM, extiende el ISA de la arquitectura con 2 nuevas instrucciones para marcar el comienzo y final de la transacción: *TX_Begin* y *TX_Commit*. Estas instrucciones pueden ser utilizadas por los compiladores para proporcionar sentencias de más alto nivel en lenguajes como OpenCL. Dentro de una transacción, todas las operaciones sobre memoria local gestionadas por la unidad de acceso a memoria son consideradas transaccionales (esto es, no existen unas instrucciones de lectura y escritura explícitas *TX_Read* y *TX_Write*). Además, nuestra propuesta no requiere la inclusión de elementos de memoria nuevos para almacenar los metadatos de la transacción, sino que aprovecha los ya existentes en las CU de la GPU. La figura A.4 muestra los elementos de la CU que se ven afectados por la implementación de GPU-LocalTM. Para implementar GPU-LocalTM se requieren cambios en el modelo de ejecución SIMT, y se necesita implementar mecanismos de detección de conflictos y gestión de versiones.

Modelo de ejecución SIMT transaccional. El modelo de ejecución SIMT presente en las GPU se basa en el uso de máscaras de ejecución. En nuestra arquitectura de base (AMD Southern Islands), estas máscaras se llaman EXEC y VCC. Cada wavefront mantiene una copia privada de estas máscaras, que tienen un bit por cada work-item en el wavefront. Estas máscaras son de 64 bits puesto que un wavefront contiene 64 work-items. La máscara EXEC indica qué work-items dentro del wavefront están activos, mientras que VCC funciona como un flag Z, indicando el resultado de las instrucciones aritméticas y de comparación. Con estas máscaras, los compiladores implementan bucles, condiciones y saltos utilizando un mecanismo de predicación.

En nuestra implementación de TM proponemos el uso de una nueva máscara *Transaction Conflict Mask* (TCM) por wavefront, mapeada en un registro escalar, que indica qué work-item ha presentado un conflicto en los accesos a memoria. La máscara TCM, de 64 bits, es inicializada a 0 al comienzo de la transacción. En un acceso a memoria, si se detecta un conflicto, se pone a 1 el bit de TCM correspondiente al work-item que ha detectado el conflicto (este mecanismo se conoce como *requester loses*). Si, al finalizar la transacción, TCM está a 0, significa que no ha habido ningún conflicto y, por tanto, la transacción se da por correcta y finalizada. Un 1 en alguno de los bits indica un conflicto. En este caso, la instrucción TX_Commit copia TCM en EXEC y provoca un salto a la instrucción TX_Begin. De esta forma, se reinicia la transacción sólo para aquellos work-items que presentaron conflictos. Este mecanismo no garantiza progreso de la transacción: es posible que la instrucción TX_Commit deba reiniciar la transacción varias veces y se encuentre ante un bucle de reintentos infinitos debido a un conflicto en los accesos por parte de 2 o más work-items. Esto se detecta al comprobar que dos veces consecutivas se ha reiniciado la transacción sin ningún cambio en TCM, indicando que no ha habido progreso desde el último reintento. Una vez detectada esta situación, proponemos una solución que funciona a 2 niveles. La primera medida es activar un mecanismo que hemos llamado *wavefront serialization* (WfS). En este caso, tras detectar la situación de no progreso, se reinicia la transacción ejecutando únicamente 1 work-item de todo el wavefront. Esto se logra cambiando uno de los bits de TCM de 1 a 0 al comienzo del reintento. De esta forma, si el conflicto ocurría entre work-items del mismo wavefront, WfS serializa esta ejecución y el conflicto desaparece. No obstante, es posible que el origen del conflicto sea un wavefront diferente. En este caso, si WfS no logra garantizar el progreso, se activa el modo *Work-group serialization* (WgS). Este modo funciona igual que WfS para el wavefront actual (es decir, permite la ejecución de un único work-item). Sin embargo, las transacciones en el resto de wavefronts del mismo work-group son reiniciadas y retenidas hasta que

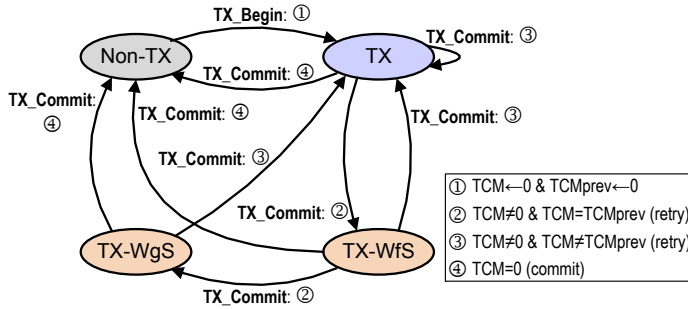


Figure A.5: Diagrama de estado mostrando los modos de ejecución transaccional y de serialización. TX significa transacción, mientras que TCM_{prev} representa el contenido de TCM en el último reintento de ejecutar una transacción.

el work-item ejecutándose termina su transacción. De esta forma se garantiza el progreso, ya que no es posible que un único work-item ejecutándose dentro del work-group detecte conflicto alguno. La figura A.5 muestra un diagrama de transición entre los diferentes modos de serialización, explicando el proceso descrito previamente.

Además de estos modos de serialización, se han propuesto 2 optimizaciones [59]. La primera de ellas es un mecanismo de selección que permite escoger el work-item a ejecutar en WfS. Dependiendo del patrón de acceso a memoria, esta selección puede tener impacto en el rendimiento de la aplicación. La segunda optimización consiste en analizar el patrón de acceso a memoria de las transacciones que han tenido un conflicto. Este análisis permite que durante WfS puedan ejecutarse en paralelo varias transacciones que previamente se ha comprobado que no generan conflicto. Estas optimizaciones permiten mejorar el rendimiento de los mecanismos de serialización en hasta un 30%.

Gestión de versiones. La gestión de versiones se encarga de mantener los valores especulativos de la transacción, y hacerlos definitivos en caso de que la transacción termine con éxito. En el caso de los valores de memoria local, se ha reservado un área llamada *shadow memory* para realizar copias de seguridad de los valores en memoria, mientras que los valores especulativos se almacenan en su ubicación final (esto se conoce como gestión de versiones *eager*). En la figura A.4 se observa el área de shadow memory reservada en cada banco de memoria local. En ella se almacena, por cada palabra de memoria local, una palabra para salvaguardar temporalmente el valor no especulativo y un byte que contiene el identificador del work-item que ha accedido a dicha palabra. Nótese que este

espacio se ha reservado de memoria local existente, por lo que no es necesario añadir nuevos elementos de almacenamiento para este propósito. No obstante, esto limita la cantidad de espacio en memoria local que los programadores tienen disponible. De la misma forma que las variables de memoria local, el estado de los registros debe ser salvaguardado, y restaurado en caso de conflicto. Para ello se propone que los registros se agrupen por pares y que, por cada registro utilizado, exista un *shadow register* para mantener el último valor consistente de dicho registro. Cabe señalar que esta nueva disposición de memoria local y registros únicamente afecta a aquellos programas que utilizan GPU-LocalTM. Los programas que no necesitan de TM para su ejecución no se ven afectados por esta configuración.

Detección de conflictos. La detección de conflictos se encarga de comprobar si dos o más transacciones acceden a la misma posición de memoria y, en ese caso, señalar un conflicto para proceder al reinicio de la transacción. En nuestra propuesta, los conflictos se detectan en el momento de acceso a memoria, y justo antes de servir dicho acceso (detección de conflictos *eager*). En GPU-LocalTM se han propuesto 4 mecanismos distintos de detección de conflicto.

- *DCD: Directory-based Conflict Detection.* Señala un conflicto si el work-item accede a una posición previamente accedida por otro. Esta información se encuentra en *shadow memory* y es establecida cuando un work-item accede a memoria sin conflicto. Nótese que dos accesos de lectura a la misma posición resulta en un (falso) conflicto, ya que no se almacena información suficiente para distinguir entre accesos de lectura y escritura.
- *SMDCD: Shared-Modified Directory-based Conflict Detection.* Este mecanismo extiende a DCD incluyendo bits S y M. El bit M indica si el acceso a memoria es de escritura, y el bit S indica si varias transacciones han leído la posición. Utilizando ambos bit se permite el acceso a la misma palabra de memoria siempre que los accesos sean únicamente de lecturas, proporcionando un método más preciso que DCD.
- *pRWsig: private Read-Write signatures.* Este método extiende a DCD. Además de en *shadow memory*, los accesos a memoria se registran en firmas [10]. El objetivo es utilizar la información almacenada en dichas firmas para acelerar la detección de conflictos, ya que su evaluación es más rápida que el acceso a *shadow memory*. En este caso, cada work-item mantiene una firma por banco de memoria en el que registra sus accesos a memoria, tanto de lectura como de escritura. Antes de acceder a memoria, se evalúan las firmas de todos los work-items en ese mismo

banco de memoria. Un positivo en una asignatura de un work-item distinto al que accede a memoria se trata como un conflicto. Este método es más rápido, pero menos preciso, que DCD y no permite distinguir entre accesos de lectura y escritura.

- *sWOSig: shared Write-Only signatures.* Es un método que extiende a pRWsig (esto es, coexisten pRWsig y sWOSig). En este método se utilizan unas firmas para registrar los accesos de escritura de un wavefront al completo. Se utiliza para diferenciar entre accesos de lectura y escritura durante la detección de conflictos. Un acceso que causa un positivo en pRWsig de un work-item distinto al actual, pero que no causa un positivo en sWOSig, es un acceso del tipo RAR (*read-after-read*) y, por tanto, permitido para la transacción. En este caso, los accesos de escritura con éxito deben registrarse tanto en la pRWsig privada al work-item que accede a memoria como a la sWOSig de su wavefront.

GPU-LocalTM se ha evaluado utilizando el simulador de GPUs Multi2Sim 4.2 simulation framework [57] que incluye un modelo de GPU de la familia Southern Islands de AMD. La tabla A.2 contiene un resumen de las características de las CU de dicha familia y los recursos necesarios para la implementación de GPU-LocalTM. Nótese que la gestión de versiones y detección de conflictos incrementan la latencia de los accesos a memoria local.

Característica	Valor	GPU-LocalTM
Unidades de cómputo (CU)	32	-
Registros vectoriales por CU	65536	2276 (2504) (~3.6%)
Registros escalares por CU	2048	136 (~7%)
Tamaño de memoria local	65536 bytes	37446 bytes (~57%)
Bancos de memoria local	32	-
Latencia de memoria local	2 ciclos	5 ciclos

Table A.2: Características más relevantes de la familia de GPUs Southern Islands de AMD según su implementación en el simulador Multi2Sim 4.2, y los recursos necesarios para implementar GPU-LocalTM.

En nuestra evaluación, utilizando un total de 8 aplicaciones comprobamos que GPU-LocalTM mejora el rendimiento tanto de cerrojos de grano grueso como de implementaciones de grano fino (con cerrojos de grano fino o atómicos) en escenarios de baja contención. Este tipo de escenarios es el más común en GPUs dado que es muy cercano a programas con paralelismo de datos. En escenarios de mayor contención, y dado que GPU-LocalTM es completamente configurable, el compi-

lador o el programador puede decidir qué implementación de los mecanismos de detección de conflicto es el más adecuado o bien realizar una transformación de código e implementar una solución con cerrojos de grano grueso.

A.5 Conclusiones

En esta tesis se ha abordado la aplicación de TM sobre dispositivos heterogéneos desde diferentes puntos de vista.

En el capítulo 3 se analiza el comportamiento de una librería de software TM en una CPU heterogénea. El resultado de dicho análisis se utiliza para diseñar un planificador para procesadores heterogéneos que tenga en cuenta las características de TM. Como resultado, en nuestra evaluación se ha conseguido disminuir el tiempo de cómputo en un 40% y reducir el consumo de energía un 15% respecto a un planificador ávido. Las publicaciones relacionadas con este capítulo son las siguientes:

Energy Efficiency of Software Transactional Memory in a Heterogeneous Architecture. Emilio Villegas, Alejandro Villegas, Angeles Navarro, Rafael Asenjo, Yash Ukidave, and Oscar Plata.

In *8th Workshop on the Theory of Transactional Memory (WTTM 2016 co-located with PODC 2016)*.

Chicago (IL), USA, July 2016

Evaluación del Consumo Energético de la Memoria Transaccional Software en Procesadores Heterogéneos. Emilio Villegas, Alejandro Villegas, Angeles Navarro, Rafael Asenjo and Oscar Plata.

In *XXVII Jornadas de Paralelismo, JP'16*.

Salamanca, Spain, September 2016

Planificación thread-to-cluster de aplicaciones que utilizan memoria transaccional sobre un procesador heterogéneo. Alejandro Villegas, Ernesto Rittwagen, Angeles Navarro, Rafael Asenjo and Oscar Plata.

In *XXVIII Jornadas de Paralelismo, JP'17*.

Málaga, Spain, September 2017

El capítulo 4 presenta APUTM, una implementación TM software para procesadores del tipo APU. APUTM permite la ejecución de transacciones tanto en GPU como en CPU utilizando la misma interfaz de programación, pero proporcionando un mecanismo ligero para la sincronización entre ambos dispositivos y utilizando implementaciones adaptadas a cada uno de ellos. Los resultados obtenidos en este capítulo han sido publicados en:

Towards a Software Transactional Memory for heterogeneous CPU-GPU processors. Alejandro Villegas, Angeles Navarro, Rafael Asenjo and Oscar Plata. In *3rd IEEE International Workshop on Reengineering for Parallelism in Heterogeneous Parallel Platforms (Repara 2017, part of ParCo2017)*. Bologna, Italy, September 2017

Por último, el capítulo 5 presenta un diseño de TM por hardware, enfocado a la memoria local de GPU. Las principales novedades de este diseño son la inclusión de un mecanismo de serialización para garantizar el progreso de las transacciones sin intervención del programador, la propuesta de un sistema configurable que reutiliza los recursos de memoria disponible, y la implementación de varios métodos de detección de conflictos permitiendo la selección del mejor adaptado a la carga de trabajo. Las publicaciones asociadas a este capítulo son:

Hardware support for Local Memory Transactions on GPU Architectures. Alejandro Villegas, Angeles Navarro, Rafael Asenjo, Oscar Plata, Rafael Ubal and David Kaeli. In *10th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2015, part of FCRC 2015)*. Portland (OR), USA. June 2015

Memoria Transaccional Hardware en Memoria Local de GPU. Alejandro Villegas, Angeles Navarro, Rafael Asenjo and Oscar Plata. In *XXVI Jornadas de Paralelismo, JP'15*. Córdoba, Spain, September 2015

Improvements in Hardware Transactional Memory for GPU Architectures. Alejandro Villegas, Rafael Asenjo, Angeles Navarro and Oscar Plata.

In *19th Workshop on Compilers for Parallel Computing, (CPC'16)*.
Valladolid, Spain, July 2016

Hardware support for scratchpad memory transactions on GPU architectures. Alejandro Villegas, Angeles Navarro, Rafael Asenjo, Oscar Plata, Rafael Ubal and David Kaeli.

In *23rd International European Conference on Parallel and Distributed Computing, EuroPar'17*

Santiago de Compostela, Spain, September 2017

Lightweight Hardware Transactional Memory for GPU Scratchpad Memory. Alejandro Villegas, Rafael Asenjo, Angeles Navarro, Oscar Plata, and David Kaeli.

In *IEEE Transactions on Computers no. 99* doi:10.1109/TC.2017.2776908

November 2017

Bibliography

- [1] ARM big.LITTLE technology. <https://developer.arm.com/technologies/big-little>. Accessed: 2017-11-15.
- [2] ODROID — Hardkernel. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127. Accessed: 2016-04-28.
- [3] A. Adir, D. Goodman, et al. Verification of transactional memory in Power 8. In *51st Ann. Design Automation Conf. (DAC'14)*, pages 1–6, 2014.
- [4] AMD. Southern Islands series instruction set architecture, 2012.
- [5] Ciedric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-Andree; Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, February 2011.
- [6] A. Baldassin, J. P. L. de Carvalho, L. A. G. Garcia, and R. Azevedo. Energy-performance tradeoffs in software transactional memory. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 147–154, Oct 2012.
- [7] A. Baldassin, F. Klein, G. Araujo, R. Azevedo, and P. Centoducatte. Characterizing the energy consumption of software transactional memory. *IEEE Computer Architecture Letters*, 8(2):56–59, Feb 2009.
- [8] Michael Bauer, Henry Cook, and Brucek Khailany. CudaDMA: Optimizing gpu memory bandwidth via warp specialization. In *Int'l. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'11)*, pages 12:1–12:11, 2011.
- [9] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd Conference*



- on Computing Frontiers*, CF '06, pages 29–40, New York, NY, USA, 2006. ACM.
- [10] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
- [11] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):40:46–40:58, September 2008.
- [12] Daniel Cederman, Philippas Tsigas, and Muhammad Tayyab Chaudhry. Towards a software transactional memory for graphics processors. In *10th Eurographics Conf. on Parallel Graphics and Visualization (EG PGV'10)*, pages 121–129, 2010.
- [13] Luis Ceze, James Tuck, Calin Cascaval, and Josep Torrellas. Bulk disambiguation of speculative threads in multiprocessors. In *33rd ACM/IEEE Int'l. Symp. on Computer Architecture (ISCA'06)*, pages 227–238, June 2006.
- [14] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE Int'l. Symp. on Workload Characterization (IISWC'09)*, pages 44–54, Oct 2009.
- [15] Shuai Che, J.W. Sheaffer, M. Boyer, L.G. Szafaryn, Liang Wang, and K. Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *IEEE Int'l. Symp. on Workload Characterization (IISWC'10)*, pages 1–11, Dec 2010.
- [16] Quan Chen and Minyi Guo. Adaptive workload-aware task scheduling for single-isa asymmetric multicore architectures. *ACM Trans. Archit. Code Optim.*, 11(1):8:1–8:25, February 2014.
- [17] Sui Chen and Lu Peng. Efficient GPU hardware transactional memory through early conflict resolution. In *22nd Int'l. Symp. on High Performance Computer Architecture (HPCA'16)*, 2016.
- [18] Woojin Choi and J. Draper. Improving utilization of hardware signatures in transactional memory. *IEEE Trans. on Parallel and Distributed Systems*, 24(11):2230–2239, Nov 2013.
- [19] Luke Dalessandro and Michael L. Scott. Strong isolation is a weak idea. In *Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'12)*, 2012.

- [20] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: Streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '10*, pages 67–78, New York, NY, USA, 2010. ACM.
- [21] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Distributed Computing*, volume 4167, pages 194–208. Springer, 2006.
- [22] Nuno Diegues, Paolo Romano, and Luís Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *23rd Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'14)*, pages 3–14, 2014.
- [23] K.R. Duffy, N. O'Connell, and A. Sapozhnikov. Complexity analysis of a decentralised graph colouring algorithm. *Information Processing Letters*, 107(2):60 – 63, 2008.
- [24] Carole Dulong. The IA-64 architecture at work. *Computer*, 31(7):24–32, 1998.
- [25] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *IEEE Trans. on Parallel and Distributed Systems*, 21(12):1793–1807, 2010.
- [26] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'08)*, pages 237–246, 2008.
- [27] Cesare Ferri, Amber Viescas, Tali Moreshet, Iris Bahar, and Maurice Herlihy. Energy implications of transactional memory for embedded architectures. *Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods (EPHAM'08)*, 2008.
- [28] Wilson W. L. Fung and Tor M. Aamodt. Energy efficient GPU transactional memory via space-time optimizations. In *46th Ann. IEEE/ACM Int'l. Symp. on Microarchitecture (MICRO'13)*, pages 408–420, 2013.
- [29] Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt. Hardware transactional memory for GPU architectures. In *44th Ann. IEEE/ACM Int'l. Symp. on Microarchitecture (MICRO'11)*, pages 296–307, 2011.

- [30] E. Gaona, R. Titos-Gil, M. E. Acacio, and J. Fernández. Dynamic serialization: Improving energy consumption in eager-eager hardware transactional memory systems. In *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 221–228, Feb 2012.
- [31] Epifanio Gaona-Ramírez, Rubén Titos-Gil, Juan Fernández, and Manuel E Acacio. Characterizing energy consumption in hardware transactional memory systems. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*, pages 9–16. IEEE, 2010.
- [32] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '08, pages 175–184, New York, NY, USA, 2008. ACM.
- [33] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., 1977.
- [34] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Ed.* Morgan & Claypool Publishers, USA, 2010.
- [35] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural support for lock-free data structures. In *20th Ann. Int'l. Symp. on Computer Architecture (ISCA '93)*, pages 289–300, 1993.
- [36] A. Holeý and A. Zhai. Lightweight software transactions on GPUs. In *43rd Int'l Conf. on Parallel Processing (ICPP'14)*, pages 461–470, 2014.
- [37] C. Jacobi, T. Siegel, and D. Greiner. Transactional memory architecture and implementation for IBM System z. In *45th Ann. IEEE/ACM Int'l. Symp. on Microarchitecture (MICRO'12)*, pages 25–36, 2012.
- [38] Khronos. The OpenCL Specification. Version 2.0.
- [39] F. Klein, A. Baldassin, G. Araujo, P. Centoducatte, and R. Azevedo. On the energy-efficiency of software transactional memory. In *Proceedings of the 22Nd Annual Symposium on Integrated Circuits and System Design: Chip on the Dunes*, SBCCI '09, pages 33:1–33:6, New York, NY, USA, 2009. ACM.
- [40] Ang Li, Gert-Jan van den Braak, Henk Corporaal, and Akash Kumar. Fine-grained synchronizations and dataflow programming on GPUs. In *29th ACM Int'l. Conf. on Supercomputing (ICS'15)*, pages 109–118, 2015.

- [41] Simone Libutti, Giuseppe Massari, and William Fornaciari. Co-scheduling tasks on multi-core heterogeneous systems: An energy-aware perspective. *IET Computers & Digital Techniques*, 10:77–84(7), March 2016.
- [42] L. Lugini, V. Petrucci, and D. Mossé. Online thread assignment for heterogeneous multicore systems. In *2012 41st International Conference on Parallel Processing Workshops*, pages 538–544, Sept 2012.
- [43] Chi Cao Minh, Jaewoong Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IEEE Int'l. Symp. on Workload Characterization (IISWC'08)*, pages 35–46, Sept 2008.
- [44] Tali Moreshet, R Iris Bahar, and Maurice Herlihy. Energy-aware micro-processor synchronization: Transactional memory vs. locks. *Fourth Annual Boston-Area Architecture Workshop*, page 21, 2006.
- [45] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Morph algorithms on GPUs. In *18th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP'13)*, pages 147–156, 2013.
- [46] Angeles Navarro, Antonio Vilches, Francisco Corbera, and Rafael Asenjo. Strategies for maximizing utilization on multi-cpu and multi-gpu heterogeneous architectures. *J. Supercomput.*, 70(2):756–771, November 2014.
- [47] NVIDIA. NVIDIA CUDA Programming Guide.
- [48] M. Popovic, B. Kordic, and I. Basicevic. Transaction scheduling for software transactional memory. In *2017 IEEE 2nd International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, pages 191–195, April 2017.
- [49] Ricardo Quisiant, Eladio Gutierrez, Emilio L. Zapata, and Oscar Plata. Leveraging irrevocability to deal with signature saturation in hardware transactional memory. *The J. of Supercomputing*, 2016.
- [50] Vignesh T. Ravi and Gagan Agrawal. A dynamic scheduling framework for emerging heterogeneous systems. In *Proceedings of the 2011 18th International Conference on High Performance Computing, HIPC '11*, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society.
- [51] Wenjia Ruan, Yujie Liu, and Michael Spear. STAMP Need Not Be Considered Harmful. pages 1–7, 2014.
- [52] Wenjia Ruan, Yujie Liu, and Michael Spear. Transactional read-modify-write without aborts. *ACM Trans. Archit. Code Optim.*, 11(4):63:1–63:24, January 2015.

- [53] D. Sainz and H. Attiya. RelSTM: A proactive transactional memory scheduler. In *8th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2013)*, 2013.
- [54] S. Sanyal, S. Roy, A. Cristal, O. S. Unsal, and M. Valero. Clock gate on abort: Towards energy-efficient hardware transactional memory. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8, May 2009.
- [55] Qi Shen, Craig Sharp, William Blewitt, Gary Ushaw, and Graham Morgan. PR-STM: Priority rule based software transactions for the GPU. In *21st Int'l. Conf. on Parallel and Distributed Computing (EuroPar'2015)*, pages 361–372, 2015.
- [56] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, page 325–335. ACM, 2006.
- [57] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2Sim: A simulation framework for CPU-GPU computing. In *21st Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'12)*, 2012.
- [58] A. Villegas, R. Asenjo, A. Navarro, O. Plata, and D. R. Kaeli. Lightweight hardware transactional memory for gpu scratchpad memory. *IEEE Transactions on Computers*, PP(99):1–1, 2017.
- [59] Alejandro Villegas, Rafael Asenjo, Angeles Navarro, and Oscar Plata. Improvements in hardware transactional memory for gpu architectures. In *19th Workshop on Compilers for Parallel Computing, CPC '16*, 2016.
- [60] Alejandro Villegas, Rafael Asenjo, Angeles Navarro, Oscar Plata, Rafael Ubal, and David Kaeli. *Hardware Support for Scratchpad Memory Transactions on GPU Architectures*, pages 273–286. Springer International Publishing, Cham, 2017.
- [61] Alejandro Villegas, Angeles Navarro, Rafael Asenjo, and Oscar Plata. Memoria transaccional hardware en memoria local de gpu. In *Proceedings of the XXVI Edición de las Jornadas de Paralelismo, CPC '15*, 2015.
- [62] Alejandro Villegas, Angeles Navarro, Rafael Asenjo, and Oscar Plata. Towards a software transactional memory for heterogeneous cpu-gpu processors. In *3rd IEEE International Workshop on Reengineering for Parallelism*

- in Heterogeneous Parallel Platforms (Repara 2017, part of ParCo2017)*, 2017.
- [63] Alejandro Villegas, Angeles Navarro, Rafael Asenjo, Oscar Plata, Rafael Ubal, and David Kaeli. Hardware support for local memory transactions on gpu architectures. In *10th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2015, part of FCRC 2015)*, 2015.
- [64] Alejandro Villegas, Ernesto Rittwagen, Angeles Navarro, Rafael Asenjo, and Oscar Plata. Planificación thread-to-cluster de aplicaciones que utilizan memoria transaccional sobre un procesador heterogéneo. In *Proceedings of the XXVIII Edición de las Jornadas de Paralelismo*, JP '17, 2017.
- [65] Emilio Villegas, Alejandro Villegas, Angeles Navarro, Rafael Asenjo, and Oscar Plata. Evaluación del consumo energético de la memoria transaccional software en procesadores heterogéneos. In *Proceedings of the XXVII Edición de las Jornadas de Paralelismo*, JP '16, 2016.
- [66] Emilio Villegas, Alejandro Villegas, Angeles Navarro, Rafael Asenjo, Yash Ukidave, and Oscar Plata. Energy efficiency of software transactional memory in a heterogeneous architecture. In *8th Workshop on the Theory of Transactional Memory (WTTM 2016 co-located with PODC 2016)*, 2016.
- [67] A. Wang, M. Gaudet, P. Wu, J.N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of BlueGene/Q hardware support for transactional memories. In *21st Int'l. Conf. on Parallel Architectures and Compilation Techniques (PACT'12)*, pages 127–136, 2012.
- [68] Anton Weber, Kim-Anh Tran, Stefanos Kaxiras, and Alexandra Jimborean. Decoupled access-execute on ARM big.LITTLE. In *5th Workshop on High Performance Energy Efficient Embedded Systems (HIP3ES'17)*, 2017.
- [69] Yunlong Xu, Lan Gao, Rui Wang, Zhongzhi Luan, Weiguo Wu, and Depei Qian. Lock-based synchronization for GPU architectures. In *ACM Int'l. Conf. on Computing Frontiers (CF'16)*, pages 205–213, 2016.
- [70] Yunlong Xu, Rui Wang, Nilanjan Goswami, Tao Li, Lan Gao, and Depei Qian. Software transactional memory for GPU architectures. In *Ann. IEEE/ACM Int'l. Symp. on Code Generation and Optimization (CGO'14)*, pages 1:1–1:10, 2014.
- [71] Ayse Yilmazer and David Kaeli. HQL: A scalable synchronization mechanism for GPUs. In *27th IEEE Int'l. Symp. on Parallel and Distributed Processing (IPDPS'13)*, 2013.

-
- [72] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *Int'l. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'13)*, pages 19:1–19:11, 2013.
- [73] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *20th Ann. Symp. on Parallelism in Algorithms and Architectures (SPAA'08)*, pages 169–178, 2008.