MASTER'S THESIS 2021

Integration of a Cycle-approximate Model Into a Cycle-accurate Environment

Andreas Hansson

ISSN 1650-2884
LU-CS-EX: 2021-08

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2021-08

Integration of a Cycle-approximate Model Into a Cycle-accurate Environment

Integration av en cykel-ungefärlig modell i en cykel-exakt miljö

Andreas Hansson

Integration of a Cycle-approximate Model Into a Cycle-accurate Environment

Andreas Hansson
tpi15aha@student.lu.se

April 26, 2021

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Jörn Janneck (LTH), Jorn.Janneck@cs.lth.se
Assistant supervisor: Reimar Döffinger (Arm), Reimar.Doffinger@arm.com
Examiner: Flavius Gruian (LTH), flavius.gruian@cs.lth.se

Abstract

Software models can simulate hardware components to varying degrees of accuracy. On the extreme ends, there are purely functional models which have no concept of time and execute requests right away, and cycle-accurate models which capture all the implementation details and clock the time they take. In this thesis we study a model of an Arm Mali GPU which is a cycle-approximate model, meaning it is somewhere between the two. The model has some timekeeping abilities, but still accesses data in a functional untimed way. This creates compatibility issues if we want to connect it with cycle-accurate models which expect timed memory requests. To bridge this gap, we implement a scheme to convert untimed requests into timed when needed by employing a scheme of coroutines to switch context until timing requirements can be satisfied. We find that our scheme enables the model to work correctly in an environment that demands timed requests, but at the cost of accuracy loss in the model's performance estimations.

Keywords: Cycle-approximate, Cycle-accurate, TLM, Coroutine, GPU, SystemC, C++, Modelling, AXI

Contents

List of Figures

Chapter 1
Introduction

As part of its operation, a graphics processing unit (*GPU*) often needs to read from and write to a system level memory that is located external to the GPU itself. As with all operations taking place inside of a computer, this transaction of data will take a certain amount of time to complete. When designing a simulator of a hardware component such as a GPU, physical properties become implementation choices, and depending on the simulator needs this allows for the handling of memory to be done in many different ways. A *functional* approach would be to disregard the mechanisms of the transactions and act on the memory directly. However, if we also care about getting performance data out of the simulation we would need to simulate the transaction mechanisms as well.

In this project we will look at a model simulating an Arm Mali GPU. The model mixes functional- and performance aspects, by letting the model directly act on the memory while simulating transaction delays separately (in the form of a request-response mechanism) but without functionality (meaning that the simulated transactions do not move any data). This creates challenges when integrating the model into an environment where the model no longer has functional access to the emulated memory. Not only do we have to tie the data access to our simulated performance transactions, but we also need to rely on the accuracy of an implementation that was done with approximativeness in mind.

1.1 Problem Statement

We have a GPU model that is implemented on the basis of having direct functional access to system memory. In order to get performance estimates the model also has a system in place to simulate the delays that memory transactions would have induced. However, this system relies on manual implementations of individual requests, and is incomplete. In a full system simulation it is assumed that the model no longer has direct functional access. Instead the data connection is tied to the manually implemented requests which were previously used for performance estimates. The consequence of this is that the data correctness of the model

now relies on a request system that is incomplete.

This restriction is illustrated in Figure 1.1. To the left we have the model as given to us. To the right a setup with an external memory is shown. In the beginning of a simulation run, the memory that was previously loaded into the model will now be loaded into the external memory. Our goal is to make the result of any given test vector run in the setup to the left to be identical to the result when run in the setup to the right.

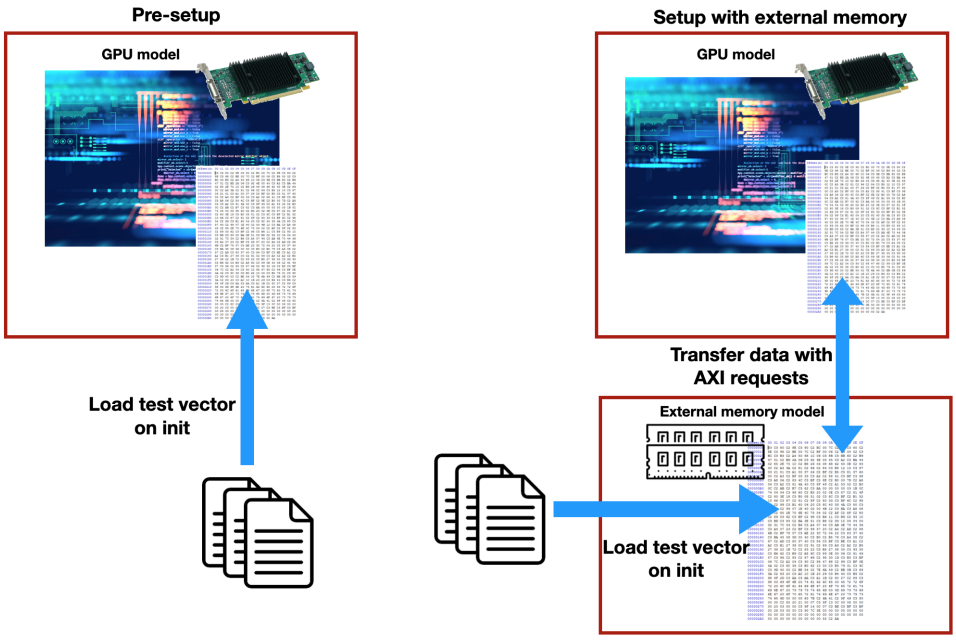


Figure 1.1: Left: The setup of the model.
Right: The setup this project aims to enable.

1.1.1 Research Questions

With the problem above in mind, our thesis will investigate and answer the following questions:

- **How can incorrect data being consumed as a result of missing or mismanaged memory requests be identified?**
- **What is the current state of the model's management of memory requests?**
- **Is it possible to set up a way to automatically handle all the missing/mismanaged requests?**
- **How would such a handling impact the performance estimates given by the model?**

1.2 Overview of the Thesis

In this thesis, we will start out by providing background on a couple of technical concepts that will be used throughout the project. These include a description about model types,

the SystemC simulation environment, an introduction to coroutines, and the AXI protocol which is used for communication between the GPU and memory systems. Following this, we describe the properties of the specific Mali GPU model that we are looking at. We define the target behaviour we want the model to have in Section 3.2. We then spend the rest of Chapter 3 to reflect on different approaches for achieving this target.

An approach that answers the first and third research question is set up in Chapter 4.The correctness of this implementation is verified in Chapter 5, along with an evaluation of the second and fourth research questions. Finally, the outcomes of these evaluations, along with some discussion on alternative approaches, is done in Chapter 6.

Chapter 2
Theoretical Background and Related Work

In this chapter we will briefly present some technical concepts and definitions which will be used throughout investigation. The topics covered may seem unrelated to each other, but we will find that they all end up central to the project.

2.1 Computer Components

The model we are working with is a model of a *graphical computational unit* (GPU). In many ways it could be likened to a highly specialised *central processing unit* (CPU) - the component responsible for computer computations in general. Just like a CPU, a GPU operates in terms of *clock cycles*, which is the smallest unit of time in which the component performs actions. It is these cycles which are referred to when talking of the more commonly used term *clock frequency*, where for example a component operating on **1 GHz** means that it performs 1 billion clock cycles per second.

The GPU is an advanced component containing multiple sub-components performing different tasks related to graphical computation. These computations use memory, and to speed up this access the GPU contains memories called *caches* which tend to be fast but small. Since these memories cannot fit everything that the GPU might need, and because the GPU might wish to share memory outputs with other components, there is a need to frequently read and write to a larger memory located outside of the GPU. This memory is usually of the type *random access memory* (RAM) and typically has a size in the range of gigabytes. The *random*-property in its name refers to the fact that the access time is fairly uniform no matter which part of the memory one tries to access - meaning that reading two randomly located parts should take roughly the same time. This is a contrast to a hard drive where the physical properties of its mechanical components makes it so that moving to different parts of the data might take different amounts of time.

Cache memories are hierarchical, with the terms **L1**, **L2**, etc denoting their proximity to a component, and the system-level RAM could be seen as highest up in this chain of work-

memory. As a rule of thumb, a lower number memory is smaller but faster. Since a component will read data from the closest memory having the desired contents, the question of coherency between memory sources becomes important (since a different component might have modified the data on a higher level). To this end, our model performs two key actions :

- *Flushing*, meaning that when a cache has a certain amount of data written to it that has not been passed up, it performs a write-back of this.
- *Invalidation*, meaning that data is cleared/removed from a cache.

2.2 Modelling Concepts

By *model*, we refer a piece of software that simulates the behaviour of a hardware component. Such models can vary widely in both accuracy and use cases. The accuracy to which they represent the hardware can be down to the wire, or they might just give the same output but arrive to it in a completely different way.

2.2.1 Transaction Level Modelling

Transaction level modelling (*TLM*) is a type of modelling where the details of communication between components in a model are separated from the components themselves. Instead, there exists abstractions in form of channels which interface with the model's computational components and thus act as a separation between computational and communicational aspects[1][2]. The level of precision of both the communicational and computational aspects of a model can vary a lot. *Cycle-accurate* as well as *untimed* (also known as *"functional"*) models are at opposite ends of the precision spectras. *Cycle-approximate* however covers a wide range between these which is often not so well-defined. [3] In an attempt to define these properties in terms of computation and communication, Cai and Gajski[2] introduce the terms *bus-arbitration model* and *bus-functional model* as shown in Figure 2.1. The process of going from approximate to accurate in either aspect is referred to as *refinement*.

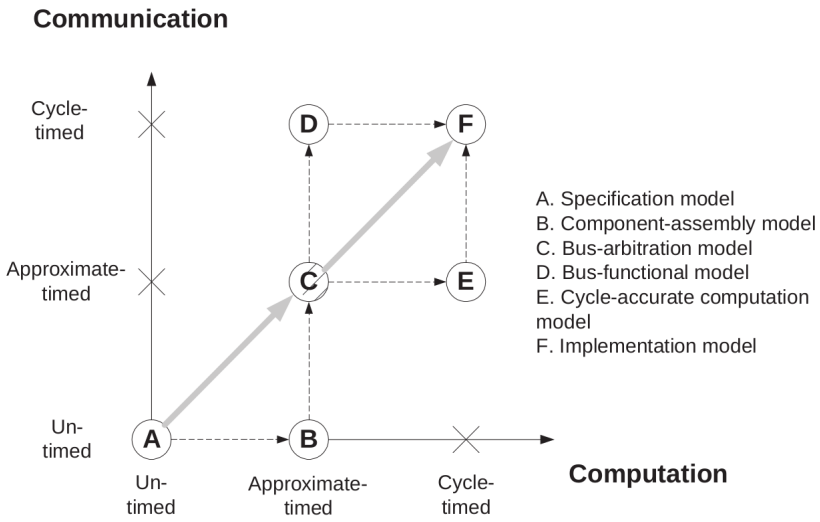


Figure 2.1: TLM classifications according to Cai and Gajski[2]

The given classifications are as follow:

- **Bus-arbitration model**: Channels between computational components have some representation, but the bus protocols can be simplified as blocking and non-blocking I/O and lack all semblance of cycle-accuracy. The transactions on the channel have time estimations specified by a wait statement.
- **Bus-functional model**: Contains time/cycle accurate communication and approximate-timed computation.

Our model would best be characterised by something in between **C** and **D**, likely closer to **D**. It does not represent communications down to the wire and the channels very much consist of higher level abstractions, but it largely adheres to the given communication protocol including handshake mechanisms and meta-data fields (though notably not the data itself). It should be noted that the cause of our problems is not related to any timing inaccuracies by the communication channels themselves, which do provide a reliable request-response mechanism. Rather, our problems are caused by the endpoints initiating these mechanisms in the wrong stage coupled in the pipeline, or not initiating them at all.

2.2.2 SystemC

SystemC is a C++-based class library and design environment for system-level design[4]. It adds timing, concurrency, and hardware data types to the standard C++ language, and is a popular choice for system level modelling. The GPU model is not written in SystemC, but it might be used in a SystemC-scope. In this project we will make use of a SystemC program that acts as a top-level that loads the model and drives its execution via interaction with the model's interface.

Methods in SystemC can be run as a **SC_METHOD** or as a **SC_THREAD**. The difference between these is that a **SC_METHOD** is started each time it is used whereas a **SC_THREAD** is started only once and can then suspend itself at will using a **wait** [5]. **SC_THREAD** is a

sequentially executing construct working much like a coroutine / user level thread. This means that using a **SC_THREAD** does not create any risk of race conditions, as only one of them can execute at the time.

For both of these method types, what drives execution is one or more *sensitivities* which are registered for each such instance. These sensitivities could be items such as custom events raising notifications, but they could also stem from a clock. The notion of time in SystemC covers clockability and lets one define a clock that can fire at every discrete amount of time elapsed. In our case, the simulation processes bound to the clock are the clocking of the model itself as well as the clocking of the interfaces managing model requests.

2.3 Context Management

As later described in Section 3.4, we will be faced with the need of returning from the middle of functions and then returning to them at a later point. To this end we will make use of a type of control structures called *coroutines*. These are powerful and versatile abstractions with a range of use cases[6]. In this project, we will not make use of the full range of these capabilities, and we will therefore not provide a description of all these either. Below is a summary that relates to the aspects that we will be useful for us, or that are likely to be useful in some extension of this work.

2.3.1 Call Stacks

A *call stack* is a contiguous section of memory set aside for a program[7]. It is made up of entries called *stack frames* (also known as *activation records*) which are structures containing information about function calls, typically local variables, parameters passed, and where it should return. This structure works based on the "last in, first out"-principle, meaning that the currently executing function in a program is on top of the program's call stack, and when it returns it is popped from it.

With that in mind, we can see the stack as a representation of the current chain of events in a program. One of the key parts of threading is to have multiple chains of events going on at the same time, which in turn means having multiple stacks. When using system level threads (also known as *kernel* level threads), the system will change back and forth between these, which is commonly referred to as *context switching*. However, it is not necessary to have system threads in order to have multiple stacks - they can also be managed on application level as part of a *user level thread*. Since these do not need system-level management they also do not need as many unique resources, leading to less overhead and faster context switching[8]. User level threads can take shape in different semantics and are commonly used as part of *coroutines*.

2.3.2 What is a Coroutine?

A *coroutine* is a generalisation of the concept of a subroutine. To understand this relationship, let us first consider the idea of a subroutine. The invocation of a subroutine consists of three basic parts:

- The invocation itself
- A range of actions that make up the subroutine, that get executed from start to end
- Returning the result of the subroutine (possibly void), and moving back the execution to the caller

We take notice of the fact that the middle part has to run to completion before control can be given back to the caller. This is one of the key differences to coroutines, which do not have this requirement. Instead, a coroutine can return control to the caller without having run to completion, and if it is given back control later, it will resume execution from the point it stopped as shown in Figure 2.2. The action of returning control in the middle of execution will henceforth be referred to as *yielding* or *suspending*. The other part which is central to the idea of coroutines is the idea of *cooperative scheduling*, meaning that rather than having a central dispatch for which coroutine should be executing at a given moment, the control is explicitly passed back and forth between the coroutines themselves (meaning the coroutine is *symmetric*), or always back to the caller (meaning the coroutine is *asymmetric*)[6].

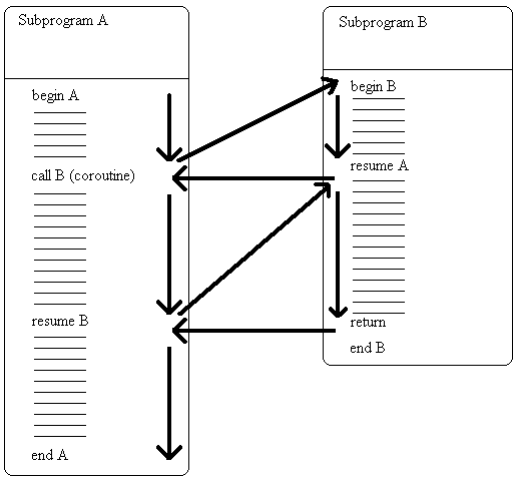


Figure 2.2: Coroutine control passing (Tevfik AKTUĞLU, Wikimedia Commons, `https://commons.wikimedia.org/wiki/File:Coroutine.png`)

2.3.3 Difference Between Stackful and Stackless Coroutines

One of the main distinctions between coroutines types is whether they are *stackful* or *stackless*. These phrases are easy to get confused, because *stackful* is in reference to the coroutine having its **own stack**. Whether this stack is actually stack- or heap-allocated with regard to the system is inconsequential and a question of optimisation.

The Boost library's documentation of coroutines notes that in contrast to stackless coroutines, stackful coroutines allow invoking the suspend operation out of arbitrary sub-stackframes, enabling escape-and-reenter recursive operations[9]. This means that if a coroutine **A** calls a subroutine **B**, then it is possible to suspend the coroutine **A** from inside **B**, because the

frame of **B** is located on this separate stack. When the coroutine is then resumed, the context including local variables is then resumed from where it was suspended inside of **B**. As a contrast, stackless coroutines can not suspend from any call depth deeper than the coroutine function itself.

2.3.4 Difference Between Coroutines and Fibers

Coroutines and *fibers* have a lot in common and might even share implementation-level details. They are both user level threads (meaning they have no kernel interaction in their switching, and run on a single kernel level thread), and they can both be used for asynchronous operations by yielding (or in the case of fibers, "blocking"). The principal difference is that when a coroutine yields, it either leaves the control to the caller (in the case of *asymmetric* coroutines), or to another coroutine (in the case of *symmetric* coroutines). Fibers on the other hand have a *scheduler* which determines the next execution context[10].

2.3.5 Methods For Context Preservation in C++

Let us now consider the different ways we can go about to freeze a process from a specific line, do something else from the same top-level context, and then resume where it was left off. We want this done with as little overhead and as few dependencies as possible. We offer no precise metrics for measuring these properties, but instead try to reason with how different specifications align with the context of our model. After doing so we end up selecting two of them, and will make two similar but separate versions of the implementation in Chapter 4.

Until C++20 there was no native support for coroutines in the C++-standard. For this reason, a number of third party libraries provide different variants of coroutine functionality[11]. Perhaps the most widely used of these is Boost's implementations of the library which provide stackful, first-class coroutines (meaning they have an passable handle) building on top of their library **Boost.Context**. Additionally, the library **Boost.Fiber** also builds on top of Boost.Context, making the underlying functionality of their coroutine- and fiber-libraries very similar. In addition to this, there exists a number of lesser known libraries proving different variations of coroutine functionality.

Additionally, there exist lower-level constructs which could be used to create custom coroutines and/or context management facilities. For POSIX-systems these functionalities include **setjmp**, which saves various information about the calling environment and then performs a non-local jump restoring this context with **longjmp**, as well as (officially deprecated) **makecontext** and **setcontext** which offer user-level context switching [12] [13]. In addition to this, there is a Windows-equivalent mechanism called **Win32 Fiber**.

Threads

One possibility would be to use separate threads, as each thread has its own call stack. A regular system-level thread could be created to start from the desired point of invocation, and then be used together with locks to make sure that only one thread execute at a given time (ensuring a sequential flow of operations). Should the C++ **std::thread** library be used for this, the library type **condition_variable** could be used along with a **mutex** to facilitate communication and synchronisation between the threads[14].

Threads however introduce overheads in other ways. A context switch might induce a delay in the range of multiple microseconds[15]. Running one work item like this each computation cycle in the model, that would result in two switches per cycle. Given that our model executes between 5000 and 10000 computation cycles per real-world second on our host machine, this means inducing up to 20000 switches per second of computation, which might not be negligible. Should some additional modification require finer granularity than our current scheme (such as running each subcomponent in its own context rather than having a single context from the top-level runner), then the number would be even larger. In addition to this, having a system-level thread creates unpredictability if the user expects the program to execute in a certain way, for example if some tests rely on a fixed number of threads interacting for which these might interfere. A few strengths and weaknesses of this approach are listed in Table 2.1.

Strengths	Weaknesses
Widely known	Cost of context switching
Quick to implement	Synchronisation mismanagement risk
-	Possible interference with other behaviours relying on specific thread setups

Table 2.1: Strengths and weaknesses of using system threads

C++20 coroutines

A newly introduced feature in the *C++20* revision of the C++-standard is native coroutine functionality as a language construct. It added three new keywords to the language; **co_await**, **co_yield**, and **co_return**. This new functionality integrates well with a few established standard library features often associated with *std::thread*-functionality, such as **std::future** which is a type that can hold a *promise*, which could be likened to a placeholder of sorts for the result of an asynchronous operation. While it would be welcome to not have any external dependencies, it does require C++20 which is not yet widely adopted, and incompatible with the default build of our model which is compiled with C++11. Most importantly of all, this implementation is *stackless*. This effectively means that in order to suspend from an arbitrary point in a nested chain of calls, every routine in this chain would need to be a coroutine of its own. Since we are dealing with hundreds of different routines in the potential call paths from which we might need to yield, this would require a huge reworking of definitions all across the model - in addition to an unknown computational overhead induced by running every invocation this way. For this reason we will not consider this a viable candidate. A few strengths and weaknesses of this approach are listed in Table 2.2.

Strengths	Weaknesses
Native language construct	Stackless, needs to be chained to suspend nested calls
Future proofing	C++20 only
-	Requires large code overhaul

Table 2.2: Strengths and weaknesses of using C++20 coroutines

Boost coroutines

The **Boost** library[9] offers two implementations for coroutine functionality - **Boost.Coroutine**, which is now deprecated, and its successor **Boost.Coroutine2**. They are stackful, as well as first-class objects giving an advantage to ease of syntax as the handles can be passed around at will. Additionally, the model already uses other Boost libraries reducing the expense of introducing another third party library. It is computationally fast, with the official documentation giving a reference figure of 26 ns for a context switch[9]. A few strengths and weaknesses of this approach are listed in Table 2.3.

Strengths	Weaknesses
Stackful	-
Well-documented	-
Model already uses Boost libraries	-

Table 2.3: Strengths and weaknesses of using Boost.Coroutine2

Note: For library versions up until 1.60, Boost.Coroutine2 required C++14. In subsequent versions, this requirement is relaxed to C++11. At the time of running tests in our experimental section we had missed this detail and as a result excluded the use of these when doing runs in an environment which did not support C++14. Changing the library version to a more recent one than 1.60 would have solved our presumed incompatibility issues.

Fibers

Returning once again to the Boost library, it provides a **Fiber** library that builds on top of the same implementation of Boost.Context which provides the underlying mechanics for stack management and context switching. Comparing this to Boost.Coroutine2 it comes down to a matter of syntactic preference. The Fiber requires an explicit scheduler which could potentially be implemented to always resume the caller in order to achieve the same as an asymmetric coroutine.

SystemC SC_THREAD

As touched upon in Section 2.2.2, SystemC provides a coroutine-like functionality of its own by allowing user-level processes suspend to the caller and then resume at a later point. Since we call the entire model as a single **SC_METHOD**, we could swap this out for a **SC_THREAD** to allow for the **wait**-functionality to be used. Since this would suspend the *entire* GPU model, including the memory channels, the state checking that pertains to the suspension state would have to be moved to the SystemC environment side. This creates a dependence on SystemC-specific facilities, but would be very convenient when these are available. A few strengths and weaknesses of this approach are listed in Table 2.4.

Strengths	Weaknesses
Easy to implement and manage	Creates SystemC dependency
-	Requires that the whole model process is suspended as an atomic

Table 2.4: Strengths and weaknesses of using SC_THREAD

Custom solutions

As mentioned in Section 2.3.5, low-level facilities may be used to craft a custom equivalent to the aforementioned alternatives. While this might result in the solution with the least overhead, it would require a larger initial effort to set up as compared to alternatives. Additionally, the most interesting candidate - the **ucontext**-functions **makecontext**, **setcontext**, and **getcontext** were officially deprecated from the POSIX standard in IEEE Std 1003.1-2001/Cor 2-2004 [16]. For these reasons we decided not to pursue these options for the scope of this project. A few strengths and weaknesses of these approaches are listed in Table 2.5.

Strengths	Weaknesses
Potentially optimal performance	Difficult implementation
Adds no extra dependencies	Partially deprecated
-	OS-specific

Table 2.5: Strengths and weaknesses of using custom solutions

Conclusion

The two solutions provide our necessary functionality in terms of quick and simple context switching are **Boost.Coroutines2** and **SC_THREAD / sc_cor**. The noted disadvantage of the latter in that state management and request handling would have to be moved to a SystemC externality could actually be an advantage in the testing phase, as the lightweight and simple SystemC program we are dealing with allows for quicker tests and changes when looking at request-specific functionalities. For this reason we will start out with **SC_THREAD** in the testing phase, and then move onto a version based on the Boost library for wider coverage once all the other parts are working.

2.4 AXI Protocol

The AMBA AXI protocol is a communication protocol for memory transfers[17]. It is a quite comprehensive protocol covering a lot of functionalities and conditions, but only a few are necessary to understand for the scope of what is being investigated in this paper.

The protocol has five channels - *write address*, *write data*, *write response*, *read address*, and *read data* (includes the control data for the response). These are all uni-directional, as shown in Figure 2.3 and Figure 2.4. In addition to addresses and data content, these contain a few auxiliary fields, the most important being the handshake signals **VALID** and **READY** which work in such a way that the *source* generates the **VALID** signal to indicate when the address, data, or control information is available. The *destination* generates the **READY** signal to indicate that it can accept the information. Transfer occurs only when *both* the **VALID** and **READY** signals are HIGH[17]. In addition to this, each memory transaction also has an ID.

Reads are always covering all bytes in a range. Writes however can be *strobed*, where strobe is a field on write data that specifies which bytes contain valid information. The strobe works similarly to a mask, and contains one bit of information per byte transferred. If for example the data on the bus is **[0xAA 0xBB 0xCC]** and the corresponding strobe is **[0 1 0]** that would mean that only **[0xBB]** is valid information.

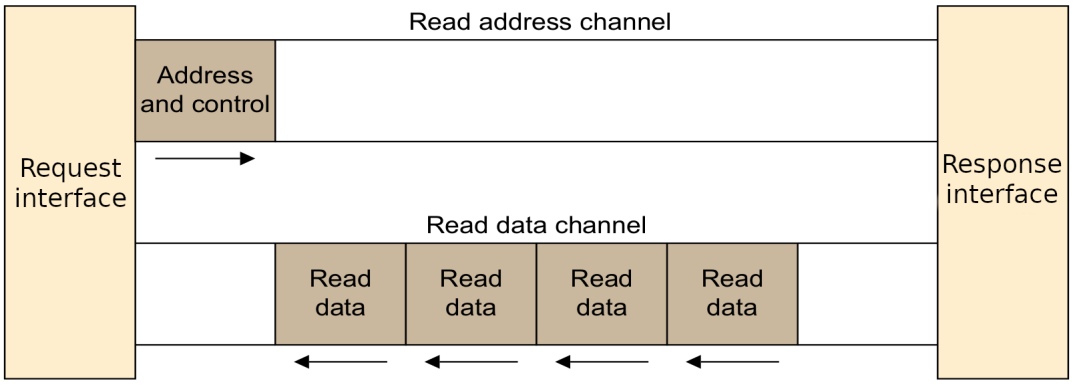


Figure 2.3: AXI Read Channels[17]

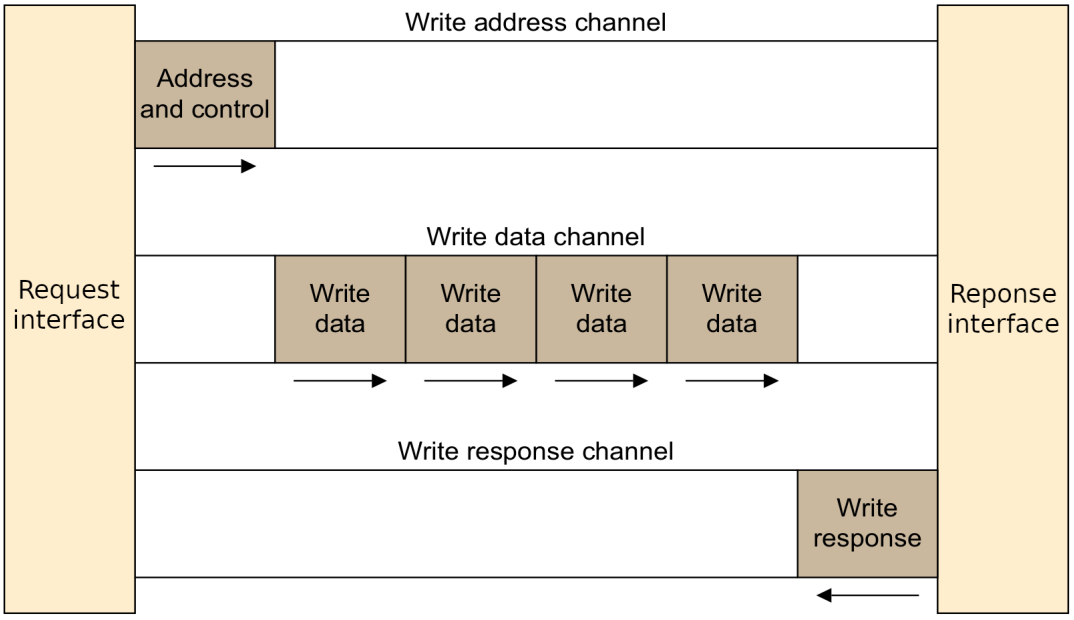


Figure 2.4: AXI Write Channels[17]

In the model these are implemented in such a way that the signals in *write address* and *write data* are conflated into one object, having the data-field of the *write data*-channel as optional (the alternative being making the functional write separately). For reads, the data-field is always omitted in the *read data*-channel (leaving only meta-data fields such as valid- and ready-indications) and the response is largely treated as an acknowledgement as well as simulation of pipeline delays, while the actual memory access uses a different method of direct access (*usually* following this response).

2.5 Related Investigations

In a paper titled "*Integrating GPGPU computations with CPU coroutines in C++*"[18], Pavel A. Lebedev investigates possibilities of using coroutine semantics to facilitate asynchronous I/O

operations in C++, something he describes as a "recently rediscovered solution to elegant asynchronous processing", which in this case involves calling GPU operations from a CPU which are blocking but do not consume any resources in the CPU itself. In this investigation the author reflects over the fact that native language facilities are lacking and makes use of Boost.Context in combination with an *asio* networking library derivative [19] to demonstrate a use case, motivating the library choices as "readily available, not depending on any particular compiler or operating system, and representing the least efficient library-based approach" (with respect to getting a lower bound on performance numbers). These demonstrations consist of a few code examples showing that the syntax modifications needed when making I/O operations asynchronous can be quite small, as well the run-time performance impact being relatively small.

Since these use cases share similarities with our need of control flow management, it gives confidence in using coroutines as an abstraction for the handling of blocking I/O in C++. Especially the following description from this paper sums up a characteristic that we will benefit a lot from in our own implementation: *"Traditional asynchronous code in C++ contains lots of relatively small functions that specify each other as a point at which the algorithm should continue once an invoked asynchronous call completes. This quickly complicates program structure in comparison with blocking synchronous code that doesn't need to split algorithms with multiple asynchronous calls into pieces. Coroutines can be used as the primary means of reverting tangled and piecewise asynchronous code back to serial and readable form."*

An interesting industrial case study exploring parallel simulation of loosely timed SystemC/TLM programs has been done by Becker, Moy and Cornet[20]. Though their investigation is directed towards parallelisation of multiple SystemC processes, it does bring up the role of concurrency in a SystemC context and the limitations caused by loose timing. They conclude that these limitations are quite severe, making is very hard to implement parallelisation schemes for SystemC models. A related work Samuel Jones titled "Optimistic Parallelisation of SystemC"[21] goes further on this by exploring ways to use *temporal decoupling*, which is the idea of letting different parts of a simulation be in different places of the virtual time. In this work the synchronisation mechanisms are between threads communicating over SystemC directly. There are two key differences between our investigation and this - first off, Jones' investigation focuses on improving **host** speed, while we try to find ways to decrease our impact on **virtual** time. Secondly, since we are not using SystemC for the communication between different parts of our model (we are just using it to bridge the model with other external components) we cannot use the suggested SystemC-native synchronisation constructs. The core concepts of time decoupling stay true though, and we will find that our ideas for a speed-up heuristic is a type of time decoupling. In addition to this, their work also touched on the difficulties of using roll-back mechanisms in SystemC, which will be a limitation for us in our heuristic exploration.

Chapter 3
Challenges and Objectives

In order to understand the nature and constraints of the problem, we will give a brief explanation on how a few aspects of the GPU model are set up. This understanding will help motivate design choices we make for our approaches. We will then go on to describe how we aim to make a modified version that satisfies a couple of added constraints. We will also outline some limitations that affect our possibilities to do so.

3.1 How the GPU Model Works

The GPU model replays known scenarios by taking a *test vector* as input. A test vector is loaded from file and contains previously captured system memory along with a list of instructions to drive the execution. The replay is managed by a small program which we will refer to as the *replay driver*. This driver interfaces with the GPU model by reading and writing memory and registers as per the instructions of the test vector. The memory in the test vector represents content that would have been present in a system-level memory, i.e. external to the GPU. However, we do not simulate this memory component separately. Instead, the replay driver will load the full contents of the file directly into the GPU model immediately when the replay starts. Inside the GPU model the memory will then be kept in a structure that is functionally accessible throughout the model. At the end of the simulation, an output will be written to disk containing the resulting frame buffer along with a range of performance estimates. The expected output is known for a replay, so the output of a given simulation run can be compared to this known result to verify the functional correctness that is expected of the model.

The key performance estimates of the model are the clock cycles that have elapsed in the model, and the memory bandwidth consumed. These both have an accuracy target of **5**% as compared to the numbers of the real GPU. In order to stay so close to the numbers of the real GPU the model has a block-based design resembling it. Each clock cycle a top level function will invoke these blocks to perform one cycle worth of computation, pass communication,

and update performance counters. The version of the model we work on in this thesis is implemented as single-threaded, although a multi-threaded version also exists.

Although the model performs the data access itself functionally and untimed, meaning an access does not affect the model's notion of time, there is a system in place to simulate this delay separately. The way this works is that in the places where memory accesses are taking place, AXI requests are manually issued. Once a response has been attained for these, the model goes ahead with the untimed functional read. This is not always the case though, and in some places functional memory access takes place without a corresponding AXI response.

In summary, the properties of the model are the following:

- Written in C++, using the C++11 standard
- Functionally accurate
- Deterministic
- Performance approximate with a target of being within **5%** from the real GPU in terms of cycle count and memory bandwidth
- Block-based design with clockable blocks
- By default running self-contained without being connected to any other system components
- Functional data access
- Simulated AXI requests which are timed. Implemented in most places, but not all.
- Has an interface exposing some functionality, such as passing memory performance requests and clocking the model. This opens up for external components in a shared setup driving its execution.

The general flow of the model is shown in Figure 3.1.

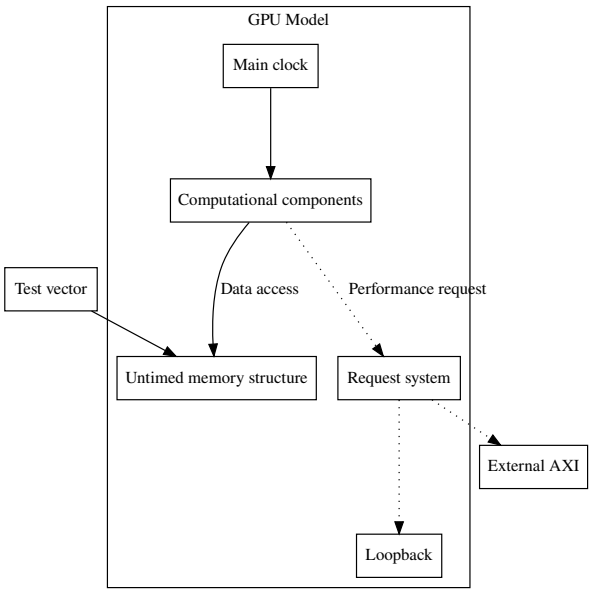


Figure 3.1: High-level layout of model

3.1.1 Overview of the Memory System

As shown in Figure 3.4a, the model contains a memory structure that holds the entire memory contained in the test vector, representing what is supposed to be system level memory. Each cell of this memory is made of a custom type that can hold either an 8-bit unsigned integer (a regular byte), or a value representing an unknown state which will be denoted as *X*. This unknown state is what the memory is initialised to, and then in the beginning of the simulation run the model will load whatever values are in the test vector file into it. Additionally, the type is set up in such a way that if it contains a value it can be implicitly converted to a regular *char*-type, but if it is marked as unknown this attempted conversion will trigger a hard fault, upon which a backtrace prints and the simulation halts.

The memory structure has a number of access methods that allow for reads and writes of this underlying data. What these access methods all have in common is that they act directly on the underlying memory and are instant, as shown in Figure 3.3. It is for this reason - that they are instant - that this memory structure will be referred to as the *untimed memory* throughout the text. Other than this, the different access methods vary a bit in their properties. One key difference is that some take in a physical address as argument, while others use virtual addresses. In this context, physical refers to the address space provided by the memory test vector file, after which the untimed memory is set up. If a virtual address is passed in, the access method must first make a secondary read for a *page table entry*, or PTE for short, to resolve the physical address it should read from or write to. For this reason a functional write can cause a read to happen.

Another difference which will have even greater importance for our problem is the fact that a few of the methods take in a regular *char*-type for the destination buffer, while others take in a reference to a buffer with the extended type described in the previous paragraph.

One last difference is that writes can optionally have a *strobe* attached to them, acting as a mask for the sequence following the given address. Reads have no option for this.

These methods are accessible in almost all parts of the model, and is in fact the only way that data is fetched from the memory system. The model contains a number of cache memories, but the underlying data has no separation – instead of the caches keeping a copy of the data they purportedly hold, they reference line addresses as shown in Figure 3.2. This means that the same functional access methods are used to consume both cached and non-cached memory.

Lastly, as part of the caches' memory coherency, they contain an *invalidation* mechanism as well as a *flushing* mechanism. These are implemented in such a way that they are done globally, meaning that the entire cache will be invalidated or flushed at once.

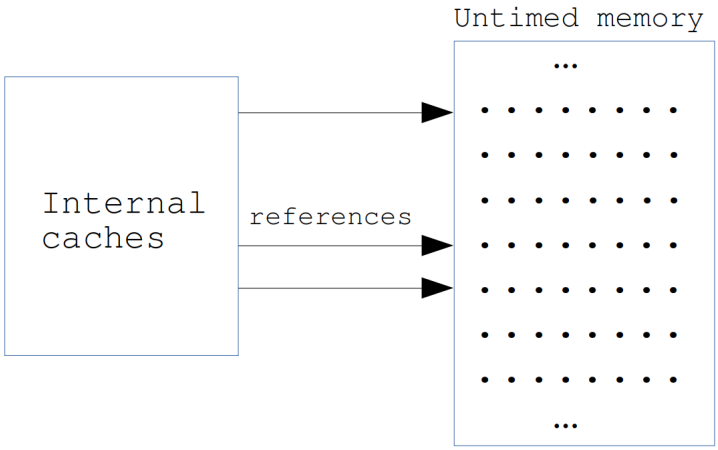
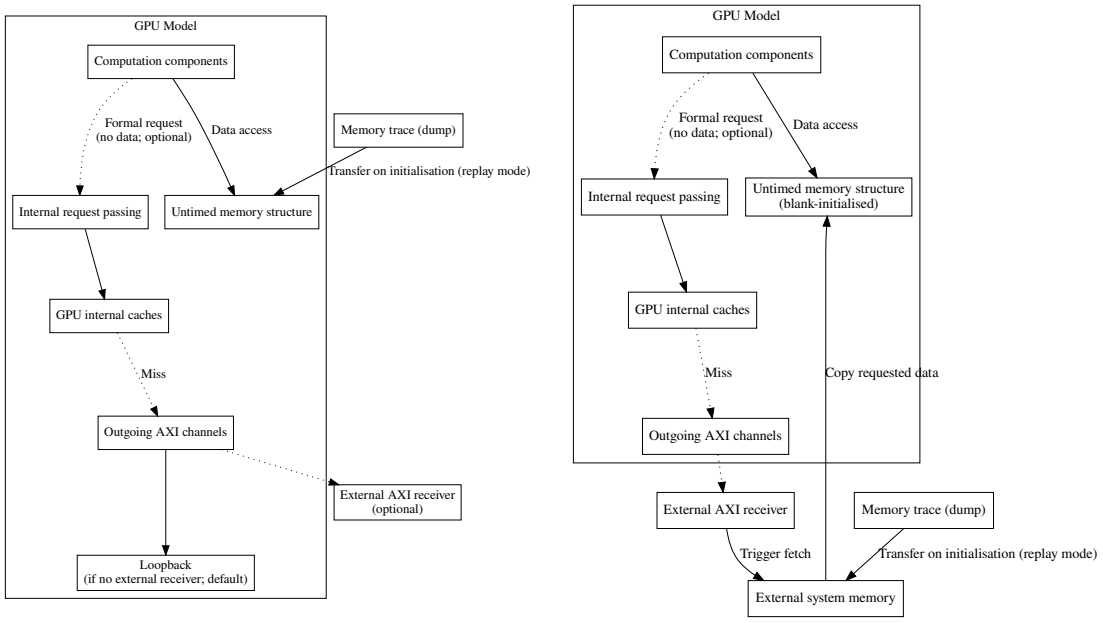


Figure 3.2: Cache-untimed memory relationship

```
...
do something()
functional read()
consume data()
...
```

```
functional read(address, size, destination)
{
    check if valid(address, size)
    copy content into destination
    return
}
```

Figure 3.3: Functional read invocation



(a) Model flow before

(b) Model flow after

Figure 3.4: Flow of memory reads and writes in the model

3.2 Target Behaviour

The golden standard will be the GPU model as it was given to us. Whenever our modified version makes a functional read or write, we wish for the data in question to be identical to what it would have been in the unmodified build. This is the only hard rule we set with respect to the changes we are about to introduce: **any memory consumed by the model must at all times be identical to the memory consumed by that same operation in the unmodified build**. Should the model consume memory with different content than intended, it will be considered undefined behaviour which is likely to lead to unpredictable crashes of the simulation or corruption of the output.

Comparing Figure 3.4b with Figure 3.4a, we have that:

- The model's memory is initialised as unknown instead of pre-loaded with test vector contents.
- The content that would otherwise be pre-loaded is now only loaded upon proper memory requests

This is another way of saying that **the model memory will only be correct if is also makes correct requests for it**.

3.3 Identifying Problems

By *problem*, we refer to a functional read of data where the underlying memory is different than it would have been in our reference build. In this section, we will reason around how these problems can be identified. Additionally, we also look at what happens when functional writes are not guaranteed to reach the target memory. Once we know the factors affecting data states, we will combine these rules into a component which we will refer to as the *coherence checker* throughout the text. The coherence checker will be a just-in-time detection tool returning the state of any given address of the internal memory.

3.3.1 Sensitivity and Specificity

An illegal read might consume memory that by chance is the same as the intended content, but with no knowledge of the context we can make no such guarantees. In order to adhere to the principle we set up regarding never consuming invalid memory, this can only be guaranteed by detecting every single instance prior to it occurring. In other words, any false negative in the detection might be fatal for the output coherency of the process. On the other side, a false positive would not be a fatal problem – it would cause a duplicate read or write of memory. However, while not an immediate problem, this would cause unnecessary performance impact so it is also not desired. Should performance not be a concern at all, a simple but valid solution would be to explicitly generate a new AXI request for every single functional memory access.

3.3.2 Pre-existing Checking Method

A checking tool to identify functional reads with no matching performance request existed prior to this investigation. It was implemented to be fast and does not cover all cases, meaning it has false negatives. It works by creating a second functional memory in the model, meaning we now have two copies of of the untimed memory. Denoting these two memories **A** and **B**, memory **A** is the one receiving the contents from the test vector file(s) upon initialisation, and memory **B** is the one that the model sub-components access when they use their functional read and write methods. **B** will be initialised to be blank, with all the entries being marked *X* rather than having a definite default value such as zero. Should any method attempt a read from a piece of memory marked *X* into a destination that does not support this extended type, the attempted conversion will trigger a run-time crash (however, some destinations *do* support this type). When an AXI read request comes through the simulated memory system it will copy the underlying data for requested memory content from memory **A** to memory **B**. Should any of this memory be marked *X*, then a run-time crash is triggered giving two pieces of information - the fact that at least one problem was present, and a stack trace showing the call path in order to identify the code component responsible for this undesired behaviour. Conversely, if an AXI write request comes in, it will trigger a copy in the other direction. The idea is that this would cause the final output of the simulation to be different than expected should some portion not have been copied at any point, indicating that some incorrect write operation had occurred.

In order to map out existing problems, we first changed the behaviour to not crash to get the relevant information out of it, by reading the data into an intermediate buffer and then verifying that this buffer does not contain any X-values. If it does, copy the relevant portion from **A** to **B** in order to make it as the data already was there, so the model can move on to identify the next problem present. Progressing this way the model would eventually run into some odd crashes though, indicating that at some point it had consumed incorrect data. This helped highlight a few missing parts that would be needed for full coherency checking.

What counts as valid?

Whenever a functional read is done, we need to make sure that there exists initialised data and that this data is not architecturally outdated. Data can exist in the model's memory as a result of either of the following things:

- AXI read response, triggering data copying from an external source to the model's internal memory
- Internal functional write

If either of these have happened for a memory location, that memory is considered **valid**.

We treat anything that is read into the model as *possibly cached*. The invalidation of cache memory in the model is done for the entire memory at once, upon which anything possibly cached is invalidated. When this happens, the set of valid memory is cleared.

Dirty Data

By *dirty data*, we refer to data which has been written to the model's untimed memory, but not to the external system memory. When a functional write is done, it might make sense to expect that the performance request (as seen from the point of the external outgoing AXI bus) follows shortly. What complicates this is that the same functional write methods that are used for the main memory representation are shared with the internal caches, as shown in Figure 3.2. If it were not for this, an idea to catch the cases of missing performance requests/responses following a functional write would be to keep track of whether they appear as expected in the outgoing bus within the immediate time-frame. However, since we do not know if this functional write was intended to go out directly or to be an intermediate cache placement, which might not flush for some time, this cannot be relied on. We have no easy way to iterate over all the caches in the model to verify the intentions, and/or presence of a generated AXI request which might have been held up at some point in the request pipeline. A solution to this would be to keep a record of all the dirty data, wait until the caches flush - at which point performance requests are generated by the caches for all outstanding writes - and for each outgoing request remove them from the dirty set. Anything left on this set at the end of a flush cycle could be assumed to have the matching performance request either missing or unordered. For these a request could be generated for it and appended to the end of the flush cycle as shown in Figure 3.5.

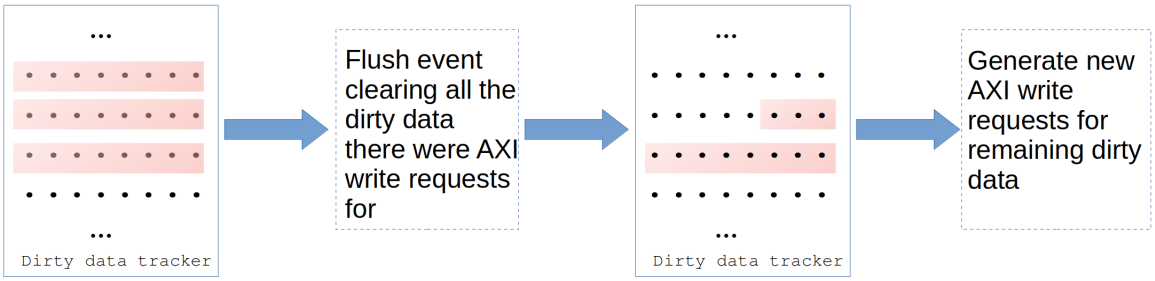


Figure 3.5: Appending outstanding dirty writes at the end of a flush cycle

Data Precedence

Because the model's internal memory might have a more recent copy of some data than the external memory, an AXI read request overlapping with this should not cause the newer memory to be overwritten. A scenario like this could occur (and it does): the model writes to just a few bytes of a previously unread and unwritten line, and then calls on the functional read to read the whole line before the dirty bytes have been flushed. Since parts of the line are invalid still, the request mechanism will cause a formal read request for it. The data written along with the read response will then have to be filtered against the dirty bytes in that same interval, so that the necessary new bytes are fetched, but the dirty ones are not overwritten.

For example, if the model was to generate a read request for eight bytes starting from **0xdef**, we would exclude the third and fourth byte from being overwritten.

3.3.3 Unordered Write Requests

There is one additional case that needs to be handled in order to not cause problems. According to the AXI protocol specification, a write transaction occurs in the following order: write request information, write data, wait for response acknowledging a completed transaction on the receiving end. The first two could be treated as atomic in the model implementation, usually happening in the same cycle in either order. No matter the internal ordering of these two the functional event, which is instant, will happen before the AXI write request comes to the outgoing as it will take at least one cycle to get into the system. When the functional write takes place the affected bytes are marked as dirty, and when the AXI write response comes back they will be de-marked. However, one of two things might also happen:

- (1) A functional write is done without sending any AXI write request at all, as discussed in the previous section where it was addressed by appending them to the next flush cycle.
- (2) A component erroneously waits for the AXI write response before performing the functional write.

The latter case can be especially insidious, because when the AXI request goes to the external interface, it will trigger a copy of the underlying data from the model's internal memory to the external memory. Since the functional write has not taken place yet, the underlying piece of memory in the model internal memory will either be uninitialised or outdated. To handle this we added a filter in the model's request endpoint before it forwards the request externally. Since an uninitialised X-state cannot be down-cast to a definite datatype it will first check the underlying content for this, and mask out any bytes with X-state in the outgoing strobe - for these we can with certainly say that the write operation has been unordered. For bytes with initialised data there is no way of knowing this though, and the external memory will copy from the old data source. This is not a big problem however, because when the functional write occurs (presumably shortly afterwards) this is essentially the same as (1), and will leave the bytes marked as dirty. Since data marked as dirty has precedence when the model is reading, the model will always consume this data instead of the outdated piece in the external memory even if that one happens to be requested.

Putting it Together

At the end of this endeavour, we can set up a problem detection with the following rules:

- Before a functional read is done, verify that all the bytes encompassed have either been externally fetched, or internally written to (i.e. that they are *valid*)
- When a functional write is done, mark it as dirty until a matching performance request has gone out. Additionally, mark it as valid for the model to use
- When a performance read is done, mark its content as valid for the model to use
- When memory gets invalidated, mark this as no longer valid for the model to use

- When the model flushes, generate requests for any outstanding dirty data at the end of it
- When a copying is called as result of a write request, make sure the model actually had the data for it

3.4 Handling Problems

In Section 3.3 we defined a *problem* as an instance of a functional read of underlying memory different than would have been read in the unmodified reference build. We then laid out rules for how these are identified in Section 3.3.3. Now that we know how to spot the problems, we will go on to find a way to automatically handling them.

3.4.1 Just-in-time suspension

The invalid data detection outlined in Section 3.3.3 relies on making checks when the model is about to perform a functional read. This means that the read detection is "just in time". The sequence typically looks look like in Figure 3.6.

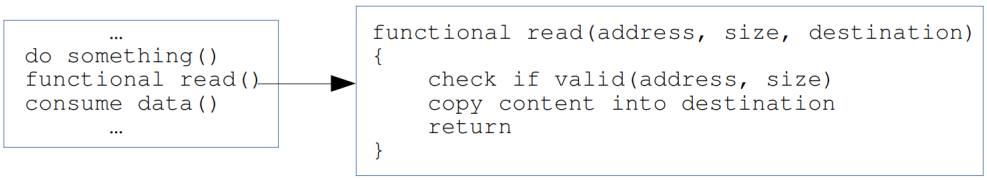


Figure 3.6: Functional read invocation

Prior to the invocation of the functional read function, which could be from anywhere in the model, it would not be known to us that it was about to be invoked without knowing subcomponent-specific behaviour. If the validity check states that the memory that will be read in the next line is uninitialised or outdated, it will break model functionality as incorrect data will be passed into the system. This means that we have to request the data from the external interface, but the problem is that this is not immediate (in the simulated case, it costs us a minimum of 170 cycles).

This is a problem with two possible solutions

- Notify the caller that the data cannot be consumed at this point in time
- Wait for the data to come in before returning it

The first option is made very difficult by the fact that the caller is opaque leaving us with no information on how and when the returned memory will be consumed. A typical case might look something like Figure 3.7:

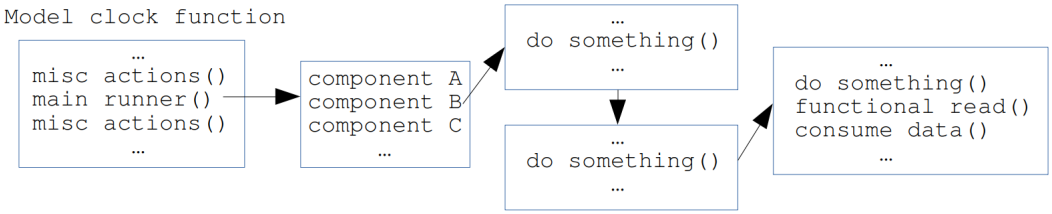


Figure 3.7: Invocation flow from toplevel

A full handling of the situation via that route would mean explicit state-checking in every component that calls the functional read. As highlighted in Figure 3.8 this involves a lot of hops, and would require a massive code overhaul.

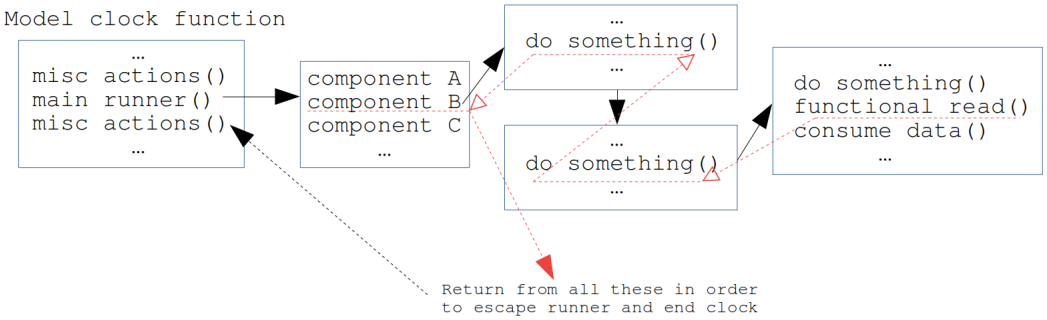


Figure 3.8: Returning to toplevel

Changing data types for the destination buffers could provide some advantage. This could be achieved by creating a datatype that contains a byte as well as a status indication of whether it is valid to consume or not. The overload for implicit conversion to a regular char would then be conditional on this status indication, potentially raising a custom flag if an invalid byte was about to be consumed. This could be useful in a case such as

```
functional_read(A)
functional_read(B)
do_something(A, B)
```

where the flag would not have to be raised until both had been read. This would however only accomplish a packing together of reads taking place back to back before consuming any of the contents (which should be a rare occurrence, if even happening at all), reducing it to a single suspension fetching the external content at once.

We will now look at the second option, waiting for the data to get back before returning. The request mechanism is not a concurrent procedure, but one of the things that get handled in every clock cycle. Pushing a request through the system takes many steps, and these steps all require cycling. However, we cannot cycle if we do not return, because as long as we have not returned we are holding up the whole model of which the memory system is also part. Assuming we do indeed intend to pause the process here, it means we got to do one of two things with respect to the request

- Clock only certain parts of the model (the parts necessary to get a request through the system) while keeping the subcomponent that made the request in suspension

- Find a way to get the request out without having it pass the internal systems

To better understand the situation we are in, consider the flow shown in Figure 3.9.

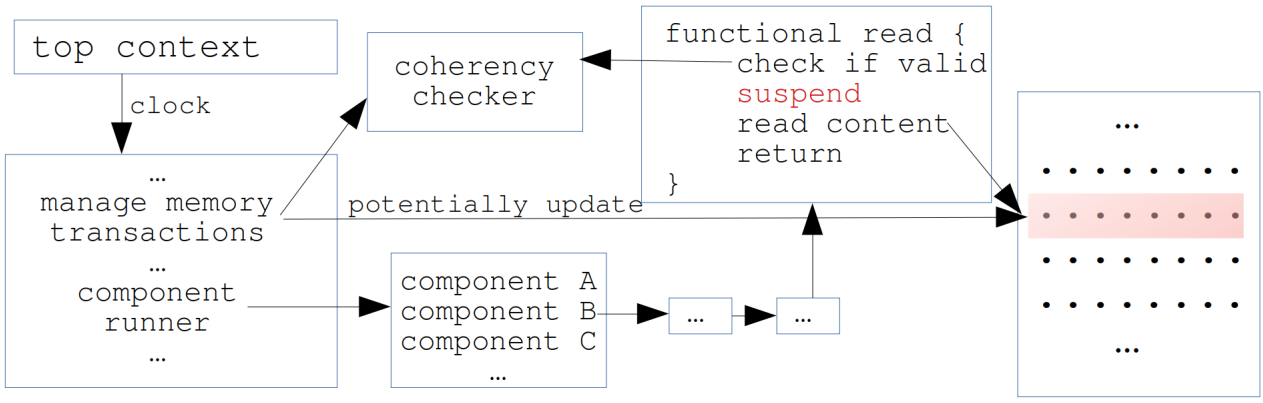


Figure 3.9: Flow showing the context from which we must yield

In this example, a chain of calls from component **B** lands in a functional read for which the coherency checker will report that the data is invalid. We wish to suspend execution before the next operation - the read - is performed, and not resume until the condition is satisfied that the correct data has arrived to the memory. The problem is that this condition can never be satisfied if we do not leave our current context and return to top-level. Additionally, we face the following circumstances:

- Any action performed in the model in the current cycle up until this point may have affected the model state. This means we cannot safely repeat the partially executed cycle without having saved the entire state of the model before.
- Because we cannot re-run the sequence that just occurred, we need to save the local variables along the current call path if we want to return to resume it from the same point.

3.4.2 Creating Syntax-friendly Abstractions

We now make an example using the **Boost.Coroutine2** library to show how it can satisfy our desired properties. An analogous version of this will be used in our implementation. This means that it should conform to the idea of clockable objects in such a way that

- Whenever the object is called, it should start from the beginning if it had run to completion in the last iteration, and resume from where it left off otherwise.
- After being called, it should run to completion or to it yielding, whichever comes first.

In order to instantiate a coroutine object with Boost we create a *pull type*, which a calling type ("run it and pull something out of it") but leave the provided template type to *void* as we do not need any data passing between the point of invocation and the clockable object. We

note that we will need to pass data out of the coroutine to give information about conditions of when it can resume, but in our case it will need to be passed to a different place than the caller for which we will use a shared object. For places where the data would be needed in the caller's scope, adding the type to the coroutine directly would be the most straightforward way.

The input argument to *pull_type* takes in a method reference which we will create with *bind*, passing in an additional argument to the destination which must take a matching *push type* as its input argument. This is shown in Listing 3.1:

Listing 3.1: Instantiating a coroutine handle

```
coroutine<void>::pull_type A { boost::bind(&Some_class::some_method,
                    &instance_of_class, boost::placeholders::_1) };
```

The first problem that we run into is shown in Listing 3.2. Once *some_method* runs to completion, nothing happens on subsequent calls - the entry point is not reset. Additionally, it is also run once on creation which is not desired in our case.

Listing 3.2: Calling a coroutine

```
Some_class::some_method(coroutine::push_type &pt)
{
    cout << "Instance created" << endl;
    pt(); // yields
    cout << "some_method invoked" << endl;
    return;
}

main()
{
    Some_class instance_of_class();
    auto A { bind(&Some_class::some_method,
        &instance_of_class, placeholders::_1) };
        // prints "Instance created"
    A(); // prints "some_method invoked"
    A(); // prints nothing
}
```

Fixing the run-once behaviour has two solutions, either regenerating a new coroutine object for each iteration of its contents, or retaining the frame by making it reset itself. To avoid the computational cost of regeneration, we opt for the latter. This behaviour is shown in Listing 3.3. By pointing the entry point of the coroutine object to an intermediary that never runs to completion the object is retained, and by having it yield between each invocation of the target method we get our desired behaviour.

Listing 3.3: Looping a coroutine

```
Some_class::method_hook(coroutine::push_type &pt)
{
    cout << "Instance created" << endl;
    while (true)
    {
        pt();
        some_method();
    }
```

```
}

Some_class::some_method()
{
    cout << "Method invoked" << endl;
    return;
}

main()
{
    ...
    A(); // prints "Method invoked"
    A(); // prints "Method invoked"
    A(); // prints "Method invoked"
    ...
}
```

Now the only part missing is making any subroutine called from the hook able to use the push type to yield, and for this reason we set it in a place that all such calls will be able to access. This could be a class base if there is a shared one for all subsequent calls, otherwise some other shared space. In the case of the latter, the handle would have to be uniquely associated with the component in some way if we want to have multiple coroutines in place at the same time. In Listing 3.4 the use of a shared handle is shown.

Listing 3.4: Using yield handle

```
Some_class::method_hook(coroutine::push_type &pt)
{
    if (!yield)
    {
        yield = &pt;
    }
    while (true)
    {
        pt();
        some_method();
    }
}
Some_class::some_method()
{
    cout << "Beginning of method" << endl;
    (*yield)();
    cout << "End of method" << endl;
    return;
}

main()
{
    ...
    A(); // prints "Beginning of method"
    A(); // prints "End of method"
    A(); // prints "Beginning of method"
    A(); // prints "End of method"
    ...
}
```

```
coroutine::push_type *yield;
```

We now have a setup in such a way that we only need to create the handle in our top-level once, and then we can go ahead and call it as usual. In any function invoked from **A**, we can freely make a call to yield.

Performance impact of reusing a frame

In the previous section we made the design decision to instantiate the coroutine to an infinitely looped proxy function instead of creating a new instance for each invocation. To verify that this was indeed the right call, we created a small clockable dummy class which we applied both approaches on, and found that the frame-retaining version was about 24 times faster than rebuilding it on each iteration.

3.5 Heuristics

In this section we will take a look at how applying different heuristics might affect cycle- and bandwidth counters.

3.5.1 Filtering Out Duplicate Requests

In the case of unordered requests triggering a suspension state, the SC wrapper will generate an AXI request that it will pass to the external AXI interface. When this request returns to the SC wrapper, the wrapper will write the raw underlying data to the GPU model's internal memory so that it can act on it when resuming. A short while later the GPU will reach the point of generating and sending out its own AXI request however, which will be passed on the regular outgoing AXI channel to the SC wrapper. This makes an impact to the bandwidth numbers of the external interface. Since it is unnecessary to bother the external memory system twice for the same request, these can be filtered out. If the model makes a request for the same address as one generated by our problem handling within a short given time-span afterwards, we can return a fake response as we know the data has already been written to the model. The reason to give such a fake response would be to update the internal statistics of the model correctly. This flow is shown in Figure 3.10

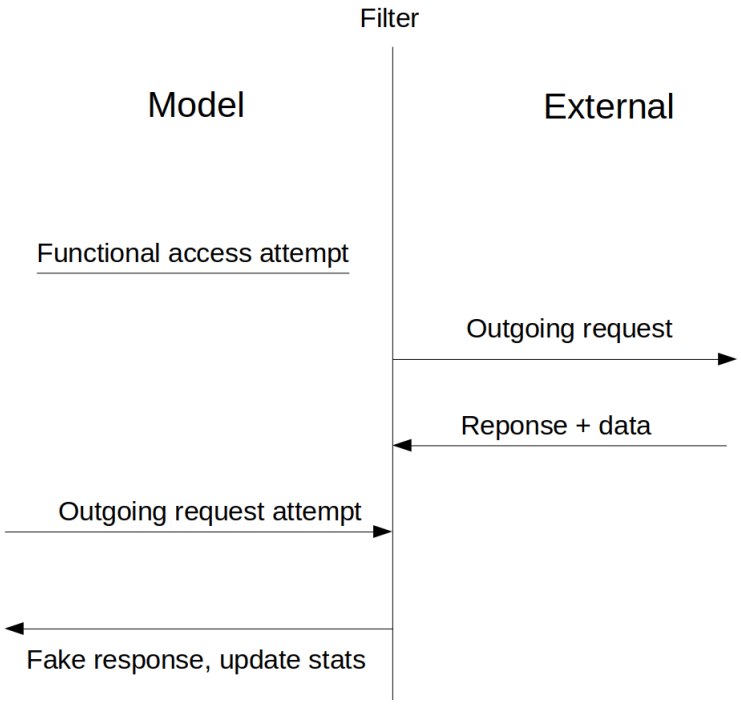


Figure 3.10: Duplicate request being filtered out

3.5.2 Some Easy Pre-fetching

An inspection of the address logs of the lines being requested through the suspension mechanism showed that they were highly clustered with large continuous swats. This indicates that fetching blocks of adjacent memory might prove beneficial. One of the easiest ideas would be to fetch the whole page the faulty read address was contained in. Disregarding the bandwidth consumed for a second, the cycle counter would benefit from even a low hit rate. Considering that making a single request has an expected latency of 170 cycles, with an additional cycle per 128 bits requested. Not considering any other stalls in the pipeline, or hardware behaviour relating to sequential fetching in SDRAM, this creates an expectation for 174 cycles penalty when suspending to fetch a single 64 byte line, compared to a 426 cycle penalty when suspending to fetch a 4k-block of 64 lines. This means that if only two more lines in that whole page turns out to need a critical suspension, it would be worth it in terms of active cycles.

For reasons discussed in Section 5.2.4, this track was not pursued in depth and no further block-fetching heuristics were tested.

3.5.3 Exploring Speeding up the Simulation

If the GPU has no interaction with the outside world for a period of time, this period of time could hypothetically run inside a single SC-cycle. This could be leveraged to find information about critical states ahead of time which could then be compensated by leveraging the ahead-state and not clock the model while the requested information is in-flight externally. For example, if the model does not have any inputs in the time period between some time T

and $T + 100$, then there would be no difference in advancing the model one clock cycle for each environment cycle, or advancing the model 100 cycles inside a single environment cycle (meaning that the environment is still in time T, but the model is now in time $T + 100$) and then advancing only the environment excluding the model for its next 100 cycles. Should there have been any output from the model, let's say at time $T + 50$ from its perspective, then this could be held back in a buffer between model and environment until the environment is also in time $T + 50$. If this event is that the model at some time T_i suspends itself because it requires some information which would take 170 environment cycles to fetch, and the environment would be at say $T_i - 250$ relative to model time, then the environment could handle that request while not clocking the model bringing down the relative difference to $250 - 170 \approx 80$ cycles.

An input event however would require the time of the model to be equal to that of the environment. For read and write request, these are preceded by output events - the outgoing requests - which can be used to anticipate this. If the request return time can be accurately predicted, that time frame could also be added to the leverage - the point where the model and the environment need to be in sync is at the point of return. Should this be unpredictable, the safe bet would be to have the requests as sync-up points (where sync-up means having the run-ahead reduced to zero). Additionally, not all input events are predicated by an output action from the model. Should there be an interrupt sent from the system to the model, then this would effectively have an added latency equal to whatever the model's run-ahead is.

We did not do an implementation of this scheme itself, however we ran an estimation of the time frames to get an idea of the I/O down-times in order to get an idea of what results it might have yielded. To this end we generate an *estimation*, where we keep a tab of the time elapsed between events requiring the model and the environment to be synced and deduct **max(runahead, suspension request time)** from the total cycle count if a suspension request event happens under these conditions. Additionally, we set a limit of 1000 cycles for the run-ahead. It should be noted that we leveraged the assumption of known (fixed) response time for this test, meaning that the numbers might be favourable as compared to proper full-system conditions. We have two separate versions of this measure, one where timing is only synced up for read responses, and one where timing is synced up both for read and write responses. The results of these estimates are presented in Section 5.2.2.

Chapter 4
Implementation

4.1 Setting up the Testing Environment

In order to test the model together with an external memory we set up both in a co-simulation environment. This is done using a SystemC program which we will refer to as the *SystemC wrapper* throughout the text. The SystemC wrapper will interface with the GPU model, the Replay Driver, and a memory structure that we set up. The wrapper has its own clock and it is this clock we will refer to when we talk about *system time*.

4.1.1 Prior Setup

Parts of the needed setup were already available to us and set up according to Figure 4.1. These parts include the following:

- A SystemC wrapper, containing logic to interact with the model's AXI channels (but with no data transfer).
- AXI TLM library, proving a back-end for request mechanisms. It is set up to work as a fixed-latency loop back that has a default delay of 170 cycles for read requests and 90 cycles for write requests, along with a throughput of 16 bytes per cycle.
- Replay Driver, which includes a C-interface.
- The GPU model itself. Provides interface functions needed for the interactions listed above. Contains a pre-existing implementation of callbacks for the replay driver.

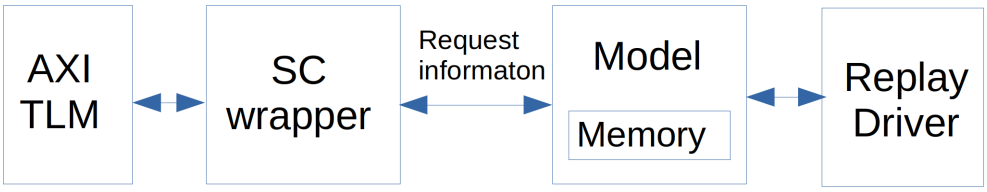


Figure 4.1: Connection between components

4.1.2 Changes to the Setup

We will add an external memory to the setup. We also want to have the GPU model simulation be controlled from the co-simulation environment. In order to do so we make a couple of changes to the parts listed in Section 4.1.1. The connection between components after these changes is shown in Figure 4.2.

- Add data passing between the SystemC wrapper and the GPU model. Tie the transfer event to the AXI responses that are in place.
- Add a data structure to act as an external system memory. This memory will be an intermediary between the Replay Driver and the model when running replays.
- Connect the Replay Driver to the SC wrapper instead of to the model itself.

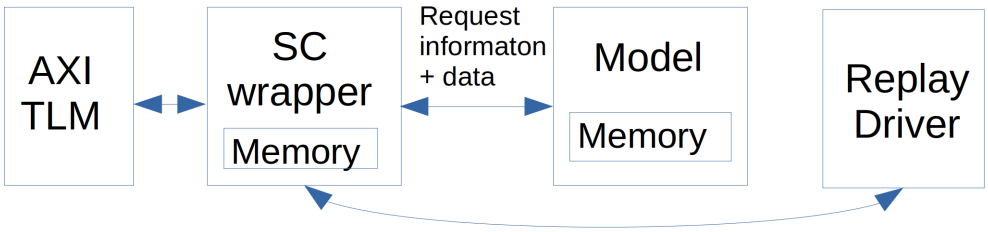


Figure 4.2: Connection between components, modified

4.1.3 Connecting the Replay Driver

The replay driver is provided as a library with a C-interface providing the callbacks for a few device-host interactions. From the driver side, this means to read and write memory, read and write registers, clear memory blocks.From the device side, this means to request a new command (which could make any of the aforementioned functionalities being called), or to report an interrupt.

Previously these functions have been implemented in a subcomponent of the GPU model responsible for test vector interaction, and in this case the functions of reading, writing, and clearing memory have acted directly on the untimed memory.

To migrate these callbacks, we use the pre-existing implementations as inspiration and switch out the memory calls to point to our external memory container. Should a proper memory component, such as a real RAM model, be added in the future this is where that

interaction should be handled. For callbacks relating to functionalities that have not been moved externally, such as register reads and writes, the internal function calls are simply swapped with externally exposed ones in the model's interface. The locations of the callback implementations, and the places from which they are called, are shown in Figure 4.3 .

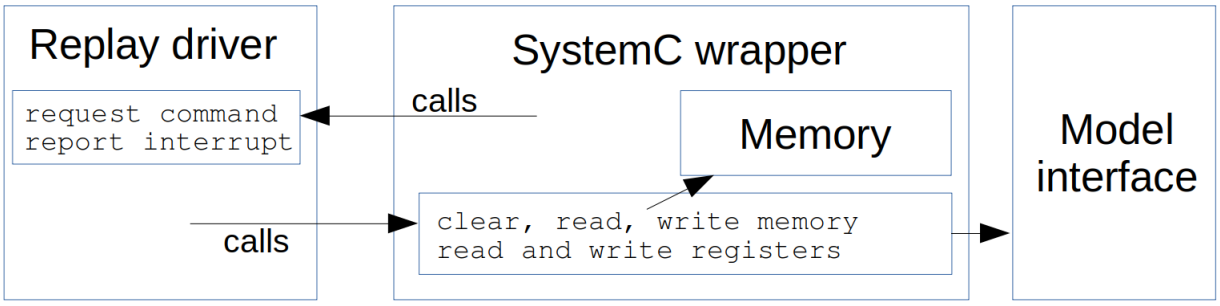


Figure 4.3: Replay driver callback flow after migration

4.1.4 Adding a Memory Container

Now that data can move between the model and the wrapper, it needs to have a place to be stored. We do not have any real system memory model at our disposal, so we create a more simple representation. To do so we set up a simple sparse type of memory, where we allocate 4k-pages on demand. We set up a fixed table to keep the page entries on, meaning that we need $2^{32-12} = 2^{20}$ entries to track 12 bit wide page addresses in a 32 bit memory space. This naturally scales rather poorly for bigger memory representations, and it would make sense to change approach if this is needed. It does however cover our test cases and provides us with easy debugging by having all functionality directly accessible in our immediate scope.

4.1.5 Data Passing

Via the GPU model's interface, we can interact with the functional read- and write methods in order to pass data between SC wrapper and the GPU model. We tie calls to these to the SC wrapper's AXI management logic, which forwards requests and responses between the AXI TLM system the the GPU model. When an AXI read response comes through, the SC wrapper copies data from the system memory to the model's internal one. Conversely, when an AXI write request comes in, data will be copied from the model's internal memory to the system memory. The separation of memories is shown in Figure 4.4.

4.2 Scheme to Catch Problems

When we talk about our *scheme* to catch problems we refer to the setup outlined in Figure 4.5. The parts in red are newly introduced by us, orange has been modified in a major way, and black ones are left untouched.

The scheme involves two principal parts – checking the functional data reads to make sure the target data is **valid**, and handling the cases where it is not. If all the data the **functional**

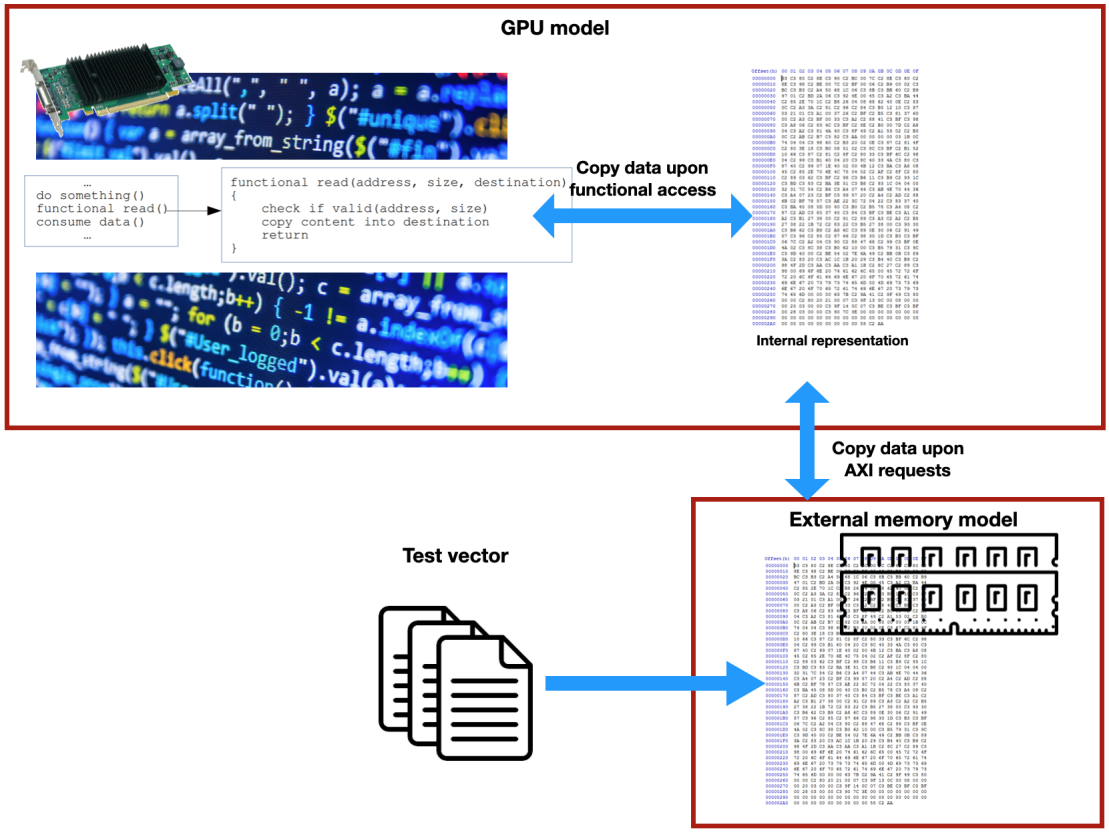


Figure 4.4: Memory copied upon AXI requests

data read tries to access is valid it will be returned as normal. If it is not, it will inform the **request management** that AXI requests covering the missing parts will have to be generated. It will also inform the **state checking** mechanism that the responses to those requests are requirements to allow computation to resume. After doing these two things, it will **suspend** the state of the current unfinished computation.

In subsequent cycles, the request management will push out the requests while the invocations of the main clock entry function returns early. Once the matching responses are back, the previously half-finished computation will be completed.

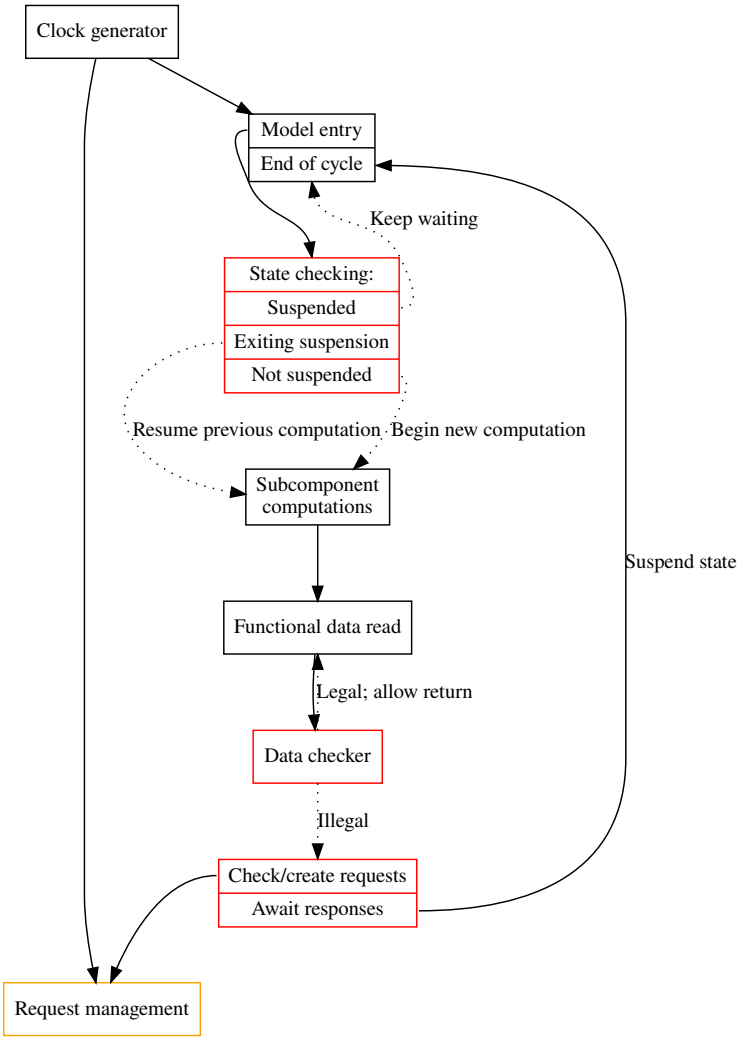


Figure 4.5: High-level layout of scheme to handle and detect problems

4.2.1 Data Checking

In order to set up the *data checker* shown in Figure 4.5, we create two classes we call *dirty data manager* and *valid data manager*. Each of these provide access methods to mark, or check the state of, data which is being referenced to. These methods are bound to model functions in accordance with Figure 4.6. These actions are summarised as follows:

- When a functional write is done, mark it as dirty until a matching performance request has gone out. Additionally, mark it as valid for the model to read.
- When a performance read is done, mark its content as valid.
- When memory gets invalidated, mark it as no longer valid for the model to use.
- When the model flushes, generate requests for any outstanding dirty data at the end of it

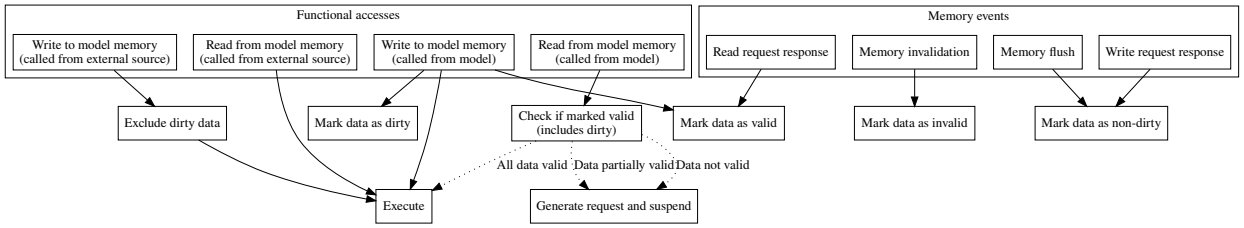


Figure 4.6: Events affecting the data checker

Underlying Data Containers

To track the valid- and dirty-states we chose to work with underlying containers of *std::unordered::_map*, containing addresses to 64-byte lines and their associated bit masks. We recognise that this is likely to be computationally inefficient when reaching large amounts of entries, but chose it for reasons of easier debugging involving frequent inspections of specific lines. Within limits of practicality, we do not seek to optimise our checker with respect to computational resources at this point - our primary objective is to make it functionally correct. This same functionality could be implemented in multiple different ways, some of which would presumably be faster, as touched upon in Section 6.3.3.

Modifying Access Methods

Inside the functional read endpoints, we now insert a call to check the content that is about to be read. For convenience, we set up the checker method in such a way that when passing in the address and size of the desired read, it will output only the lines in the span for which the range is not fully valid in order to reduce redundancy. The output of this function, if any, represents the lines that should be fetched in order to make the read operation return the expected data.

The write methods also gets a modification. When data is written, it will mark it in both the set of dirty data and the set of valid data.

We do not want to trigger the checks and allocations when interacting with the replay driver. Any call to the functional access methods which are called from the model's external interface (which will be used by the system integration later in this chapter) is given a separate analogue of the functional read and write which is identical, but with the coherency component removed. Additionally, it is in this input method we chose to out our filtering of reads overlapping with dirty bytes.

Duplicate Request Filtering

As touched on in Section 3.5.1, model-generated requests for which we have already generated our own requests can get a fake response. We let the cut-off of what is considered "close in time" to be 10000. This arbitrarily chosen number should be high enough that it catches most of the unordered request duplicates, but low enough that it would not accidentally disregard an unrelated request for the same address.

In order to not throw off the model's performance numbers for these requests, the fake response is sent after a delay equal to the delay incurred by the matching previous request.

4.2.2 Handling

Add logic to handle a suspended state and manage the additional requests. This is done in two analogous variants, one where this functionality is placed in the SC wrapper, and one where it is placed in the GPU model itself. The reason for this duplication is that the SC wrapper, having less overhead than the GPU model, is easier to modify to quickly try different tests. However, to not make the functionality heavily dependent on the co-simulation environment, we also make a version in which these parts are placed inside the GPU model itself. For this one Boost coroutines will be used for the context management.

Both versions start the same way, as shown in Listing 4.1. A call to the coherence checker is inserted into the functional read method. If the checker does not report the data to be valid, the execution suspends. Depending on the variant, this is done either by notifying *event_suspend_model* as seen in Figure 4.8, or by invoking the yield handle for the Boost coroutine instance. In addition to this, addresses of the invalid portions of data is forwarded.

Listing 4.1: Function read intercepted by check

```
functional_read(address A, destination buffer)
{
    [check if data at A is marked valid]
    [if not:
    invoke yield handle OR call wait in SC] ---> execution suspends here
    [read from A into destination buffer] <--- execution resumes here
    [return to caller]
}
```

After this step the two variants diverge slightly.

4.2.3 Variant 1: Boost

The top-level caller has been divided up from a flow as shown in Listing 4.2 to the flow shown in Listing 4.3. When the coroutine suspends the toplevel function will also immediately return and not perform any remaining actions. For subsequent cycles where the *main_clock_function* is invoked, it will also immediately return as long as the suspended state remains in place.

Listing 4.2: Toplevel

```
main_clock_function()
{
    misc
    main GPU computation
    misc
}
```

Listing 4.3: Toplevel with coroutine

```
main_clock_function()
{
    if (suspended)
        return;
    if (!first cycle post suspension)
        misc
```

```
    coroutine: main GPU computation
    if (suspended)
        return;
    misc
}
```

As shown in Figure 4.5 the **Request management** is called independently of the main clock function. This interaction is shown as **Get outputs** and **Set inputs** in Figure 4.7. The call to **Get outputs** will push out the requests for the needed data as soon as the receiving end (i.e. the SC wrapper) can accept them. The IDs of these requests are remembered to form the condition for when the suspension state should end. At the same time the call to **Set input** will feed in responses. If a response has a corresponding ID to one sent out, it will be ticked off. If an incoming response has an unrelated ID, this request is stored in a separate buffer. Once all the matching responses have come in, the suspension state is cleared. Once the model is active again, any response that might be in the input buffer is popped ahead of any new responses coming in. The whole state of the partial computation has been saved in the coroutine object and resumed from the data return as shown in Listing 4.1.

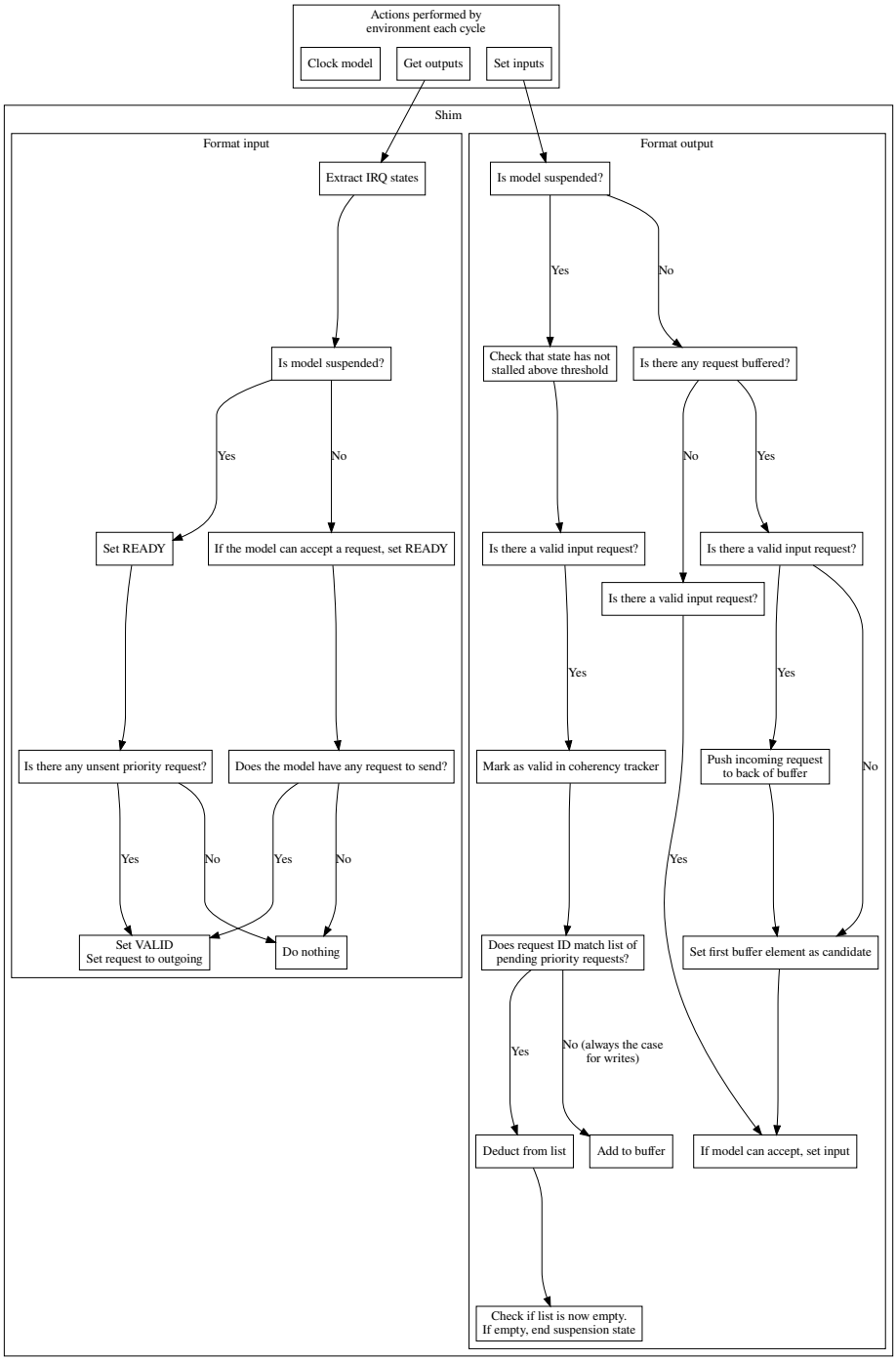


Figure 4.7: Transaction handling in the outermost layer of our modified model

4.2.4 Variant 2: SystemC wait

Our second variant manages the suspension by calling a wait on the **SC_THREAD** the model has been launched on. The principal difference between this flow, shown in Figure 4.8, and the one previously shown in Figure 4.7 is that while suspended no functions inside the GPU model will be called, not even the clock function. Aside from this the actions performed are

analogous.

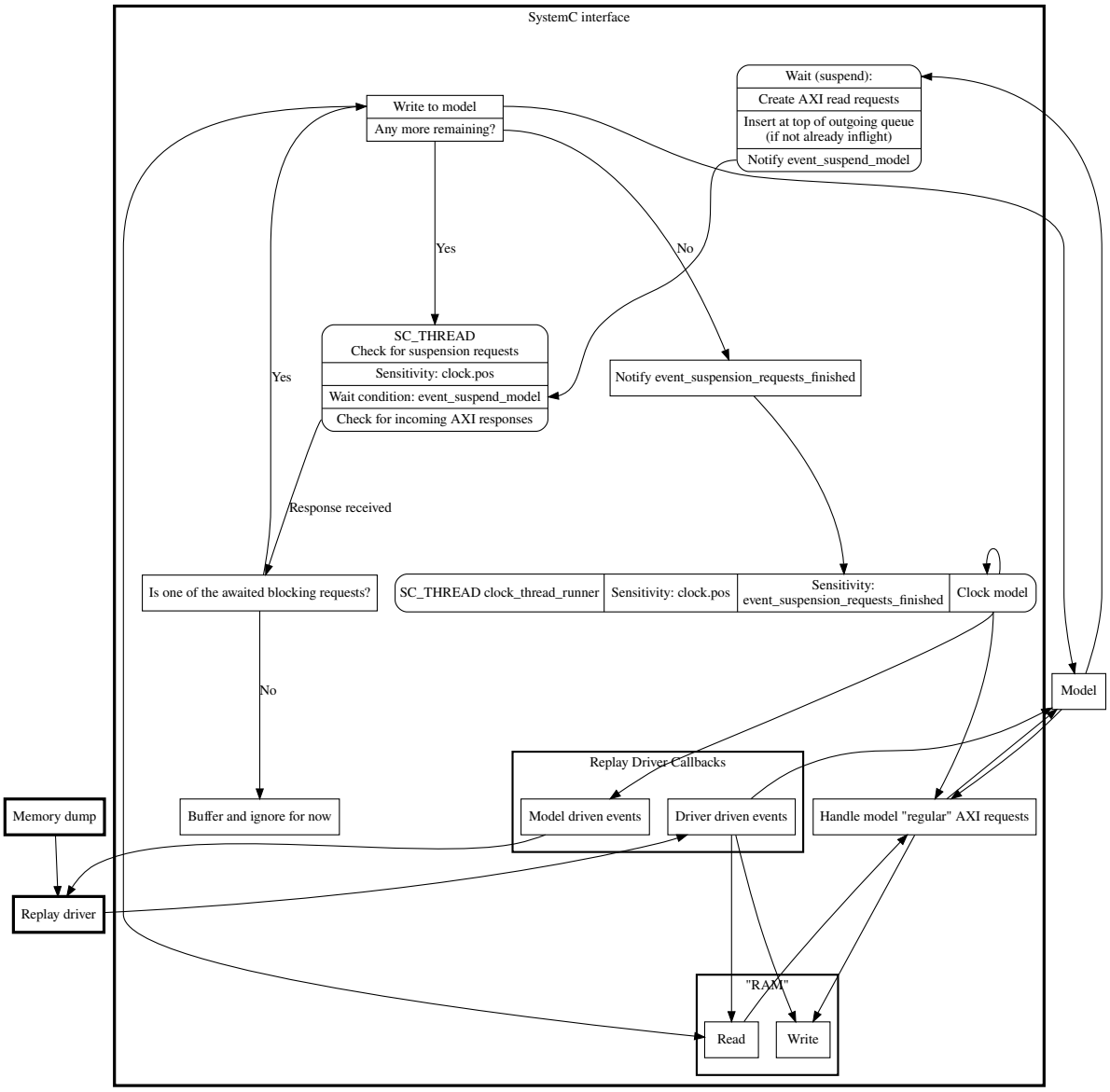


Figure 4.8: SC Logic

Chapter 5
Experimental Setup and Results

In this chapter, we will run a number of test vectors on models containing our changes. Depending on the aspects tested different variations of our changes are be applied, as specified in Section 5.1.

The first test we will perform is verifying that the output files generated by the modified models are identical to the output files generated by the unmodified model. In addition to functional correctness, simulated performance impact will be looked at. As stated in Chapter 3.1, the two key indicators the GPU model aims to stay close to are the cycle count of the computation and the memory bandwidth it consumes. For this reason we will see how our changes affect these counters. We will also look at how the wall-clock run time of the simulation is affected by our changes.

Lastly we will present some statistics on how many causes of problems are currently present in the model as was given to us. This will help to give some context to both the results of the simulation performance, and the actions we can take for future work.

5.1 Setup

Six different builds are used in order to measure different performance aspects:

1. *Variant 1* from Chapter 4.2.3. Model connected to the SystemC interface. Applies Boost coroutines. Manages request states inside the model. Returns duplicate requests and emulates their delay, otherwise has no additional heuristics.

2. *Variant 2* from Chapter 4.2.4. Model connected to the SystemC interface. Suspends the model via parent SC_THREAD. Manages request states inside the SystemC wrapper. Returns duplicate requests and emulates their delay, otherwise has no additional heuristics.

3. Clean model build, not connected to the SystemC interface. The results of this build are used as the **baseline** figures to which the other builds are compared.

4. Clean model build, connected to the SystemC interface, but with none of our added functionality.
5. Model build applying Boost coroutines for the main computation. Invoked the coroutine once each computation cycle. Does not apply coherence checking. The purpose of this build is to get an idea of how much run time performance impact the coroutines are responsible for.
6. Same as *Variant 2*, but for any read requiring a suspension it will fetch the whole page of the affected address. This is done in order to get an idea of the kind of changes that just a simple pre-fetch heuristic can yield.

In addition to this, Variant 2 also measures the potential savings that a speed-up scheme could have achieved. For reasons of time limitation this heuristic is not actually implemented, so the results should be considered an unverified guess, but might still contribute with an indication of whether it might be an interesting thing to pursue in the future.

The following variables are being measured:

- **GPU active cycles**
- **Setup total cycles**, which is the total cycles the GPU has been active as well as inactive
- **Bandwidth, read and written**
- **Wall-clock run time of the tests**
- **The code locations of functional read calls deemed illegal**, in order to find out whether the problems are caused by a few broken request sources, or many unique ones
- **The ratio of the transactions requiring suspension that got a proper request shortly afterwards**, to determine how many of the problematic reads were caused by late requests as compared to completely missing ones.

These performance indicators were logged on a set of 20 randomly selected test vectors with relatively long run time, assumed to be representative of general content.

5.1.1 Tooling Version Information

As the test vectors had long run times, a remote cluster was used to run them in parallel. Due to some difficulties with file management, all tests could not be easily run on this. For these tests, a laptop machine was used instead. For the laptop, the following tools were used:

- GCC 10.2
- SystemC 2.3.1
- Boost 1.74

For the cluster build, the following was used:

- GCC 4.8.5
- SystemC 2.3.1
- Boost 1.60

5.2 Results

5.2.1 Verifying Functional Correctness

As mentioned in Chapter 3.1, at the end of a simulation run an output file will be generated containing the memory that would have been written to a connected system. When running the modified builds, the outputs of both the *SC_THREAD* version and the *Boost* version were identical to the reference outputs. For the *SC_THREAD* version all 20 test vectors were tested and showed no difference to expectation. For the *Boost* version it was verified on three randomly selected model test vectors. The reason for the smaller number for the latter was due to version incompatibility on the cluster. For this reason the dumps were run on the laptop computer instead and due to the long run times of the tests not all could be tested.

5.2.2 Simulated Performance Impact

Model Active Cycles

The model's *active* cycles is the number of times the GPU model's computational components have been clocked. To measure the effect our changes made to this, the set of test vectors were first run on the clean build (3) to get a baseline. Afterwards, they were run on *Version 2*. Comparing the results of *Version 2* to baseline gave a deviation of **mean** −0.35%, **SD** 0.49%.

Environment Total Cycles

The environment total cycles refers to the times the SystemC clock that drives our wrapper has run. In the case where the GPU model is never suspended, this figure is identical to the active counter of the GPU, which is our baseline figure (meaning that no suspension would yield a deviation of 0%). On the set of our 20 tests, shown in Table 5.1, we got a mean deviation of 181%, with a standard deviation of 245%.

17%	21%	22%	25%	42%
57%	67%	73%	94%	103%
105%	159%	160%	171%	190%
219%	222%	285%	498%	1098%

Table 5.1: List of the $\frac{total}{active}$ deviations in ascending order

Running the list on build (6), the average deviation was reduced from 181% to 20%, offering a hit rate of 26% on average. The reduction estimates for the speed-up hack discussed in Section 3.5.3 landed on 5% when constraining both reads and writes, and 55% when constraining reads only.

Bandwidth Change

To test the bandwidth change, the total bandwidth read and written to the external memory was counted in the SystemC-wrapper, and compared to the figure reported in the internal counter of the clean build (3).

Running the list, the average change to baseline was 5% (min 4.9%, max 5.1%). It was also measured if this change was induced by extra requests generated by the wrapper, or by an increased amount on the AXI request channels between the model and the wrapper. This showed that 96% of the change to baseline was caused by change on the AXI channels between the model and the wrapper.

Additionally, the bandwidth induced when running build (6) was also compared to baseline, showing an average increase of 70% (min 37%, max 177%).

5.2.3 Run Time of the Simulation

By run time of the simulation, we refer to the real world seconds elapsed on the host machine from the start of the simulation to the end of it. For this metric, we ran two sets of tests:

- All 20 tests were run a single time each on *Variant 2*, and the baseline build (3).
- A single test (randomly picked) was run ten times on builds 1-5.

For the full set of tests, the runs showed an average increase of 19% (min 5%, max 36%) when comparing *Variant 2* to the baseline build. The individual results can be seen in Table 5.4.

For the second test, we found that neither connecting the model to the SC environment, nor making use of Boost coroutines, had any performance impact outside the confidence interval. We also found that applying our scheme on the model gave an increase of about 6% for this test, and that there is no significant difference between the run times of *Variant 1* and *Variant 2*. The results for all five runs are shown in Table 5.2.

Build	Mean (s)	SD	95% CI
1	172	2.6	1.6
2	170	3.6	2.2
3	161	5.7	3.5
4	160	4.5	2.8
5	162	7.1	4.4

Table 5.2: Results in seconds of 10 runs for a single test

5.2.4 Characteristics of the Underlying Problems

In order to get an understanding of the amount of problems currently present in the model we measure the number of incorrect data access attempts taking place. The circumstances of these are also looked at to determine if they are the result of the underlying requests being missing, or merely unordered. In addition to this, we also enumerate all the lines of code that at any point call the functional read methods in order to identify which may cause such incorrect attempts, and the frequency of how often they do so.

Unordered Versus Missing

To determine whether an improper functional read was due to the matching request being wrongly ordered or completely missing, a circular buffer keeping a reference to each line that has recently been requested in suspension mode. As a length for what should be considered recent, 10000 cycles was chosen. This means that if an illegal functional read takes place, and then an AXI request for this same address is generated by the model in the next 10000 cycles, it is considered to be matching but unordered.

Running the full set of 20 tests on build (3), it showed that on average 99.4% (SD 0.39%) of the requested lines in suspension mode got a matching request in the immediate period afterwards. This result means that almost all the problems present in the model have a corresponding mechanism implemented to send out AXI requests, but that these do not occur at the right state in the pipeline.

Proportion of the Total Reads That Induced a Problem

To give a sense of scale to what proportion of the external reads in the model that were causing issues, we take a look at the ratio of total reads that triggered a suspension. Our metric for this number is $\frac{A}{A-B+C}$ where A is the number of memory lines read via the suspension mechanism, B is the number of suspension events induced, and C is the amount of total transactions coming in to the proper channel. This shows that approximately 46% of the functional reads did not have a correctly timed AXI request corresponding to it.

Distribution of Problems Throughout the Model

Figure 5.1: Distribution of unique locations in the model code that trigger a suspension

For this test, each code instance in the model performing any functional memory access

was enumerated by preceding the call with setting an id that then would be added to a counter in the read methods.

The results in Figure 5.1 show that two single instances cause almost all the problems. The third, which happens to be the PTE-resolving, is also closely related to these as unordered requests, in most cases stemming from the first two, will trigger unordered PTE look-ups also.

An interesting thing to note is that there is a large number of problems which have an extremely low frequency, occurring only a handful of times throughout the tests. Assuming that the verification method is correct, this would indicate that there are a number of otherwise correctly implemented request mechanism which get wrongly handled under rare circumstances.

5.2.5 Lines of Code

As a measure of code complexity, we show lines of code amended to pre-existing components. We exclude blank lines.

Type	LOC
Coroutine-specific	34
State checking	114
Calls to checker	18
Coherency checker	332

Table 5.3: Lines of code by type

As seen in Table 5.3, the total lines of code is concentrated to the coherency checking part itself. This has been implemented as a separate self-contained class, limiting the added complexity. The other entries are scattered, but smaller in number.

5.2.6 Full Table of Results

We now present Table 5.4 containing the deviations for key indicators for all 20 test vectors. When collating the results the file containing the run time for test number 17 was accidentally overwritten and is therefore missing.

Tst	Active cycles (%) Δ	Total cycles(%) Δ	Bandwidth (%) Δ	Bandwidth no dual (%) Δ	Host run-time (%) Δ	# unique causes (w/ diff)
AVG	**-0.35**	**181.49**	**32.07**	**5.09**	**19.42**	**16.45 (10.10)**
1	−0.45	105.15	16.53	5.03	20.9	11(10)
2	0.08	22.12	17.03	5.13	19.69	11(10)
3	0.02	73.13	26.97	4.99	8.45	10(10)
4	−0.04	20.58	17.79	5.33	14.72	10(10)
5	−0.39	160.47	36.24	4.94	22.92	11(10)
6	−0.7	285.21	45.65	5.02	23.35	21(10)
7	−0.35	94.05	28.73	5.14	10.55	20(10)
8	−0.07	16.93	8.27	5.08	4.61	10(10)
9	−0.65	1097.94	64.84	4.89	30.49	10(10)
10	−2.2	497.55	78.01	5.15	15.81	9(9)
11	0	42.19	14.25	5.41	17.41	10(10)
12	−0.4	222.05	40.52	5.21	36.03	22(10)
13	−0.26	159.41	45.07	5.03	17.22	20(10)
14	−0.16	103.44	31.67	5.19	19.27	21(10)
15	−0.27	219.17	42.74	5.2	22.72	21(10)
16	−0.39	190.23	34.15	5.05	20.32	21(10)
17	0.01	25.17	12.77	4.76		11(10)
18	−0.14	57.43	34.98	5.13	20.58	22(10)
19	−0.16	170.94	25.41	5.13	28.41	29(10)
20	−0.5	66.69	19.68	4.93	15.52	29(13)

Table 5.4: Key indexes for all test vectors

Chapter 6
Conclusions and Future Work

All in all, this case study has shown that the examined model can be suspended by coroutines with no effect to functionality and little effect to run-time. Additionally, an exhaustive mechanism to identify memory accesses that lack corresponding AXI requests was implemented. Together these allow the model to run in an environment with a timed external memory without changes to any underlying behaviour, other than a few top-level checks. The invariance to the underlying model behaviour could make our setup (Figure 4.5) a useful catch-all.

With the current amount of optimistic memory access done by the model, applying this scheme imposes delays that are outside of the range of deviation that would give useful performance data from the model. However, since a few problematic instances are responsible for the vast majority of cases, fixing just these would bring the numbers down to an acceptable range (counter deviation $< 5\%$). Once in an acceptable range, the catch-all could provide a useful tool to make sure that minor occurrences do not break simulation functionality while having a very minor impact to the performance data.

6.1 Research Questions

In the beginning of this thesis we posed four questions which we have now examined. We will go through these one by one and briefly reflect over the conclusions we have reached.

- **How can incorrect data being consumed as a result of missing or mismanaged memory requests be identified?**

When the model tries to read data, we want two things to hold: there should be data to read, and that data should not be architecturally outdated. To cover this, we consider that there are two ways for there to be data in the model's memory: either an AXI read request has been performed, or the model itself has made an internal write. Without knowing whether the model intends to actually cache incoming memory, we treat any memory resulting of

these two actions as possibly cached. As the model performs cache invalidation for the entire cache at once, we can then clear the set of possible cached data alongside this event. When the model at any point performs a functional read, we look at the set of possibly cached data and if it is there the functional read is considered legal.

In order to also identify the missing AXI write requests we use to our advantage that the cache flushing is also a memory-wide event in the model. By keeping a set of all the functional writes that have been performed since the last flush event, we let the next flush event deduct from this set. Any entries that remain in this set at the end of the flush event is assumed to be the result of an missing AXI write request.

- **What is the current state of the model's management of memory requests?**

Currently **46%** of the read accesses performed by the model do not handle corresponding AXI requests correctly. Out of these instances **99.4%** are caused by the functional access happening before the corresponding AXI response, and the remaining **0.6%** are caused by the corresponding AXI read request not being issued in the first place.

There are **29** unique places in the model code where incorrect calls to the functional read functions are made. To ensure whether these are true problems the underlying data was also looked at. In **13** of the places the read resulted in data that was different from expectation, as shown in Table 5.4. This means that these are guaranteed to be correctly identified problems. Since there was no observed data difference for the remainders, we cannot say for certain whether they are caused by mismanaged AXI requests, or if they are false positives caused by incorrect constraints. In case the model's use of flushing and invalidation should be incorrect, false detection could follow as the constraints are based on these.

- **Is it possible to set up a way to automatically handle all the missing/mismanaged requests?**

Yes, such a setup was shown to be possible. By coupling an exhaustive detection of missing and wrongly ordered AXI requests with coroutines, the computation can be frozen in the middle of a cycle whenever a problem is detected. When the needed data has returned to the model, the frozen computation can be resumed in the next clock cycle. The flow of this scheme is outlined in Figure 4.5. Two functionally identical implementations of this scheme was made, as detailed in Chapter 4. The implementation referred to as *Variant 2* launches the entire model instance as a **SC_THREAD** and suspends it using the SystemC command **wait**. The state management is then done in the **SystemC wrapper** component, which is external to the GPU model (see Figure 4.2). By contrast, the implementation referred to as *Variant 1* keeps both the suspension of the computation and the state management inside the GPU model itself. The suspension itself is accomplished by launching the main computational event as a **Boost coroutine**. For technical details on these implementations, see Figure 4.7 and Figure 4.8.

- **How would such a handling impact the performance estimates given by the model?**

There are two performance indicators that we deem important - clock cycles of the simulation run, and the memory bandwidth consumed. For the clock cycle count, there will be an increase correlating directly with the amount of incorrect requests that need to be handled. It will roughly follow a formula of average delay $\times$ suspensions needed $+ \frac{\text{data per suspension}}{\text{data throughput}}$,

which for our constants becomes $170\times$ suspensions needed $+4\times$ memory lines covered. Our measured results deviated only by 0.4% from this formula, indicating that there is nothing else in the model having any significant effect on the delay.

Bandwidth on the other hand can be handled without increase in the case of incorrectly ordered AXI requests. Our scheme will generate new AXI requests for memory accesses that do not have a corresponding one already. However, by comparing these with the ones generated by the model shortly afterwards, the ones generated by the model could be blocked if matching. Since most of the problematic memory accesses were the result of incorrectly ordered rather than fully missing AXI requests, we did not see any significant change in the memory bandwidth in the tests.

6.2 Comparing the Two Variants

As outlined in Section 4.2.3 and Section 4.2.4 respectively, we made two versions of the modified setup. They are functionally identical in terms of output and model performance statistics. The difference between them is where in the code certain events are handled. The results shown in Table 5.2 showed that there is no measurable performance impact between the two variants.

A principal difference is that in *Variant 1* the main clock function of the model still gets invoked, potentially allowing for the resumption of some model functionality in the suspended state. In *Variant 2*'s suspension, the entire GPU model is treated as a single entity. Another advantage of *Variant 1* is that the solution is portable, since the state behaviour is contained within the GPU model itself. This means that no extra work would be needed if it were to be set up using a different simulation environment than that provided by the SystemC wrapper.

The reason we attempted *Variant 2* was due to ease of debugging by having request and state management placed in a small isolated section of the code-base. This proved very useful in our initial effort to get the problem identification working right. Additionally, the setup can provide a template for different models with the same properties being connected to the SystemC wrapper. However, if there is no need for this, there is no obvious benefit of *Variant 2* over *Variant 1*.

In conclusion, *Variant 2* is a bit easier for fast modifications, but *Variant 1* is portable which is a desirable property.

6.3 Future Work

There are several aspects of this project which could have been done differently or be expanded upon. Here follow a few ideas that come to mind that might be interesting for future investigations:

6.3.1 Testing in a System With Real Memory Models

In the tests conducted with the replay driver, the GPU model's internal active counter deviated very little from what was expected given the fixed latencies of the simple memory

used. This is a good indication that our changes do not induce any unexpected behaviours by themselves. However, it is possible that stalling the GPU might give different performance numbers with a more complex memory system, in particular with one that also has system caches. It would for that reason be very interesting to observe up the same scheme in a full-system setup in order to test both how system performance is affected by intermittent down-times of the GPU, and to explore if it changes the ability to exploit any heuristics.

6.3.2 Threading Compatibility

Our proposed scheme in its current form (see Figure 4.5) is incompatible with the multi-threaded version of the model. The multi-threading strategy for the multi-threaded version of the model revolves around spawning worker threads on the level of the component runner. If we are running the entire component runner parent as a coroutine, this means the model will attempt to spawn multiple threads from within a coroutine context which breaks the idea of having an specific context to yield from. A solution to this might be to launch coroutines inside each thread and suspend that one instead of the thread it runs on. Since the Boost library does not support migration of coroutines between threads, a new frame would have to be reconstructed every time the thread pool re-balances. Aside from this, functionality could be left mostly intact.

6.3.3 Improving the Efficiency of Underlying Containers

As mentioned in Section 4.2.1, the data types we use for the coherency checking are not well optimised for efficiency, and as shown in Section 5.2.3 they are roughly responsible for a 10% increase in host run-time. Even a small effort could likely improve this. One idea would be to set up an extra sparse memory where each byte representation could contain two bits, one to indicate the valid state and the other to indicate the dirty state. Another option would be to include these two status bits directly in the extended byte type which makes up the memory representation.

References

[1] F. Ghenassia *et al.*, *Transaction-level modeling with SystemC*. Springer, 2005, vol. 2.

[2] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and Systems Synthesis (IEEE Cat. No. 03TH8721)*. IEEE, 2003, pp. 19–24.

[3] M. Radetzki and R. S. Khaligh, "Modelling alternatives for cycle approximate bus tlms." in *FDL*. Citeseer, 2007, pp. 74–79.

[4] M. Modarressi, "Lecture 12: An introduction to systemc," http://ce.sharif.edu/courses/86-87/2/ce333/resources/root/Lecture%20Notes/L12-Introduction%20to%20SystemC%20[Compatibility%20Mode].pdf. (Accessed 2021-04-21 23:38:00).

[5] D. C. Black, J. Donovan, B. Bunton, and A. Keist, *SystemC: From the ground up; Second Edition*. Springer Science & Business Media, 2010, vol. 71.

[6] A. L. D. Moura and R. Ierusalimschy, "Revisiting coroutines," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 31, no. 2, pp. 1–31, 2009.

[7] University of Maryland cmsc311, "Understanding the stack," https://web.archive.org/web/20130225162302/http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/stack.html. (Accessed 2021-04-21 23:38:00).

[8] Oracle Corporation, "Multithreaded programming guide - user-level threads," https://docs.oracle.com/cd/E19455-01/806-5257/mtintro-54820/index.html. (Accessed 2021-04-21 23:38:00).

[9] Boost, "Coroutines (boost documentation)," https://www.boost.org/doc/libs/1_75_0/libs/coroutine/doc/html/coroutine/coroutine.html#ftn.coroutine.coroutine.f0. (Accessed 2021-04-21 23:38:00).

[10] N. Goodspeed and O. Kowalke, "Distinguishing coroutines and fibers," *ISO/IEC JTC1/SC22–Programming languages and operating systems–C++(WG21)*, vol. 2, 2014.

[11] Wikipedia, "Coroutine - implementations for c++," https://en.wikipedia.org/wiki/Coroutine#Implementations_for_C++. (Accessed 2021-04-21 23:38:00).

[12] Linux Manual Page, "makecontext(3) — linux manual page," https://man7.org/linux/man-pages/man3/makecontext.3.html. (Accessed 2021-04-21 23:38:00).

[13] X. Xu and G. Li, "Research on coroutine-based process interaction simulation mechanism in c++," in *Asian Simulation Conference*. Springer, 2012, pp. 178–187.

[14] cppreference, "std::condition_variable," https://en.cppreference.com/w/cpp/thread/condition_variable. (Accessed 2021-04-21 23:38:00).

[15] Tsuna's blog , "How long does it take to make a context switch?" https://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html. (Accessed 2021-04-21 23:38:00).

[16] The Open Group Base Specifications Issue 6, "Ieee std 1003.1, 2004 edition - <ucontext.h>," https://pubs.opengroup.org/onlinepubs/009695399/basedefs/ucontext.h.html. (Accessed 2021-04-21 23:38:00).

[17] ARM, "Amba® axi™ and ace™ protocol specification," https://developer.arm.com/documentation/ihi0022/latest/. (Accessed 2021-01-25 02:23:00).

[18] P. A. Lebedev, "Integrating GPGPU computations with CPU coroutines in C," *Journal of Physics: Conference Series*, vol. 681, p. 012048, feb 2016. [Online]. Available: https://doi.org/10.1088/1742-6596/681/1/012048

[19] C. Kohlhoff, "Executors and asynchronous operations, revision 2," http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0113r0.html. (Accessed 2021-04-21 23:38:00).

[20] D. Becker, M. Moy, and J. Cornet, "Parallel simulation of loosely timed systemc/tlm programs: challenges raised by an industrial case study," *Electronics*, vol. 5, no. 2, p. 22, 2016.

[21] S. Jones *et al.*, "Optimistic parallelisation of systemc," *Universite Joseph Fourier: MoSiG DEMIPS, Tech. Rep*, 2011.

Appendices

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2021-02-19

EXAMENSARBETE Integration of a Cycle-approximate Model Into a Cycle-accurate Environment
STUDENT Andreas Hansson
HANDLEDARE Jörn Janneck (LTH), Reimar Döffinger (Arm)
EXAMINATOR Flavius Gruian (LTH)

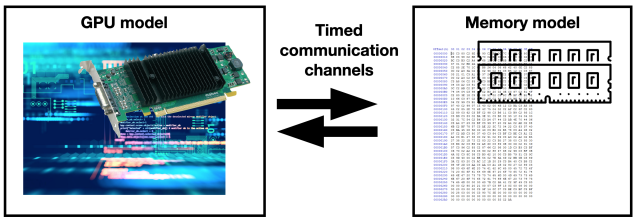
Getting Simulated Hardware Components to Work Together

POPULAR SCIENCE SUMMARY **Andreas Hansson**

Software models are used to aid the development of hardware components. To simulate systems with multiple components, multiple models can be connected together. In our project we show how mistimed communication can complicate such setups, and how this can be handled.

Over a billion phones are sold each year. The phone market is very competitive, with customers expecting constant improvements. As companies compete to satisfy these demands there is a need for optimised development processes. To speed up the development of chips, software models that mimic hardware behaviour are used. By having virtual versions of the chips being designed, designers and developers can try a lot of different changes without having to rebuild the actual parts. Different testing needs have led to the existence of a lot of different types of models, and the degree to which they stay true to the hardware behaviour varies.

In this investigation, we have looked at a model simulating an Arm Mali GPU – a popular chip used to handle graphics in many phone models. When a graphics processing unit (GPU) runs, it frequently needs to use data that is located in a memory chip that is external to the GPU itself. Transferring between these takes a bit of time, so the GPU has to plan accordingly when it wants to fetch data. However, this simulation model has been implemented under the assumption that it can instantly access the data when needed. This makes a lot of testing easier, but makes the model incompatible with memory models that only work with timed communication channels and do not allow for such instant access.



In order to enable compatibility our project has looked at the different ways the GPU model tries to access data before it has been fully transferred. To address these we have created a mechanism that tracks all transfers taking place and successfully identifies the incorrect data access attempts. When we discover an incorrect attempt we send out for the needed transfers and wait for these to be handled before we continue with the computation. This ensures that the GPU model never makes use of any different data than intended, which is necessary for the result of the computation to be correct. As we successfully identify and handle all the incorrect attempts taking place, we can now connect the GPU model to simulated systems that require timed transfers - enabling new test cases.