University of Kaiserslautern

Department of Electrical Engineering and Information Technology

Microelectronic Systems Design Research Group

Bachelor Thesis

Development of a System-Level Simulation Frontend for DRAMSys

Entwicklung eines System-Level-Simulations-Frontends für DRAMSys

---

Presented:	August 16, 2022
Author:	Derek Christ (409571)
Research Group Chief:	Prof. Dr.-Ing. N. Wehn
Tutor:	Dr.-Ing. M. Jung
	M.Sc. Lukas Steiner

Statement

I declare that this thesis was written solely by myself and exclusively with help of the cited resources.

Kaiserslautern, August 16, 2022

Derek Christ

Abstract

The performance of today's computing systems depends in particular on the memory system utilized. With the increasing usage of DRAMs, also in mobile and embedded systems, it is important to select a memory configuration that fits the application well to provide high performance. However, this is a complex task within the system design due to the overwhelming number of possible configurations and their advantages and disadvantages. In particular, bandwidth and latency requirements must be satisfied. Consequently, to verify these requirements, simulations of the system are essential to evaluate whether the configuration parameters used are suitable for the application. Such a simulation can be accomplished with the DRAM simulation environment DRAMSys. A simulation requires a realistic stimuli for the memory system that matches the application's behavior, which can be created by the time-consuming simulation of the application using processor core models. To overcome this drawback of very long simulation time, a faster method of creating stimuli for DRAMSys is developed in this thesis. In this method, access patterns are created by analyzing the application's behavior using dynamic binary instrumentation while it is running on real hardware. With our approach, we are able to simulate 73% faster compared to gem5 FS while only losing 7% in accuracy in respect of the bandwidth.

Zusammenfassung

Die Leistung heutiger Rechensysteme hängt insbesondere von dem eingesetzen Speichersystem ab. Mit der zunehmenden Verbreitung von DRAMs auch in mobilen und eingebetteten Systemen ist es wichtig, eine Speicherkonfiguration zu wählen, welche gut zur Anwendung passt, um eine hohe Leistungsfähigkeit zu erzielen. Dies ist jedoch aufgrund der überwältigenden Anzahl möglicher Konfigurationen und ihrer Vor- und Nachteile eine komplexe Aufgabe innerhalb des Systemdesigns. Insbesondere Anforderungen an Bandbreite und Latenzen müssen erfüllt werden. Folglich sind zur Überprüfung dieser Anforderungen Simulationen des Systems unerlässlich, um zu bewerten, ob die verwendeten Konfigurationsparameter für die Anwendung geeignet sind. Solch eine Simulation kann mit der DRAM Simulationsumgebung DRAMSys durchgeführt werden. Eine Simulation erfordert realitätsnahe Stimuli für das Speichersystem, das dem Verhalten der Anwendung entspricht, welches mit einer zeitaufwändigen Simulation der Anwendung mit Prozessorkernmodellen erstellt werden kann. Um diesen Nachteil der sehr langen Simulationszeit zu überwinden, wird in dieser Arbeit eine neue Methode zur Erstellung von Stimuli für DRAMSys entwickelt. Bei dieser Methode werden Zugriffsmuster durch die Analyse des Verhaltens der Anwendung mittels Instrumentierung erstellt, während sie auf echter Hardware ausgeführt wird. Mit unserem Ansatz sind wir in der Lage, die Simulationen im Vergleich zu gem5 FS um 73% zu beschleunigen, während wir in Bezug auf die Bandbreite nur 7% an Genauigkeit verlieren.

Contents

1 Introduction

Today's computing systems accompany us in almost all areas of life in the form of smart devices, computers, or game consoles. With the increasing performance requirements on these devices, not only faster processors are needed, but also high-performance memory systems, namely dynamic random-access memories (*DRAMs*), which are supposed to deliver a lot of bandwidth at a low latency. While these storage systems are very complex and offer a lot of room for configuration, e.g., the DRAM standard, the memory controller configuration or the address mapping, there are different requirements for the very different applications [1]. Consequently, system designers are entrusted with the complex task of finding the most effective configurations that match the performance and power contraints with good optimizations applied for the specific use case.

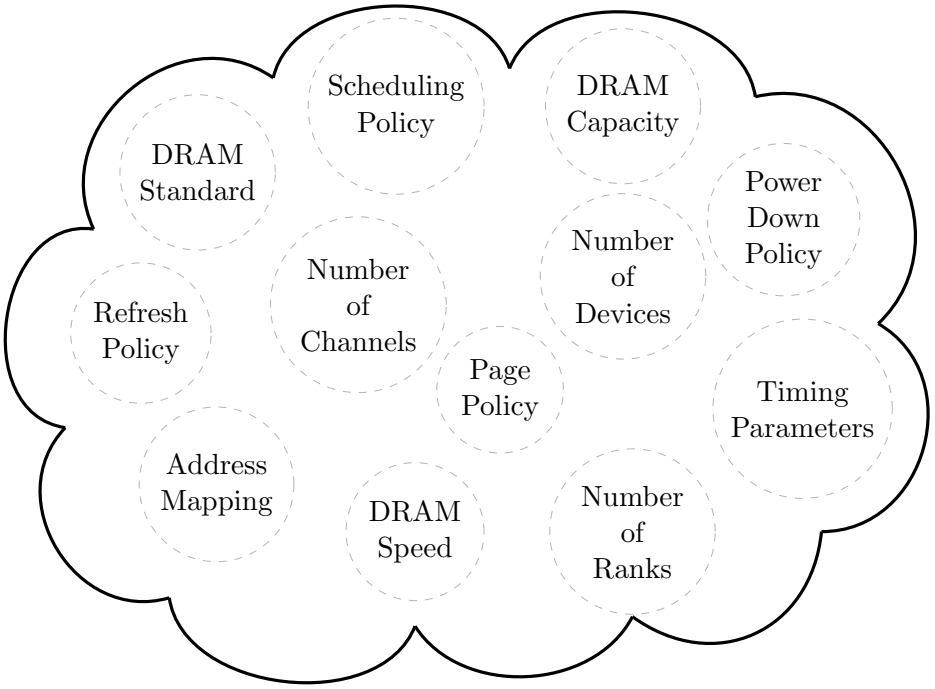

Figure 1.1: Exemplary DRAM configuration parameters to consider when designing a system.

For the exploration of the design space of these configurations, it is impractical to use real systems as they are too cost-intensive and not modifyable and therefore not suitable for rapid prototyping. To overcome this limitation, it is important to simulate the memory system using a simulation framework with sufficient accuracy.
Such a simulation framework is DRAMSys [2, 3], which is based on SystemC transaction level modeling (*TLM*) and enables the fast simulation of numerous DRAM standards and controller configurations with cycle-accuracy. Stimuli for the memory system can either be generated using a prerecorded trace file with timestamps, a traffic generator that acts as a state machine and initiates different request patterns or a detailed processor model of the gem5 [4] simulation framework.
However, the two former methods lack in accurary whereas the latter may provide the sufficient precision but is a very time-consuming effort. To fill this gap of fast but accurate traffic generation, a new simulation frontend for DRAMSys is developed and presented in this thesis.

The methology this new framework is based on is dynamic binary instrumentation. It allows the extraction of memory accesses of multi-threaded applications while they are executed on real hardware. These memory access traces are then played back using a simplified core model and are filtered by a cache model before the memory requests are passed to the DRAM. This allows an accurate modeling of the system and the variation of numerous configuration parameters in a short time.
The remainder of the thesis is structured as follows: In Section 2 the used dynamic binary instrumentation framework, DynamoRIO, is introduced. Section 3 presents the modeling language SystemC, on which the developed core and cache models are based on. After that, Section 4 gives a short overview of modern cache architectures and their high-level implementations. Section 5 introduces the DRAMSys simulation framework and its basic functionalities. Section 6 explains the implementation of the cache model, the processor model and the instrumentation tool in detail. In Section 7 the accuracy of the new framework is compared against the gem5 and Ramulator [5] simulators, whereas Section 8 finally denotes future improvements that can be achieved.

2 DynamoRIO

This section will give a short overview of the dynamic binary instrumentation tool DynamoRIO, which will be used throughout this thesis. The explained topics are mainly based on the chapters *"DynamoRIO"*, *"Code Cache"* and *"Transparency"* of [6] as well as on [7].

2.1 Dynamic Binary Instrumentation

Dynamic binary instrumentation (*DBI*) is a method to analyze, profile, manipulate and optimize the behavior of a binary application while it is executed. This is achieved through the injection of additional instructions into the instruction trace of the target application, which either accumulate statistics or intervene the instruction trace.
In comparison, debuggers use special breakpoint instructions (e.g., INT3 on x86 or BKPT on ARM) that are injected at specific places in the code, raising a debug exception when reaching it. At those exceptions a context switch to the operating system kernel will be performed. However, those context switches result in a significant performance penalty as the processor state has to be saved and restored afterwards, making it less efficient than DBI.
DBI tools can either invoke the target application by themselfes or are attached to the application's process dynamically. The former method allows instrumentation of even the early startup stage of the application whereas the latter method might be used if the application has to be first brought into a certain state or the process cannot be restarted due to reliability reasons. Some DBI tools also allow to directly integrate the DBI framework into the applications source code. While this eliminates the flexibility of observing applications that are only available in binary form, it enables the control over the DBI tool using its application interface. With this method, it is possible to precisely instrument only a specific code region of interest and otherwise disable the tool for performance reasons.
In all cases, the instrumentation tool executes in the same process and address space as the target application. While this enables great control of the DBI tool over the target application, it becomes important that the tool operates transparently, meaning that the application's behavior is not affected in unintended ways. This is a special challenge as the instrumentation tool as well as the user-written instrumentation clients are not allowed to use library routines for memory operations/allocation, synchronization or input/output buffering that interfere with the target application [7]. In particular, this is the case with library routines that are not *reentrant*, which means they are unsafe to call concurrently. The dispatcher of the DBI tool can run in arbitrary places, also during non-reentrant routines. When the instrumentation tool or user-written client calls the same non-reentrant routine concurrently, undefined behavior would be the consequence. Although it is evident, the user-written client should make no assumptions on the running system's behavior and should restore all modified registers and processor states unless it is an intentional interference with the application. Most DBI tools offer the use of two distinct methods of injecting user code into the applications trace; in one case, the framework saves all relevant registers and flags by itself and dispatches the execution to a user-defined function. This is the easiest method, but comes at the cost of the described context switch. The more advanced approach is the injection of few but sufficient instructions directly into the applications instruction trace. Here, it is the responsibility of the user to save and restore all altered states.

Generally speaking, the application should have no possibility to be able to detect that it is being instrumented by a DBI tool and should execute the same way as it would do normally, even when the application itself commits incorrect behavior such as accessing invalid memory regions.
In summary, dynamic code analysis has the full runtime information available, unlike static code analysis, which cannot predict the execution path of the program. So DBI can be a mature choice for examining the runtime behavior of a binary application in a performant way.
The following Section 2.2 will explain how the core functionality of the DBI tool DynamoRIO works.

2.2 Core Functionality

A simple way to observe and potentially modify the instructions of an application during execution is the use of an interpretation engine that emulates the binary executable in its entirety. One widely used framework that uses this technique is for example Valgrind [8]. At its core, Valgrind uses a virtual machine and just-in-time compilation to instrument the target application. This approach might be powerful, but it comes at the cost of significantly reduced performance.
DynamoRIO, on the other hand, uses a so-called *code cache* where *basic blocks* are copied into prior to execution. A basic block is a sequence of instructions extracted from the target application's binary that end with a single control transfer instruction. In the code cache, the instrumentation instructions will directly be inserted.
To be able to execute the modified code, basic blocks in the code cache are extended by two *exit stubs*, ensuring that at the end the control is transferred back to DynamoRIO via a context switch. From there the application's and processor's state is saved and the next basic block is copied into the code cache, modified and executed after restoring the previously saved state. Basic blocks that are already located in the code cache are directly executed without copying, however, a context switch is still needed to determine the next basic block to execute.
To reduce this overhead and avoid a context switch, DynamoRIO can *link* two basic blocks together that were targeted by a direct branch, i.e., branches whose target address will not change during runtime. To achieve this, the target address has to be converted in-place to point to the new address in the code cache and not the original one in the mapped binary executable. For indirect branches, i.e., branches whose target address is calculated at runtime, it is not possible to link them as their target basic blocks may vary. However, basic blocks that are often executed in a sequence are merged into a *trace*. At the end of each basic block, an additional check is performed to determine if the indirect branch target will stay in the same trace, possibly preventing the context switch. Those regularly executed parts of the application code are also referred to as *hot code* and their optimization using traces improves the performance but introduces the minor disadvantage of multiple copies of the same basic block in the code cache. The generic term for a basic block or a trace is *fragment*.
Figure 2.1 illustrates the internal architecture and functionality of DynamoRIO. The application code is loaded by the dispatcher, modified by the basic block builder, copied into the code cache and finally executed from there.
As mentioned in Section 2.1, it is important for a DBI tool to operate transparently. DynamoRIO takes a number of measures to achieve this goal, some of which are now explained [6]. As sharing libraries with the target application can cause transparency issues, especially when using non-reentrant routines or routines that alter static state

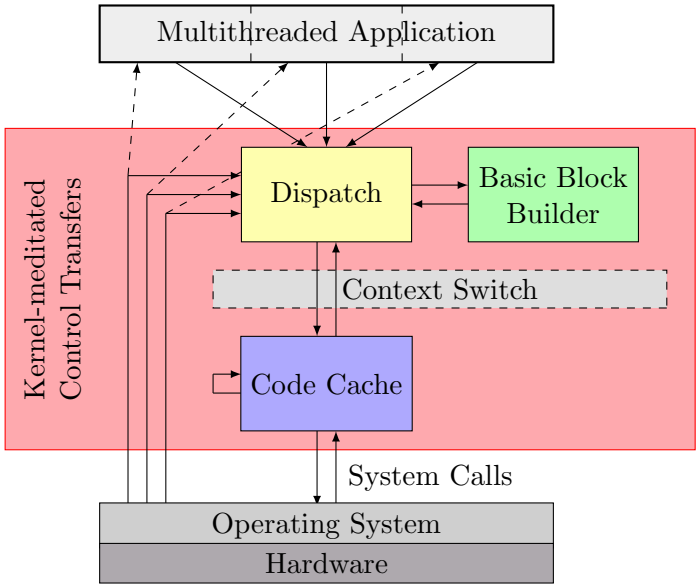

Figure 2.1: DynamoRIO runtime code manipulation layer [6].

such as error codes, DynamoRIO directly interfaces with the system using system calls and even avoids to use the C standard library (e.g., *glibc* on Linux). The same should also apply for user-written instrumentation clients (introduced in more detail in Section 2.3), but the direct usage of system calls is discouraged as this bypasses the internal monitoring of DynamoRIO for changes that affect the processes address space. Instead, DynamoRIO provides a cross-platform API for generic routines as file system operations and memory allocation. To guarantee thread transparency, DynamoRIO does not spawn new threads by itself, but uses the application threads instead and creates one DynamoRIO context for each. When an instrumentation client needs to spawn threads, they should be hidden from introspection of the application. Client code should also not alter the application stack in any way, as some specialized applications access data beyond the top of the stack. Alternatively, DynamoRIO provides a separate stack that should be used instead to store temporary data. To remain undetected, it is also required for DynamoRIO to protect its own memory from malicious reads or writes from the application. Those should, like in the native case, raise an exception as unallocated data is accessed. However, as these memory regions are actually allocated, DynamoRIO has to produce those exceptions itself to remain transparent. When the application branches to a dynamically calculated address, DynamoRIO has to translate this address to the corresponding address of the basic block in the code cache. But also in the backward case, whenever a code cache address is exposed to the application, it has to be converted back to the corresponding address to the mapped address region of the binary executable.
As it can be seen, DynamoRIO makes significant effort to ensure transparency. However, factors such as timing deviations cannot be taken into account, since the instrumentation code consists of additional instructions that must be executed. So a sophisticated application could try to detect the presence of an instrumentation tool by estimating and comparing the execution time of its own routines.

Table 2.1: Client routines that are called by DynamoRIO [7].

Client Routine	Description
void dynamorio_init()	Client initialization
void dynamorio_exit()	Client finalization
void dynamorio_thread_init(void *context)	Client per-thread initialization
void dynamorio_thread_exit(void *context)	Client per-thread finalization
void dynamorio_basic_block(void *context, app_pc tag, IntrList *bb)	Client processing of basic block
void dynamorio_trace(void *context, app_pc tag, IntrList *trace)	Client processing of trace
void dynamorio_fragment_deleted(void *context, app_pc tag)	Notifies client when a fragment is deleted from the code cache
void dynamorio_end_trace(void *context, app_pc trace_tag, app_pc next_tag)	Asks client whether to end the current trace

2.3 Clients

With the inner workings introduced so far, the presence of DynamoRIO does not have an effect other than that the application is executed from the code cache. DynamoRIO provides a programming interface to develop external so-called *clients* [6]. Clients are user-written instrumentation tools and make it possible to dynamically modify the basic blocks, either to alter the application behavior or to insert observational instructions. A DynamoRIO client is compiled into a shared library and passed to the *drrun* utility using a command line option. Clients implement a number of hook functions that will be called by DynamoRIO for certain events such as the creation of a basic block or of a trace. Generally, there are two classes of hooks: those that execute on basic block creation, which instrument all of the application code, and those that execute on trace generation, which are only interested in frequently executed code. It is important to note that the hooks for basic block and trace generation are not called every time when this code sequence is executed, but only when these basic blocks are generated and placed into the code cache. So the required instructions have to be inserted into the basic block instruction stream in this stage, rather than implementing the observational or manipulative behavior in the hook function itself.
Table 2.1 lists some of the most important hooks that a client can implement.
Most of the hooks receive a `void *context` pointer to the thread-local machine context through its parameter list, which then needs to be passed to the code manipulation routines. Those routines are available through DynamoRIO's rich code manipulation API that enables the generation, the encoding and the decoding of instructions. Since the processor's flag and general purpose registers might be altered by executing those new instructions, it is necessary to store them before and restoring them after execution to guarantee transparency. DynamoRIO also provides client routines to store those flags and registers in thread-local slots. An alternative to manually storing and restoring are, as previously mentioned in Section 2.1, so-called *clean calls* where DynamoRIO takes the responsibility for storing and restoring the processor's state. The clean call then dispatches to a user-defined function that will be run every time the basic block executes by modifying the program counter. This comes at the great advantage of not having

to implement the observational or manipulative behavior using assembly instructions; instead the compiler of the client takes care of converting the clean call function into machine code. However, since DynamoRIO can not know which registers have to be stored as this depends on the user code, it has to preserve the whole processors state. The dispatching to the clean call function is essentially a context switch and therefore has a great impact on the performance. So it is up to the user to decide whether the gain in performance by avoiding clean calls outweighs the higher development effort.
An exemplary client that already comes with DynamoRIO is *DrCacheSim*. Together with the *DrMemtrace-Framework*, this client provides an easy way to trace the executed instructions of the application and the memory accesses it makes. This framework will be further explained in Section 6.1.

3 SystemC

This section covers the basics of virtual prototyping, SystemC and transaction level modeling.
Virtual prototypes (*VPs*) are software models of physical hardware systems, that can be used for software development before the actual hardware is available. They make it easier to test the product as VPs provide visiblity and controllability across the entire system and therefore reduce the time-to-market and development cost [9].
SystemC is a C++ class library with an event-driven simulation kernel, used for developing complex system models (i.e., VPs) in a high-level language. It is defined under the *IEEE 1666-2011* standard [10] and is provided as an open-source library by Accellera.
All SystemC modules inherit from the `sc_module` base class. Those modules can hierarchically be composed of other modules or implement their functionality directly. Ports are used to connect modules with each other, creating the system structure of the simulation. There are two options to implement a process in a module:

- An `SC_METHOD` is sensitive to `sc_event`s or other signals. Methods are executed multiple times, each time they are triggered by their sensitivity list.
- An `SC_THREAD` is started at the beginning of the simulation and should not terminate. Instead, threads should contain infinite loops and should explicitly call `wait()` to wait a specific time or for events.

Moreover, there is the event queue type `sc_event_queue`, which makes it possible to queue multiple pending events, where as an `sc_event` ignores further notifications until it is waited on.
The concepts presented are used in Section 6, where the implementation of various SystemC modules will be discussed.
SystemC supports a number of abstraction levels for modeling systems, namely *cycle-accurate*, the most accurate but also the slowest abstraction, *untimed*, *approximately-timed* and *loosley-timed*. The latter two abstraction levels belong to transaction level modeling (*TLM*), which will be discussed in the next Section 3.1.

3.1 Transaction Level Modeling

TLM abstracts the modeling of the communication between modules using so-called transactions, which are data packets transferred by function calls [11]. In comparison to pin and cycle accurate modeling, this greatly reduces the simulation overhead at the cost of reduced accuracy.
Modules communicate with each other through *initiator* sockets and *target* sockets. For example, a processor sends requests to a memory through its initiator socket, while the memory responds through its target socket. Interconnection modules, which can be used to model a bus, use both socket types to communicate with both initiator and the target modules. This concept is illustrated in Figure 3.1.
The transaction object itself is a generic payload (*GP*), which consists of the target address, whether the transaction is a read or write command, status information and other transaction parameters, the actual data to transfer as well as user-defined payload extensions. GPs are passed as references, so they do not need to be copied between modules.
SystemC defines two coding styles for the use of TLM, called loosley-timed (*LT*) and approximately-timed (*AT*). In the LT coding style, a transaction is blocking, meaning that the transaction will be modeled by only one function call. This comes at the cost of

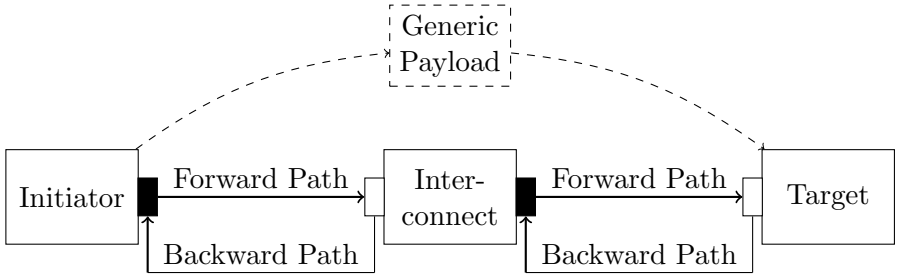

Figure 3.1: Forward and backward path between TLM sockets [11]. ■ denotes an initiator socket, □ denotes a target socket.

limited temporal accuracy, as only the start and end times of the transaction are modeled, and the initiator must wait until the transaction is completed before making the next request. However, the fast simulation time, especially when the so-called concept of *temporal decoupling* with *timing quantums* is used, makes it possible to use this coding style for rapid software development; LT is suitable for developing drivers for a simulated hardware component.
The AT coding style is non-blocking and can therefore be used to model with a higher timing accuracy than LT. This high accuracy makes it possible to use AT for hardware-level design space exploration. With AT, a special protocol is used that uses a four-phase handshake: `BEGIN_REQ`, `END_REQ`, `BEGIN_RESP` and `END_RESP`.
When an initiator requests data from a target, it starts the transaction with the `BEGIN_REQ` phase by calling its `nb_transport_fw()` method. This method in turn calls the receiving module's target socket and the target module then enqueues the payload into its payload event queue (*PEQ*). The PEQ pretends it has received the payload after the delay, that the initiator has specified with its call to the transport method. If the target is not yet ready to accept the new request, it defers its `END_REQ` phase until it is ready. During this time, the initiator is blocked from sending further requests to this module as the target applies *backpressure* on the initiator. This concept is called the *exclusion rule*. Otherwise, the target directly responds the `END_REQ` phase back to the initiator.
The target then prepares the response and sends the `BEGIN_RESP` phase through its `nb_transport_bw()` method when the data is available. The initiator can now also apply backpressure to the target by deferring its `END_RESP` phase. When the `END_RESP` phase is received by the target, the transaction is completed. Figure 3.2 shows an exemplary handshake sequence diagram of three different transactions.
SystemC defines various special cases and shortcuts that can be used troughout the protocol. In the `BEGIN_REQ` phase, it is possible for the target to directly send the `END_REQ` phase using the return value of the forward transport function call. This requires setting the return type `tlm_sync_enum` to `TLM_UPDATED` instead of `TLM_ACCEPTED` in the normal case. Analogously, it is also possible for the initiator to directly respond with the `END_RESP` phase using the return value during the `BEGIN_RESP` phase.
Besides this, it is also possible for the target to directly respond with the `BEGIN_RESP` phase after it has received the `BEGIN_REQ` phase and therefore skip the `END_REQ`. The initiator has to react accordingly and must detect that the `END_REQ` has been skipped. However, since the initiator is blocked due to backpressure during this period, this shortcut should only be used if the response is ready to send after a short delay. Another form of this shortcut is the combination with the return path of the forward transport function call. Here, the return path is used to directly send the `BEGIN_RESP` phase, with-

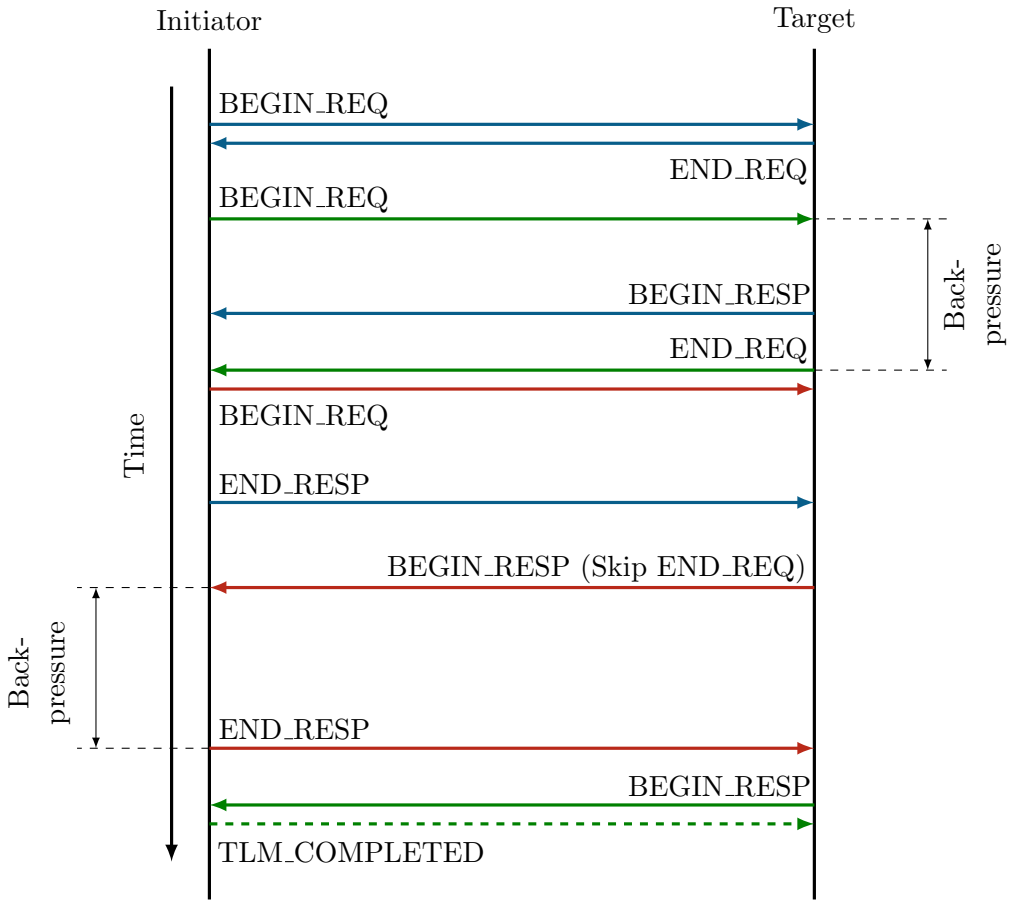

Figure 3.2: Sequence diagram of exemplary transactions.

out invoking the backward transport function altogether, reducing the required number of transport calls to only two.
The last shortcut that can be made is the so-called *early completion*. When the target receives the `BEGIN_REQ` phase, it can already place the requested data into the payload and pass `TLM_COMPLETED` as the return value back to the initiator. This notifies that the whole transaction is already completed at this point, so no further transport calls are required. Note that this form of early completion is very similar to the LT coding style, where a transaction also is modeled using only one function call. Early completion can also be used by the initiator to skip the `END_RESP` phase. Here, `TLM_COMPLETED` is returned during the backward transport call of the `BEGIN_RESP` phase.
SystemC also supports additional user-defined phases through its `DECLARE_EXTENDED_PHASE()` macro for special cases.
In contrast to the TLM-LT protocol, TLM-AT allows to model the pipelining of transactions; multiple transactions can be processed simultaneously by one target. The responses also do not need to be in the same order as the initiator has sent them; they can be *out-of-order*.
The TLM-AT coding style is the protocol used to implement the processor model and the cache model in Section 6 of this thesis. Some of the earlier described shortcuts are taken advantage of throughout those models.

4 Caches

In this section, the necessity and functionality of caches in modern computing systems is explained as well as the required considerations resulting from virtual memory addressing. A special focus is also placed on non-blocking caches. The theory is based on the chapters *"An Overview of Cache Principles"* and *"Logical Organization"* of [12] and on [13].
With the advancement of faster multi-core processors, the performance difference to the main memory is increasing, commonly referred to as the *memory wall*. Therefore, caches, whose goal is to decrease the latency and increase the bandwidth of a memory access, play an important role when it comes to the overall performance of computing systems. Caches are faster than DRAM, but only provide a small capacity as the area cost is a lot higher. For this reason, at least the *working set*, the data that the currently running application is working on, should be stored in the cache to improve performance.
The two most important heuristics that make this possible will be explained in Section 4.1. After that, the typical structure of a cache will be discussed in 4.2. Replacement policies will be explained in 4.3 and write policies in 4.4, followed by the considerations to make when it comes to virtual addressing in Section 4.5. Section 4.6 gives a short introduction on cache coherency and snooping. Finally, the advantage of non-blocking caches is the topic of Section 4.7.

4.1 Locality Principles

Access patterns of a typical application are not random. They tend to repeat themselfes in time or are located in the near surrounding of previous accesses. Those two heuristics are called *temporal locality* and *spatial locality*.

4.1.1 Temporal Locality

Temporal locality is the concept of referenced data being likely to be referenced again in the near future. Taking advantage of this is the main idea behind a cache: when new data is referenced, it will be read from the main memory and buffered in the cache. The processor can now perform operations on this data and use its end result without needing to access the main memory.

4.1.2 Spatial Locality

Programs have a tendency to reference data that is nearby in the memory space of already referenced data. This tendency, spatial locality, arises because related data is often clustered together, for example in arrays or structures. When calculations are performed on those arrays, sequential access patterns can be observed as one element is processed after the other. Spatial locality can be exploited by organizing blocks of data in so-called *cache blocks* or *cache lines*, which are larger than a single data word. This is a passive form of making use of spatial locality, as referenced data will also cause nearby words to be loaded into the same cache line, making them available for further accesses. An active form of exploiting spatial locality is the use of *prefetching*. Here, the program causes the cache to fetch more than one cache line from the underlying memory system.

4.2 Logical Organization

This section revolves about the question where to store the retrieved data in the cache. Because the cache is much smaller than the DRAM, only a subset of the memory can

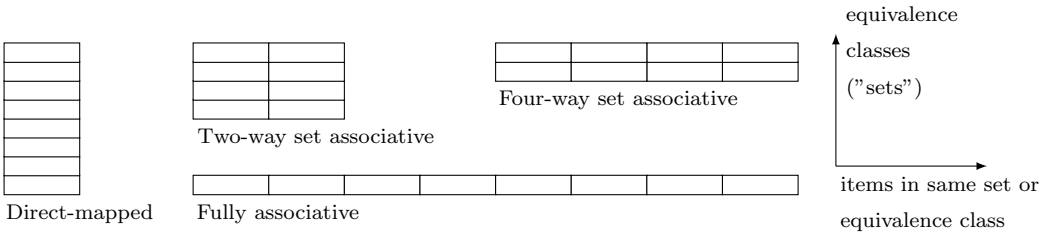

Figure 4.1: Four organizations for a cache of eight blocks [12].

be held in the cache at a time. The cache line into which a block is placed is determined by the *placement policy*. There are three main policies:

- In *direct-mapped caches* the cache is divided into multiple sets with a single cache line in each set. For each address there is only one cache line where the data can be placed in.
- In a *fully associative cache* there is only one large set, containing all available cache lines. Referenced data has no restriction in which cache line it can be placed.
- *Set-associative caches* are a hybrid form of the former two: there are multiple sets containing several cache lines each. The address determines the corresponding set, in that the data can be placed in any of the cache lines.

Figure 4.1 illustrates four different organizations for a cache of eight cache lines. As an example, a data block with the address `0x40` may be placed in the second set for the direct-mapped, two-way associative and four-way associative cache configurations. However, in the latter two configurations, the cache can choose the horizontal placement of the block within the set. For the fully associative cache, every cache line is a valid placement as it consists of only one set.
In each cache configuration, the least significant portion of the physical address of the referenced data, the *index*, determines the set in which the data is to store. However, several entries in the DRAM map to the same set, so the remaining most significant portion of the address is used as a *tag* and is stored next to the actual data in the cache line. After an entry is fetched from the cache, the tag is used to determine if the entry actually corresponds to the referenced data. An exemplary subdivision of the address in the index, tag and byte offset is shown in Figure 4.2.

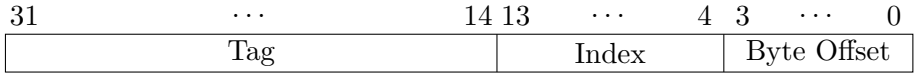

Figure 4.2: Exemplary address mapping for the tag, index and byte offset.

Direct-mapped caches have the advantage that only one tag has to be compared with the address. However, every time new data is referenced that is placed into the same set, the cache line needs to be evicted. This leads to an overall lower cache hit rate compared to the other two policies.

In a fully associative cache, a memory reference can be placed anywhere, consequently all cache lines have to be fetched and compared to the tag. Although this policy has the highest potential cache hit rate, the area cost due to additional comparators and high power consumption due to the lookup process, make it non-feasible for many systems. The hybrid approach of set-associative caches offers a trade-off between both policies. The term *associativity* denotes the number of cache lines that are contained in a set.

4.3 Replacement Policies

In case of contention, cache lines have to be evicted. To determine which cache line in the corresponding set is evicted, there are several replacement policies:

- The random policy selects a cache line of a set at random.
- The least recently used (*LRU*) policy selects the cache line whose last usage is the longest time ago. An LRU algorithm is expensive to implement, as a counter value for every cache line of a set has to be updated every time the set is accessed.
- An alternative is a pseudo LRU (*PLRU*) policy, where an extra bit is set to 1 every time a cache line is accessed. When the extra bit of every cache line in a set is set to 1, they are reset to 0. In case of contention, the first cache line whose extra bit is 0 is evicted, which indicates that the last usage was likely some time ago.
- In the least frequently used (*LFU*) policy, every time a cache line is accessed, a counter value will be increased. The cache line with the lowest value, the least frequently used one, will be chosen to be evicted.
- The first in first out (*FIFO*) policy evicts the cache lines in the same order they were placed.

4.4 Write Policies

To maintain consistency to the underlying memory subsystem, special care has to be taken when a write access occurs. In case of a *write-through* cache, the underlying memory is updated immediately, meaning the updated value will also directly be written into the DRAM. Because the DRAM provides a significantly lower bandwidth than the cache, this comes at a performance penalty. To mitigate the problem, a write buffer can be used, which allows the processor to make further progress while the data is written. An alternative is a so-called *write-back* cache. Instead of writing the updated value immediately to the underlying memory, it will be written back when the corresponding cache line is evicted. To identify if a cache line has to be written back, a so-called *dirty-bit* is used; it denotes if the value has been updated while it has been in the cache. If this is the case, it must be written back to ensure consistency, otherwise it is not necessary. Also here, a write buffer can be used to place the actual write back requests into a queue.

4.5 Virtual Addressing

Operating systems use virtual addressing to isolate the memory spaces of user space programs from each other, giving each process an own virtual address space.
Virtual addresses are composed of a *virtual page number* and a *page offset*. The virtual page number is the actual part that is virtual, the page offset is the same for the virtual

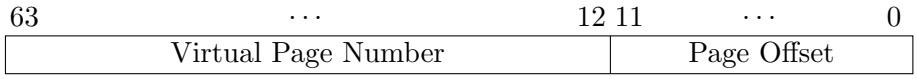

Figure 4.3: Exemplary division of the virtual address into a virtual page number and page offset.

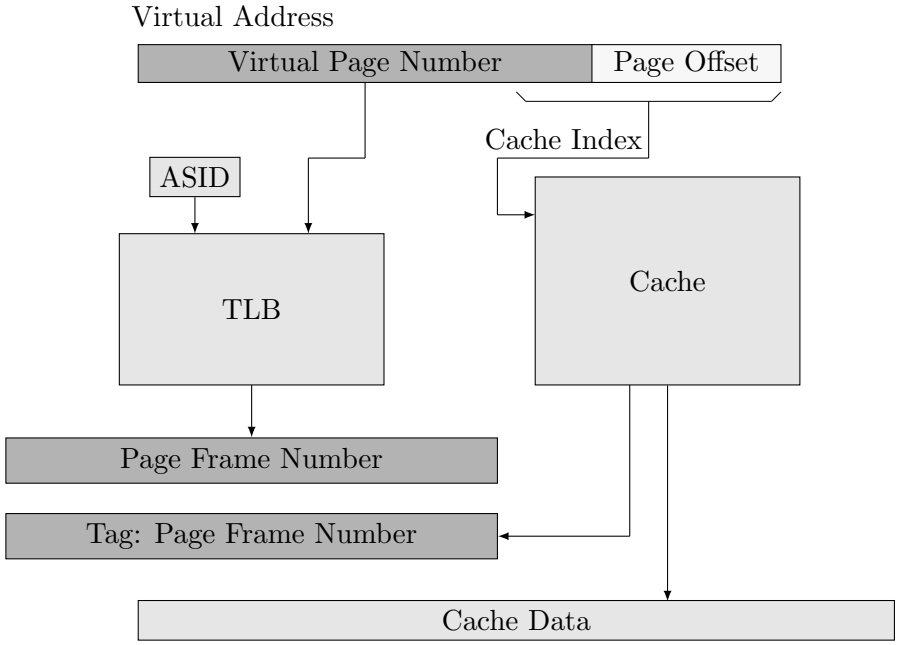

Figure 4.4: Virtually indexed, physically tagged cache [12]. ASID refers to address-space identifier.

and the physical address. Figure 4.3 shows an exemplary division of a virtual address into its components.
Before a process can access a specific region in memory, the kernel has to translate the virtual page number into a physical page number. For conversions, so-called *page tables* are used to look up the physical page number. Page tables are usually multiple levels deep (e.g., 4-levels on x86), so a single conversion can cause a number of memory accesses, which is expensive. To improve performance, a translation lookaside buffer (*TLB*) is used, which acts like a cache on its own for physical page numbers.
However, as long as the physical address is not present, the data cache cannot look up its entries as the index is not known yet. So the cache has to wait for the TLB or even multiple memory accesses in case the physical page number is not stored in it. To circumvent this problem, the cache can be indexed by the virtual address, which makes it possible to parallelize both procedures. Such a cache is called *virtually indexed* and *physically tagged* and is illustrated in Figure 4.4.
The result from the TLB, which is the physical page number, needs to be compared to the tag that is stored in the cache. When the tag and the physical page number match, then the cache entry is valid for this virtual address.

4.6 Cache Coherency

In multi-core environments, caches become a distributed system. As every core uses its own set of caches and possibly shares a cache at the last stage with the other processors,

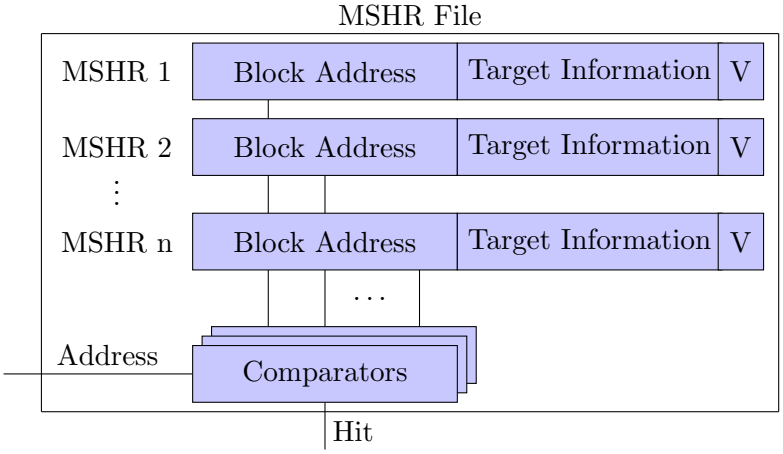

Figure 4.5: Miss Status Holding Register File [13]. *V* refers to a valid bit.

a new problem arises. If two or more cores operate on the same shared data, multiple copies of the data will be placed in the private caches and it must be guaranteed that all cores agree on the actual value the data has at any point in time. Different perceptions of the same data should be considered as errors.
Therefore, it is important to guarantee *cache coherency*. One of the solutions for cache coherency is the use of a so-called snooping protocol. A cache will snoop the cache coherence bus to examine if it already has a copy of requested data. Snooping packets are then used to update or invalidate other copies of the data. Snooping protocols are complex and it is difficult to formally verify that they in fact guarantee cache coherence. For this reason, they are not further discussed in this thesis.

4.7 Non-Blocking Caches

In blocking caches, cache misses require the processor to stall until the data is fetched from the underlying memory. As this is a major slowdown, non-blocking caches try to solve this problem, making it possible for the processor to make further progress while waiting on the value.
Similarly to the write buffer, previously discussed in Section 4.4, a new buffer will be introduced: the miss status hold register (*MSHR*). The number of MSHRs correspond to the number of misses the cache can handle concurrently; when all available MSHRs are occupied and a further miss occurs, the cache will block. An MSHR entry always corresponds to one cache line that is currently being fetched from the underlying memory subsystem.
There are two variants of cache misses: *primary misses* are misses that lead to another occupation of an MSHR, where as *secondary misses* are added to an existing MSHR entry and therefore cannot cause the cache to block. This is the case when the same cache line as accessed.
A possible architecture of an MSHR file is illustrated in Figure 4.5.
When the data for a cache miss is returned from the underlying memory, the cache will be updated, all targets of the MSHR entry will be served with the value and the MSHR entry will eventually become deallocated.

5 DRAMSys

DRAMSys is an open-source design space exploration framework, capable of simulating the latest Joint Electron Device Engineering Council (*JEDEC*) DRAM standards. It is optimized to achieve high simulation speeds and utilizes the TLM-AT coding style while still achieving cycle-accurate results [2].
DRAMSys is composed of an arbitration and mapping unit (also called arbiter) and independent channel controllers, each driving one DRAM channel. The general architecture of DRAMSys is illustrated in Figure 5.1.

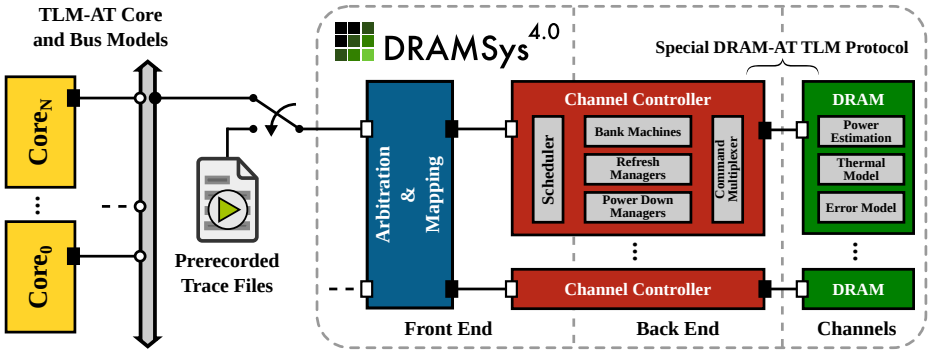

Figure 5.1: Structure of DRAMSys [3].

Multiple initiators can be connected to DRAMSys simultaneously and send requests to the DRAM subsystem independently. An initiator can either be a sophisticated processor model like the gem5 out of order processor model [4] or a more simple trace player that replays a trace file containing a sequence of memory requests with timestamps.
To support a variety of DRAM standards in a robust and error-free manner, DRAMSys uses a formal domain-specific language based on *Petri nets* called *DRAMml*. Using this language, all timing dependencies between DRAM commands of a standard can be defined. From this formal description, the source code of internal timing checkers is generated, which ensure compliance to the specific standard [14].
Since a single memory access can result in the issuance of multiple commands (e.g., a precharge (`PRE`), an activate (`ACT`), a read (`RD`) or a write (`WR`)), the four-phase handshake of the TLM-AT protocol is not sufficient to model the communication between the DRAM controller and the DRAM device. Therefore, a custom TLM protocol called DRAM-AT is used as the communication protocol between the channel controller and the DRAM device [2]. This custom protocol introduces a `BEGIN` and `END` phase for every available DRAM command. Which commands can be issued depends on the DRAM standard used.
Some of the internal modules of DRAMSys and their functionalities will now be explained. The task of the *arbiter* is to accept the incoming transactions from the various initiators and to decode the address according to the configured address mapping. From there, the transactions are passed to the channel controller.
The channel controller is the most important module of the DRAM simulation, consisting of a *scheduler*, *bank machines*, *refresh managers*, *power down managers*, a *response queue* and a *command multiplexer*. New incoming requests are placed into the scheduler. The purpose of the scheduler is to group transactions by their corresponding memory bank and reorder the payloads according to a predefined policy. Available policies are, for example, the *first-in, first-out* or the *first-ready - first-come, first-served* policy. The former policy does not reorder payloads and therefore optimizes for a short response

latency, whereas the latter policy reorders payloads and optimizes for a high memory bandwidth.
A bank machine, whose responsibility is to manage the state of its corresponding memory bank, then fetches the next transaction from the scheduler. There are also a number of available policies for the bank machines, each of which determine in which state the bank is being held after memory request is completed.
With the fetched transaction, the bank machine then selects the command that it needs to send to its memory bank. However, the selected command can not be sent instantaneously to the DRAM, as complex timing constraints need to be satisfied before the issuance of a specific command. To meet these timing requirements, the bank machine uses the so-called *timing checker* to check whether the selected command may be sent to memory. The bank machine then tries to enque the command, so that the controller can send it to the DRAM.
The task of the command multiplexer is to select one command out of all commands that have been enqueued by the bank machines, the refresh managers or the power down managers. The command multiplexer also has a set of configurable policies, that determine the individual priorities of the commands. The selected command is then sent to the DRAM by the controller.
The last important module to mention is the response queue. A completed DRAM transaction is enqueued into the response queue by the controller to send the responses back to the initiators. In the response queue, transactions can either be returned to the initiator according to the scheme *first-in, first-out* or be reordered in the queue. A reordering might be necessary to be able to support initiators that can not handle *out-of-order* responses.
DRAMSys also provides the so-called *Trace Analyzer*, a graphical tool that visualizes database files created by DRAMSys. An exemplary trace database, visualized in the Trace Analyzer, is shown in Figure 5.2. Furthermore, the Trace Analyzer is capable of calculating numerous metrics and creating plots of interesting characteristics.
In Section 6 of this thesis, a new simulation frontend for DRAMSys will be developed.

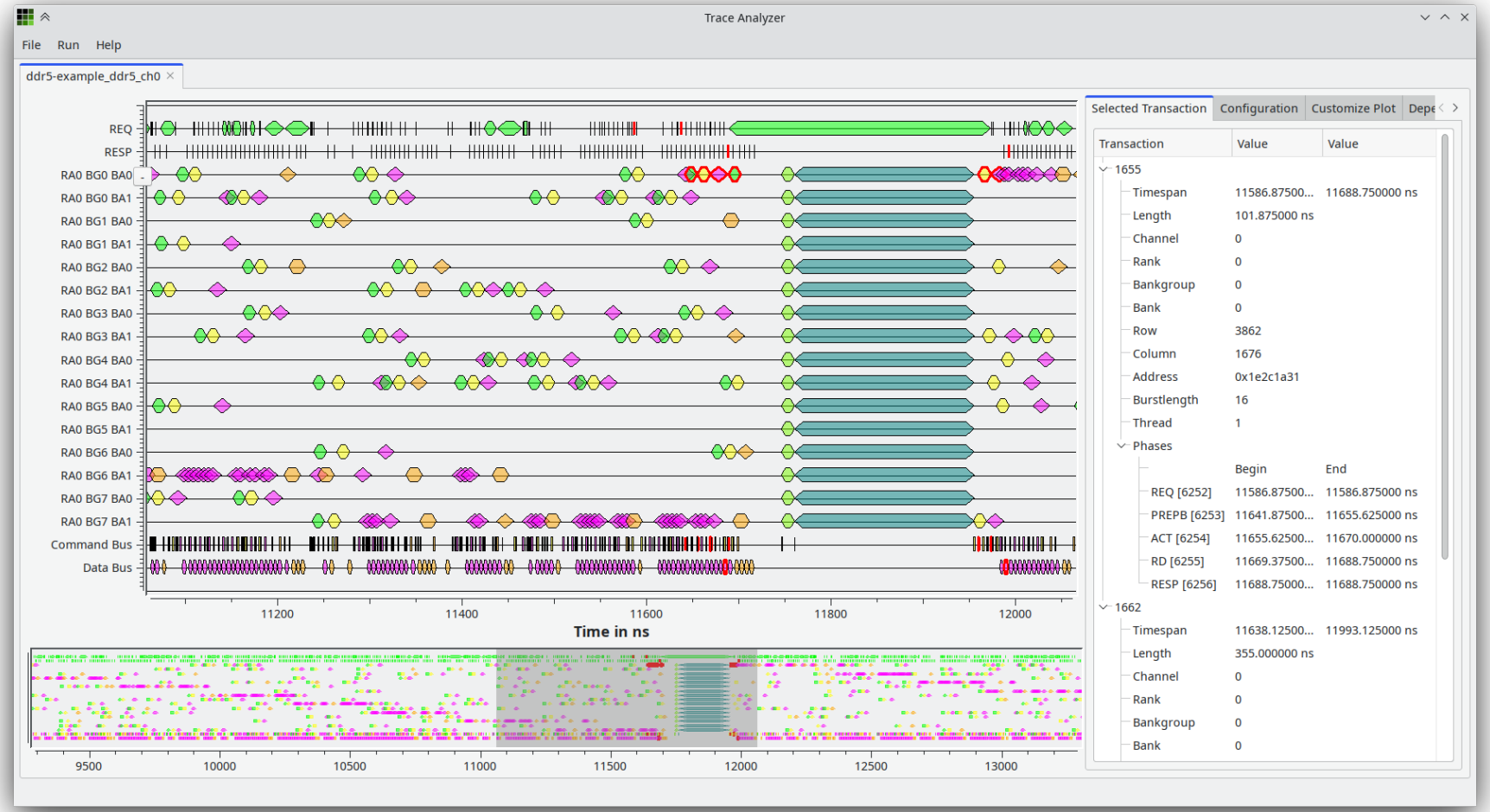

Figure 5.2: Exemplary visualization of a trace database in the Trace Analyzer. The used DRAM consists of one rank and eight bank groups with two banks each.

6 Implementation

In this section, the developed components for the new simulator frontend, which enable the tracing of an arbitrary application in real-time, as well as the replay of the recorded traces in DRAMSys, will be introduced.
To briefly summarize which components are necessary to implement the new simulation frontend, they are listed below:

- A DynamoRIO client that traces memory accesses from a running application.
- A simplified core model that replays those traces by sending transactions to DRAMSys.
- A cache model that simulates the cache filtering of memory requests of the processor.

The following sections will first explain the DynamoRIO analysis tool that generates the memory access traces and its place in the DrMemtrace framework. Furthermore, the new trace player for DRAMSys will acquire special attention as well as the mandatory cache model that is used to model the cache-filtering in a real system. The last part will concentrate on the special architecture of the new trace player interface and challenges, that the internal interconnection solves.

6.1 Analysis Tool

As described in Section 2, the dynamic binary instrumentation tool DynamoRIO will be used to trace the memory accesses while the target application is running. Instead of writing a DynamoRIO client from the ground up, the DrMemtrace framework, which comes bundled with DynamoRIO, is used.
DrCacheSim is a DynamoRIO client that builds on top of the DrMemtrace framework, which gathers memory and instruction access traces from the target application and forwards them to one or multiple analysis tools. In addition, so-called marker records are sent to the analysis tools when certain events occur, which are used to transmit meta information such as the CPU core used, kernel events or timestamps. These markers are also essential for a processor simulation, for example to reconstruct the thread interleaving, as it is intended for the new simulator frontend. DrCacheSim is a purely observational client and does not alter the behavior of the application.
Using one of many possible configuration parameters, it is possible for DrCacheSim to convert the addresses of the memory accesses from virtual addresses into the corresponding physical addresses, which is an important step for simulating a real memory system: As the virtual address space is unique for every running process and does not match the true address space of the processor, where the memory and also peripherals are mapped into, it needs to be translated to physical addresses by the operating system kernel to access the real memory. These physical addresses should be traced instead of the virtual addresses to account for effects such as paging in the simulated system.
It should be noted that in most systems the physical addresses do not directly represent the addresses that the memory subsystem perceives. The physical memory is mapped at a specific address region in the physical address space, so an address offset also has to be considered. On Linux systems, this mapping can be obtained by investigating the contents of the virtual file `/proc/iomem`, which is provided by the kernel. The trace player then substracts this offset as it will be explained in more detail in Section 6.3.

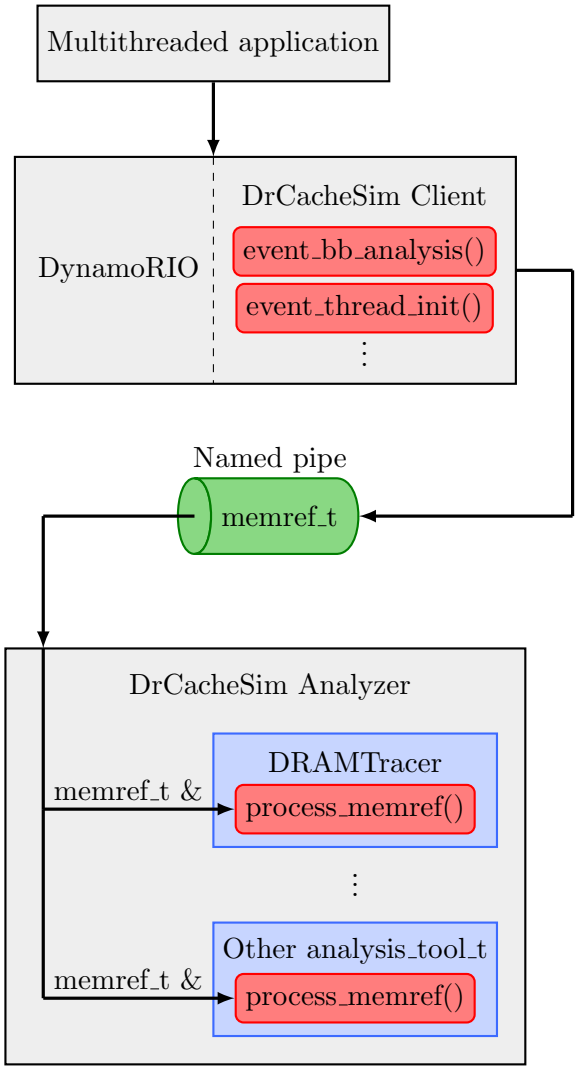

Figure 6.1: Structure of the DrCacheSim online tracing.

The physical address conversion only works on Linux and, in modern kernel versions, requires root privileges (or alternatively the CAP_SYS_ADMIN capability).
There are two different operation modes for an analyzer tool that DrCacheSim provides: The analyzer tool can either run alongside with DrCacheSim (online) or run after the target application has exited and operate on an internal trace format (offline). Offline tracing has the additional advantage of being able to disassemble the executed instructions afterwards. For this, the mapping of the executable binaries and shared libraries is stored alongside with the trace, enabling the decoding of the instructions from the traced program counter values. The instruction decoding is currently not natively supported by the online execution model, but this feature received limited attention in the development of the new frontend. As of writing this thesis, the offline tracing mode has only recently gained support for the physical address conversation. Nevertheless, the online execution model will be used throughout this thesis as the physical address support is still limited for offline tracing.
In the case of online tracing, DrCacheSim consists of two separate processes:

- A client-side process (the DynamoRIO client) which injects observational instructions into the application's code cache. For every instruction, memory access or marker event, a data packet of the type `memref_t` is generated and sent to the analyzer process.
- An analyzer-side process which is connected to the client and processes the `memref_t` data packets. The analyzer-side can contain many analysis tools that operate on this stream of records.

The inter-process communication (*IPC*) between the two processes is achieved through a *named pipe*. Figure 6.1 illustrates the structure of the online tracing mechanism.
A `memref_t` can either represent an instruction, a data reference or a metadata event such as a timestamp or a CPU identifier. Besides the type, the process identifier (*PID*) and thread identifier (*TID*) of the initiating process and thread is included in every record. For an instruction marker, the size of the instruction as well as the address of the instruction in the virtual address space of the application is provided. For data references, the address and size of the desired access is provided as well the program counter (*PC*) from where it was initiated. In offline mode, DrCacheSim stores the current mapping of all binary executables and shared libraries in a separate file, so that it is possible to decode and disassemble the traced instructions even after the application has exited. As mentioned earlier, instruction decoding is not natively supported for online tracing, but to work around the problem, the analyzer can examine the memory map of the client-side process and read the encoded instructions from there.
Using command line options, it is also possible to instruct DrCacheSim to trace only a portion of an application, rather than everyting from start to finish. This region of interest can be specified by the number of instructions after which the tracing should start or stop.
All analysis tools implement the common `analysis_tool_t` interface as this enables the analyzer to forward a received record to multiple tools in a polymorphic manner. In particular, the `process_memref_t()` method of any tool is called for every incoming record. Virtual functions, such as `initialize()` and `print_results()`, which are called by the analyzer in appropriate places, should also be implemented.
It is possible for an analysis tool to implement parallel processing of the received `memref_t` types by splitting up the trace into *shards*. However, in this thesis the sequential processing of a single sorted and interleaved trace is used because of missing support for parallel processing for the online execution model.
The newly developed DRAMTracer tool creates a separate trace file for every application thread. Since it is not known a priori how many threads an application will spawn, the tool will listen for records with new TIDs that it did not register yet. For every data reference, a new entry in the corresponding trace file is created which contains the size and the physical address of the access, whether it was a read or write, and also a count of (computational) instructions that have been executed since the last data reference. To compute the instruction count, a counter is incremented for every registered instruction record and reset again for any data reference. This instruction count is used together with the clock period to approximate the delay between two memory accesses when the trace is replayed by DRAMSys. Lastly, the analysis tool inserts a timestamp into the trace for every received timestamp marker. The use of this timestamp will be further explained in Section 6.3. Listing 6.1 presents an exemplary memory trace. Lines consisting of a number between two angle brackets represent a timestamp whereas lines for memory references consist of the instruction count, a character denoting a read or write, the size

```
# instruction count,read/write,data size,data address
# <timestamp>
<13300116157764414>
3,r,8,1190cf3f0
9,w,16,1190cf270
2,r,8,10200be48
0,w,16,1190cf280
1,w,16,1190cf290
2,w,16,1190cf2a0
1,w,16,1190cf2b0
0,w,16,1190cf2c0
```

Listing 6.1: Example of a memory access trace with a timestamp. For each thread, a seperate trace file is generated.

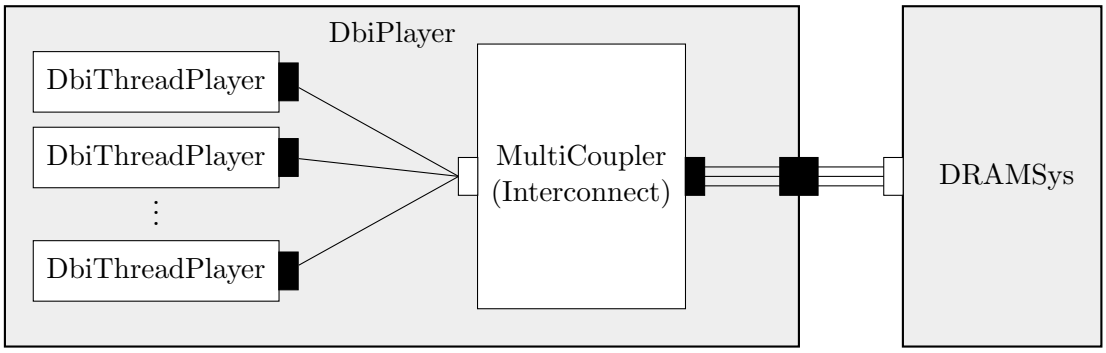

Figure 6.2: Architecture of the *DbiPlayer* without caches.

and the physical address of the access. Also, comments which are ignored by the trace player can be added by starting the line with a number sign.
As of writing this thesis, there is no application binary interface for analysis tools defined for the DrMemtrace framework. Therefore it is not possible to load the DRAMTracer tool as a shared library but rather it is required to modify the DynamoRIO source code to integrate the tool.

6.2 Trace Player Architecture

This section covers the general architecture of the *DbiPlayer*, the new trace player for DRAMSys that replays the captured trace files.
For every recorded thread, a traffic initiator thread, a so-called *DbiThreadPlayer*, is spawned, which is a standalone initiator for memory transactions. Because those threads need to be synchronized to approximate real thread interleaving, they need to communicate among each other. This communication, however, brings up the necessity to containerize the thread players into a single module that can directly be connected to DRAMSys. With the old DRAMSys interface for trace players, this was not easily realizable, so a new generic initiator interface was developed that allows components to be connected to DRAMSys whose internal architecture can be arbitrary. This new interface will be further discussed in Section 6.5.
For the *DbiPlayer*, an additional interconnect module will bundle up all `simple_initiator_sockets` into a single `multi_passthrough_initiator_socket`. So the *DbiPlayer* is a hierarchical module that consists of a more complex architecture with multiple traffic initiators, illustrated in Figure 6.2.

As the memory accesses are directly extracted from the executed instructions, simply sending a transaction to the DRAM subsystem for every data reference would completely neglect the caches of today's processors. Therefore, also a cache model is required whose implementation will be explained in more detail in Section 6.4. Many modern cache hierarchies are composed of three cache levels: two caches for every processor core, the L1 and L2 cache, and one cache that is shared across all cores, the L3 cache. This cache hierarchy is also reflected in the *DbiPlayer* shown in Figure 6.3, but also more simplistic hierarchies such as an L1 cache for every processor core and one shared L2 cache are configurable. In order to connect the different SystemC socket types, one additional interconnect is required which is explained in more detail in Section 6.6.

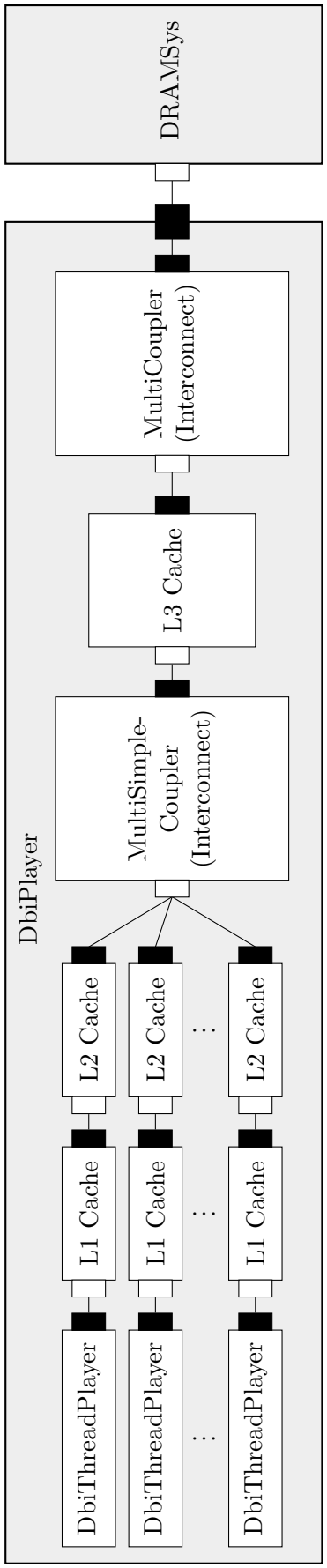

Figure 6.3: Architecture of the *DbiPlayer* with caches.

6.3 Trace Player Functionality

With the overall architecture of the main initiator module introduced, this section explains the internal functionality of the *DbiPlayer* and its threads.
The threads of the *DbiPlayer* are specialized initiator modules that inherit from the more generic `TrafficInitiatorThread` class. Each `TrafficInitiatorThread` consists of a `sendNextPayloadThread() SC_THREAD`, which in turn calls the virtual method `sendNextPayload()` each time the `sc_event_queue sendNextPayloadEvent` is being notified. `sendNextPayload()` is implemented in the `DbiThreadPlayer`.
Each `DbiThreadPlayer` iterates through the lines of its trace file and stores the entries in an internal buffer. In `sendNextPayload()`, a new generic payload object is created from the following entry of this buffer. The address of the payload is calculated from the physical address stored in the trace file entry. As previously discussed, the trace player now needs to account for the offset the RAM was placed at in the physical memory map and substract this offset from the physical address. The instruction count field of the trace is used to approximate the delay between two consecutive memory accesses: the count is multiplied with the trace player clock period to defer the initiation of the next transaction by the resulting value. Additionally, this count can be multiplied by an approximation of the average cycles per instruction (CPI) value. While this does not take into account the type of the instructions executed, it is still a simple approximation that can be used to model the system more accuratly.
The individual initator threads should not run by themselves without paying attention to the others; rather, they require synchronization to ensure the simulated system replicates the real running application as closely as possible. The analysis tool appends timestamps into the memory access traces. When such a timestamp is reached, it will be used to pause the execution of a thread if the global time has not yet reached this far, or to advance the global time when the thread is allowed to continue. Note that the term global time in this context does not correspond to the SystemC simulation time, but denotes a loose time variable that only the *DbiPlayer* uses to schedule its threads.
A set of rules determine if a thread is allowed to make progress beyond a timestamp that is greater than current global time:

1. The main thread at the start of the program is always allowed to run.
2. Threads do not suspend themselves when they would produce a deadlock. This is the case when they are the only thread currently running.
3. When a previously running thread exits and all other threads are suspended, then they will be resumed.
4. As a fallback, when currently all threads are suspended, one thread will be resumed.

Those rules reconstruct the thread interleaving of the instrumented application as it was running while being traced. The two latter rules ensure that always at least one thread is running so that the simulation does not come to a premature halt.

6.4 Non-Blocking Cache

This section gives an overview of the cache model that is used by the new trace player. It is implemented as a non-blocking cache that, as explained in Section 4.7, can accept new requests even when multiple cache misses are being handled.
The cache inherits from the `sc_module` base class and consists of a target socket to accept requests from the processor or a higher level cache as well as an initiator socket to send

requests to a lower level cache or to the DRAM subsystem. It has a configurable size, associativity, cache line size, MSHR buffer depth, write buffer depth and target depth for one MSHR entry.
To understand how the cache model works, a hypothetical request from the CPU will be assumed to explain the internal processing of the transaction in detail:
When the transaction arrives, it will be placed in the PEQ of the cache, from where, after the specified amount of delay has elapsed, the handler for the `BEGIN_REQ` phase is called. The handler verifies that the cache buffers are not full[1] and checks if the requested data is stored in the cache. If it is the case (i.e., a cache hit), the cache model sends immediately an `END_REQ` and, when the target socket is not currently occupied with a response, accesses the cache and sends the `BEGIN_RESP` phase to the processor. During a cache access, the content of the cache line is copied into the transaction in case of a read request, or the cache line is updated with the new value in case of a write request. Further, in both cases the timestamp of the last access is updated to the current simulation time. The processor then finalizes the transaction with the `END_RESP` phase, the target backpressure of the cache will be cleared as the postponed request from the CPU (if it exists) is placed into the PEQ once again.
If, on the other hand, the requested data is not in the cache (i.e., a cache miss), first it will be checked if there is already an existing MSHR entry for the corresponding cache line. If this is the case[2], the transaction is appended to it as an additional target. If not, a cache line is evicted to make space for the new cache line that will be fetched from the underlying memory. The cache model implements the optimal replacement policy LRU, so the cache line with the last access time, which lies furthest back in the past, is chosen to be evicted. When an eviction is not possible, the transaction will be postponed. An eviction is not possible when the selected cache line is allocated but not yet filled with requested data from the underlying cache, the cache line is currently present in the MSHR queue, or a hit for this cache line is yet to be handled. When the *dirty* flag of the old cache line is set, it has to be placed into the write buffer and written back to the memory. The newly evicted cache line is now *allocated*, but not *valid*. Then, the transaction is put into an MSHR entry and the `END_REQ` phase is sent back to the processor.
To process the entries in the MSHR and in the write buffer, the `processMshrQueue()` and `processWriteBuffer()` methods are called at appropriate times. In the former, a not yet issued MSHR entry is selected for which a new fetch transaction is generated and sent to the underlying memory. Note that special care has to be taken when the requested cache line is also present in the write buffer: To ensure consistency, no new request is sent to the DRAM and instead the value is snooped out of the write buffer. Since the cache line in the write buffer is now allocated again in the cache, the entry in the write buffer can be removed to prevent an unnecessary write-back. In the latter, the processing of the write back buffer, a not yet issued entry is selected and a new write transaction is sent to the memory.[3]
Incoming transactions from the memory side are accepted with an `END_RESP` and, in case of a fetch transaction, used to update the cache contents and possibly preparing a new response transaction for the processor as described before.
This example works analogously with another cache as the requesting module or another cache as the target module for a fetch or write back accesses.

[1] Otherwise the cache will apply backpressure on the CPU and postpone the handling of the transaction.
[2] And if the target list of the MSHR entry is not full. Otherwise the transaction will be postponed.
[3] Both `processMshrQueue()` and `processWriteBuffer()` also need to ensure that currently no backpressure is applied onto the cache from the memory.

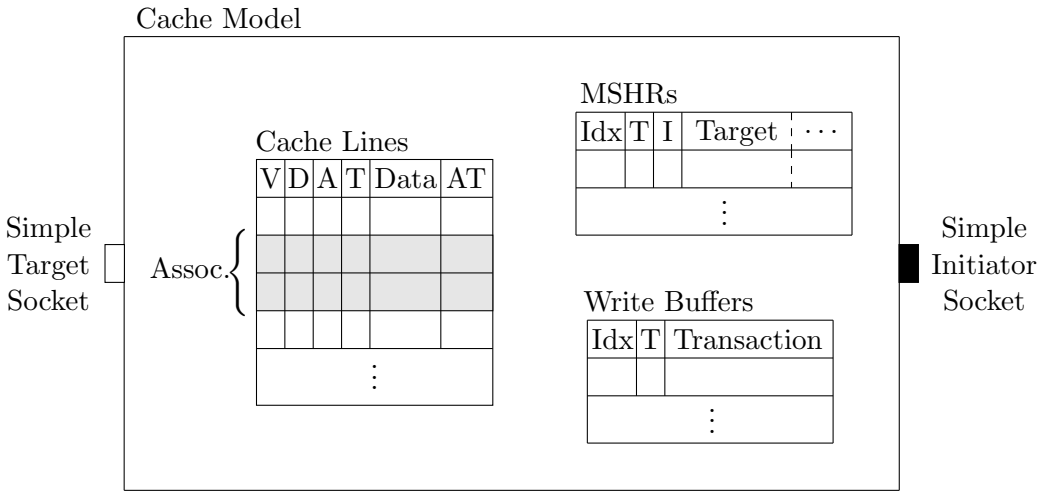

Figure 6.4: Internal architecture of the cache model. *V* stands for *valid*, *D* for *dirty*, *A* for *allocated*, *T* for *tag*, *AT* for *access time*, *I* for *issued* and *Idx* for *index*. In the cache line array, adjacent lines with the same addressing index are colored in the same gray shade. The size of such a group is the *associativity*.

The rough internal structure of the cache model is shown again in Figure 6.4.
It is to note that the current implementation does not utilize a snooping protocol. Therefore, cache coherency is not guaranteed and memory shared between multiple processor cores will result in incorrect results as the values are not synchronized between the caches. However, it is to expect that this will not drastically affect the simulation results for applications with few shared resources.

6.5 Trace Player Interface

Previously, initiators could only represent one thread when they were connected to DRAMSys. This, however, conflicted with the goal to develop a trace player module that is internally composed of multiple threads, which communicate with each other and initiate transactions to DRAMSys independently.
To be able to couple such hierarchical initiator modules with DRAMSys, a new trace player interface was developed. The `TrafficInitiatorIF` is a generic interface that every module that connectes to DRAMSys needs to implement. It requires to implement the `bindTargetSocket()` method so that top-level initiators can be coupled regardless of the used initiator socket type (e.g., `simple_initiator_socket` or `multi_passthrough_initiator_socket`).
When coupling a `multi_passthrough_initiator_socket` to a `multi_passthrough_target_socket`, the SystemC `bind()` method has to be called multiple times - once for each thread. Because of this, a wrapper module also has to overwrite the `getNumberOfThreads()` method of the new interface and use this number to bind the target socket in `bindTargetSocket()` the correct number of times.
This makes it possible to polymorphically treat all initiator modules as this interface, whether they are simple threads or more complex wrapper modules, and connect them to DRAMSys with the provided bind method, abstracting away the concrete type of initiator socket used.

With the new trace player interface, a top-level initiator can either be a single thread, like in previous versions, or a more complex hierarchical module with many internal threads.

6.6 Interconnect

As already seen in Figure 6.3, interconnection modules are needed to connect the caches to each other. While the implementation of the *MultiCoupler* component is trivial as it only passes the transactions from its so-called `multi_passthrough_target_socket` to its `multi_passthrough_initiator_socket`, the *MultiSimpleCoupler* is more complex because it has to internally buffer transactions.
In order to understand why this buffering is needed, consider the scenario where the L3 cache applies backpressure to one L2 cache. The L2 cache is not allowed to send further requests due to the exclusion rule. But since the target socket of the L3 cache is occupied, this also applies to all other L2 caches. This information, however, is not propagated to the other caches, leading to an incorrect behavior if not addressed, as the other caches will send further requests.
To solve this problem, the MultiSimpleCoupler only forwards requests to the L3 cache when it is able to accept them. If this is not the case, the request is internally buffered and forwarded when an earlier request is being completed with the `END_REQ` phase.
For illustrating this further, a simple example can be assumed: one L2 cache needs to request a cache line from the underlying L3 cache. The MultiSimpleCoupler receives the `BEGIN_REQ` phase and places it into its PEQ. From there, a hash table used as an internal routing table is updated to be able to send the response back through the correct multi-socket binding afterwards. As the L3 cache is currently not applying backpressure onto the interconnect, it can forward the transaction with the `BEGIN_REQ` phase to the L3 cache. Until the L3 cache responds with the `END_REQ` phase, the interconnect defers any new request from any L2 cache and buffers the payload objects in an internal data structure. When the `END_REQ` phase is received, the next transaction from this request buffer is sent to the L3 cache. After some time, the L3 cache will respond with the requested cache lines. During this `BEGIN_RESP` phase, the L2 cache that requested this line is looked up using the routing table and the payload is sent back to it. Until the L2 cache responds with an `END_RESP`, the exclusion rule also has to be honored here: when a new response from the L3 cache is received, it has to be buffered in another internal data structure until the corresponding target socket binding is clear again. Once the L2 cache sends out the `END_RESP` phase, the interconnect will forward the `END_RESP` to the L3 cache, and initiate new response transactions in case the response buffer is not empty.
In conclusion, this special interconnect module with a multi-target socket and a simple-initiator socket ensures that the exclusion rule is respected in both directions.

7 Simulation Results

This section evaluates the accuracy of the new simulation frontend. After a short discussion about the general expections regarding the accuracy and considerations to make, the simulation results will be presented. The presentation is structured into two parts: At first simulation statistics of numerous benchmarks are compared against the gem5 [4] simulator, which uses detailed processor models and can be considered as a reference. Secondly, the new simulation frontend is compared against the memory access trace generator tool of the Ramulator DRAM simulator [15].

7.1 Accuracy

Generating memory access traces using dynamic binary instrumentation as a faster alternative to the simulation of detailed processor models introduces several inaccuracies, which of some will now be enumerated.
The most important aspect to consider is that DBI can only instrument the target application but fails to also take the operating system the application is running on into account. That includes the inability to observe the execution of kernel routines that are directly invoked by the application through system calls, but also the preemptive scheduling of other programs that are running on the system at the same time.
The fetching of the instructions themselves should also be considered: In a real system the binary executable of the target application is placed in the DRAM, along with its data, and is loaded into the instruction cache while executing. Since the DBI cannot observe the fetching of those instructions, the new simulator frontend cannot model this memory traffic.

7.2 Comparison to the gem5 Simulator

At first, the micro-benchmark suite TheBandwithBenchmark [16], consisting of various streaming kernels, will be used to compare the gem5 full-system simulation as well as the gem5 syscall-emulation simulation with the newly developed frontend.
The gem5 syscall-emulation does not simulate a whole operating system, rather it utilizes the host system's Linux kernel and therefore only simulates the binary application. In contrast, the gem5 full-system simulation boots into a complete Linux system including all processes, that may run in the background. Therefore, syscall-emulation is conceptually closer to the DynamoRIO approach than full-system simulation.
In both cases, the simulation setup consists of a two-level cache hierarchy with the following parameters:

Table 7.1: Cache parameters used in simulations.

Cache	Size	Associativity	Line size	MSHRs	MSHR targets	WB entries
L1	32 kiB	8	64	4	20	8
L2	256 kiB	4	64	20	12	8

In this configuration, every processor core has its own L1 data cache (in case of gem5 also a L1 instruction cache) whereas the L2 cache is shared between all cores. The gem5 simulator uses four ARM CPU core models (TimingSimpleCPU, an in-order core model) at *1000 MHz*, whereas the DynamoRIO traces are obtained using a QEMU [17]

ARM virtual machine, configured to use four cores as well. The DRAM subsystem will be varied between a single-channel DDR3 memory (1600 MT/s) and a single-channel DDR4 memory (2400 MT/s). To match the same configuration as in gem5, the memory controller in DRAMSys is set to use a first-ready - first-come, first-served (*FR-FCFS*) scheduling policy, a first-in, first-out (*FIFO*) response queue policy, and a row-rank-bank-column-channel address mapping (explained in more detail in Appendix 9.1). The trace player operates at the same clock frequency as the gem5 core models.
It is important to configure the CPI value of the new trace player to a sensible value to approximate the delay between two consecutive memory accesses. For the simulations, the CPI value that gem5 SE reports in its statistics is used. It has been found that the CPI results in an approximate value of *10* if only computation instructions are considered and load and store operations are ignored, since those are affected by the latency of the memory subsystem.
The micro-benchmarks itself are multi-threaded and make use of all available cores. Furthermore, the compiler optimizations are set to `-Ofast` for all benchmarks. Their access patterns are as followed:

Table 7.2: Access patterns of the micro-benchmark kernels [16].

Kernel	Description	Access Pattern
INIT	Initialize an array	a = s (store, write allocate)
SUM	Vector reduction	s += a (load)
COPY	Memory copy	a = b (load, store, write allocate)
UPDATE	Update vector	a = a * scalar (load, store)
TRIAD	Stream triad	a = b + c * scalar (load, store, write allocate)
DAXPY	Daxpy	a = a + b * scalar (load, store)
STRIAD	Schönauer triad	a = b + c * d (load, store, write allocate)
SDAXPY	Schönauer triad	a = a + b * c (load, store)

In the following, the simulation results of the new simulation frontend, the gem5 full-system emulation and the gem5 syscall-emulation are now presented.

Table 7.3: Results for bandwidth and bytes read/written with DDR4-2400. *FS* denotes gem5 full-system, *SE* denotes gem5 syscall-emulation, *DS* denotes DRAMSys.

Benchmark	Avg. Bandwidth [GB/s]			Bytes Read [MB]			Bytes Written [MB]		
	FS	SE	DS	FS	SE	DS	FS	SE	DS
COPY	2.201	2.794	2.130	238.4	268.8	307.8	140.2	134.3	134.4
DAXPY	2.157	2.721	1.600	238.2	268.8	302.0	140.2	134.4	134.4
INIT	2.058	2.737	2.040	141.9	172.6	216.1	140.0	134.1	134.4
SDAXPY	2.239	2.813	2.080	335.1	364.8	403.0	140.3	134.4	134.4
STRIAD	2.246	2.803	2.350	335.1	460.9	494.4	140.4	134.4	134.4
SUM	1.429	1.982	1.110	142.0	172.7	189.1	44.0	38.4	38.5
TRIAD	2.246	2.853	2.110	335.1	364.9	402.6	140.4	134.4	134.4
UPDATE	1.995	2.611	1.430	142.0	172.7	220.0	140.1	134.2	134.4

Listed in Table 7.3 are three key parameters, specifically the average memory bandwidth and the number of bytes that has been read or written for the DDR4-2400 configuration. The results show that all parameters of DRAMSys correlate well with the gem5 statistics. While for the average bandwidth the DynamoRIO results are on average 31.0% lower compared to gem5 SE, this deviation is only 11.1% for gem5 FS. The numbers for the total amount of bytes read result in a deviation of 35.5% in comparison to gem5 FS and only to 14.6% to gem5 SE. The amount of bytes written, on the other hand, shows a very small deviation of 5.2% for gem5 FS and only 0.07% for gem5 SE. Therefore, it can be stated that almost the same number of bytes were written back to the DRAM due to cache write-backs.
Those numbers are also illustrated in Figure 7.1.

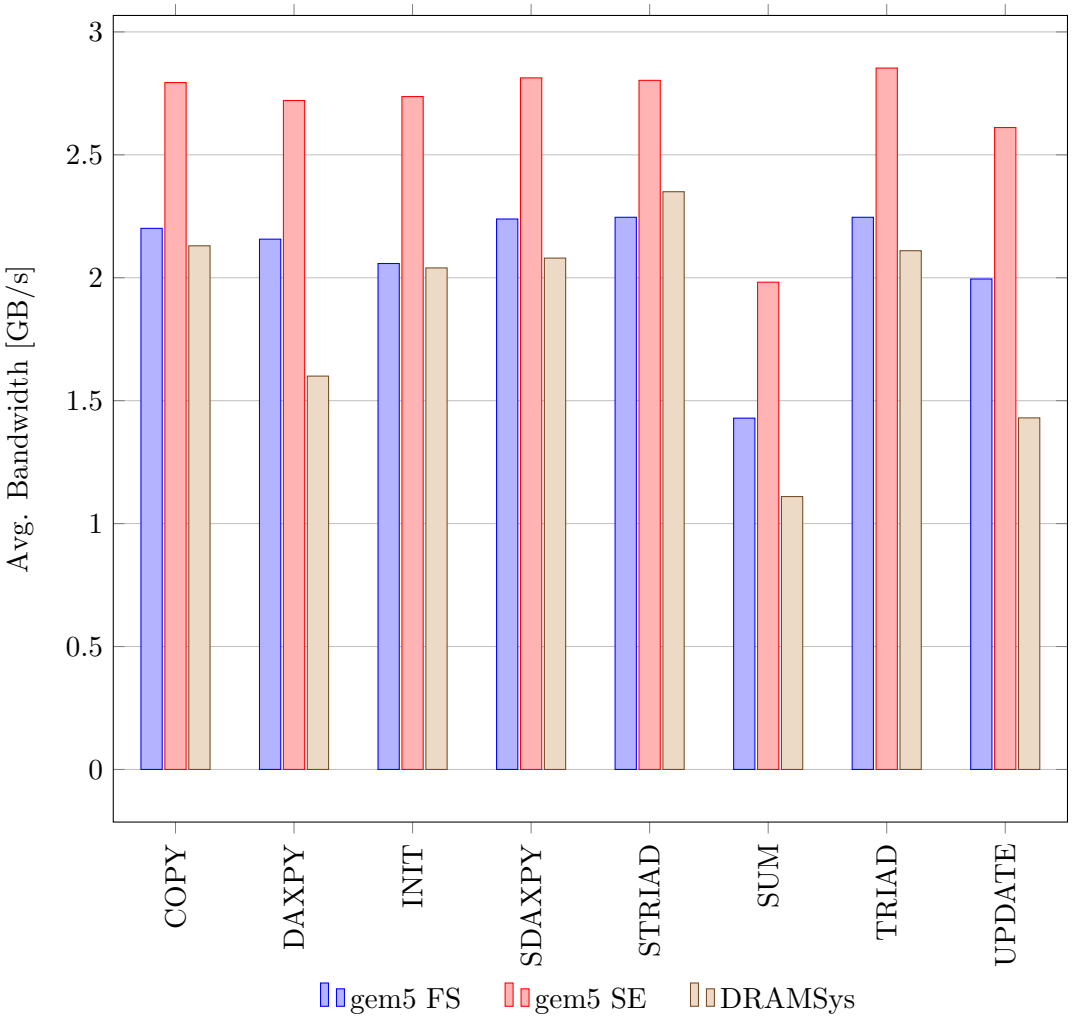

Figure 7.1: Average Bandwidth with DDR4-2400.

Table 7.4 and Figure 7.2 show those same key parameters for the DDR3 configuration. Here, the absolute deviations in the average memory bandwidth amount to 27.5% and 7.0% for gem5 SE and gem5 FS respectively. The differences for the amount of bytes read result to 31.6% for gem5 FS and to 14.7% to gem5 SE. Also here, the bytes written only show small deviations of 5.2% for gem5 FS and 0.02% for gem5 SE.

Table 7.4: Results for bandwidth and bytes read/written with DDR3-1600.

Benchmark	Avg. Bandwidth [GB/s]			Bytes Read [MB]			Bytes Written [MB]		
	FS	SE	DS	FS	SE	DS	FS	SE	DS
COPY	2.031	2.698	2.160	238.3	268.8	310.1	140.3	134.3	134.4
DAXPY	2.070	2.627	1.610	238.2	268.9	301.9	140.2	134.4	134.4
INIT	2.028	2.629	2.070	141.9	172.9	216.0	140.1	134.4	134.4
SDAXPY	2.101	2.755	2.110	335.1	364.8	404.0	140.4	134.4	134.4
STRIAD	2.228	2.613	2.370	431.6	460.9	494.7	140.4	134.4	134.4
SUM	1.393	1.969	1.120	142.0	172.9	189.1	44.1	38.5	38.5
TRIAD	2.162	2.725	2.140	335.1	364.9	403.8	140.4	134.4	134.4
UPDATE	1.938	2.528	1.430	142.0	172.8	220.0	140.1	134.3	134.4

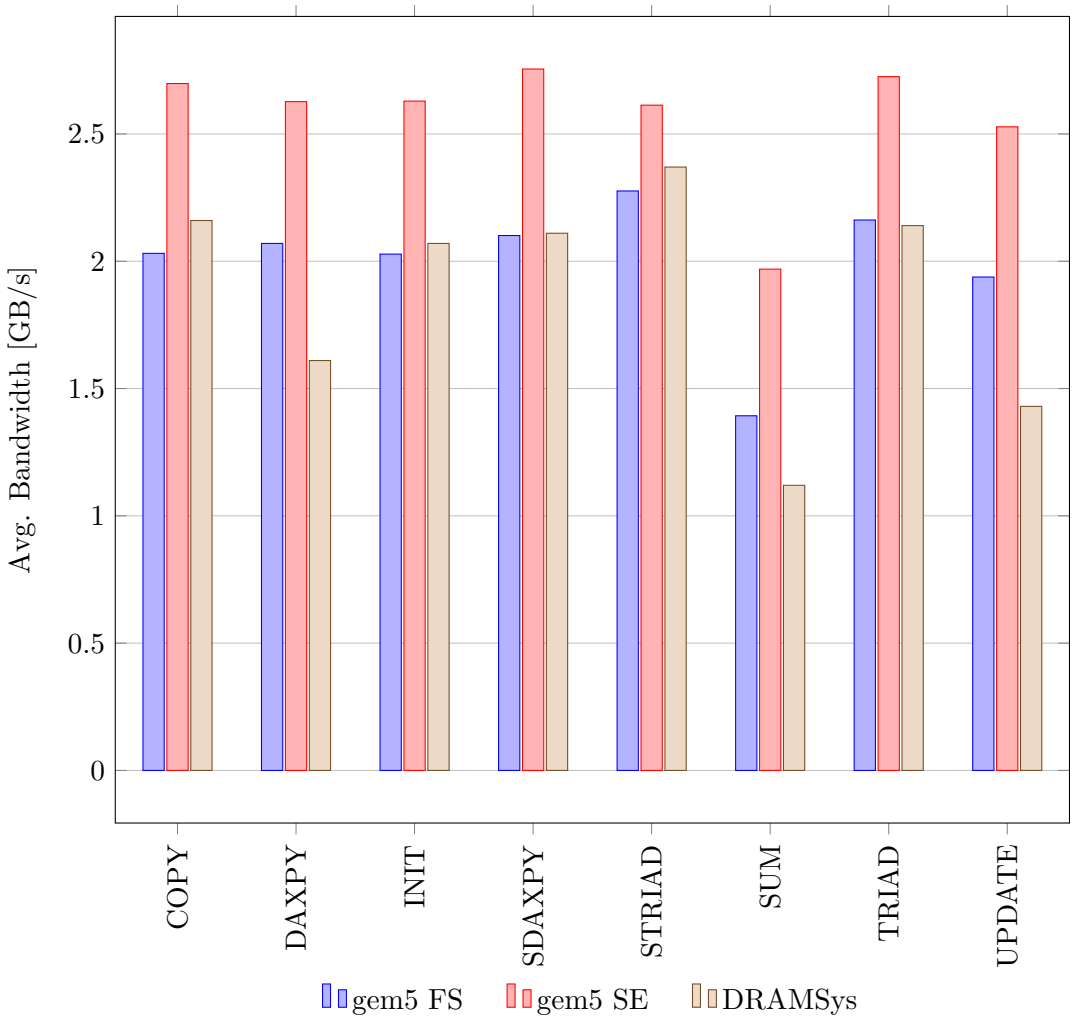

Figure 7.2: Average Bandwidth with DDR3-1600.

Another important metric in the evaluation of a memory subsystem is the average response latency of a memory access. In Figure 7.3, the average latencies of the DRAM are illustrated for the DDR4-2400 configuration.

While the latencies reported by DRAMSys are always higher for the respective benchmark, it averages to a deviation of 36.0% in comparison to gem5 SE and to 24.9% to gem5 FS.
Those numbers can be looked up in greater detail in Table 9.6 for the DDR3-1600 and in Table 9.7 for the DDR4-2400 configuration. These tables also provide information about the simulation time of the different benchmarks.

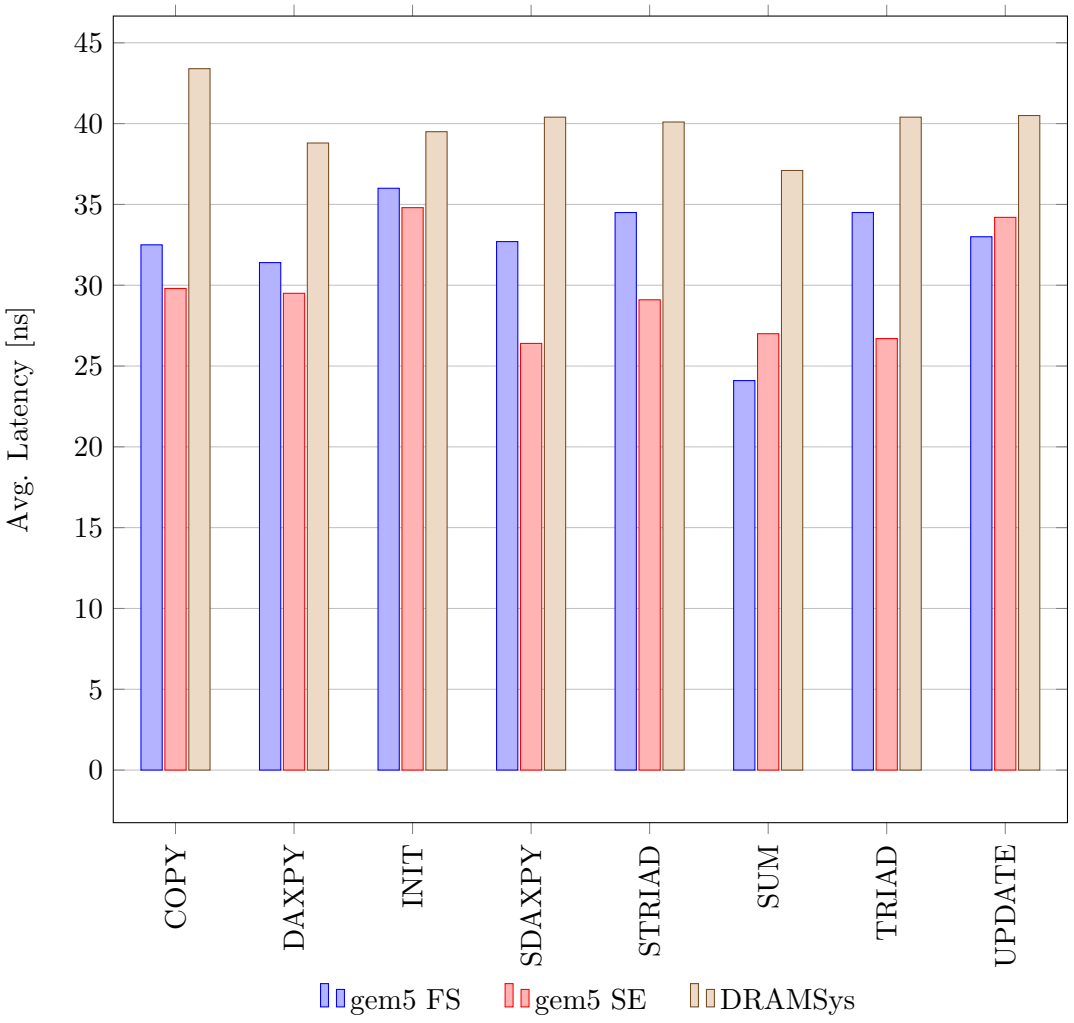

Figure 7.3: Average response latency with DDR4-2400.

In order to compare not only the total average bandwidth, but also its behavior over time, all benchmarks were run consecutively on gem5 SE and DRAMSys and plotted as a bandwidth-time diagram in Figure 7.4.
Similar to the previous comparisons, the average bandwidth of DRAMSys is marginally lower than gem5. Furthermore, an increased fluctuation around a bandwidth value can be observed. However, the overall time behavior is the same: The highest bandwidth value is reached at the beginning of the simulation and the bandwidth drops to a low plateau at the end of the simulation.

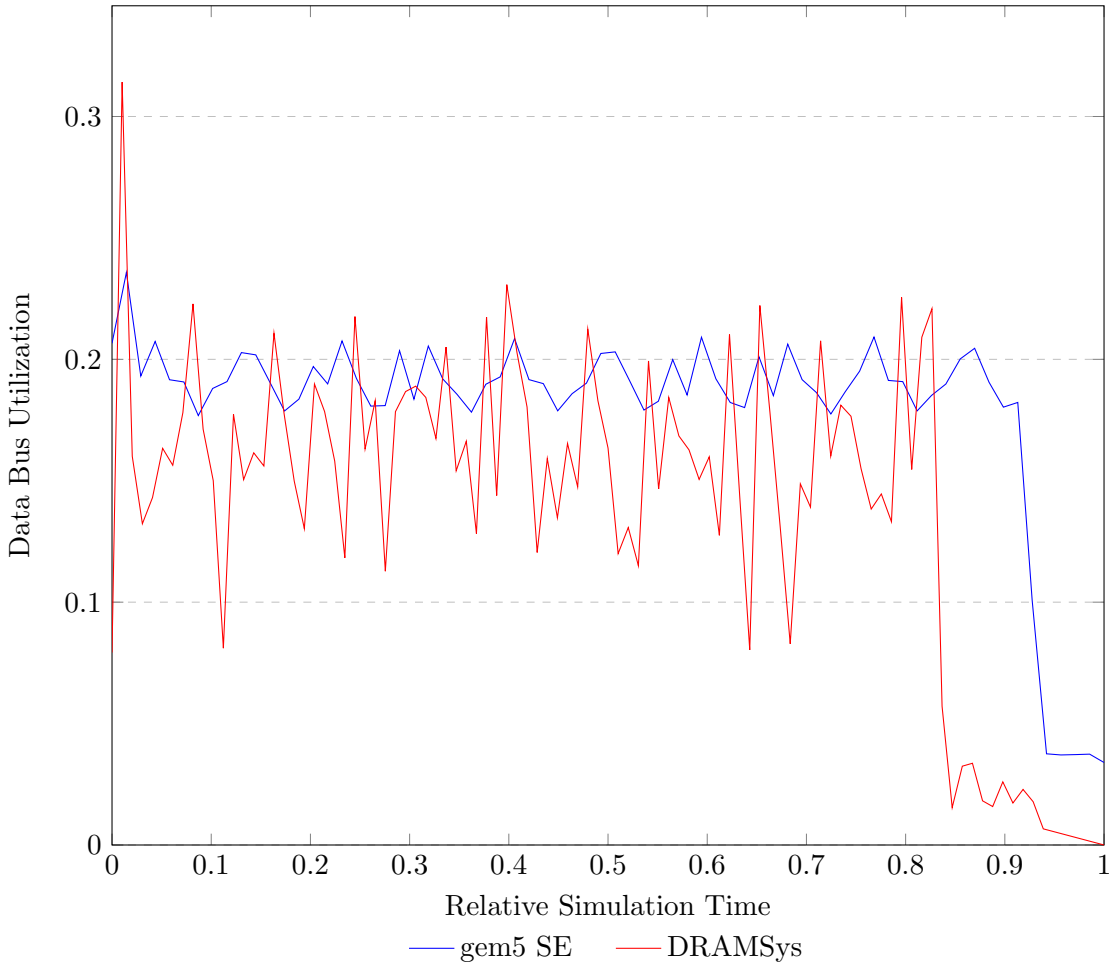

Figure 7.4: Data bus utilization over the simulation time with DDR4-2400.

The mean absolute percentage error ($MAPE$) is used to evaluate the deviations of the two bandwidth curves. With the total number of data points n, the individual actual values A_t and the individual forecasted values F_t, the measure is defined as follows:

$$MAPE = \frac{1}{n} \sum_{t=1}^{n} \left| \frac{A_t - F_t}{A_t} \right| \tag{7.1}$$

For the bandwidth simulations, the resulting MAPE value is 22.3%.

7.3 Comparison to Ramulator

In order to evaluate the new simulation frontend with a simulator that uses a similar approach, the benchmarks are compared with Ramulator in this section. This approach is also based on DBI, more specifically Ramulator uses the Intel Pin-Tool to create a memory access trace of a running application. Here, the cache filtering takes place when the trace is created instead of while the trace is played back by the simulator. This means that the simulation of the cache cannot take into account the feedback from the DRAM system and therefore the latencies of the cache are neglected. Ramulator also uses the count of computational instructions to approximate the delay between two

memory accesses. Since Ramulator uses a CPI value of *4* by default, this is also the value that DRAMSys is configured with.
The cache configuration remains the same as in the gem5 simulations, and the simulation is also performed again with a DDR3-1600 and DDR4-2400 configuration. However, the address mapping has changed, namely to a row-bank-rank-column-channel address mapping with only one rank and channel respectively. The exact configuration is listed in Section 9.1.
In contrast to the previous simulations, the benchmarks are now single-threaded.

Table 7.5: Results for bandwidth and bytes read/written with DDR3-1600.

Benchmark	Avg. Bandwidth [GB/s]		Avg. Latency [ns]	
	Ramulator	DRAMSys	Ramulator	DRAMSys
COPY	3.053	2.930	66.7	74.4
DAXPY	3.049	2.940	66.7	54.9
INIT	3.063	2.760	66.6	63.8
SDAXPY	3.047	2.840	66.3	60.9
STRIAD	3.058	3.180	66.6	63.8
SUM	3.039	2.650	66.6	56.3
TRIAD	3.057	3.310	66.9	57.4
UPDATE	3.064	2.480	66.5	64.1

Table 7.6: Results for bandwidth and bytes read/written with DDR4-2400.

Benchmark	Avg. Bandwidth [GB/s]		Avg. Latency [ns]	
	Ramulator	DRAMSys	Ramulator	DRAMSys
COPY	3.462	3.740	74.9	46.7
DAXPY	3.454	3.240	74.9	43.0
INIT	3.480	3.340	74.5	48.9
SDAXPY	3.475	3.430	74.1	39.5
STRIAD	3.490	3.830	74.1	41.7
SUM	3.496	3.040	73.7	38.0
TRIAD	3.468	4.210	75.1	35.8
UPDATE	3.478	3.130	74.6	43.6

In Tables 7.5 and 7.6, it can be seen that the average memory bandwith of Ramulator matches well with the results of DRAMSys. On average, the absolute deviation is about 19.1% for the DDR4 simulation, whereas it only amounts to about 10.0% for the DDR3 configuration. The differences in the average access latency equal to 41.5% and 3.6% for the DDR4 and DDR3 simulations, respectively.
One noticeable aspect is that with Ramulator, the latencies are greater with DDR4 than with DDR3. In the DRAMSys configuration, this is the opposite case. A possible explanation could be that Ramulator, as already mentioned, cannot take into account the feedback from the memory system during cache filtering and therefore deviations can occur.

7.4 Simulation Runtime Analysis

The last topic for comparison is to analyze the speed increase (i.e., the reduction in *wall clock time*) by using the new simulation frontend compared to a detailed processor simulation.
For this DRAMSys is again compared with gem5 SE and FS. A comparison with Ramulator would not be meaningful, because the cache filtering takes place at different times: while with Ramulator the trace generation takes longer than with DynamoRIO, the simulation itself is faster. The database recording feature of DRAMSys is also disabled for these measurements, since the additional file system accesses for this functionality severely degrade the simulator's performance.
Figure 7.5 presents the runtimes of the various benchmarks and simulators.

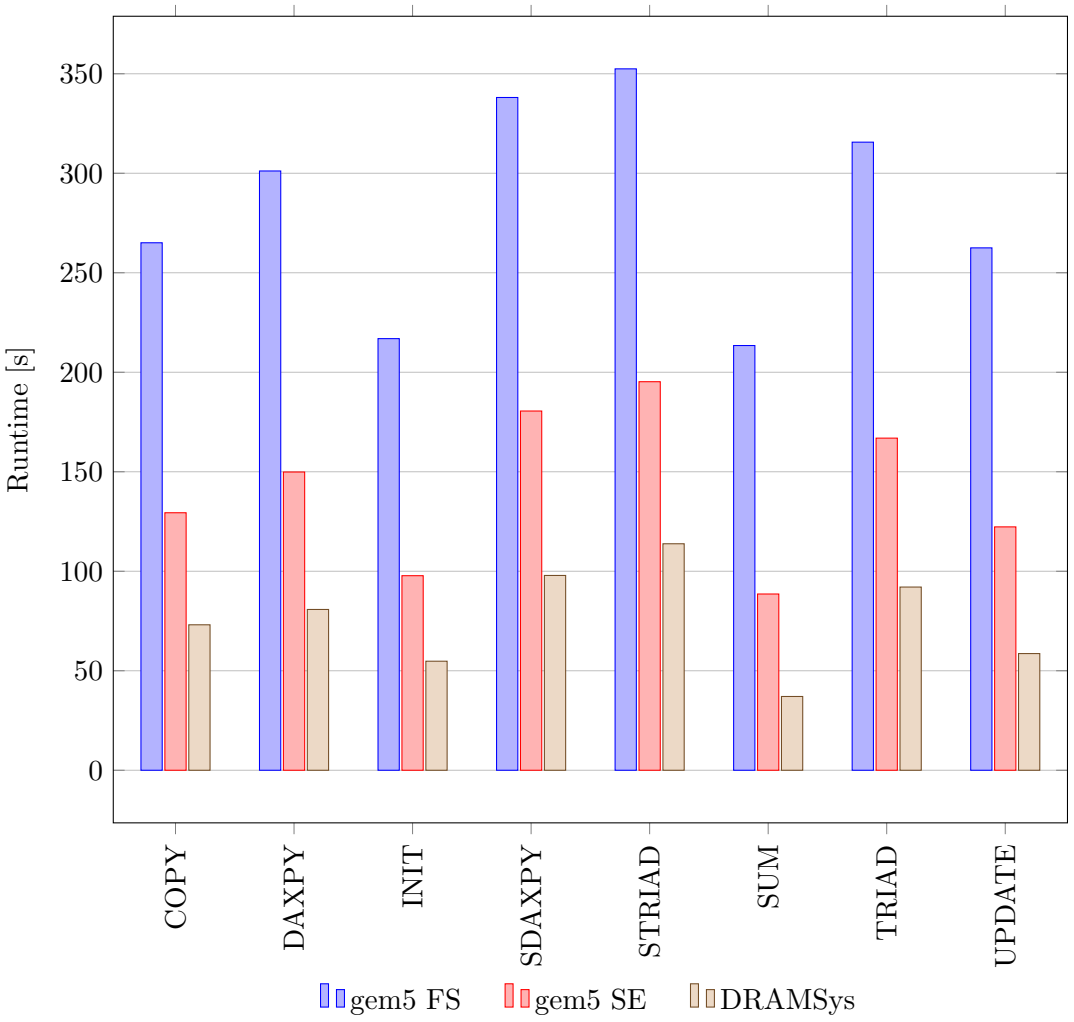

Figure 7.5: Runtimes for the utilized benchmarks with DDR4-2400.

As expected, DRAMSys outperforms the gem5 full-system and syscall-emulation simulators in every case. On average, DRAMSys is 47.0% faster than gem5 SE and 73.7% faster than gem5 FS, with a maximum speedup of 82.6% for the benchmark `SUM`. While gem5 SE only simulates the target application using the detailed processor model, gem5 FS has to simulate the complete operating system kernel and applications, that run in the background concurrently. However, the bootup process of the operating system was

not included in the simulations. These conceptual differences explains the large runtime deviations between the two simulation modes.

8 Conclusion and Future Work

Due to the complexity of possible memory subsystem configurations, simulation is an indispensable part of the development process of today's systems. It not only has a high impact on the development cost but also significantly reduces the time-to-market and enables the rapid release of new products. However, the accurate simulation of a specific application takes a large period of time because of the detailed processor core models. On the other hand, fixed or relative time memory traces allow faster simulation at the expense of accuracy, which makes them often unsuitable. To fill this gap, this thesis introduced a new simulation frontend for DRAMSys, which fastens the process while only making few compromises on accuracy.
In conclusion, the newly developed instrumentation tool provides a flexible way of generating traces for arbitrary multi-threaded applications. The mature DRAMSys simulator framework then can be used to explore the design space and vary numerous configuration parameters of the DRAM subsystem to find a well-suited set of options.
It was shown that in comparison to the well-established full-system simulation framework gem5, only some deviations have to be accepted. Also, the Pin-Tool based memory access tracing of the Ramulator DRAM simulator was compared to the new frontend. Although Ramulator takes a slightly different approach to trace generation than this thesis, a very good correlation in the results could be demonstrated. A noteworthy advantage of the newly developed tool is its support for all hardware architectures that DynamoRIO provides (currently IA-32, x86-64, ARM, and AArch64) in contrast to the supported architectures of Pin (IA-32 and x86-64).
Still, there is room for improvement. To improve the simulation runtime, a binary trace format could be used instead of the text-based format. Both the performance during tracing and parsing should increase by using such a binary format.
As mentioned in 6.4, the cache models do not yet guarantee cache coherency due to the lack of a snooping protocol. Although this can be a complex task, it is possible to implement this in future work.
A less impactful inaccuracy results from the scheduling of the applications threads in the new simplified core models. While an application can spawn an arbitrary number of threads, the platform may not be able to process them all in parallel. Currently, the new trace player does not take this into account and runs all threads in parallel. This deviation could be prevented by recording used processor cores on the initial system and using this information to better match the scheduling.
Another inaccuracy can be caused by the hyperthreading of some of today's processors: While hyperthreading enables the parallel processing of two pipelines in a processor core, those threads do share the same first level cache. Currently, this is not taken into account, and each application thread is assigned its own first level cache.
Further room for improvement offers the consideration of the special prefetch and instructions the architectures provide. DynamoRIO already offers an interface to catch those instructions without much effort. Support for this would have to be added to the core and cache models as well as the memory trace format.
The recorded number of computational instructions between each memory access, which are used to esimate the time between those accesses, is multiplied with the clock period of the trace player. However, this is a vast simplification of the real timing behavior of a processor. In the future, the DynamoRIO tool could decode those computational instructions and create a better estimate of the execution time of those instructions, based on statistical estimates that have been published before [18, 19].

One significant improvement that still could be applied is the consideration of dependencies between the memory accesses. Similarily to the elastic trace player of gem5 [20], which captures data load and store dependencies by instrumenting a detailed out-of-order processor model, the DynamoRIO tool could create a dependency graph of the memory accesses using the decoded instructions. By using this technique, it is possible to also model out-of-order behavior of modern processors and make the simulation more accurate, whereas the current implementation is entirely in-order.

9 Appendix

9.1 Simulation Address Mappings

Table 9.1: Memory configuration used in comparison simulations against gem5.

DRAM	Ranks per Channel	Banks per Rank	Rows	Columns	Devices per Rank	Width
DDR3	2	8	65536	1024	8	8
DDR4	2	16	65536	1024	8	8

Table 9.2: Address mappings used in comparison simulations against gem5.

DRAM	Byte	Column	Bankgroup	Bank	Rank	Row
DDR3	0-2	3-12	-	13-15	16	17-32
DDR4	0-2	3-12	13-14	15-16	17	18-33

Table 9.3: Memory configuration used in comparison simulations against Ramulator.

DRAM	Ranks per Channel	Banks per Rank	Rows	Columns	Devices per Rank	Width
DDR3	1	8	32768	1024	8	8
DDR4	1	16	32768	1024	8	8

Table 9.4: Address mappings used in comparison simulations against Ramulator.

DRAM	Byte	Column	Bankgroup	Bank	Rank	Row
DDR3	0-2	3-12	-	13-15	-	16-30
DDR4	0-2	3-12	13-14	15-16	-	17-31

9.2 Simulation Results

Table 9.5: Last-level cache (L2) statistics.

Benchmark	Miss Rate [%] (DDR3-1600)			Miss Rate [%] (DDR4-2400)		
	FS	SE	DS	FS	SE	DS
COPY	96.8	100.0	76.9	96.7	100.0	76.3
DAXPY	96.7	100.0	74.8	96.7	100.0	74.9
INIT	94.9	100.0	70.3	94.8	99.8	70.3
SDAXPY	96.7	100.0	080.9	96.6	100.0	80.7
STRIAD	97.1	100.0	083.1	96.6	100.0	83.0
SUM	94.8	100.0	89.4	94.8	99.9	89.5
TRIAD	96.6	100.0	80.9	96.6	100.0	80.6
UPDATE	94.9	100.0	71.6	94.8	99.9	71.6

Table 9.6: Results for the total simulation time and the average response latency with DDR3-1600.

Benchmark	Simulation Time [s]			Avg. Latency [ns]		
	FS	SE	DS	FS	SE	DS
COPY	0.186	0.149	0.206	47.4	35.2	55.8
DAXPY	0.183	0.153	0.271	39.1	34.8	49.4
INIT	0.139	0.117	0.169	39.0	42.2	48.1
SDAXPY	0.226	0.181	0.255	43.2	29.3	50.6
STRIAD	0.251	0.228	0.266	36.6	37.6	50.4
SUM	0.134	0.107	0.204	29.0	28.4	44.2
TRIAD	0.220	0.183	0.251	40.6	32.5	51.2
UPDATE	0.146	0.121	0.248	39.8	40.3	49.9

Table 9.7: Results for the total simulation time and the average response latency with DDR4-2400.

Benchmark	Simulation Time [s]			Avg. Latency [ns]		
	FS	SE	DS	FS	SE	DS
COPY	0.172	0.144	0.208	32.5	29.8	43.4
DAXPY	0.175	0.148	0.273	31.4	29.5	38.8
INIT	0.137	0.112	0.172	36.0	34.8	39.5
SDAXPY	0.212	0.177	0.259	32.7	26.4	40.4
STRIAD	0.210	0.212	0.268	34.5	29.1	40.1
SUM	0.130	0.107	0.205	24.1	27.0	37.1
TRIAD	0.212	0.175	0.254	34.5	26.7	40.4
UPDATE	0.141	0.118	0.247	33.0	34.2	40.5

List of Figures

List of Tables

List of Listings

List of Abbreviations

References

[1] Manil Dev Gomony, Christian Weis, Benny Akesson, Norbert Wehn, and Kees Goossens. DRAM selection and configuration for real-time mobile systems. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 51–56, 2012.

[2] Lukas Steiner, Matthias Jung, Felipe S. Prado, Kirill Bykov, and Norbert Wehn. DRAMSys4.0: A Fast and Cycle-Accurate SystemC/TLM-Based DRAM Simulator. In Alex Orailoglu, Matthias Jung, and Marc Reichenbach, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 110–126, Cham, 2020. Springer International Publishing.

[3] Matthias Jung. *System-level Modeling, Analysis and Optimization of DRAM Memories and Controller Architectures*. Forschungsberichte Mikroelektronik. Technische Universität Kaiserslautern, 2017.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2), August 2011.

[5] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters*, 15(1):45–49, 2016.

[6] Derek Bruening. Efficient, transparent, and comprehensive runtime code manipulation. *Massachusetts Institute of Technology*, 2004.

[7] Derek Bruening, Timothy Garnett, and Saman P. Amarasinghe. An infrastructure for adaptive dynamic optimization. *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, 2003.

[8] Valgrind. *Valgrind is an instrumentation framework for building dynamic analysis tools.* https://valgrind.org/. Accessed: 2022-07-05.

[9] Pablo Oliveira Antonino, Matthias Jung, Andreas Morgenstern, Florian Faßnacht, Thomas Bauer, Adam Bachorek, Thomas Kuhn, and Elisa Yumi Nakagawa. Enabling Continuous Software Engineering for Embedded Systems Architectures with Virtual Prototypes. In Carlos E. Cuesta, David Garlan, and Jennifer Pérez, editors, *Software Architecture*, pages 115–130, Cham, 2018. Springer International Publishing.

[10] IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, 2012.

[11] Christian Menard, Jeronimo Castrillon, Matthias Jung, and Norbert Wehn. System simulation with gem5 and SystemC: The keystone for full interoperability. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 62–69, 2017.

[12] Bruce Jacob, Spencer W. Ng, and David T. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008.

[13] Magnus Jahre and Lasse Natvig. Performance Effects of a Cache Miss Handling Architecture in a Multi-core Processor. 2007.

[14] Matthias Jung, Kira Kraft, and Norbert Wehn. A new state model for DRAMs using Petri Nets. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2017.

[15] Saugata Ghose, Tianshi Li, Nastaran Hajinazar, Damla Senol Cali, and Onur Mutlu. Demystifying Complex Workload-DRAM Interactions: An Experimental Study. *Proc. ACM Meas. Anal. Comput. Syst.*, 3(3), dec 2019.

[16] Erlangen National High Performance Computing Center. The Bandwidth Benchmark. https://github.com/RRZE-HPC/TheBandwidthBenchmark. Accessed: 2022-06-28.

[17] QEMU. *A generic and open source machine emulator and virtualizer*. https://www.qemu.org/. Accessed: 2022-06-28.

[18] Andreas Abel and Jan Reineke. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *ASPLOS*, ASPLOS '19, pages 673–686, New York, NY, USA, 2019. ACM.

[19] Agner Fog. Instruction tables. *Technical University of Denmark*, June 2022. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs.

[20] Radhika Jagtap, Stephan Diestelhorst, Andreas Hansson, Matthias Jung, and Norbert When. Exploring system performance using elastic traces: Fast, accurate and portable. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 96–105, 2016.