A Maude Framework for Cache Coherent Multicore Architectures *

Shiji Bijo, Einar Broch Johnsen, Ka I Pun, S. Lizeth Tapia Tarifa

Department of Informatics, University of Oslo, Norway
{shijib,einarj,violet,sltarifa}@ifi.uio.no

Abstract. On shared memory multicore architectures, cache memory is used to accelerate program execution by providing quick access to recently used data, but enables multiple copies of data to co-exist during execution. Although cache coherence protocols ensure that cores do not access stale data, the organisation of data in memory and the scheduling of tasks may significantly influence the performance of a parallel program in this setting. As a step towards understanding how the data organisation impacts the performance of a given parallel program using shared memory, this paper proposes a framework defined in Maude for the executable modelling of program execution on cache coherent multicore architectures, formalising the interactions between cores executing tasks, their caches, and main memory. The framework allows the specification and comparison of program execution with different design choices for the underlying hardware architecture, such as the number of cores, the data layout in main memory, and the cache associativity.

1 Introduction

Program execution on multicore architectures can be accelerated by cache memory, which provides quick access to recently used data but allows multiple copies of data to co-exist during execution. Cache coherence protocols ensure that the data in the caches is consistent and that cores never access stale data from the caches. Requested data which has become stale or which is not in the cache, leads to a so-called *cache miss*; the requested data needs to be loaded into the cache before it can be accessed, resulting in a performance penalty. With the current dominating position of multicore architectures in hardware design, we believe that language designers may benefit from a better understanding and ability to reason about interactions between cores executing tasks, their caches, and main memory. For this purpose we need clear and precise operational models which allow us to reason about such interactions. However, work on operational semantics for parallel programs generally abstracts from the caches of multicore architectures, and assumes that there are only single copies of data in memory

* This work is partly funded by the EU research project FP7-612985 *UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimisations* (www.upscale-project.eu).

(i.e., memory is directly accessed from threads). As a consequence, these semantic models provide little guidance for language designers in making efficient use of cache memory.

Maude has been proposed as a unifying framework for language semantics which supports a wide range of definitional styles (e.g., [22, 30]). Its usefulness as a semantic framework has been widely demonstrated, including low-level and highly complex languages such as C [13]. In this paper, we develop a Maude framework for modelling execution on cache consistent multicore architectures, capturing how data movement between cores and main memory is triggered by the execution of program tasks on the different cores. Our purpose is not to evaluate the specifics of a concrete cache coherence protocol, but rather to capture program execution on shared data at locations with coherent caches in a formal yet highly configurable way. We let the Maude specification keep track of the cache hit/miss ratio per core as a way to evaluate the performance of an execution path. This allows runtime design choices for programs to be compared by means of Maude's analysis techniques. We illustrate how models of multicore architectures can be configured and compared with some small examples.

Related work. Previous work by the authors developed an SOS for cache coherent multicore architectures using multi-set label synchronisation on transitions to model parallel instantaneous broadcast, and proved that this semantics guaranteed properties such as data race free access to data and no access to stale data [2]. This paper extends the previous work in several ways: (1) it makes the source-level programming language more expressive by supporting loops, choice, and dynamic task creation, (2) it allows the modeller to configure the cache associativity of the architecture, and (3) it refines the abstracted declarative definitions of the previous work, resulting in an executable semantics in Maude. Rewriting logic has previously been used to specify and analyse interaction between a single microprocessor and its cache [1, 14], and to model and analyse memory safety and garbage collection for hardware architectures [24, 29].

Other approaches to the analysis of multicore architectures include on the one hand simulators for evaluating the efficiency of specific cache coherence protocols and on the other hand formal techniques for proving their correctness. Simulation tools allow cache coherence protocols to be specified in order to evaluate their performance on different architectures (e.g., gems [19] and gem5 [3]). These tools run benchmark programs written as low-level read and write instructions to memory and perform measurements, e.g., the cache hit/miss ratio. Advanced simulators such as Graphite [23] and Sniper [5] can handle multicore architectures with thousands of cores by running on distributed clusters. A framework, proposed in [17], statically estimates the worst-case response times for concurrent applications running on multiple cores with shared cache.

Both operational and axiomatic formal models have been used to describe the effect of parallel executions on shared memory under relaxed memory models, including abstract calculi [7], memory models for programming languages such as Java [16], and machine-level instruction sets for concrete processors such as POWER [18, 31] and x86 [32]. The behaviour of programs executing under

total store order (TSO) architectures is studied in [12, 33]. However, work on weak memory models abstracts from caches, and is as such largely orthogonal to our work which does not consider the reordering of source-level syntax. Cache coherence protocols can be formally specified as automata and verified by (parametrised) model checking (e.g., [9, 25, 27]) in terms of operational formalisations which abstract from the specific number of cores to prove the correctness of the protocols (e.g., [10, 11, 34]). For example, Maude's model checker has recently been used to verify the correctness of configurations of the MSI and ESI protocols [20, 28]. In contrast, our work, which also uses MSI, focuses on specifying the abstract interactions between caches and shared memory for parallel programs executing on a multicore architecture.

Paper overview. Sect. 2 reviews background concepts on rewriting logic and Maude, and on multicore architectures, Sect. 3 presents our abstract model of cache coherent multicore architectures, Sect. 4 details the Maude model, Sect. 5 presents examples with different configurations, and Sect. 6 concludes the paper.

2 Preliminaries

2.1 Maude

The semantics of our framework is defined in Maude [6], a specification and analysis system based on rewriting logic (RL) [21]. In a rewrite theory (Σ, E, R), the signature Σ defines the ground terms, E equations between terms, and R a set of labelled rewrite rules. Rewrite rules apply to terms of given sorts, specified in (membership) equational logic (Σ, E). When modelling computational systems, different system components are typically modelled by terms of suitable sorts and the global state configuration is represented as a multi-set of these terms. RL extends algebraic specification techniques with transition rules which capture the dynamic behaviour of a system. *Conditional rewrite rules* on the form **crl** `[label]:` $t \longrightarrow t'$ **if** *cond* transform an instance of the pattern t into the corresponding instance of the pattern t', where the condition *cond* is a conjunction of rewriting conditions and equalities that must hold for the rule to apply (the *label* identifies the rule). When auxiliary functions are needed, these can be defined in equational logic, and thus evaluated in between the state transitions [21]. In a *conditional equation* **ceq** $t = t'$ **if** *cond* , the condition must similarly hold for the equation to apply. Unconditional rewrite rules and equations are denoted by the keywords **rl** and **eq**, respectively.

To structure specifications in Maude, equations and rewrite rules can be scoped in modules which may be parameterised. In particular, Maude supports the modelling of systems as multi-sets of objects in a standardised format, with suitable communication mechanisms. The pre-defined Maude module `CONFIGURATION` provides a notation for object syntax. An object in a given state has the form $\langle$ `O : C |` $\mathtt{a}_1$ `:` $\mathtt{v}_1$`, ..., ` $\mathtt{a}_n$ `:` $\mathtt{v}_n$ $\rangle$, where `C` is the class name and `O` the object identifier, and there is a set of attributes with identifiers $\mathtt{a}_i$ which have corresponding values $\mathtt{v}_i$ in the current state. Given an initial configuration, Maude supports simulation and breadth-first search through reachable

states, and model checking of systems with a finite number of reachable states. In this paper, Maude is used to implement executable operational semantics of a framework to experiment with architectural models.

2.2 Overview of Multicore Architectures

Processing units, or *cores*, in modern multicore architectures use *caches* to store their most recently accessed data. Caches are usually small and fast compared to *main memory*, and they can be either private to one core or shared among multiple cores. Main memory in general consists of *blocks*, each with a unique *address*. Each block is organised in multiple continuous *words*, and each word is associated to a *memory reference*. Cache memory, on the other hand, consists of *cache lines*, each of which may contain several words. In general, a cache line can contain at least as many words as one memory block.

A cache *hit* refers to the case that the desired data is found in the cache, the other case is called a cache *miss*. In the case of a miss, the cache fetches data from the next levels in the memory hierarchy (e.g., main memory). The fetched data, which includes the requested data, consists of consecutive words starting at the beginning of a memory block and corresponds in size to a cache line. Since caches are small in size compared to main memory, fetching data from main memory to caches may require the eviction of existing cache lines. The choice of cache lines to evict depends on the organisation of the cache lines, the so-called *cache associativity*, and on the *replacement policy*. In k-way *set associative caches*, the cache lines are grouped as sets with k cache lines and the fetched data can go anywhere in a particular set. *Direct mapped caches* have one-way set associativity as these caches are organised in single-line sets. In *fully associative caches*, the entire cache is considered as a single set and the fetched data can be placed anywhere in the cache. The modulo operation $n\%c$ calculates the index of the cache set to which the fetched data starting at the block with address n should be placed, where c is the number of sets per cache. Replacement policies select the line to evict from a specific cache set when the fetched data is placed into that set, e.g., random, FIFO, LRU (Least Recently Used).

Since multiple copies of data may be stored in different local caches and in main memory, cache coherence protocols are used to keep all copies consistent. In particular, *invalidation-based protocols* broadcast invalidation messages when a particular core requires write access to a memory address. Typical invalidation protocols are MSI and its extensions (e.g., MESI and MOESI). In MSI, a cache line can be in one of three states: **M**odified, **S**hared, and **I**nvalid. A line in *modified* state indicates that it is the most updated copy and that all other copies in the system are *invalid*, whereas a line in *shared* state implies that all copies in the system are consistent. A cache has a miss when the requested line is either invalid or is not present in the cache; in this case, the cache broadcasts a request. Upon receiving this request, the cache which has the modified copy of the relevant cache line flushes the cache line to the main memory. The state of the cache line will then be updated to shared in both the cache and the main memory. An invalidation message will be broadcast in the system before writing in a cache

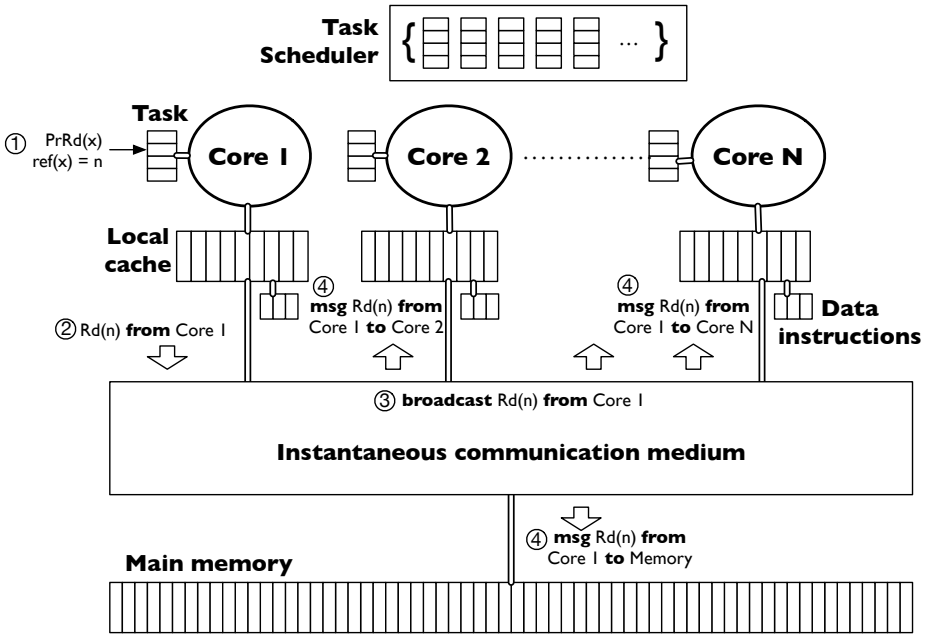

Fig. 1. Abstract model of multicore architecture (illustration). Here we let circled numbers ① – ④ suggest the ordering of communication events.

line. The state of the cache line will be updated to modified if the write operation succeeds. Upon the arrival of an invalidation message, a cache will invalidate its shared copy. For more details on variations of multicore architectures, coherence protocols, and memory consistency, the reader may consult, e.g., [8, 15, 26].

3 An Abstract Model of Execution on Cache Coherent Multicore Architectures

This section outlines an abstract model of execution for parallel programs running on multicore architectures. We first explain the source-level language that the model executes, then the abstract model of the hardware architecture, and finally how the relation between these two is captured by the execution model.

The source-level language. Programs are written in a source-level input language with the following syntax, where r is a memory reference and T a task name:

$$\begin{aligned} &LLP ::= \overline{Task}\ \texttt{main}\{sst\} \\ &Task ::= \texttt{task}\ T\{sst\} \\ &\quad sst ::= \varepsilon \mid sst; sst \mid \texttt{PrRd}(r) \mid \texttt{PrWr}(r) \mid \texttt{commit}(r) \mid sst \sqcap sst \mid sst^* \mid \texttt{Spawn}\ T \end{aligned}$$

A (low-level) program LLP consists of a set of tasks, denoted by the overbar $\overline{Task}$, and a main block. Each task $\texttt{task}\ T\{sst\}$ has a name T and a body

with statements *sst*. Statements include **PrRd**(r) for reading the word from main memory reference r, **PrWr**(r) for writing to r, **commit**(r) for flushing the content of the cache line with reference r to main memory, and **Spawn** T to dynamically create a task T. In addition, there are standard operators for sequential composition $sst; sst$, non-deterministic choice $sst \sqcap sst$, and repetition sst^*.

Example 1. Let r_0, r_1, r_2, and r_3 be references inside memory blocks. This is a simple example of a low-level program:

task $T_1\{$**PrRd**$(r_0);$**PrWr**$(r_1)\}$
task $T_2\{$**PrRd**$(r_2);$**PrWr**$(r_3)\}$
main$\{$**Spawn** $T_1;$ **Spawn** $T_2\}$ □

The hardware architecture. As depicted in Figure 1, our hardware architecture model consists of multiple cores with shared memory, where each core has a *private one-level* cache. Communications between different components are through messages. Cores are responsible for sending messages to other components in the architecture, if needed. The exchange of messages between different cores and main memory is captured by a *communication medium*, which abstracts from different concrete topologies such as bus, ring, or mesh. Observe that the communication medium needs to arbitrate between requests from different cores (e.g., to exclude conflicting invalidation messages). We here abstract from how this arbitration is realised by a specific communication medium by modelling the communication as *instantaneous*. For simplicity, we ignore the data contained in memory blocks and cache lines. We further assume that the size of a cache line is same as the size of a memory block, and data move from one core to another one indirectly via the main memory. Our model includes an abstract version of the MSI protocol.

The execution model. A runtime configuration consists of several components: the cores with their associated cache, where each cache has its own associativity relation, a main memory, a scheduler with a global pool of dynamically created tasks that are waiting to be executed, a task lookup table $[T \mapsto sst]$ mapping task names to bodies, and a reference lookup table $[r \mapsto n]$ binding references to block addresses.

To execute *LLP* programs, new tasks are scheduled on the idle cores in the abstract model. Cores execute *runtime statements rst* by interacting with their local caches. Runtime statements include the *source-level statements sst* and statements **PrRdBl**(r), **PrWrBl**(r) and **commit**. Statements such as **PrRd**(r) and **PrWr**(r) may trigger data movement between the cache and main memory. Such data movements are captured by *data instructions* **fetch**(n) and **flush**(n), where n is the address of the memory block in which the word with reference r resides. A task executing on a core may be *blocked* waiting for data to be fetched, which is captured by statements **PrRdBl**(r) and **PrWrBl**(r). To ensure that modified data is stored in main memory before another task is scheduled, we use **commit** to force all modified lines in the cache to be flushed after

```
op {_} : Configuration → GlobalSystem [ctor] .
op ref : Int → Reference [ctor] .
op Assoc : Int → MapPolicy [ctor] .

op CR : → Cid [ctor format(n d)] .
op CM:_ : Cache{Int, MemoryMap} → Attribute [ctor] .
op Rst:_ : stList → Attribute [ctor] .
op D:_ : stList → Attribute [ctor] .
op Miss:_ : Int → Attribute [ctor] .
op Hit:_ : Int → Attribute [ctor] .
op CacheSz:_ : Int → Attribute [ctor] .
op ~:_ : MapPolicy → Attribute [ctor] .

op ⟨_: MM|M:_, fetchCount:_⟩ : Oid MemoryMap Int → Object [ctor] .
op ⟨_: Task|Data:_⟩ : Oid task{Qid, stList} → Object [ctor] .
op ⟨_: Qu|TidSet:_⟩ : Oid Set{Qid} → Object [ctor] .
op ⟨_: TBL|Addr:_⟩ : Oid Tbl{Reference, Address} → Object [ctor] .
```

Fig. 2. Runtime syntax: Relevant sorts and Object representation in Maude

the execution of a task. In an *initial state*, all memory blocks in the main memory have status shared, the task pool contains a main block *sst* generated from **main**{*sst*}, and each core has an empty cache, no data instructions and no executing task.

To illustrate a behaviour in our model, let a reference r map to a memory address n, and consider a core which does not have the memory block n available in the local cache and attempts to access n by executing a statement **PrRd**(r) (or **PrWr**(r)). To retrieve the memory block, the core first broadcasts the request as messages to other cores and main memory through the communication medium. Messages include requests for *read* and *read exclusive* (for write operation) access to memory address n, which are of the form $Rd(n)$ and $RdX(n)$, respectively. Once the message is broadcast, the execution of the task in the core will be blocked by a **PrRdBl**(r) (or **PrWrBl**(r)) statement and a **fetch**(n) instruction is added to the data instructions. The request will be instantaneously broadcast to the other cores and to main memory (thereby mimicking the use of label synchronisation in SOS such as [2]). If the requested cache line is modified in another cache, the protocol ensures that the line in the other cache is first *flushed* to main memory. The requesting core can then proceed to *fetch* the memory block n from main memory after the **flush**(n) instruction, and continue with the execution of its task. Note that a request message for read exclusive access will invalidate all copies of the relevant block in the other caches as well as in main memory. Consequently, data race freedom and the consistency of copies of data in different caches in this model are maintained by an abstracted version of the basic coherence protocol MSI.

4 Formalising Multicore Architectures in Maude

We formalise the abstract model presented in Sect. 3 as an object-oriented specification in Maude [6], and focus the presentation on the main parts of this

specification. Section 4.1 explains the main sorts and signatures of the model, Sect. 4.2 focuses on a subset of the rewriting rules which capture task execution and Sect. 4.3 on a subset of the equations and rewriting rules for coordination and communication between cores and main memory using message passing[1].

4.1 Sorts and signatures of the model

Figure 2 shows the main sorts and objects used in our model. The components (e.g., cores, memory, and scheduler) are modelled as Maude objects floating in a global configuration. A system wide operator {_} is used to wrap complete configurations into the sort `GlobalSystem`, and ensures that communication messages are correctly propagated to every part of the system. Each memory block has an address of sort `Address` and may contain multiple words. The operator `ref(r)` of sort `Reference` means the reference r to a word in a memory block. Observe that multiple references can be mapped to the same block address, which gives us flexibility to specify the size of memory blocks and cache lines. The `Assoc` operator of sort `MapPolicy` takes an integer k as parameter to specify that each cache set has k cache lines. As shown in Figure 2, the object `MM` models the main memory. The attribute `M` is a map of sort `MemoryMap` that binds `Address` to `Status`, which can be modified `mo`, shared `sh` or invalid `inv`, as in the MSI protocol. The attribute `fetchCount` logs the total number of *fetch* operations which have been used in the whole configuration during an execution.

Cores are modelled by a class identifier `CR` and a number of attributes, such as `CM` stores the cache memory, `Rst` the task to be executed, `D` the data instructions (fetch and flush), `CacheSz` the size of the cache, `~` the cache associativity, `Miss` the number of local cache misses, and `Hit` the number of local cache hits. As the core object has a number of attributes, we model it using the pre-defined object syntax from the `CONFIGURATION` module, which introduces an object format with a set of attributes. This allows a compact presentation format for the rewrite rules as a variable `Atts` of sort `AttributeSet` can be used to replace all attributes which are not important for a specific rule as shown later in Figure 3. The cache memory `CM` of sort `Cache{Int, MemoryMap}` is a collection of cache sets where each set has an integer index as the identifier. Similar to the attribute `M`, a cache set is also of sort `MemoryMap`, which maps each cache line in the set to its status. The sort `stList` refers to the sequence of statements to be executed in both `Rst` and `D`.

Object `Task` models a task lookup table which binds task `Qid` to sequences of statements (for dynamic task allocation). The scheduler is modelled by the object `Qu`, which selects a task id from its pool and allocates the sequence of statements (using the object `Task`) to an idle core. Object `TBL` models the reference lookup table by mapping each reference r to a word in the main memory to the address of the memory block where the word resides.

[1] The complete Maude model is available from `http://folk.uio.no/~shijib/wrla2016maude.zip`.

4.2 Rewriting rules for task execution

Rewriting rules are used to capture the execution of parallel tasks in the multi-core hardware architecture. For brevity in the presented rules, we omit the static table `TBL` and use `addr[ref(r)]` to denote the block address of a memory reference r. Figure 3 contains the subset of rules that are explained in this section, gray boxes are used to highlight the evolving patterns in each rule.

Rule *PrRd1* describes the case when a read statement `PrRd(r)` proceeds with a cache hit, because the block `N` (= `addr[ref(r)]`) is either in status `mo` or `sh` in the cache. To simplify the presentation of the rules, we let σ denote the frequently recurring expression `selectStatus(`*mapPol*`,` *Ca*`,` *size*`,` `addr[ref(`*r*`)])`, where *mapPol*, *Ca*, *size*, and r are the parameters which respectively refer to the cache associativity $\sim$, the cache memory, the number of cache lines and references. The function returns the status of block `N`; if `N` is not present in the cache, it returns `unknown`. In the case of a cache miss, rule *PrRd2* adds a `fetch`-instruction to the data instructions `D`, (which will later fetch data from the main memory) and replaces the read statement by a blocking statement `PrRdBl(r)` to indicate that the core is waiting for the memory block to be fetched. To request block `N`, rule *PrRd2* broadcasts a read request `Rd(N)` to the communication medium from which all other cores instantaneously receive the request; upon receipt, a core which has a modified copy of the requested block will make a flush. For each cache hit and cache miss, the related local counters are incremented.

A write statement `PrWr(r)` is treated as a hit if the status of the relevant block `N` is `mo`, where the corresponding rule is similar to the rule *PrRd1* (and therefore omitted here). In case block `N` is in `sh` status, which is also counted as a cache hit, the core needs to broadcast an invalidation message `RdX(N)` to ensure exclusive access to the block `N`, captured by rule *PrWr2*. Upon receiving the invalidation message, all other cores and the main memory will instantaneously invalidate their shared copy of block `N`. In the case of a cache miss, the core first requests the block by sending a read request `Rd(N)`, as captured by rule *PrWr3*, and the write statement is replaced by a blocked statement `PrWrBl(r)`, indicating that the core is waiting for the block to be fetched. Once the block is fetched, the core will request exclusive access to `N`, similar to rule *PrWr2*. Observe that before requesting exclusive access, the cache line may get invalidated by a `RdX(N)` message from another core in the configuration. In this situation, the core has to broadcast the `Rd(N)` message again as in rule *PrWr3*. Rule *PrWrBlock2* captures this situation; here, the `occurs` operation avoids repeated execution of the same rule by checking whether the `fetch` instruction for the requested block is already added to the data instructions `D` or not. Rule *Commit1* forces a core to flush the modified copy of a particular memory block to main memory.

Only memory blocks with status `sh` can be fetched from main memory. Otherwise, the fetch instruction is blocked until the data is flushed from another cache. In the rule *Fetch1*, the `allModified` function returns `true` if the cache set in which the memory block should be placed is full with modified cache lines. This function returns `false` if there is vacant space in the set or all cache lines in

```
crl [PrRd1] :
⟨C1: CR | CM: Ca, Rst:(PrRd(r);rst), Hit:h, CacheSz: size, ~: mapPol, Atts⟩
⟶
⟨C1: CR | CM: Ca, Rst:rst, Hit:(h+1), CacheSz: size, ~: mapPol, Atts⟩
if σ = sh or σ = mo .

crl [PrRd2] :
⟨C1: CR | CM: Ca, Rst:(PrRd(r);rst), D:d, Miss:m, CacheSz: size, ~: mapPol, Atts⟩
⟶
⟨C1: CR | CM: Ca, Rst:(PrRdBl(r);rst), D:(d;fetch(addr[ref(r)])), Miss:(m + 1),
CacheSz: size, ~: mapPol, Atts⟩ (broadcast Rd(addr[ref(r)]) from C1)
if σ = inv or σ = unknown .

crl [PrWr2] :
⟨C1: CR | CM: Ca, Rst:(PrWr(r);rst), Hit:h, CacheSz: size, ~: mapPol, Atts⟩
⟶
⟨C1: CR | CM: Ca, Rst:(PrWrBl(r);rst), Hit:(h+1), CacheSz: size, ~: mapPol,
Atts⟩ (broadcast RdX(addr[ref(r)]) from C1)
if σ = sh .

crl [PrWr3] :
⟨C1: CR | CM: Ca, Rst:(PrWr(r);rst), D:d, Miss:m, CacheSz: size, ~: mapPol, Hit: h⟩
⟶
⟨C1: CR | CM: Ca, Rst:(PrWrBl(r);rst), D:(d;fetch(addr[ref(r)])), Miss:(m+1),
CacheSz: size, ~: mapPol, Hit: h ⟩ (broadcast Rd(addr[ref(r)]) from C1)
if σ = inv or σ = unknown .

crl [PrWrBlock2] :
⟨C1: CR | CM: Ca, Rst: (PrWrBl(r);rst), D:d, CacheSz: size, ~: mapPol, Atts⟩
⟶
⟨C1: CR | CM: Ca, Rst: (PrWrBl(r);rst), D:(d;fetch(addr[ref(r)])), ~: mapPol,
CacheSz: size, Atts ⟩ (broadcast Rd(addr[ref(r)]) from C1)
if( σ = inv or σ = unknown) and (occurs(fetch(addr[ref(r)]), d) = false) .

crl [Commit1] :
⟨C1: CR | CM: Ca, Rst:(commit(r);rst), D:d, CacheSz: size, ~: mapPol, Atts⟩
⟶
⟨C1: CR | CM: Ca, Rst:rst, D:(d;flush(addr[ref(r)])), CacheSz: size, ~: mapPol,
Atts⟩ if σ = mo .

crl [Fetch1] :
⟨C1: CR | CM:Ca, D:(fetch(N);d), CacheSz: size, ~: mapPol, Atts⟩
⟨Me: MM | M: mapSet, fetchCount: x' ⟩
⟶
⟨C1: CR | CM:fetchUpdateLine(Ca,size,N,sh,mapPol), D:d, CacheSz: size,
~: mapPol, Atts⟩ ⟨Me: MM | M: mapSet, fetchCount:(x'+1)⟩
if allModified(cacheLineSet(Ca,size,N,mapPol),mapPol) = false and
selectStatus(mapSet, N) = sh .

crl [Flush1] :
⟨Me: MM | M:mapSet, fetchCount: x⟩
⟨C1: CR | CM:Ca, D:(flush(N);d), CacheSz: size, ~: mapPol, Atts⟩
⟶
⟨Me: MM | M:updateLine(mapSet,N,inv), fetchCount: x⟩
⟨C1: CR | CM:updateLine(mapPol,Ca,size,N,sh,mo), D:d, CacheSz: size,
~: mapPol, Atts ⟩ if σ = mo .
```

Fig. 3. Subset of rewriting rules in Maude (part 1)

```
eq {(broadcast Re from C1) REST} =
    {(multicast Re from C1 to objectIds(REST)) REST} .
eq multicast Re from C1 to none = none .
eq multicast Re from C1 to C2 ; ReSet = (msg Re from C1 to C2)
                                        (multicast Re from C1 to ReSet) .

ceq (msg RdX(N) from C2 to C1)
⟨ C1 : CR | CM: Ca, CacheSz: size, ~: mapPol, Atts ⟩
=
⟨ C1 : CR | CM: updateLine(mapPol,Ca,size,N,inv,sh), CacheSz: size,~: mapPol,
Atts ⟩ if σ = sh and C2 ≠ C1 .

ceq (msg Rd(N) from C2 to C1)
⟨ C1 : CR | CM: Ca, D: d, CacheSz: size, ~: mapPol, Atts ⟩
=
⟨ C1 : CR | CM: Ca, D: (flush(N) ; d), CacheSz: size,~: mapPol,Atts ⟩
if σ = mo .
```

Fig. 4. Subset of rewriting rules and equations in Maude (part 2)

the set are not modified (resulting in an *eviction* without flushing). Apart from fetching the cache line, this rule uses `fetchUpdateLine` to update the status of the fetched block to `sh`. The global counter `fetchCount` is used to log the total number of fetch operations. As explained in rule *PrWrBlock2*, executing one read (or write) statement may entail multiple fetch operations, and therefore the value of `fetchCount` can be greater than the sum of local cache misses from all cores. Rule *Flush1* stores a modified cache line in main memory, setting its status to `sh` both in the cache and main memory. The rules for the remaining statements (e.g., spawn, choice, repetition) are standard.

4.3 Synchronisation through message passing

Figure 4 contains the main equations used to coordinate the accesses to memory blocks. To access a memory block `N`, a core broadcasts a read request `Rd(N)` to get the most recent copy of `N` or a write request `RdX(N)` to get exclusive access to `N`. Each broadcast request is of the form **`broadcast`** `Re` **`from`** `C1`, where `Re` denotes the read or write request and `C1` the identity of the sender. As the requests are broadcast to all the other cores and to the main memory, each broadcast request is recursively transformed into individual messages of the form **`msg`** `Re` **`from`** `C1` **`to`** `C2` where `Re` is the request and `C1` and `C2` are the sender and receiver, respectively. The function `objectIds` collects the identities of objects from the configuration `REST`.

Upon receiving a read request `Rd(N)`, if the core has the requested block `N` in its cache with status `mo`, the block will be flushed to the main memory. This allows the core sending the request to eventually get access to the most recent copy of `N`. If a core receives a read request about a block that is not locally modified, the message is ignored. Note that the main memory will discard all messages with a read request. When receiving a write request `RdX(N)`, the main memory and all cores with a local copy of `N` (except the sender) will update the

status of the relevant block to `inv` by the function `updateLine`. If the sender and the receiver are the same core, it indicates that exclusive access to the relevant block is granted to that core. In this case, the status of the block is updated from `sh` to `mo`. Note that the rewriting rules for write operations ensure that a write request can only be sent when the block is in `sh` status.

5 Examples

The presented Maude specification constitutes a highly configurable modelling framework for programs executing on cache coherent multicore architectures. In this section we consider small examples to show how the cache associativity relation, cache size, and memory layout can be configured in concrete models, and how these parameters can influence the *cache hit/miss ratio*. In addition, we consider search conditions to check the correctness of concrete instances of our modelling framework. Maude's **`search`** command evaluates all reachable states from a given initial configuration with respect to a given condition. Out of these reachable states of the execution of terminating *LLP*s, we are interested in the *worst case scenarios*, that is, states which give the highest number of cache misses or the worst hit/miss ratios.

Below, we present the execution paths which give the highest number of total cache misses. To observe differences in the status of cache lines and memory blocks in main memory, we inspect the state where all the tasks have been executed, but before the cache is flushed (using the `commit`[2] statement). We refer to this state as the *observing state* of the execution path.

Example 2. In this example we illustrate the effect of changing the cache size and cache associativity. Consider an initial configuration `config0` consisting of one core `C1` with a cache of size five, `CacheSz: 5`, and with direct map associativity `~: Assoc(1)`. The size of a memory block is one word and the reference table `TBL` maps exactly one memory reference to each block. Assume that a program which consists of a main block that spawns a task executes in this configuration:

```
eq config0 =
⟨ C1: CR | M: empty, Rst: nil, D: nil, Miss: 0, Hit: 0, CacheSz: 5, ~: Assoc(1) ⟩
⟨ M: MM | M: 0↦sh, 1↦sh, 2↦sh, 3↦sh, 4↦sh, 5↦sh, fetchCount: 0 ⟩
⟨ Sch: Qu | TidSet: ('main) ⟩
⟨ Tbl: TBL | Addr: ref(0)↦0, ref(1)↦1, ref(2)↦2, ref(3)↦3, ref(4)↦4, ref(5)↦5 ⟩
⟨ Ta: Task | Data: 'main↦Spawn('T1), 'T1↦(PrWr(0);PrWr(5);PrWr(0)) ⟩ .
```

In the following observing state (which omits the static tables `TBL` and `Task`), observe that due to the cache size and associativity, memory blocks with addresses `0` and `5` will compete for the same cache line with index `0`. All the statements in task `'T1` entail a cache miss, which gives the global counter `fetchCount: 3` in the main memory and the local counter `Miss: 3` in the cache:

[2] Recall that the `commit` statement forces the flushing of all modified cache lines, which does not affect the number of cache misses.

```
⟨ c1 : CR | M : 0↦(0↦mo),Rst : commit,D : nil,Miss : 3,Hit : 0,CacheSz : 5,~ : Assoc(1) ⟩
⟨ m : MM | M : 0↦inv, 1↦sh, 2↦sh, 3↦sh, 4↦sh, 5↦sh, fetchCount : 3 ⟩
⟨ sch : Qu | TidSet : empty ⟩
...
```

Let us change the cache size to `CacheSz : 10` and cache associativity to `~ : Assoc(2)`, which means that each cache set has two cache lines. In this setting, there is enough space for two memory blocks `0` and `5` to be in the same cache set with index `0` at the same time:

```
⟨ C1 : CR | M : 0 ↦(0↦mo,5↦mo),Rst : commit,D : nil,Miss : 2,
       Hit : 1,CacheSz : 10,~ : Assoc(2) ⟩
⟨ M : MM | M : 0↦inv, 1↦sh, 2↦sh, 3↦sh, 4↦sh, 5↦inv, fetchCount : 2 ⟩
⟨ sch : Qu | TidSet : empty ⟩
...
```

Observe that the same program executing in this configuration has only two cache misses in the observing state. □

Example 3. In this example we illustrate the effect of changing the main memory organisation. Consider an initial configuration `config1` consisting of two cores `C1` and `C2` with caches of size five, `CacheSz : 5`, and with direct map associativity `~ : Assoc(1)`. The size of a memory block is four words, so the reference table `TBL` maps four different memory references to one block. Consider the program in Example 1, which consists of tasks trying to concurrently access the memory block `0`, executes in this configuration:

```
eq config1 =
⟨ C1 : CR | M : empty,Rst : nil,D : nil,Miss : 0,Hit : 0,CacheSz : 5,~ : Assoc(1) ⟩
⟨ C2 : CR | M : empty,Rst : nil,D : nil,Miss : 0,Hit : 0,CacheSz : 5,~ : Assoc(1) ⟩
⟨ M : MM | M : 0↦sh, 1↦sh, 2↦sh, 3↦sh, 4↦sh, fetchCount : 0 ⟩
⟨ Sch : Qu | TidSet : ('main) ⟩
⟨ Tbl : TBL | Addr : ref(0)↦0, ref(1)↦0, ref(2)↦0, ref(3)↦0,
                ref(4)↦1, ref(5)↦1, ref(6)↦1, ref(7)↦1 ⟩
⟨ Ta : Task | Data : 'main↦(Spawn('T1);Spawn('T2)),'T1↦(PrRd(0);PrWr(1)),
               'T2↦(PrRd(2);PrWr(3)) ⟩ .
```

Observe that in a possible path: `C1` executes the main block and then spawns two tasks where `C2` executes `'T1` and later `C1` executes `'T2`. The concurrent execution of `'T1` and `'T2` may lead to the mutual invalidation of memory block `0` in both caches due to the presence of the so-called *false sharing* pattern in which parallel cores operate on independent words located in the same memory block. This pattern causes mutual cache invalidations as if cores were truly using shared memory block, which increases the number of cache misses. For this configuration, the total number of cache misses is three in the observing state:

```
⟨ C1 : CR | M : 0↦(0↦inv),Rst : commit,D : nil,Miss : 1,Hit : 1,CacheSz : 5,~ : Assoc(1) ⟩
⟨ C2 : CR | M : 0↦(0↦mo),Rst : commit,D : nil,Miss : 2,Hits : 1,CacheSz : 5,~ : Assoc(1) ⟩
⟨ M : MM | M : 0↦inv, 1↦sh, 2↦sh, 3↦sh, 4↦sh, fetchCount : 3 ⟩
⟨ sch : Qu | TidSet : empty ⟩
...
```

Let `config2` be a variant of `config1`, where the main memory layout in the `TBL` object is organised such that `'T1` and `'T2` access different memory blocks, preventing the false sharing pattern. In this case, the observing state is:

```
⟨ C1 : CR | M : 0↦(0↦mo) , Rst : commit , D : nil , Miss : 1 , Hit : 1 , CacheSz : 5 , ~ : Assoc(1) ⟩
⟨ C2 : CR | M : 1↦(1↦mo) , Rst : commit , D : nil , Miss : 1 , Hit : 1 , CacheSz : 5 , ~ : Assoc(1) ⟩
⟨ M : MM | M : 0↦inv , 1↦inv , 2↦sh , 3↦sh , 4↦sh , fetchCount : 2 ⟩
⟨ sch : Qu | TidSet : empty ⟩
⟨ Tbl : TBL | Addr : ref(0)↦0 , ref(1)↦0 , ref(4)↦0 , ref(5)↦0 ,
                    ref(2)↦1 , ref(3)↦1 , ref(6)↦1 , ref(7)↦1 ⟩
...
```

Observe that the highest number of total cache misses is two from this initial configuration, which reorganises the memory layout in the `TBL` object. □

We now consider *search* conditions to show a *correctness property* for concrete instances of our modelling framework with respect to data race freedom. Consider the initial configuration `config1` in Example 3. We consider a search condition stating that after a core executes a `PrWr(n)` statement, only that core has the modified copy of `n`, while all other copies are invalid. This property, for which Maude's search finds no reachable states, can be formulated as follows:

```
search {config1} ⟶ *{C : Configuration
⟨ C1 : Oid : CR | CM : (x : Int ↦(y : Address ↦stA : Status) , CaA : Cache{Int,MemoryMap}) ,
Atts : AttributeSet ⟩
⟨ C2 : Oid : CR | CM : (x : Int ↦(y : Address ↦stB : Status) , CaB : Cache{Int,MemoryMap}) ,
Atts' : AttributeSet ⟩
 ⟨ O : Oid : MM | M : (y : Address ↦stM : Status , D : MemoryMap) , fetchCount : N : Int ⟩
 } such that ((stM : Status = sh) and (stA : Status = mo)) or
      ((stA : Status = mo) and (stB : Status = mo)) or
      ((stA : Status = mo) and (stB : Status = sh)) .
```

We can similarly verify the absence of deadlock and livelock which result in statements or data instructions to be remained in the final state.

```
search {config1} ⟶ ! {C : Configuration
⟨ A : Oid : CR | Rst : rstA : stList , D : dA : stList , Atts : AttributeSet ⟩
 } such that (rstA : stList ≠ nil) or (dA : stList ≠ nil) .
```

6 Conclusions

This paper has presented a Maude framework for modelling programs executing on cache coherent multicore architectures. The framework formalises in a highly configurable way how task execution on cores with explicit caches triggers interactions between the different caches and between the caches and main memory in cache coherent multicore architectures. The framework allows the specification and comparison of program execution with different design choices for the underlying hardware architecture, such as the number of cores, the data layout in main memory and the cache associativity. We have illustrated by examples that the presented framework may be used to help us understand how the data organisation and the properties of the caches influence the performance of parallel programs executing on shared memory.

There are several interesting possible extensions of the presented work. At the level of the presented model, ongoing work considers the inclusion of multi-level caches and abstract notions of data locality. It is also interesting to see how

models such as the one presented in this paper can be used to understand real programs, for example by extracting LLP programs from real code; here we run into the problem of optimisations and instruction reordering which could perhaps be addressed by extracting worst-case LLP representations. In the context of the actor-based programming language Encore [4], we plan in future work to try that approach and use the presented framework to study the effects of program specific optimisations of data layout and scheduling derived by, e.g., locking disciplines, annotations such as behavioural type systems, or static analyses.

Acknowledgements We are grateful to the anonymous reviewers for their very thorough reviews and for giving helpful and critical feedback.

References

1. M. Ayala-Rincón, R. M. Neto, R. P. Jacobi, C. H. Llanos, and R. W. Hartenstein. Applying ELAN strategies in simulating processors over simple architectures. *Electronic Notes in Theoretical Computer Science*, 70(6):84 – 99, 2002.
2. S. Bijo, E. B. Johnsen, K. I Pun, and S. L. Tapia Tarifa. An operational semantics of cache coherent multicore architectures. In *Proc. 31st Annual ACM Symp. on Applied Computing (SAC'16)*. ACM, 2016. To appear.
3. N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, 2011.
4. S. Brandauer, E. Castegren, D. Clarke, K. Fernandez-Reyes, E. B. Johnsen, K. I Pun, S. L. Tapia Tarifa, T. Wrigstad, and A. M. Yang. Parallel objects for multicores: A glimpse at the parallel language Encore. In *Formal Methods for Multicore Programming*, *LNCS* 9104, pages 1–56. Springer, 2015.
5. T. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC'11)*, pages 1–12, 2011.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, *LNCS* 4350. Springer, 2007.
7. K. Crary and M. J. Sullivan. A calculus for relaxed memory. In *Proc. 42nd Annual ACM Symp. on Principles of Programming Languages (POPL 2015)*, pages 623–636. ACM, 2015.
8. D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., 1997.
9. G. Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.
10. D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proc. Intl. Conf. on Computer Design: VLSI in Computer & Processors (ICCD'92)*, pages 522–525. IEEE Computer Society, 1992.
11. D. L. Dill, S. Park, and A. G. Nowatzyk. Formal specification of abstract memory models. In *Proc. Symp. on Research on Integrated Systems*, pages 38–52. MIT Press, 1993.

12. B. Dongol, O. Travkin, J. Derrick, and H. Wehrheim. A high-level semantics for program execution under total store order memory. In *Theoretical Aspects of Computing (ICTAC 2013)*, *LNCS* 8049, pages 177–194. Springer, 2013.
13. C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *Proc. 39th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 2012)*, pages 533–544. ACM, 2012.
14. N. Harman. Verifying a simple pipelined microprocessor using Maude. In *Recent Trends in Algebraic Development Techniques*, *LNCS* 2267, pages 128–151. Springer, 2002.
15. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2011.
16. R. Jagadeesan, C. Pitcher, and J. Riely. Generative operational semantics for relaxed memory models. In *Proc. 19th European Symp. on Programming (ESOP 2010)*, *LNCS* 6012, pages 307–326. Springer, 2010.
17. Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *Real-Time Systems Symp. (RTSS 2009)*, pages 57–67. IEEE Computer Society, Dec 2009.
18. S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams. An axiomatic memory model for POWER multiprocessors. In *Proc. 24th Intl. Conf. on Computer Aided Verification (CAV'12)*, *LNCS* 7358, pages 495–512. Springer, 2012.
19. M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, Nov. 2005.
20. O. Martín, A. Verdejo, and N. Martí-Oliet. Model checking TLR* guarantee formulas on infinite systems. In *Specification, Algebra, and Software*, *LNCS* 8373, pages 129–150. Springer, 2014.
21. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
22. J. Meseguer and G. Rosu. The rewriting logic semantics project: A progress report. *Inf. Comput.*, 231:38–69, 2013.
23. J. E. Miller, H. Kasture, G. Kurian, C. G. III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *16th Intl. Conf. on High-Performance Computer Architecture (HPCA-16)*, pages 1–12. IEEE Computer Society, 2010.
24. G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *Proc. 7th Intl. Conf. on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 66–77. ACM, 1995.
25. J. Pang, W. Fokkink, R. F. H. Hofman, and R. Veldema. Model checking a cache coherence protocol of a Java DSM implementation. *Journal of Logic and Algebraic Programming*, 71(1):1–43, 2007.
26. D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., 2013.
27. F. Pong and M. Dubois. Verification techniques for cache coherence protocols. *ACM Comput. Surv.*, 29(1):82–126, 1997.
28. S. Ramirez and C. Rocha. Formal verification of safety properties for a cache coherence protocol. In *10th Colombian Computing Conf. (10CCC)*, pages 9–16. IEEE, 2015.
29. G. Roşu, W. Schulte, and T. F. Şerbănuţă. Runtime verification of C memory safety. In *Runtime Verification*, *LNCS* 5779, pages 132–151. Springer, 2009.

30. V. Rusu, D. Lucanu, T. Serbanuta, A. Arusoaie, A. Stefanescu, and G. Rosu. Language definitions as rewrite theories. *J. Log. Algebr. Meth. Program.*, 85(1):98–120, 2016.
31. S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *Proc. 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'11)*, pages 175–186. ACM, 2011.
32. P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.
33. G. Smith, J. Derrick, and B. Dongol. Admit your weakness: Verifying correctness on TSO architectures. In *11th Intl. Symp. on Formal Aspects of Component Softwares*, *LNCS* 8997, pages 364–383. Springer, 2014.
34. X. Yu, M. Vijayaraghavan, and S. Devadas. A proof of correctness for the Tardis cache coherence protocol. *arXiv preprint*, arXiv:1505.06459, 2015.