A Formal Model of Data Access for Multicore Architectures with Multilevel Caches ☆

Shiji Bijo[a], Einar Broch Johnsen[a], Ka I Pun[a,b], S. Lizeth Tapia Tarifa[a]

[a] *University of Oslo, Norway*
[b] *Western Norway University of Applied Sciences, Norway*

Abstract

The performance of software running on parallel or distributed architectures can be severely affected by the location of data. In shared memory multicore architectures, data movement between caches and main memory is driven by data accesses from tasks executing in parallel on different cores and by a protocol to ensure cache coherence. This paper integrates cache coherence in a formal model of data access, to capture such data movement from an application perspective. We develop an executable model which captures cache coherent data movement between different cache levels and main memory, for software described by task-level data access patterns. The proposed model is generic in the number of cache levels and cores, and abstracts from the concrete communication medium. We show that the model guarantees expected correctness properties for cache coherence, in particular data consistency. This paper further presents a proof-of-concept implementation of the proposed model in rewriting logic, which allows different choices for the underlying hardware architecture of dynamically created parallel data access patterns to be specified and compared at the modelling level.

Keywords: Data access; cache memory; multicore architectures; multicore memory systems; formal models; operational semantics

1. Introduction

Parallel computing enhances the performance of software applications by distributing the execution of tasks across multiple cores or processors. Cache memory provides quick access to recently used data, but it also allows multiple copies of data to co-exist during execution. Since tasks running in parallel on different cores may need to access

☆The EU project FP7-612985 *UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations* (www.upscale-project.eu), the *SIRIUS Centre for Scalable Data Access* (www.sirius-labs.no) and the FRINATEK-274515 *ADAPt: Exploiting Abstract Data-Access Patterns for Better Data Locality in Parallel Processing* (https://www.mn.uio.no/ifi/english/research/projects/adapt/).

Email addresses: shijib@ifi.uio.no (Shiji Bijo), einarj@ifi.uio.no (Einar Broch Johnsen), violet@ifi.uio.no (Ka I Pun), sltarifa@ifi.uio.no (S. Lizeth Tapia Tarifa)

the same shared data, software engineers developing applications for multicore architectures need to answer questions about data locality, data access, and data movement: Is data organized in the most convenient way for efficient data access by the different tasks of the application? Are tasks organized and ordered optimally for a given data layout? How does a given data layout fit with the targeted cache hierarchy? Is the chosen data layout robust for different cache hierarchies? These questions are important for software quality, but difficult to answer in an intuitive and straightforward way. To fully benefit from multicore architectures, it is essential to understand how software applications interact with their underlying hardware architectures at runtime; that is, we need to understand multicore architectures from the programmers' perspective.

Formal models contribute to our understanding of parallel execution, but neither software nor hardware models currently provide much guidance for software developers in understanding how data is transferred between main memory and caches, driven by the data access requirements of tasks executing in parallel. This is because (1) formal models of parallel programs generally abstract from local copies of data in caches of multicore architectures and either assume single copies of data in shared memory or record only the local ordering of write operations in buffers [12, 22, 17, 41] (i.e., they assume that all threads have direct access to main memory and only look into their local buffers for recent write operations that have not yet been flushed), and (2) formal models of hardware architectures and consistency protocols, such as cache coherence, focus on low-level correctness properties and completely abstract from the programming level [27, 39, 40, 14, 15]. In both cases, the models do not account for how the workflows of different task compete for access to shared data. Integrating models of parallel execution and hardware architecture enable reasoning about how data movement in the architecture is triggered by parallel access to data in shared memory.

```
static int sum = 0;
void worker(int n) { int i = 0;
  while (i < n) {sum := m(sum, i); i ++;}}
int m(int x, int y) { ... }
```

Figure 1: A program repeatedly accessing a shared variable.

To illustrate data access, consider the code in Figure 1, where a method `worker` iterates `n` times over a read access to a shared variable `sum`, followed by a write access to the same variable. Data access for this program can be described using a pattern, such as the regular expression $(\mathbf{read}(\mathtt{sum});\mathbf{write}(\mathtt{sum}))^n$, which abstracts from the actual computation. In this paper, we combine such patterns with models of hardware architectures to account for how tasks running on different cores interact with a memory system consisting of multilevel caches and shared memory. The purpose of this work is not to evaluate the specifics of a concrete hardware architecture, but to formally describe data access in a setting with parallel execution on multiple consistent copies of shared data. Inspired by programming language semantics, we formalise data access patterns and hardware architecture in terms of an operational semantics capturing cache coherent parallel execution. Our model abstracts from the actual communication medium (e.g., a bus or a ring), and orchestrates parallel executions by restricting data access to the different components of the memory system in the architecture to ensure consistency between co-existing copies of data. The technical contributions of this paper are:

1. a formal, operational model of data access in multicore architectures with mul-

tilevel caches for tasks describing data access patterns with loops, choice and spawn;

2. correctness properties for this formal model, expressed as invariants over an arbitrary number of cores and an arbitrary number of multilevel caches; and
3. a proof-of-concept implementation of the model which allows executions to be compared based on *penalties*, an abstract performance indicator.

Whereas the authors' previous work [5, 6] studied the much simpler setting of statically given, purely sequential data access patterns and single-level caches, this paper extends to multilevel caches, and addresses data access patterns with dynamically created tasks with loops and branching. A short version of this paper appeared in the proceedings of FACS 2017 [7]; the long version provides the full operational model, as well as further correctness properties, including instrumentation for reasoning over the operational model and the details of all proofs, and discusses the implementation of the proof-of-concept tool.

Paper overview. Section 2 reviews background concepts on shared memory multicore architectures, Section 3 presents our abstract formal model of data access patterns for multicore architectures, Section 4 details correctness properties embodied in this model, Section 5 presents a proof-of-concept implementation and an example, Section 6 discusses the related work, and Section 7 concludes the paper. The correctness proofs are included in an appendix.

2. Shared Memory Multicore Architectures

In this section, we briefly introduce the basic concepts of shared memory multicore architectures. For further details see, e.g., [13, 20, 35, 42, 43]. The components of multicore architectures are parallel processing units called *cores* for executing tasks, a *main memory* for data storage, and memory units called *caches* which give the cores rapid access to recently used data. A *multicore memory system* consists of main memory and one or more caches private to a core. Each core has an associated hierarchy of caches $L_1, \ldots, L_m$, organized in terms of size, speed, and distance: the L_1 cache is the smallest, fastest, and closest to the core and the L_m cache is the slowest, largest, and furthest away. To facilitate inter-core communication, the memory systems of the different cores are connected via a communication medium with a given topology such as a bus, a ring, or a mesh. A *cache hit* expresses that data required by the core is found in its caches, a *cache miss* that the data needs to be fetched from main memory. The hierarchy can be generalized to architectures in which caches may be shared among cores.

Data is stored in main memory as *words*, each with a unique reference. Multiple continuous words constitute a *memory block*, which has a distinct memory address. Cache memory is organized in cache lines storing memory blocks. During program execution, cores access a piece of data in memory by referencing a word, but the cache fetches the entire memory block containing the required word and stores it in a cache line. As the cache is filled up, memory blocks in cache lines may need to be evicted

to make space for newly fetched blocks. The choice of which memory block to evict depends on the cache line organisation, the so-called *associativity*, and the *replacement policy* of the cache. In *k-way* set associative caches, cache memory is organized as sets containing k cache lines and a memory block can go anywhere in a particular set. *Fully associative* caches treat the entire cache memory as a single set. A *direct mapped* cache consists of singleton sets; thus, a particular memory block can only go to one specific cache line. If the set in which a newly fetched memory block should be placed is full, another memory block is either evicted or swapped from the set, to free space using a replacement policy such as random, first in first out (FIFO) or least recently used (LRU).

Multilevel caches can be organized in different ways. In *inclusive caches*, blocks in the level i cache are also included in all lower level caches j ($j > i$). Consequently, the last level cache contains the blocks in all other caches in the hierarchy. In *exclusive caches*, data exists in at most one of the caches in the hierarchy. With *NINE* (non-inclusive non-exclusive), neither an inclusive nor an exclusive policy is enforced; i.e., memory blocks in a cache may or may not be in the corresponding lower level caches.

A *memory consistency model* [2] for cache-based architectures combines a local memory model (which can be either weak or strong) with a multicore memory system. Note that the local memory model and the multicore memory system are traditionally completely orthogonal: a weak memory model may be built on top of a multicore memory system which (normally) guarantees *coherence* [13]: for each memory block, there is some serial order of all accesses to the block that is consistent with the results of the execution of a program, and in this serial order, the program order of data accesses from a task executing on a core is preserved and any read access to a reference always obtains the last value written to that reference. In multicore memory systems, cache coherence protocols regulate the movement of data to achieve coherence among the caches of different cores. The cost of such coherence is that the writing to non-exclusive cache lines needs to be broadcast to other components of the multicore memory system.

Invalidation-based protocols notify all other affected caches when a core performs a write operation. The most common invalidation-based 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 or **i**nvalid. A *modified* state indicates that the block in that cache line has the most recently updated data and that all other copies in the memory system are *invalid* (including the copy in main memory), while a *shared* state indicates that all copies of the block have consistent data (including the copy in main memory). These protocols broadcast messages through the communication medium of the multicore architecture. Following standard nomenclature, messages of the form *Rd* request read access to a memory block while messages of the form *RdX* request exclusive read access to a memory block (for writing purposes), and thereby invalidate other copies of the same memory block in other caches.

3. A Formal Model of Parallel Data Access on Shared Memory Multicore Architectures

This paper proposes a model that captures data movement triggered by tasks executing on parallel hardware architecture. In this model, tasks are modelled as data

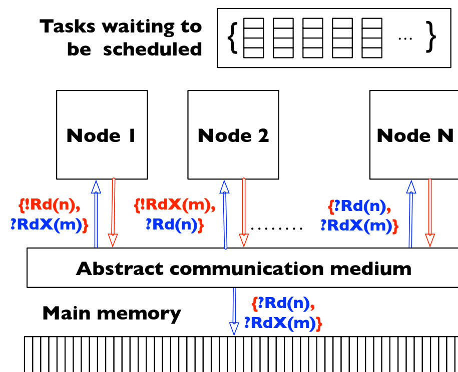


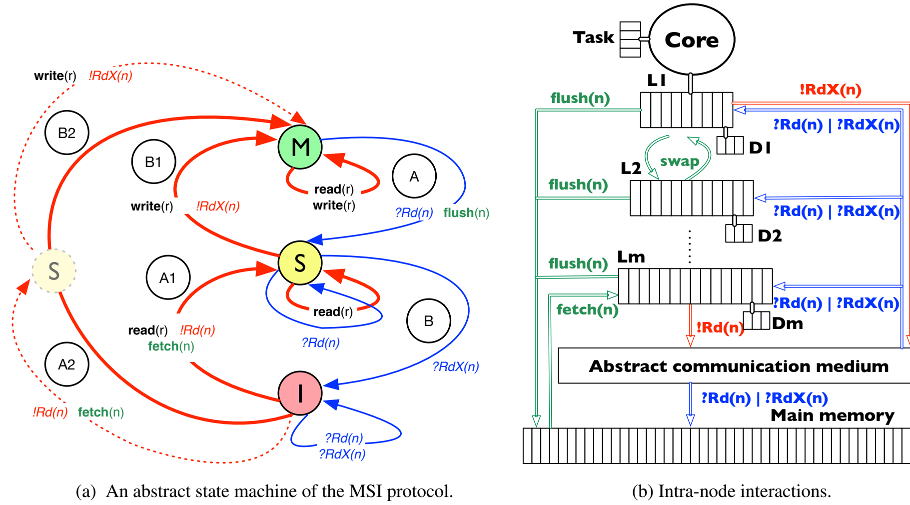
Figure 2: Overview of the model.

access patterns and the architecture consists of cores with multilevel caches and shared memory. Since our purpose is to capture data movement in a coherent multicore memory system and not to evaluate the specifics of a concrete hardware architecture, we opt to abstract from certain details and consider simple choices when available; in particular, (1) the model uses MSI for cache coherence since it is the most basic protocol with the smallest abstract state machine, (2) tasks are scheduled randomly from a shared pool (this can be easily extended to more advanced scenarios such as local schedulers and work stealing), and (3) the model uses standard textbook nomenclature, in particular for messages in the communication medium. We now present the formal model: first the abstractions introduced in the model are discussed, then its syntax, and finally its operational semantics.

3.1. Abstractions in the Formal Model

We consider a model of multicore architectures with a communication medium that abstracts from concrete topologies but ensures cache coherence using MSI. The main components of the model are the cores, the caches, and the main memory. In this model, illustrated in Figure 2, a *node* consists of a core and its hierarchy of private caches. Each node in the model executes tasks scheduled from a shared pool. To communicate with the other components, the node broadcasts messages $!Rd(n)$ and $!RdX(n)$ via the medium to respectively request read and write access to a memory block with address n. Let $?Rd(n)$ and $?RdX(n)$ denote the reception of read and invalidation requests for address n by a cache or by main memory; they are the duals of the aforementioned broadcast messages. Note that in this figure, the red lines capture messages broadcast by the node to the other components via the medium and the blue lines capture messages received by main memory and by components in each node.

Figure 3 depicts both the abstract state machine of the MSI protocol (Figure 3a) and the intra-node interactions (Figure 3b). In these figures, in addition to red and blue lines, green text and lines capture the data transfer between components. The abstract state machine of the MSI protocol, depicted in Figure 3a, describes how the state of a local cache line changes depending on the flow of these messages. In the figure, the three states M, S and I correspond to *modified*, *shared* and *invalid*, respectively. If a



(a) An abstract state machine of the MSI protocol.

(b) Intra-node interactions.

Figure 3: The structure of the formal model of multicore architectures.

node requests a memory block n that is found in its local caches in state I (or it is not found in any of its caches), it will broadcast a $!Rd(n)$ message and perform a fetch operation to get the block from main memory. Once this operation succeeds, which corresponds to annotations Ⓐ1 and Ⓐ2 in the figure, the state of n will be updated to S. If the node wants exclusive access to n for writing, it will broadcast $!RdX(n)$. Once the broadcast succeeds, which corresponds to annotations Ⓑ1 and Ⓑ2 in the figure, the state of n will be updated to M. Note that to successfully obtain exclusive access to block n, it first needs to be fetched in a shared state (depicted by the dotted state S), and later be updated to modified state. When a node receives a $?Rd(n)$ message and if the block n is found in state M in its caches, which corresponds to annotation Ⓐ in the figure, the block will be flushed to main memory and the state will be updated to S. Similarly, when a node receives a $?RdX(n)$ message and if it has this memory block in state S, which corresponds to annotation Ⓑ in the figure, it will then be updated to I. Finally, reading a memory block n with state S does not have any effect. It is similar to reading or writing to a block n with state M. If the block n is found in state I (or it is not found), then the messages $?Rd(n)$ and $?RdX(n)$ will be ignored.

Although each transition in the MSI state machine appears as an atomic step, it in fact involves one or more transitions in the architecture (as shown in Figures 2 and 3b). It is because a single node comprises a core and may have multiple levels of exclusive caches $L_1, L_2, \cdots, L_m$, and the instantaneous broadcast of messages synchronise all the nodes in the architecture. To decouple such transitions, each cache L_i in the cache hierarchy has a data instruction queue D_i for *flush* and *fetch* instructions, which move memory blocks to or from main memory or transfer cache lines between caches. In this setting, to read data from a memory block n, the node looks for n by traversing its local caches in the hierarchical order (i.e., from the first level L_1 to the last level, here L_m). If we get a *cache miss*, the last level cache broadcasts a *read request* $!Rd(n)$ via the communication medium to the other nodes and main memory. The last level cache

Syntactic categories.	***Definitions.***	
$cid \in CoreId$	$Config$	$::= M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR}$
$T \in TaskId$	$CR \in Core$	$::= cid \bullet rst$
$Tb \in TaskTable$	$Ca \in Cache$	$::= caid \bullet M \bullet dst$
$caid \in CacheId$	$st \in Status$	$::= \{mo, sh, inv\}$
$n \in Address$	$dap \in AccessPatterns$	$::= \varepsilon \mid dap;dap \mid \mathbf{read}(r) \mid \mathbf{write}(r) \mid \mathbf{commit}(r)$
$r \in Reference$		$\mid \mathbf{commit} \mid dap \sqcap dap \mid dap^{*} \mid \mathbf{skip} \mid \mathbf{spawn}(T)$
	$rst \in RuntimeLang$	$::= dap \mid rst;rst \mid \mathbf{readBl}(r) \mid \mathbf{writeBl}(r)$
	$dst \in DataLang$	$::= \varepsilon \mid dst;dst \mid \mathbf{fetch}(n) \mid \mathbf{flush}(n) \mid \mathbf{fetchBl}(n)$
		$\mid \mathbf{flush}$

Figure 4: Syntax for the formal model of multicore architectures, where over-bar denotes collections (i.e., sets $\overline{CR}$ or multisets $\overline{T}$ as appropriate), and n represents memory addresses and r references.

fetches the memory block when it is available in main memory. Eviction is required if the last level cache is full. From a cache L_i, memory block n is propagated to the first level L_1 through intermediate caches. The memory block is transferred from L_i to L_{i-1} if the cache has free space; otherwise a memory block is selected from L_{i-1} and *swapped* with n in L_i.

For simplicity, we abstract from the actual data stored in the memory blocks, and let memory blocks be transferred between nodes via the main memory. Furthermore, without compromising the validity of the model, we assume that a cache line always has the same size as a memory block. We model read and invalidation requests in the communication medium to be instantaneous; this is justified by message transfer being an order of magnitude faster than data transfer, and by the focus of the work on data movement. We can then match dual labels in a labelled transition system during composition to coordinate messages in a transition, as commonly done in process algebra, abstracting from the concrete communication medium. By lifting this matching of dual labels to sets of labels, we capture a *true concurrency* execution model for an arbitrary number of cores.

3.2. Runtime Syntax of Data Access Patterns and Multilevel Architectures

This section explains the syntax of the formal model, which is shown in Figure 4. A configuration *Config* captures the runtime state of a multicore architecture, it consists of main memory M, a set of cores $\overline{CR}$, a set of caches $\overline{Ca}$, and a multiset of tasks $\overline{T}$ (we syntactically abuse set operations, including union $\cup$ and subtraction $\setminus$, for multisets). A core CR with identifier cid executes runtime statements rst. A cache Ca has a memory M, an identifier $caid$, and a sequence of data instructions dst to be performed. We assume that a cache identifier $caid$ encodes the cid of the core to which the cache belongs and its level in the cache hierarchy where it is located. We denote by $Status \cup \{\bot\}$ the extension of the set of status tags with the undefined value $\bot$. Thus, a memory $M : Address \rightarrow Status \cup \{\bot\}$ maps addresses n to either a status tags st or to $\bot$ if the memory block with address n is not found in M. Each memory block has a unique address n, while each word located in a memory block has a unique reference r. The status tags mo, sh, and inv refer to the three states *modified*, *shared*, and *invalid* of the MSI protocol. Blocks in main memory can only be in sh or inv status. The task

table $Tb : TaskId \to AccessPatterns$ associates task identifiers T to data access patterns dap.

Data access patterns are sequences of basic operations $\mathbf{read}(r)$ and $\mathbf{write}(r)$ to read from and write to a memory reference r, $\mathbf{commit}(r)$ to flush r to main memory, and control flow statements $dap_1 \sqcap dap_2$ to non-deterministically select either dap_1 or dap_2 for execution, dap^* to repeat the execution of dap zero or more times, and $\mathbf{spawn}(T)$ to add a copy of task T to the pool of tasks to be scheduled. To ensure data consistency, the statement **commit** is used at the end of each task to flush the entire cache after task execution. Since the task table is statically given, we assume that it is always available and do not represent it explicitly in the configurations. The cores execute *runtime statements rst*, which extend *dap* with the additional control statements $\mathbf{readBl}(r)$ and $\mathbf{writeBl}(r)$ to indicate that the core gets temporarily *blocked* due to a cache miss during a read or write operation, respectively.

Each cache executes *data instructions dst*, which can be $\mathbf{fetch}(n)$ to fetch a block n from the next level cache or from main memory, $\mathbf{flush}(n)$ to flush the modified copy of n to the main memory, and **flush** to flush all modified copies in the cache. The instruction $\mathbf{fetchBl}(n)$ indicates that execution in the cache is temporarily blocked while waiting for block n to arrive in the next level cache.

Observe that the syntax presented above contains sets of different entities, e.g., $\overline{CR}$ and $\overline{Ca}$. An alternative representation of this syntax could be defined to more directly reflect the hierarchical structure in the architecture; for instance, introducing nodes containing a core and a hierarchy of caches. Such a structure would help to syntactically ensure wellformedness, but may require more complicated or additional rules to reflect the nesting structure and propagate information and control between the different entities.

3.3. An Operational Semantics of Parallel Data Access on Multicore Architectures

We define a parallel execution model for data access patterns, which captures true concurrency in the multicore setting by means of a structural operational semantics (SOS) [36], and use labels on transitions to synchronise read and invalidation requests. The semantics has a local and a global level. The *local level* captures local transitions in main memory, task execution in each core and intra-node communications to ensure data consistency between different components of the same node. The *global level* captures transitions involving data transfer between caches and main memory, the broadcasting of messages in the abstract communication medium, and the scheduling of tasks. The global level enforces data consistency by restricting how labels match in the composition rules. Multiple nodes may successfully request different memory blocks at the same time by parallel instantaneous broadcast, using (possibly empty) sets of labels on transitions. The formal syntax for the label mechanism is defined as follows:

$$\xi ::= !Rd(n) \mid !RdX(n) \qquad \pi ::= ?Rd(n) \mid ?RdX(n)$$
$$S ::= \emptyset \mid \{\xi\} \mid S \cup S \qquad R ::= \emptyset \mid \{\pi\} \mid R \cup R$$

where S and R represent (possibly empty) sets of send requests ξ and receive requests π, respectively.

Well-formed and initial well-formed configurations. In an *well-formed* configuration, all caches are associated to a core and are hierarchically organized. An *initial well-formed* configuration is a well-formed configuration such that all cores are idle (i.e., *rst* is ε), all caches are empty and have no data instructions in *dst*, all blocks in main memory M have status tag *sh*, and the task pool in $\overline{T}$ contains a single task representing the main block of a program.

Reachable configurations. Let $Config \rightarrow^* Config'$ denote an execution starting from a configuration *Config* and resulting in another configuration $Config'$ in zero or more steps by applying rules at the global level, which in turn recursively apply rules at the local level for each component. A configuration $Config'$ is *reachable* if there exists an execution starting from a well-formed configuration *Config* such that $Config \rightarrow^* Config'$. Moreover, any reachable configuration $Config'$ is well-formed.

Auxiliary functions. For a given reference r and cache identifier *caid*, we assume the following auxiliary functions: Function $addr(r)$ returns the block address n containing r. Note that which memory address to return depends on the data layout in main memory. Function $cid(caid)$ returns the core identifier associated with the cache, and $lid(caid)$ returns the level at which the cache is located in the hierarchy. The predicate $first(caid)$ is true when $lid(caid) = 1$, otherwise false; similarly, $last(caid)$ is true when $lid(caid) = l$ and l is the number of cache levels in a node, otherwise false. Setting a block n to the undefined value $\perp$ in memory M, denoted as $M[n \mapsto \perp]$, indicates that n is removed from M. Function $status(M,n)$ returns the status of block n in the map M or $\perp$ if $M(n) = \perp$.

Local semantics. The local semantics reflects the execution of statements, the interactions between caches in a node, and how the local state changes in each cache line according to the finite state machine that enforces the MSI protocol during the execution (see Figure 3a). Note that a different protocol (e.g., MESI or MOESI) will have a different state machine, and therefore different local rules. The local transition rules for nodes and main memory are given in Figures 5, 6 and 7.

The transition rules involving a core and its first level cache are shown in Figure 5. Reading reference r succeeds in rule PrRd_1 if the memory block containing r is available in the first level cache. Otherwise, rule PrRd_2 appends a **`fetch`**(n) instruction to the data instructions *dst* of the first level cache and blocks further execution of the core with the statement **`readBl`**(r). Execution may proceed once the memory block n is fetched into the cache with status *sh* or *mo*, captured by rule PrRd_3. Repeated invalidation may occur if the cache line gets invalidated by another core while the core is blocked, as captured in rule PrRd_4. Writing to reference r only succeeds in rule PrWr_1 if the associated memory block has *mo* status in the first level cache. If the memory block is in shared a state, the core broadcasts a $!RdX(n)$ request, which appears as a label in rule PrWr_2, to acquire exclusive access. Similar to reading a memory block, if the cache line is invalid (or the memory block is not in the cache) the core needs to fetch the block from main memory and execution is blocked by the statement **`writeBl`**(r), as in rule PrWr_3. Once the memory block n is copied into the cache, execution may proceed if the status of the cache line is modified, captured by

$$\textsc{(PrRd}_1\textsc{)}\quad \frac{n = addr(r) \quad first(caid) = true \quad cid(caid) = c \quad status(M,n) \in \{sh, mo\}}{(caid \bullet M \bullet dst) \circ (c \bullet \mathtt{read}(r); rst) \rightarrow (caid \bullet M \bullet dst) \circ (c \bullet rst)}$$

$$\textsc{(PrRd}_2\textsc{)}\quad \frac{n = addr(r) \quad first(caid) = true \quad cid(caid) = c \quad status(M,n) \in \{inv, \perp\}}{(caid \bullet M \bullet dst) \circ (c \bullet \mathtt{read}(r); rst) \rightarrow (caid \bullet M[n \mapsto \perp] \bullet dst; \mathtt{fetch}(n)) \circ (c \bullet \mathtt{readBl}(r); rst)}$$

$$\textsc{(PrRd}_3\textsc{)}\quad \frac{n = addr(r) \quad first(caid) = true \quad cid(caid) = c \quad status(M,n) \in \{sh, mo\}}{(caid \bullet M \bullet dst) \circ (c \bullet \mathtt{readBl}(r); rst) \rightarrow (caid \bullet M \bullet dst) \circ (c \bullet rst)}$$

$$\textsc{(PrRd}_4\textsc{)}\quad \frac{n = addr(r) \quad first(caid) = true \quad cid(caid) = c \quad status(M,n) = inv}{(caid \bullet M \bullet dst) \circ (c \bullet \mathtt{readBl}(r); rst) \rightarrow (caid \bullet M[n \mapsto \perp] \bullet dst; \mathtt{fetch}(n)) \circ (c \bullet \mathtt{readBl}(r); rst)}$$

$$\textsc{(PrWr}_1\textsc{)}\quad \frac{n = addr(r) \quad first(caid) = true \quad cid(caid) = c \quad status(M,n) = mo}{(caid \bullet M \bullet dst) \circ (c \bullet \mathtt{write}(r); rst) \rightarrow (caid \bullet M \bullet dst) \circ (c \bullet rst)}$$

$$\textsc{(PrWr}_2\textsc{)}\quad \frac{n = addr(r) \quad first(caid) = true \quad cid(caid) = c \quad status(M,n) = sh}{(caid \bullet M \bullet dst) \circ (c \bullet \mathtt{write}(r); rst) \xrightarrow{!RdX(n)} (caid \bullet M[n \mapsto mo] \bullet dst) \circ (c \bullet rst)}$$

$$\textsc{(PrWr}_3\textsc{)}\quad \frac{n = addr(r) \quad first(caid) = true \quad cid(caid) = c \quad status(M,n) \in \{inv, \perp\}}{(caid \bullet M \bullet dst) \circ (c \bullet \mathtt{write}(r); rst) \rightarrow (caid \bullet M[n \mapsto \perp] \bullet dst; \mathtt{fetch}(n)) \circ (c \bullet \mathtt{writeBl}(r); rst)}$$

$$\textsc{(PrWr}_4\textsc{)}\quad \frac{n = addr(r) \quad first(caid) = true \quad cid(caid) = c \quad status(M,n) = mo}{(caid \bullet M \bullet dst) \circ (c \bullet \mathtt{writeBl}(r); rst) \rightarrow (caid \bullet M \bullet dst) \circ (c \bullet rst)}$$

$$\textsc{(PrWr}_5\textsc{)}\quad \frac{n = addr(r) \quad first(caid) = true \quad cid(caid) = c \quad status(M,n) = sh}{(caid \bullet M \bullet dst) \circ (c \bullet \mathtt{writeBl}(r); rst) \xrightarrow{!RdX(n)} (caid \bullet M[n \mapsto mo] \bullet dst) \circ (c \bullet rst)}$$

$$\textsc{(PrWr}_6\textsc{)}\quad \frac{n = addr(r) \quad first(caid) = true \quad cid(caid) = c \quad status(M,n) = inv}{(caid \bullet M \bullet dst) \circ (c \bullet \mathtt{writeBl}(r); rst) \rightarrow (caid \bullet M[n \mapsto \perp] \bullet dst; \mathtt{fetch}(n)) \circ (c \bullet \mathtt{writeBl}(r); rst)}$$

$$\textsc{(Commit)}\quad \frac{n = addr(r) \quad first(caid) = true \quad cid(caid) = c}{(caid \bullet M \bullet dst) \circ (c \bullet \mathtt{commit}(r); rst) \rightarrow (caid \bullet M \bullet dst; \mathtt{flush}(n)) \circ (c \bullet rst)}$$

$$\textsc{(CommitAll)}\quad \frac{first(caid) = true \quad cid(caid) = c}{(caid \bullet M \bullet dst) \circ (c \bullet \mathtt{commit}; rst) \rightarrow (caid \bullet M \bullet dst; \mathtt{flush}) \circ (c \bullet rst)}$$

$$\textsc{(Skip)}\quad (caid \bullet M \bullet dst) \circ (c \bullet \mathtt{skip}; rst) \rightarrow (caid \bullet M \bullet dst) \circ (c \bullet rst)$$

$$\textsc{(Choice)}\quad (caid \bullet M \bullet dst) \circ (c \bullet dap_1 \sqcap dap_2; rst) \rightarrow (caid \bullet M \bullet dst) \circ (c \bullet dap_1; rst)$$

$$\textsc{(Repetition)}\quad (caid \bullet M \bullet dst) \circ (c \bullet dap^*; rst) \rightarrow (caid \bullet M \bullet dst) \circ (c \bullet (dap; dap^*) \sqcap \mathtt{skip}; rst)$$

Figure 5: Local semantics of task execution in a core

PRWR$_4$. If the status of the cache line is shared, an invalidation message $!RdX(n)$ will be broadcast to all other nodes before the execution proceeds, captured by PRWR$_5$. Rule PRWR$_6$ handles repeated invalidations of the memory block n. Note that rules PRWR$_2$ and PRWR$_5$, which are labelled transitions, will globally synchronise the execution with all the nodes. The execution of these rules will be triggered by the global rules GLOBAL-SYNCH and NODE-SYNCH in Figure 8 and the rule SEND-MESSAGE$_1$ in Figure 7, which are explained later in this section.

Rules COMMIT and COMMITALL are used to flush a single cache line and the entire cache, respectively. Rule SKIP consumes the **skip**-statement and continues executing the rest of the task. With the commutative $\sqcap$-operator, rule CHOICE non-deterministically selects a statement from $dap_1 \sqcap dap_2$ for execution. Finally, rule REPETITION repeats the statement *dap* zero or more times. In order to ensure data consistency between main memory and individual caches, a task is always appended with a **commit**-statement when it is scheduled (see rule PAR-TASK-SCHEDULER in Figure 8), which ensures that all modified data in the cache will be flushed before a new task is assigned to that core.

The rules which are local to the cache hierarchy in a node are shown in Figure 6. Since caches are exclusive, rule COMMIT-FLUSH-IGNORE$_1$ discards a **flush**(n) if the block n is in either *sh* or *inv* state. If it is the last level cache and block n is undefined in the cache, **flush**(n) is also discarded, as captured in rule COMMIT-FLUSH-IGNORE$_2$. Rule COMMIT-FLUSH-LC captures the case where the cache is not at the last level and block n is undefined, i.e., $M_i(n) = \perp$, the **flush**(n)-instruction will then be propagated to the next level cache. Rule COMMITALL-FLUSH flushes all the modified memory blocks in a cache, by recursively reducing it to a sequence of **flush**(n)-instructions, for all blocks n whose state is *mo*. Rule COMMITALL-FLUSH-LC propagates the **flush**-instruction to the next level cache if there are no modified blocks in the current cache and if the current cache is not at the last level; otherwise, the instruction will be discarded by rule COMMITALL-Flush-Ignore. The remaining rules of Figure 6 deal with interactions between two adjacent cache levels. Rules LC-HIT$_1$ and LC-HIT$_2$ capture the case where cache $caid_i$ needs to fetch block n and finds it in either *sh* or *mo* state in the next level cache. The function $select(M_i, n)$ determines the address where the block should be placed, based on a cache associativity (e.g., random, set associativity and direct map) and a replacement policy (e.g., random and LRU). If a block needs to be evicted in $caid_i$ to give space to block n, which is determined by the premise $select(M_i, n) = n_i$, the selected block n_i in $caid_i$ is swapped with block n from $caid_j$, as shown in rule LC-HIT$_1$; otherwise, as captured in rule LC-HIT$_2$ with condition $select(M_i, n) = n$, block n can be directly transferred to $caid_i$ and removed from $caid_j$, without moving any block from $caid_i$ to $caid_i$. Rule LC-MISS$_1$ shows how **fetch**(n)-instructions are propagated to lower levels in the cache hierarchy by replacing the instruction **fetch**(n) with **fetchBl**(n) in $caid_i$ and append the **fetch**(n)-instruction to the data instruction queue in $caid_j$. Rule LC-MISS$_2$ capture the case in which the newly fetched block n in the next level $caid_j$ gets invalided by the other core while the cache $caid_i$ is blocked. When a cache is blocked waiting for block n, which is found at the next level cache, execution may be resumed by rules LC-BLOCK-FINISH$_1$ and LC-BLOCK-FINISH$_2$, which are similar to rules LC-HIT$_1$ and LC-HIT$_2$ respectively. If the block cannot be found in any local cache, we have a *cache miss*: execution is

(COMMIT-FLUSH-IGNORE$_1$)

$$\frac{status(M,n) \in \{sh, inv\}}{(caid \bullet M \bullet \texttt{flush}(n); dst) \rightarrow (caid \bullet M \bullet dst)}$$

(COMMIT-FLUSH-IGNORE$_2$)

$$\frac{last(caid) = true \qquad M(n) = \bot}{(caid \bullet M \bullet \texttt{flush}(n); dst) \rightarrow (caid \bullet M \bullet dst)}$$

(COMMITALL-FLUSH)

$$\frac{status(M,n) = mo \qquad \texttt{flush}(n) \notin dst}{(caid \bullet M \bullet \texttt{flush}; dst) \rightarrow (caid \bullet M \bullet \texttt{flush}(n); \texttt{flush}; dst)}$$

(COMMIT-FLUSH-LC)

$$\frac{M_i(n) = \bot \quad lid(caid_j) = lid(caid_i) + 1 \qquad cid(caid_i) = cid(caid_j)}{(caid_i \bullet M_i \bullet \texttt{flush}(n); dst_i) \circ (caid_j \bullet M_j \bullet dst_j) \rightarrow (caid_i \bullet M_i \bullet dst_i) \circ (caid_j \bullet M_j \bullet dst_j; \texttt{flush}(n))}$$

(COMMITALL-FLUSH-LC)

$$\frac{\forall n \in dom(M_i).status(M_i,n) \neq mo \quad lid(caid_j) = lid(caid_i) + 1 \qquad cid(caid_i) = cid(caid_j)}{(caid_i \bullet M_i \bullet \texttt{flush}; dst_i) \circ (caid_j \bullet M_j \bullet dst_j) \rightarrow (caid_i \bullet M_i \bullet dst_i) \circ (caid_j \bullet M_j \bullet dst_j; \texttt{flush})}$$

(COMMITALL-FLUSH-IGNORE)

$$\frac{last(caid) = true \quad \forall n \in dom(M).status(M,n) \neq mo}{(caid \bullet M \bullet \texttt{flush}; dst) \rightarrow (caid \bullet M \bullet dst)}$$

(LC-HIT$_1$)

$$\frac{\begin{array}{lll} status(M_i,n_i) = s_i & status(M_j,n) = s_j & s_j \in \{sh, mo\} \\ lid(caid_j) = lid(caid_i) + 1 & cid(caid_i) = cid(caid_j) & select(M_i,n) = n_i \end{array}}{(caid_i \bullet M_i \bullet \texttt{fetch}(n); dst_i) \circ (caid_j \bullet M_j \bullet dst_j) \rightarrow (caid_i \bullet M_i[n_i \mapsto \bot, n \mapsto s_j] \bullet dst_i) \circ (caid_j \bullet M_j[n \mapsto \bot, n_i \mapsto s_i] \bullet dst_j)}$$

(LC-HIT$_2$)

$$\frac{lid(caid_j) = lid(caid_i) + 1 \quad cid(caid_i) = cid(caid_j) \quad select(M_i,n) = n \quad status(M_j,n) = s_j \quad s_j \in \{sh, mo\}}{(caid_i \bullet M_i \bullet \texttt{fetch}(n); dst_i) \circ (caid_j \bullet M_j \bullet dst_j) \rightarrow (caid_i \bullet M_i[n \mapsto s_j] \bullet dst_i) \circ (caid_j \bullet M_j[n \mapsto \bot] \bullet dst_j)}$$

(LC-MISS$_1$)

$$\frac{lid(caid_j) = lid(caid_i) + 1 \quad cid(caid_i) = cid(caid_j) \quad status(M_j,n) \in \{inv, \bot\}}{(caid_i \bullet M_i \bullet \texttt{fetch}(n); dst_i) \circ (caid_j \bullet M_j \bullet dst_j) \rightarrow (caid_i \bullet M_i \bullet \texttt{fetchBl}(n); dst_i) \circ (caid_j \bullet M_j[n \mapsto \bot] \bullet dst_j; \texttt{fetch}(n))}$$

(LC-MISS$_2$)

$$\frac{lid(caid_j) = lid(caid_i) + 1 \quad cid(caid_i) = cid(caid_j) \quad status(M_j,n) = inv}{(caid_i \bullet M_i \bullet \texttt{fetchBl}(n); dst_i) \circ (caid_j \bullet M_j \bullet dst_j) \rightarrow (caid_i \bullet M_i \bullet \texttt{fetchBl}(n); dst_i) \circ (caid_j \bullet M_j[n \mapsto \bot] \bullet dst_j; \texttt{fetch}(n))}$$

(LC-BLOCK-FINISH$_1$)

$$\frac{\begin{array}{lll} lid(caid_j) = lid(caid_i) + 1 & cid(caid_i) = cid(caid_j) & select(M_i,n) = n_i \\ status(M_i,n_i) = s_i & status(M_j,n) = s_j & s_j \in \{sh, mo\} \end{array}}{(caid_i \bullet M_i \bullet \texttt{fetchBl}(n); dst_i) \circ (caid_j \bullet M_j \bullet dst_j) \rightarrow (caid_i \bullet M_i[n_i \mapsto \bot, n \mapsto s_j] \bullet dst_i) \circ (caid_j \bullet M_j[n \mapsto \bot, n_i \mapsto s_i] \bullet dst_j)}$$

(LC-BLOCK-FINISH$_2$)

$$\frac{\begin{array}{l} lid(caid_j) = lid(caid_i) + 1 \quad cid(caid_i) = cid(caid_j) \\ select(M_i,n) = n \quad status(M_j,n) = s_j \quad s_j \in \{sh, mo\} \end{array}}{(caid_i \bullet M_i \bullet \texttt{fetchBl}(n); dst_i) \circ (caid_j \bullet M_j \bullet dst_j) \rightarrow (caid_i \bullet M_i[n \mapsto s_j] \bullet dst_i) \circ (caid_j \bullet M_j[n \mapsto \bot] \bullet dst_j)}$$

(LLC-MISS)

$$\frac{last(caid) = true \qquad status(M,n) \in \{inv, \bot\}}{(caid \bullet M \bullet \texttt{fetch}(n); dst) \xrightarrow{!Rd(n)} (caid \bullet M[n \mapsto \bot] \bullet \texttt{fetchBl}(n); dst)}$$

Figure 6: Local semantics of flush and fetch instructions in the cache hierarchy.

$$\frac{status(M,n) = sh}{caid \bullet M \bullet dst \xrightarrow{?RdX(n)} caid \bullet M[n \mapsto inv] \bullet dst} \text{ (INVALIDATE-ONE-LINE)}$$

$$\frac{status(M,n) \in \{inv, \perp\}}{caid \bullet M \bullet dst \xrightarrow{?RdX(n)} caid \bullet M \bullet dst} \text{ (IGNORE-INVALIDATE-ONE-LINE)}$$

$$\frac{status(M,n) = mo}{caid \bullet M \bullet dst \xrightarrow{?Rd(n)} caid \bullet M \bullet \texttt{flush}(n); dst} \text{ (FLUSH-ONE-LINE)}$$

$$\frac{status(M,n) \in \{sh, inv, \perp\}}{caid \bullet M \bullet dst \xrightarrow{?Rd(n)} caid \bullet M \bullet dst} \text{ (IGNORE-FLUSH-ONE-LINE)}$$

$$\frac{\overline{Ca} \xrightarrow{R} \overline{Ca}' \qquad \overline{Ca}' \circ CR \xrightarrow{\xi} \overline{Ca}'' \circ CR'}{\overline{Ca} \circ CR \xrightarrow{R \cup \{\xi\}} \overline{Ca}'' \circ CR'} \text{ (RECEIVE-SEND-MESSAGE)}$$

$$\frac{\overline{Ca}_1 \xrightarrow{R} \overline{Ca}_1' \qquad \overline{Ca}_2 \xrightarrow{R} \overline{Ca}_2'}{\overline{Ca}_1 \circ \overline{Ca}_2 \xrightarrow{R} \overline{Ca}_1' \circ \overline{Ca}_2'} \text{ (RECEIVE-MESSAGE}_1\text{)}$$

$$\frac{Ca \xrightarrow{\pi} Ca' \qquad Ca' \xrightarrow{R} Ca''}{Ca \xrightarrow{\{\pi\} \cup R} Ca''} \text{ (RECEIVE-MESSAGE}_2\text{)}$$

$$\frac{Ca \circ CR \xrightarrow{!RdX(n)} Ca' \circ CR'}{\{Ca\} \uplus \overline{Ca} \circ CR \xrightarrow{!RdX(n)} \{Ca'\} \uplus \overline{Ca} \circ CR'} \text{ (SEND-MESSAGE}_1\text{)}$$

$$\frac{Ca \xrightarrow{!Rd(n)} Ca'}{\{Ca\} \uplus \overline{Ca} \circ CR \xrightarrow{!Rd(n)} \{Ca'\} \uplus \overline{Ca} \circ CR} \text{ (SEND-MESSAGE}_2\text{)}$$

$$Ca \xrightarrow{\emptyset} Ca \text{ (REC-MSG-EMPTY)}$$

$$\frac{M \xrightarrow{R} M' \qquad M' \xrightarrow{\pi} M''}{M \xrightarrow{R \cup \{\pi\}} M''} \text{ (MAIN-MEMORY}_1\text{)}$$

$$M \xrightarrow{\emptyset} M \text{ (MAIN-MEMORY}_2\text{)}$$

$$M \xrightarrow{?RdX(n)} M[n \mapsto inv] \text{ (ONE-BLOCK-MAIN-MEMORY}_1\text{)}$$

$$M \xrightarrow{?Rd(n)} M \text{ (ONE-BLOCK- MAIN-MEMORY}_2\text{)}$$

Figure 7: Local semantics for message passing in cache coherent multicore architectures.

blocked by the instruction **`fetchBl`**(n), and a read request $!Rd(n)$ will be broadcast, represented by the label on the transition in rule LLC-MISS.

The transition steps which capture the sending and receiving of broadcast messages for a node and in main memory, are shown in Figure 7. These steps use the label mechanism described in the beginning of Section 3.3, where label ξ represents an individual sent message ($!Rd(n)$ or $!RdX(n)$), label π represents an individual received message ($?Rd(n)$ or $?RdX(n)$), and S and R are sets of ξ and π, respectively. Each node which receives a set R of messages from other nodes, may broadcast a read or write request ξ in the same synchronized step (see rule GLOBAL-SYNCH in Figure 8). These received messages are broadcast to all caches in the same node, which are handled by the rules in Figure 7. The rules are divided in three categories, the first category contains rules handling one single received message in a cache, the second contains rules for distributing a set R of received messages to all caches in the node and one send message ξ to the broadcast medium, and the third contains rules dealing with messages which arrive at main memory.

When a cache receives an invalidation request $?RdX(n)$ and has block n with status

sh, the cache will update the status of block *n* to *inv* in rule INV-ONE-LINE; otherwise, the message will be ignored in rule IGNORE-INVALIDATE-ONE-LINE. When a cache receives a read request $?Rd(n)$ and has block *n* with status *mo*, as in rule FLUSH-ONE-LINE, a **`flush`**-instruction is appended to *dst* to prioritise the flushing of the modified copy (to avoid a potential deadlock caused by cyclic waiting for modified data to be flushed from other node to main memory). The received messages are ignored in all other cases by rule IGNORE-FLUSH-ONE-LINE.

Rule RECEIVE-SEND-MESSAGE ensures that a node can receive multiple messages *R*, but can only send *one* message ξ. In the first premise, the received messages are broadcast to caches at different levels in the same node. The second premise handles the sending of messages by a cache at either the first or last level. In rule SEND-MESSAGE$_1$, the cache *Ca* sends an invalidation message; note that here *Ca* must be a first level cache to match the rules in the local semantics from Figure 5. Similarly, in rule SEND-MESSAGE$_2$ the cache *Ca* can send a read request as; note that here *Ca* must be a last level cache to match the rule LLC-MISS in Figure 6. The receipt of messages is captured by the three rules REC-MSG-EMPTY, RECEIVE- MESSAGE$_1$, and RECEIVE-MESSAGE$_2$, which propagate the set of received messages to all levels of caches in a node.

The remaining rules of Figure 7 cover transition steps for receiving messages by the main memory. In general, the main memory ignores all read requests, but responds to invalidation requests by setting the status of the corresponding block to *inv* as in rule ONE-BLOCK-MAIN-MEMORY$_1$. Rule MAIN-MEMORY$_1$ shows that main memory receives all messages involved in a synchronous transition. Note rule GLOBAL-SYNCH in Figure 8 ensures that *R* contains only one request for each memory block, which avoids data races when accessing multiple locations in parallel.

Global Semantics. The rules at the global level model the abstract communication medium: these rules capture interactions between different components in the configuration and ensure coherence between caches and main memory. The global transition rules are given in Figure 8. Rules GLOBAL-SYNCH and GLOBAL-ASYNCH are the two topmost rules performing the global steps. These rules coordinate and synchronise all transitions.

The first topmost rule GLOBAL-SYNCH captures global synchronisation for a non-empty set *S*. In this rule, the different read and invalidation requests are being broadcast. To maintain data consistency, the different components must process these requests at the same time. Note that to apply rule GLOBAL-SYNCH, the set *S* must contain at most one request per address. This is ensured by the premise $oneReqPerAddr(S)$, which is defined by the predicate

$$oneReqPerAddr(S) \hat{=} \forall n \in Addr \cdot \neg(!Rd(n) \in S \wedge !RdX(n) \in S).$$

The set *R* of receive labels is given as the *dual* of *S*. For synchronisation, the transition is decomposed into a premise for main memory with labels *R*, and another for the nodes with labels *S*. The former is handled by rules corresponding to main memory in Figure 7 and the latter by rule NODE-SYNCH.

Rule NODE-SYNCH handles synchronisation among nodes via messages. It distributes the labels over the nodes by recursively decomposing *S* into sets of send and

GLOBAL RULES

$$\frac{S \neq \emptyset \quad oneReqPerAddr(S) \quad R = dual(S) \quad M \xrightarrow{R} M' \quad \overline{Ca} \circ \overline{CR} \xrightarrow{S} \overline{Ca}' \circ \overline{CR}'}{M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} \xrightarrow{S} M' \circ \overline{T} \circ \overline{Ca}' \circ \overline{CR}'} \text{ (GLOBAL-SYNCH)}$$

$$\frac{\begin{array}{c}\overline{CR} = \overline{CR}_1 \uplus \overline{CR}_2 \uplus \overline{CR}_3 \quad \overline{Ca} = \overline{Ca}_1 \uplus \overline{Ca}_2 \uplus \overline{Ca}_3 \uplus \overline{Ca}_4 \\ belongs(\overline{Ca}_3, \overline{CR}_3) \quad M \circ \overline{Ca}_1 \to M' \circ \overline{Ca}_1' \quad \overline{Ca}_2 \to \overline{Ca}_2' \\ \overline{T} \circ \overline{CR}_2 \to \overline{T}' \circ \overline{CR}_2' \quad \overline{Ca}_3 \circ \overline{CR}_3 \to \overline{Ca}_3' \circ \overline{CR}_3' \\ \overline{CR}' = \overline{CR}_1 \cup \overline{CR}_2' \cup \overline{CR}_3' \quad \overline{Ca}' = \overline{Ca}_1' \cup \overline{Ca}_2' \cup \overline{Ca}_3' \cup \overline{Ca}_4\end{array}}{M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} \xrightarrow{\emptyset} M' \circ \overline{T}' \circ \overline{Ca}' \circ \overline{CR}'} \text{ (GLOBAL-ASYNCH)}$$

SYNCHRONOUS RULES

$$\frac{\begin{array}{c}\overline{CR} = \overline{CR}_1 \uplus \overline{CR}_2 \quad \overline{Ca} = \overline{Ca}_1 \uplus \overline{Ca}_2 \quad S = S_1 \uplus S_2 \\ belongs(\overline{Ca}_1, \overline{CR}_1) \quad belongs(\overline{Ca}_2, \overline{CR}_2) \quad R_1 = dual(S_1) \quad R_2 = dual(S_2) \\ \overline{Ca}_1 \circ \overline{CR}_1 \xrightarrow{S_1 \cup R_2 \cup R} \overline{Ca}_1' \circ \overline{CR}_1' \quad \overline{Ca}_2 \circ \overline{CR}_2 \xrightarrow{S_2 \cup R_1 \cup R} \overline{Ca}_2' \circ \overline{CR}_2' \\ \overline{CR}' = \overline{CR}_1' \cup \overline{CR}_2' \quad \overline{Ca}' = \overline{Ca}_1' \cup \overline{Ca}_2'\end{array}}{\overline{Ca} \circ \overline{CR} \xrightarrow{S \cup R} \overline{Ca}' \circ \overline{CR}'} \text{ (NODE-SYNCH)}$$

ASYNCHRONOUS RULES

$$\frac{\begin{array}{c}M[n \mapsto st] \circ Ca \to M[n \mapsto st'] \circ Ca' \\ M' = M[n \mapsto \bot] \quad M' \circ \overline{Ca} \to M'' \circ \overline{Ca}'\end{array}}{\begin{array}{c}M[n \mapsto st] \circ \{Ca\} \cup \overline{Ca} \to \\ M''[n \mapsto st'] \circ \{Ca'\} \cup \overline{Ca}'\end{array}} \text{ (PAR-MEMORY-ACCESS)}$$

$$\frac{status(M', n) = mo \quad status(M, n) = inv}{\begin{array}{c}M \circ (caid \bullet M' \bullet \texttt{flush}(n); dst) \to \\ M[n \mapsto sh] \circ (caid \bullet M'[n \mapsto sh] \bullet dst)\end{array}} \text{ (FLUSH)}$$

$$\frac{\begin{array}{c}last(caid) = true \\ select(M', n) = n \quad status(M, n) = sh\end{array}}{\begin{array}{c}M \circ (caid \bullet M' \bullet \texttt{fetchBl}(n); dst) \to \\ M \circ (caid \bullet M'[n \mapsto sh] \bullet dst)\end{array}} \text{ (FETCH}_1\text{)}$$

$$\frac{\begin{array}{c}last(caid) = true \quad select(M', n) = n' \\ n' \neq n \quad status(M', n') \neq mo \quad status(M, n) = sh\end{array}}{\begin{array}{c}M \circ (caid \bullet M' \bullet \texttt{fetchBl}(n); dst) \to \\ M \circ (caid \bullet M'[n' \mapsto \bot, n \mapsto sh] \bullet dst)\end{array}} \text{ (FETCH}_2\text{)}$$

$$\frac{\begin{array}{c}last(caid) = true \quad select(M', n) = n' \\ n' \neq n \quad status(M', n') = mo\end{array}}{\begin{array}{c}M \circ (caid \bullet M' \bullet \texttt{fetchBl}(n); dst) \to \\ M \circ (caid \bullet M' \bullet \texttt{flush}(n'); \texttt{fetchBl}(n); dst)\end{array}} \text{ (FETCH}_3\text{)}$$

$$\frac{\begin{array}{c}\overline{Ca} = \overline{Ca}_1 \uplus \overline{Ca}_2 \quad \overline{Ca}_1 \to \overline{Ca}_1' \\ \overline{Ca}' = \overline{Ca}_1' \cup \overline{Ca}_2' \quad \overline{Ca}_2 \to \overline{Ca}_2'\end{array}}{\overline{Ca} \to \overline{Ca}'} \text{ (PAR-CACHE)}$$

$$\frac{\begin{array}{c}\overline{CR} = \{CR_1\} \uplus \overline{CR}_2 \quad \overline{Ca} = \overline{Ca}_1 \uplus \overline{Ca}_2 \quad belongs(\overline{Ca}_1, CR_1) \quad belongs(\overline{Ca}_2, \overline{CR}_2) \\ \overline{Ca}_1 \circ CR_1 \to \overline{Ca}_1' \circ CR_1' \quad \overline{Ca}_2 \circ \overline{CR}_2 \to \overline{Ca}_2' \circ \overline{CR}_2' \quad \overline{CR}' = \{CR_1'\} \cup \overline{CR}_2' \quad \overline{Ca}' = \overline{Ca}_1' \cup \overline{Ca}_2'\end{array}}{\overline{Ca} \circ \overline{CR} \to \overline{Ca}' \circ \overline{CR}'} \text{ (PAR-INTERNAL-STEPS)}$$

$$\frac{\overline{T} \circ \overline{CR} \to \overline{T}' \circ \overline{CR}'}{\begin{array}{c}\overline{T} \circ \overline{CR} \uplus \{(cid \circ \texttt{spawn}(T); dap)\} \to \\ (\overline{T}' \cup \{T\}) \circ \overline{CR}' \uplus \{(cid \bullet dap)\}\end{array}} \text{ (PAR-TASK-SPAWN)}$$

$$\frac{\overline{T}' = \overline{T} \setminus \{T\} \quad dap = Tb(T) \quad \overline{T}' \circ \overline{CR} \to \overline{T}'' \circ \overline{CR}'}{\overline{T} \circ \overline{CR} \uplus \{(cid \bullet \varepsilon)\} \to \overline{T}'' \circ \overline{CR}' \uplus \{(cid \bullet dap; \texttt{commit})\}} \text{ (PAR-TASK-SCHEDULER)}$$

Figure 8: Global semantics for cache coherent multicore architectures. The partition operator $\uplus$ is defined as $X_1 \uplus X_2 = X$ iff $X_1 \cup X_2 = X$ and $X_1 \cap X_2 = \emptyset$.

receive labels for sets of nodes containing cores ($\overline{CR}_1$ and $\overline{CR}_2$) and caches ($\overline{Ca}_1$ and $\overline{Ca}_2$), such that each set eventually contains at most one send label ξ (either $!Rd(n)$ or $!RdX(n)$) to match transitions in the local rules. The predicate $belongs(\overline{Ca}, \overline{CR})$ expresses that each cache in $\overline{Ca}$ belongs to a core in $\overline{CR}$. The decomposition of S repeats recursively until the dual labels have been generated for each single node. Since the sets S_1 and S_2 are disjoint by construction, their dual sets R_1 and R_2 are also disjoint. Consequently, the messages in S_1 and R_2, correspondingly S_2 and R_1, are disjoint. Together with rule RECEIVE-SEND-MESSAGE in Figure 7, this ensures that the sender of a message ξ does not receive its dual π. Note that rules GLOBAL-SYNCH and NODE-SYNCH together with SEND-MESSAGE$_1$ in Figure 7 enforce synchronisation between the different nodes.

The other topmost rule GLOBAL-ASYNCH captures parallel transitions when the label set is empty. These asynchronous transitions can be local to individual nodes and caches (e.g., rules PAR-INTERNAL-STEPS and PAR-CACHE), parallel memory accesses (e.g., rule PAR-MEMORY-ACCESS), or the parallel spawning and scheduling of new tasks (e.g., rules PAR-TASK-SPAWN and PAR-TASK-SCHEDULER). Rule PAR-MEMORY-ACCESS allows parallel access by multiple caches to different addresses in main memory, but only sequential access to the same address by multiple caches to ensure data consistency. Rule PAR-TASK-SPAWN adds a new task identifier to the task pool and PAR-TASK-SCHEDULER looks up a task identifier T in the task table and schedules the corresponding task to a core. Adding the statement **`commit`** to the end of the scheduled task in this rule ensures that all modified data will be flushed before the next task is executed on the same core.

Rules FLUSH and FETCH capture the data movement between main memory M and a cache. A cache at any level can flush data to main memory. Rule FLUSH updates a block in main memory with the modified copy in the cache and sets the status to *sh* in both the cache and main memory. On the contrary, only the last level cache can fetch data from main memory. Rules FETCH$_1$ and FETCH$_2$ copy the data to the cache if no eviction is required, or if the block to be evicted has status *sh*. If eviction is needed and the block chosen by the *select* function has status *mo*, it will be first flushed in rule FETCH$_3$ before the requested block can be fetched.

4. Correctness of the Model

In this section, we discuss the correctness of our model. We consider standard correctness properties for data consistency and cache coherence, based on the literature [13, 43], including the preservation of program order in each core, the absence of data races, and no access to stale data. In addition, we show that nodes cannot mutually block each other in any reachable configuration in the model. We prove these properties for architectures with any number of cores and caches, orchestrated by the MSI protocol for cache coherence. To prove these properties, the formal model will be *instrumented with logging information* which does not affect the possible executions of a model. The preservation of these properties ensures that the model correctly captures coherence for multicore memory systems.

4.1. Instrumentation of the Formal Model

To show that task execution respects program order, we introduce *histories* to capture the local and global order of successful data accesses at runtime. Let an event ev, which reflects a successful data access to memory address n by a core cid, be written as either $R(cid,n)$ for read access or $W(cid,n)$ for write access. A local history h captures the data accesses local to a core, where a single event is appended at a time. Thus, the local history h logs all successful read and write operations executed so far by the current task in the core. The global history H records the *concurrent execution* of statements in different cores $\overline{CR}$; a set of events $\overline{ev}$ is appended to H in each global transition step. The local history h of a particular core is a projection of the global history H with respect to that core.

Let ε denote the empty history, ";" the concatenation operator, and $\preceq$ the reflexive prefix relation on histories. Formally, events ev, local histories h and global histories H are defined as follows:

$$\begin{array}{rcl} h & ::= & \varepsilon \mid h;ev \\ H & ::= & \varepsilon \mid H;\overline{ev} \\ ev & ::= & W(cid,n) \mid R(cid,n) \end{array}$$

In the transitions, local configurations $\overline{Ca} \bullet CR$ decorated with local histories are written as $\overline{Ca} \bullet CR : h$. The local reduction rules in Figures 5–7 are extended to describe the changes to the local history. Similarly, global configurations $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR}$ decorated with global histories are written $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$ and the global reduction rules of Figure 8 are similarly extended.

To show that all read and write operations access the most recent value of a memory block, we introduce *version numbers* to capture and track the changes in the values in an address (since our model abstracts from the actual data). Following Section 3.2, let $Int \times Status \cup \{\perp\}$ denote the extension of the set of pairs $\langle k, st \rangle$, where $k : Int$, with the undefined value $\perp$. We lift the memory mapping M to contain version numbers $k : Int$ and increment k whenever a core successfully gains write access to block address n. Thus, M has signature $M : Address \rightarrow Int \times Status \cup \{\perp\}$. Let the function $version(M,n)$ return the version number k of block address n in M or $\perp$ if $M(n) = \perp$.

Figure 9 shows the instrumented local transition rules which affect the local history or version number. These rules deal with successful read/write operations. The remaining rules of Figures 5–7, which have no affect on the history or version number, are omitted. In rules $\textsc{PrWr}_2$ and $\textsc{PrWr}_5$, which capture succesful write operations, the version number of block n is incremented when the status changes from sh to mo in the first level cache.

Figure 10 shows the instrumented global rules which affect the global history. These rules show that if local histories are extended with events, they will be merged into a set of events which extends the global history H after the parallel transition step. The remaining rules of Figure 8, which have no effect on the history, are unchanged and omitted.

4.2. Properties of the Instrumented Model

We now show that the multicore memory system defined in Section 3 is coherent, using the instrumented semantics of the previous section. We first prove that (1) the

$$\frac{n = addr(r) \quad first(caid) = true \quad cid(caid) = c \quad status(M,n) \in \{sh, mo\}}{(caid \bullet M \bullet dst) \circ (c \bullet \mathbf{read}(r); rst) : h \to (caid \bullet M \bullet dst) \circ (c \bullet rst) : h; R(c,n)} \text{ (PRRD}_1\text{)}$$

$$\frac{n = addr(r) \quad first(caid) = true \quad cid(caid) = c \quad status(M,n) \in \{sh, mo\}}{(caid \bullet M \bullet dst) \circ (c \bullet \mathbf{readBl}(r); rst) : h \to (caid \bullet M \bullet dst) \circ (c \bullet rst) : h; R(c,n)} \text{ (PRRD}_3\text{)}$$

$$\frac{n = addr(r) \quad first(caid) = true \quad cid(caid) = c \quad status(M,n) = mo}{(caid \bullet M \bullet dst) \circ (c \bullet \mathbf{write}(r); rst) : h \to (caid \bullet M \bullet dst) \circ (c \bullet rst) : h; W(c,n)} \text{ (PRWR}_1\text{)}$$

$$\frac{n = addr(r) \quad first(caid) = true \quad cid(caid) = c \quad M(n) = \langle k, sh \rangle}{(caid \bullet M \bullet dst) \circ (c \bullet \mathbf{write}(r); rst) : h \xrightarrow{!RdX(n)} (caid \bullet M[n \mapsto \langle k+1, mo \rangle] \bullet dst) \circ (c \bullet rst) : h; W(c,n)} \text{ (PRWR}_2\text{)}$$

$$\frac{n = addr(r) \quad first(caid) = true \quad cid(caid) = c \quad status(M,n) = mo}{(caid \bullet M \bullet dst) \circ (c \bullet \mathbf{writeBl}(r); rst) : h \to (caid \bullet M \bullet dst) \circ (c \bullet rst) : h; W(c,n)} \text{ (PRWR}_4\text{)}$$

$$\frac{n = addr(r) \quad first(caid) = true \quad cid(caid) = c \quad M(n) = \langle k, sh \rangle}{(caid \bullet M \bullet dst) \circ (c \bullet \mathbf{writeBl}(r); rst) : h \xrightarrow{!RdX(n)} (caid \bullet M[n \mapsto \langle k+1, mo \rangle] \bullet dst) \circ (c \bullet rst) : h; W(c,n)} \text{ (PRWR}_5\text{)}$$

$$\frac{\overline{Ca} \xrightarrow{R} \overline{Ca}' \qquad \overline{Ca}' \circ CR : h \xrightarrow{\xi} \overline{Ca}'' \circ CR' : h'}{\overline{Ca} \circ CR : h \xrightarrow{R \cup \{\xi\}} \overline{Ca}'' \circ CR' : h'} \text{ (RECEIVE-SEND-MESSAGE)}$$

$$\frac{Ca_f \circ CR : h \xrightarrow{!RdX(n)} Ca_f' \circ CR' : h'}{\{Ca_f\} \uplus \overline{Ca} \circ CR : h \xrightarrow{!RdX(n)} \{Ca_f'\} \uplus \overline{Ca} \circ CR' : h'} \text{ (SEND-MESSAGE}_1\text{)}$$

Figure 9: Instrumented local semantics.

result of any execution of the global system is equivalent to interleaving the results of the data accesses from each core in some serial order, (2) task execution preserves program order, (3) for all memory blocks and for any synchronous or asynchronous parallel global step, cores cannot access stale data, and (4) there does not exist mutually blocked nodes in any reachable configuration in the model.

The first lemma shows that for reachable configurations there is at most one modified copy of each memory block among all caches, which captures the absence of data races in accessing memory blocks from main memory.

Lemma 1 (No data races)**.** *Let Ca_x be the cache $(caid_x \bullet M_x \bullet dst_x)$ and $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$ a reachable configuration. The following properties hold for $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$:*

(a) $\forall n \in dom(M).(status(M,n) = inv \Leftrightarrow \exists\, Ca_i \in \overline{Ca}.\ status(M_i,n) = mo)$

(b) $\forall n \in dom(M).\ (status(M,n) = inv \Leftrightarrow (\exists\, Ca_i \in \overline{Ca}.\ status(M_i,n) = mo)$
$\quad \wedge \forall\, Ca_j \in \overline{Ca} \setminus \{Ca_i\}\ .\ status(M_j,n) \in \{inv, \bot\})$

(c) $\forall n \in dom(M).\ status(M,n) = sh \Leftrightarrow \forall\, Ca_i \in \overline{Ca}.\ status(M_i,n) \neq mo$

(d) $\forall\, Ca_i \in \overline{Ca}\ , \forall n \in dom(M_i).\ (status(M_i,n) = sh \Rightarrow status(M,n) = sh)$

Proof. We show that these properties hold jointly as an invariant by induction over execution sequences. The proof is detailed in Appendix A.1. □

(Global-Synch)

$$\frac{S \neq \emptyset \qquad oneReqPerAddr(S) \qquad R = dual(S) \qquad M \xrightarrow{R} M' \qquad \overline{Ca} \circ \overline{CR} : H \xrightarrow{S} \overline{Ca}' \circ \overline{CR}' : H'}{M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H \xrightarrow{S} M' \circ \overline{T} \circ \overline{Ca}' \circ \overline{CR}' : H'}$$

(Node-Synch)

$$\frac{\begin{array}{c} \overline{CR} = \overline{CR}_1 \uplus \overline{CR}_2 \qquad \overline{Ca} = \overline{Ca}_1 \uplus \overline{Ca}_2 \qquad S = S_1 \uplus S_2 \\ belongs(\overline{Ca}_1, \overline{CR}_1) \qquad belongs(\overline{Ca}_2, \overline{CR}_2) \qquad R_1 = dual(S_1) \qquad R_2 = dual(S_2) \\ \overline{Ca}_1 \circ \overline{CR}_1 : H/cid(\overline{CR}_1) \xrightarrow{S_1 \cup R_2 \cup R} \overline{Ca}'_1 \circ \overline{CR}'_1 : H/cid(\overline{CR}_1); \overline{ev}_1 \\ \overline{Ca}_2 \circ \overline{CR}_2 : H/cid(\overline{CR}_2) \xrightarrow{S_2 \cup R_1 \cup R} \overline{Ca}'_2 \circ \overline{CR}'_2 : H/cid(\overline{CR}_2); \overline{ev}_2 \\ \overline{CR}' = \overline{CR}'_1 \cup \overline{CR}'_2 \qquad \overline{Ca}' = \overline{Ca}'_1 \cup \overline{Ca}'_2 \qquad H' = H; (\overline{ev_1} \cup \overline{ev_2}) \end{array}}{\overline{Ca} \circ \overline{CR} : H \xrightarrow{S \cup R} \overline{Ca}' \circ \overline{CR}' : H'}$$

(Global-Asynch)

$$\frac{\begin{array}{c} \overline{CR} = \overline{CR}_1 \uplus \overline{CR}_2 \uplus \overline{CR}_3 \qquad \overline{Ca} = \overline{Ca}_1 \uplus \overline{Ca}_2 \uplus \overline{Ca}_3 \uplus \overline{Ca}_4 \\ belongs(\overline{Ca}_3, \overline{CR}_3) \qquad M \circ \overline{Ca}_1 \rightarrow M' \circ \overline{Ca}'_1 \qquad \overline{Ca}_2 \rightarrow \overline{Ca}'_2 \\ \overline{T} \circ \overline{CR}_2 \rightarrow \overline{T}' \circ \overline{CR}'_2 \qquad \overline{Ca}_3 \circ \overline{CR}_3 : H/cid(\overline{CR}_3) \rightarrow \overline{Ca}'_3 \circ \overline{CR}'_3 : H/cid(\overline{CR}_3); \overline{ev} \\ \overline{CR}' = \overline{CR}_1 \cup \overline{CR}'_2 \cup \overline{CR}'_3 \qquad \overline{Ca}' = \overline{Ca}'_1 \cup \overline{Ca}'_2 \cup \overline{Ca}'_3 \cup \overline{Ca}_4 \qquad H' = H; \overline{ev} \end{array}}{M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H \xrightarrow{\emptyset} M' \circ \overline{T}' \circ \overline{Ca}' \circ \overline{CR}' : H'}$$

(Par-Internal-Steps)

$$\frac{\begin{array}{c} belongs(\overline{Ca}_1, CR_1) \qquad belongs(\overline{Ca}_2, \overline{CR}_2) \qquad H' = H; (\{ev_1\} \cup \overline{ev}_2) \\ \overline{CR} = \{CR_1\} \uplus \overline{CR}_2 \qquad \overline{CR}' = \{CR'_1\} \cup \overline{CR}'_2 \qquad \overline{Ca}_1 \circ CR_1 : H/cid(CR_1) \rightarrow \overline{Ca}'_1 \circ CR'_1 : H/cid(CR_1); ev_1 \\ \overline{Ca} = \overline{Ca}_1 \uplus \overline{Ca}_2 \qquad \overline{Ca}' = \overline{Ca}'_1 \cup \overline{Ca}'_2 \qquad \overline{Ca}_2 \circ \overline{CR}_2 : H/cid(\overline{CR}_2) \rightarrow \overline{Ca}'_2 \circ \overline{CR}'_2 : H'/cid(\overline{CR}); \overline{ev}_2 \end{array}}{\overline{Ca} \circ \overline{CR} : H \rightarrow \overline{Ca}' \circ \overline{CR}' : H'}$$

Figure 10: Instrumented global semantics.

We next show that shared for reachable configurations, copies of a memory block in different caches always have the same version number.

Lemma 2 (Consistent shared copies)**.** *Let* $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$ *be a reachable configuration where* $status(M, n) = sh$*. If* $(caid_i \bullet M_i \bullet dst_i) \in \overline{Ca}$ *such that* $status(M_i, n) = sh$*, then* $version(M, n) = version(M_i, n)$*.*

Proof. The proof is by induction over execution sequences and is detailed in Appendix A.2. □

The following lemma states that successful parallel read accesses to a memory block always get the same version.

Lemma 3 (Consistent parallel read access)**.** *Let* CR_x *be a core* $(c_x \bullet rst_x)$ *and* Ca_y *be a cache* $(caid_y \bullet M_y \bullet dst_y)$*. Assume further that* $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$ *is a reachable configuration where* $CR_k, CR_l \in \overline{CR}$ *and* $Ca_k, Ca_l \in \overline{Ca}$*. If*

$$M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H \xrightarrow{S} M' \circ \overline{T}' \circ \overline{Ca}' \circ \overline{CR}' : H; \overline{ev}$$

such that $cid(caid_k) = c_k$ *and* $cid(caid_l) = c_l$ *with* $first(caid_k) = first(caid_l) = true$*, then* $\forall i, j, \forall n \, . \, R(c_i,n), R(c_j,n) \in \overline{ev}$*,* $version(M'_i,n) = version(M'_j,n)$*.*

Proof. The proof is by induction over execution sequences and is detailed in Appendix A.3. □

Lemma 3 shows that parallel read accesses can be broken down into a sequence of read operations. Since Lemma 1 guarantees that at most one modified copy of a memory block exists among the caches, parallel read and write accesses to the same memory block are impossible. Therefore, a global step consisting of multiple parallel read and write accesses is equivalent to a sequence of read and write accesses. The lemma can be easily lifted to a sequence of global steps, which entails a sequence of sets of events.

Theorem 1 (Serialisation of global transitions)**.** *Let* S_1 *and* S_2 *be (possibly empty) sets of labels such that* $S_1 \uplus S_2 = S$ *and* $oneReqPerAddr(S)$*.*
If $Config : H \xrightarrow{S} Config' : H;\overline{ev}$*, then* $Config : H \xrightarrow{S_1} \xrightarrow{S_2} Config' : H;\overline{ev_1};\overline{ev_2}$ *where* $\overline{ev_1} \cup \overline{ev_2} = \overline{ev}$*.*

Proof. Follows directly from Lemma 3 since the predicate $oneReqPerAddr(S)$ ensures that S contains at most one request per address. □

In the following, we formalise the semantics of runtime statements *rst* in terms of sets of local histories (or traces).

Definition 1 (Trace semantics of local task execution)**.** *Let* $addr(r) = n$*. The* traces *of a task rst executed on core c, written* $[\![rst]\!]_c$*, is defined inductively as follows:*

$$
\begin{array}{ll}
[\![\texttt{read}(r)]\!]_c = \{R(c,n)\} & [\![\texttt{commit}(r)]\!]_c = \{\varepsilon\} \\
[\![\texttt{readBl}(r)]\!]_c = \{R(c,n)\} & [\![\texttt{commit}]\!]_c = \{\varepsilon\} \\
[\![\texttt{write}(r)]\!]_c = \{W(c,n)\} & [\![\texttt{skip}]\!]_c = \{\varepsilon\} \\
[\![\texttt{writeBl}(r)]\!]_c = \{W(c,n)\} & [\![\texttt{spawn}(T)]\!]_c = \{\varepsilon\} \\
[\![(dap_1 \sqcap dap_2)]\!]_c = [\![dap_1]\!]_c \cup [\![dap_2]\!]_c & [\![dap^*]\!]_c = [\![dap;dap^*]\!]_c \cup [\![\texttt{skip}]\!]_c \\
[\![(rst_1;rst_2)]\!]_c = \{\tau_1;\tau_2 | \tau_1 \in [\![rst_1]\!]_c, \tau_2 \in [\![rst_2]\!]_c\} &
\end{array}
$$

Intuitively, $[\![rst]\!]_c$ reflects the possible *program orders* of *rst* in terms of read and write accesses when executing *rst* directly on main memory. The following lemma and corollary show that executions in a core preserve this program order.

Lemma 4 (Preservation of trace semantics)**.** *If* $(c \bullet rst) : \varepsilon \rightarrow^* (c \bullet rst') : h$*, then* $\{h;\tau \mid \tau \in [\![rst']\!]_c\} \subseteq [\![rst]\!]_c$.

Proof. The proof is by induction over local transitions. It is detailed in Appendix A.4. □

Corollary 1 (Preservation of program order)**.** *If* $(c \bullet rst) : h_1 \rightarrow^* (c \bullet rst') : h_1;h_2$*, where* h_2 *is the sequence of events produced by the transition step(s) from rst to rst'*, *then* $h_2 \preceq h$ *for some* $h \in [\![rst]\!]_c$*.*

Proof. Since h_2 is the sequence of events produced by the transition from *rst* to *rst*$'$, we get $\{h_2;\tau \mid \tau \in [\![rst']\!]_c\} \subseteq [\![rst]\!]_c$ by Lemma 4. Thus, $h_2 \preceq h$ for some $h \in [\![rst]\!]_c$. □

Corollary 1 establishes the local program order for the data access operations of each individual core. In fact, the model's formalisation of the MSI protocol preserves *sequential consistency* [24] in the sense that the result of any execution of the proposed model of multicore memory architectures is equivalent to interleaving the results of executing the operations of each core in some sequential order.

To show that cores in our formal model never access stale values from memory blocks, we now define *the most recent value* of a memory block as follows:

Definition 2 (Most recent value)**.** *Let $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$ be a global configuration, n a memory location, and $Ca_i \in \overline{Ca}$ a cache such that $Ca_i = (caid_i \bullet M_i \bullet dst_i)$. Then $M_i(n)$ has* the most recent value *if the following holds:*

(a) If $M_i(n) = \langle k, sh\rangle$, then $M(n) = \langle k, sh\rangle$ and $\forall\ (caid_j \bullet M_j \bullet dst_j) \in \overline{Ca}\backslash\{Ca_i\}.\ status(M_j,n) = sh \Rightarrow M_j(n) = \langle k, sh\rangle$; or

(b) If $M_i(n) = \langle k_i, mo\rangle$, then $M(n) = \langle k, inv\rangle$ where $k_i > k$, and $\forall\ (caid_j \bullet M_j \bullet dst_j) \in \overline{Ca}\backslash\{Ca_i\}.\ M_j(n) = \langle k_j, inv\rangle$ where $k_i > k_j$.

Based on Definition 2, it follows from Lemma 2 that if a core succeeds to access a memory block, it will always get the most recent value.

Theorem 2 (No access to stale data)**.** *Let $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$ be a reachable configuration such that $CR_i = (c_i \bullet rst_i)$ for $CR_i \in \overline{CR}$ with local history $h_i = H/c_i$, $Ca_i = (caid_i \bullet M_i \bullet dst_i)$ for $Ca_i \in \overline{Ca}$ and $belongs(Ca_i, CR_i)$. Consider a block address n and an event $e \in \{R(c_i,n), W(c_i,n)\}$.*

If $Ca_i \circ CR_i : h_i \rightarrow Ca_i' \circ CR_i' : (h_i;e)$ or $Ca_i \circ CR_i : h_i \xrightarrow{!RdX(n)} Ca_i' \circ CR_i' : (h_i;e)$, then $M_i(n)$ has the most recent value.

Proof. The proof is by induction over execution sequences. The proof is detailed in Appendix A.5. □

Coherence of the multicore memory system. The properties presented above show that the formal model correctly captures a coherent multicore memory system [13]: Lemmas 1 and 2, together with Theorem 1, ensure that all accesses to a memory block can be performed in some serial order which is consistent with the result of the execution. Corollary 1 shows that the memory access by any order complies with the local program order of a task executing on a core; Theorem 2 shows that accesses to a memory block always get the most recently written value to that block.

To show nodes can not block each other in any reachable configuration in our formal model, we define in the following a *blocked* node and a set of *mutually blocked* nodes. We assume CR_x is a core $(c_x \bullet rst_x)$ and Ca_y is a cache $(caid_y \bullet M_y \bullet dst_y)$, and let a node be $\overline{Ca_z} \circ CR_z$ where $belongs(\overline{Ca_z}, CR_z)$.

Definition 3 (Blocked nodes)**.** *Consider a node $\overline{Ca_i} \circ CR_i$ in a configuration $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$, where $CR_i \in \overline{CR}$ and $\overline{Ca_i} \subseteq \overline{Ca}$. The node is* blocked *on an address n if the following holds:*

(a) $rst_i = \texttt{readBl}(r); rst'_i \vee rst_i = \texttt{writeBl}(r); rst'_i$, *where* $n = addr(r)$,

(b) $\forall Ca_j \in \overline{Ca}_i \,.\, Ca_j = (caid_j \bullet M_j \bullet \texttt{fetchBl}(n); dst_j)$,

(c) $status(M,n) = inv$.

Definition 4 (Mutually blocked nodes)**.** *Consider a set of k unique memory block addresses* $\{n_1,\dots,n_k\}$*, and a set of k nodes* $\{\overline{Ca}_1 \circ CR_1,\dots,\overline{Ca}_k \circ CR_k\}$ *in a configuration* $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$ *where* $k \geq 2$*. The set of nodes is* mutually blocked *if the following holds:*

(a) $\forall 1 \leq i \leq k$. $\overline{Ca}_i \circ CR_i$ *is blocked on* $n_i \in \{n_1,\dots,n_k\}$*, and for the node* $CR_{i+1} \circ \overline{Ca}_{i+1}$, $\exists (caid_{i+1} \bullet M_{i+1} \bullet dst_{i+1}) \in \overline{Ca}_{i+1} \,.\, status(M_{i+1},n_i) = mo$ *and*

(b) $\forall n_i \in \{n_1,\dots,n_k\} \,.\, status(M,n_i) = inv$.

Based on Definitions 3 and 4, we can show that no mutually blocked nodes can exist in any reachable configuration in our model,

Theorem 3 (No mutually blocked nodes)**.** *Given a reachable configuration* $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$*, there does not exits a set nodes that are mutually blocked.*

Proof. We assume a set of mutually blocked nodes in a reachable configuration and derive a contradiction. The proof is detailed in Appendix A.6. □

5. Proof-of-Concept Implementation

To explore the behaviour of different configurations in the formal model, we have developed a proof-of-concept tool in the rewriting logic system Maude [11]. This tool implements the formal model presented in Section 3, and it allows to specify and compare configurations in which the design choices for the underlying hardware architecture are different; e.g., the number of cores, cache levels and the data layout in main memory (which specifies how the references are organized in memory), the cache associativity and replacement policy may vary. In contrast to the formal semantics, the implementation features *weighted penalties* associated with accessing data from memory other than the first level cache in order to facilitate comparisons between different configurations. These penalties constitute an abstract performance indicator. Penalties can be accumulated during a run of the model. Exploring such design decisions is beneficial when developing software for multicore systems, where hardware features and data layout influence data movement, and consequently the performance of an application.

A second noteworthy difference between the formal semantics and the implementation concerns the abstractions in the semantics: the number of cores and caches, the size of caches, the cache associativity, replacement policies and memory layout. The implementation requires these parameters of the model to be explicit such that the model with a particular configuration, containing an explicit parallel architecture and a number of parallel tasks that are specified by the user, can be executed. This enables the behaviour of different configurations to be observed and compared.

A third difference between the formal semantics and the implementation is that the formalisation makes use of true concurrency to allow multiple data accesses to main memory in parallel by using the label mechanism, whereas the implementation uses interleaving concurrency. A global, parallel step in the formal semantics will therefore be translated into one or more interleaving steps in the implementation. This serialization is justified by Theorem 1 and does not otherwise affect the properties discussed in Section 4.

5.1. Rewriting Logic and Maude

Maude [11] is a specification and analysis system based on rewriting logic (RL) [30]. RL extends algebraic specification techniques with transitions rules which capture the dynamic behaviour of a system. In a rewrite theory (Σ, E, R), the signature Σ defines the ground terms, E a set of equations between terms, and R a set of labelled rewrite rules. *Conditional rewrite rules* of 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`* just identifies the rule). Rewrite rules apply to terms of given sorts, specified in (membership) equational logic (Σ, E). In a *conditional equation* **`ceq`** $t \; = t'$ **`if`** *`cond`*, the condition must similarly hold for the equation to apply. Rewrite rules transform equivalence classes of terms; i.e., it is assumed that terms can be reduced to unique normal forms by means of the equational theory in between applications of the rewrite rules. When auxiliary functions are needed, these can be defined in equational logic, and thus evaluated between the state transitions [30]. Unconditional rewrite rules and equations are denoted by the keywords *`rl`* and *`eq`*, respectively.

When modelling computational systems, different system components are typically modelled by terms of suitable sorts and the global state configuration is represented as a multiset of these terms [32]. In particular, Maude supports the modelling of systems as multisets of objects in a standardized 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 |` a_1`:` v_1`, ... ,` a_n`:` v_n $\rangle$, where `C` is the class name, `O` the object identifier, and a_i are attributes with corresponding values v_i (for $i \in \{1, \ldots, n\}$) 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.

5.2. An Implementation of the Multicore Model in Maude

The proof-of-concept implementation is realized as an object-oriented specification in Maude[1]. We first consider the term structure of system configurations and the use of equations for auxiliary functions, and then the rewrite rules of the dynamic behaviour for the local execution of tasks and for the global coordination and communication between cores, local caches and main memory. Rewriting rules are used to implement

[1]The proof-of-concept implementation and the complete example scenarios can be downloaded from `https://github.com/ShijiBijo84/Multilevel-Caches`.

```
-- Maude's object representation:
⟨C1: CR |Rst: nil, Levels: 1⟩
⟨Lev(1,C1): Cache |CM: empty,D: nil,CacheSz: 10,Assoc: Assoc(1),Lflag: false,Penalty: 1⟩
⟨M: MM |M: 0↦sh,1↦sh,2↦sh,3↦sh,4↦sh,fetchCount: 0;Penalty: 0⟩
⟨Sch : Qu |TidSet: empty ⟩
⟨Ta : Task |Data: empty⟩
⟨Mp(C1) : MP |mp: 0 ⟩
⟨Tbl : TBL |Addr: empty ⟩
```

Figure 11: Objects in Maude.

the majority of the rules of the operational semantics, and equations are used for coordination and to implement the instantaneous communication of the label mechanism as well as for auxiliary functions.

5.2.1. Configurations

The main components of the model are represented by Maude objects as shown in Figure 11. These objects float in a global Maude configuration (i.e., technically the different objects are modelled as subsorts of the sort `Configuration`, which has an associative and commutative composition operator denoted by whitespace). A system wide operator {_} is used to wrap complete configurations into the sort `GlobalSystem` ; observe that patterns $\{t\}$ in rules and equations will match t with the *entire* configuration, which we use to ensure that communication messages are correctly propagated to every part of the system (this technique is used in the first equation of Figure 15).

Maude objects of class `CR` represent cores, with attributes `Rst` for the task to be executed and `Levels` for the number of caches in that core. Maude objects of class `Cache` have an identifier `Lev(1,C1)`, where the first element represents the cache level and the second identifies the associated core. The attribute `CM` stores the memory blocks as a map from `Int` to `MemoryMap`, where `MemoryMap` represents cache line sets and `Int` is an integer representing the identity of each set, `D` stores the data instructions, `CacheSz` defines the size of the cache in terms of a number of cache lines, `Assoc` indicates the cache associativity, `Lflag` is a boolean value indicating whether the cache is at the last level, and `Penalty` stores the data access cost associated to the memory component. We use a parametrised initial function to create a well-formed initial configuration taking into account the parameters.

Compared to the formal semantics, the Maude implementation features *penalties* for accessing different parts of the memory hierarchy. In a typical memory hierarchy with three levels of cache, these penalties could be 1 for accessing the first level, 10 for the second, 100 for the third and 1000 for main memory, roughly approximating the differences in access time to the different levels. In the cache object, we use an `Assoc` operator of sort `MapPolicy`, parametrized by an integer k, to specify the number of cache lines in each associativity set. In addition to the wellformedness requirements described in Section 3.3, we here assume an initial configuration in which there is a relationship between the cache associativity and the cache size from the different levels, such that all caches belonging to the same node have the same number of sets. This ensures the swapping of cache lines between two adjacent caches will manipulate the same set in both levels.

```
ceq SwapLinesi(LCi,(LCj,x↦(M',N↦s)),N,sz) = update(x,(N↦s),LCi)
if LCi[x] = undefined.

eq SwapLinesi((LCi,x↦(M,N↦inv)),(LCj,x↦(M',N↦s)),N,sz) = (LCi,x↦(M,N↦s)).

ceq SwapLinesi((LCi,x↦M),(LCj,x↦(M',N↦s)),N,sz) = (LCi,x↦(M,N↦s))
if |M| < sz ∧ N ∉ dom(M).

ceq SwapLinesi((LCi,x↦M),(LCj,x↦(M',N↦s)),N,sz) = (LCi,x↦(M\line,N↦s))
if |M| = sz ∧ line:=selectStrategy(M).

ceq SwapLinesj((LCi,x↦M),(LCj,x↦(M',N↦s)),N,sz) = (LCj,x↦(M',line))
if |M| = sz ∧ line:=selectStrategy(M) ∧ selectStatus(line) ≠ inv.

eq SwapLinesj(LCi,(LCj,x↦(M',N↦s)),N,sz) = (LCj,x↦M')[owise].

ceq selectStrategy(M) = pick(selectSet(M,empty,inv))if inv ∈ stSet(M).

ceq selectStrategy(M) = pick(selectSet(M,empty,sh))
if sh ∈ stSet(M) ∧ inv ∉ stSet(M).

ceq selectStrategy(M) = pick(selectSet(M,empty,mo))
if sh ∉ stSet(M) ∧ inv ∉ stSet(M) .

eq selectSet((M,(n↦s)),M',s) = selectSet(M,(M',(n↦s)),s).
ceq selectSet((M,(n↦s)),M',s') = selectSet(M,M',s')if s ≠ s'.
eq selectSet(empty,M',s') = M'.

eq pick((M,n↦s,n'↦s)) = pick((M,min(n, n')↦s)).
eq pick(n↦s) = n↦s.

eq stSet(M) = stSet(M,empty).
eq stSet((M,(n↦s)),st) = stSet(M,(st,s)).
eq stSet(empty,st) = st.
```

Figure 12: Equations for swapping cache lines between adjacent caches.

The Maude object of class `MM` represents the main memory, where the attribute `M` maps addresses to status values, `fetchCount` records the total number of fetch operations in the configuration and `Penalty` stores the cost associated to accessing memory block from main memory. The pre-defined Maude module `MAP`, which is used both in `CM` and `MM`, denotes a binding from key to value as `key↦value`. A lookup of a `key` in a `MAP` as `map[key]` returns a constant, or `undefined` if the key of the entry is not found in the set.

The Maude object of class `Sch` corresponds to a task scheduler, with an identifier `Qu`. The task scheduler arranges tasks based on the task identifiers stored in attribute `TidSet`, and uses the task lookup table `Task` to obtain the task body, represented as a sequence of statements, for a given task identifier. The Maude object of class `MP` stores the accumulated penalty for each core during task execution and the identifier `Mp(C1)` identifies the core. Configurations also include the table `TBL` that maps references to addresses of memory blocks, which specifies the data layout in the main memory.

5.2.2. Auxiliary Functions

Auxiliary functions, which are used to make the formal model of Section 3 more abstract, need to be concrete in Maude. Some functions, such as *first* or *last*, are realized by direct pattern matching over the attributes of the Maude objects. The function

addr, which returns the block address of a reference, makes use of the address table object `TBL`. The function *status*, which returns the status of a cache line, is implemented by an equation `selectStatus`. Additional functions are used to simplify the Maude rules; e.g., the predicate `validStatus` checks if the status of a memory block is `sh` or `mo`.

The function *select* abstracts from the placement and eviction policy in the rules LC-HIT$_1$, LC-HIT$_2$, LC-BLOCK-FINISH$_1$ and LC-BLOCK-FINISH$_2$ of Figure 6. These rules capture the swapping of blocks between two adjacent caches. Let us consider two adjacent caches `i` and `j`, (where `j = i+1`). When a memory block with address `N` is fetched from level `j`, a block may be evicted from level `i` to give space for `N`. The evicted block will be stored in level `j` if its status is `sh` or `mo`; otherwise, it will be discarded. In Maude, we have modelled the swapping process and the different criteria for *select* using the auxiliary functions `SwapLinesi` and `SwapLinesj` (shown in Figure 12), which respectively handle the updates at level `i` and `j`. Here, the variables `LCi` and `LCj` refer to maps containing sets of cache lines (the associativity group) from the caches at levels `i` and `j`, respectively. These two functions complement each other and are used together (see, e.g., rule `LC-Hit1` in Figure 13).

The first equation of `SwapLinesi` handles the case in which we add a new entry to an empty cache or a cache without the considered set `x` of cache lines. In the second equation, where `LCi` has block `N` in `inv` state, the status is updated with the one from `LCj`. In the third equation, `LCi` has space in the cache line set `M`, in which `N` should be placed. The predicate `|M| < sz` checks whether the cardinality of `M` is less than the maximum capacity `sz` of the set. The fourth equation captures the case when the set is full and a cache line needs to be evicted; this cache line is selected by the function `selectStrategy(M)`. The function embodies an *eviction strategy*: it first looks for a cache line in `inv` state; if no such cache line is found, it looks for one in `sh` state; otherwise returns a modified cache line. The function `selectSet(M)` returns a set of cache lines with a particular status according to this eviction strategy, and the function `pick` returns the smallest address in a set.

The remaining equations in Figure 12 define the auxiliary function `SwapLinesj`, by similarly removing the corresponding cache line and storing the evicted one (if needed) in the cache level `j`. The evicted line will be stored only if the status is either shared or modified.

5.2.3. Local Semantics

We now discuss the Maude representation of the local semantics (cf. Figures 5 and 6). We focus on some representative rules, shown in Figure 13; the remaining rules are similar. To facilitate comparison between the operational semantics and the Maude representation, rule names from the semantics are used as labels for the rewrite rules and the highlighting patterns of Section 3 are kept.

Rule `PrRd1` describes a read operation when the required block has status `sh` or `mo` (checked by the predicate `validStatus`) in the first level cache. The penalty is incremented according to the specified cost of fetching data from the first level cache. Rule `PrRd2` describes a read miss in the first level cache, which blocks the core by the `readBl` statement and adds a `fetch` instruction to the data instruction list `D`. Rule `PrWr1` describes a write operation when the first level cache has the required block

```
crl [PrRd1] :
⟨Mp(C1):MP |mp:k⟩ ⟨Lev(1,C1):Cache |CM:LCi,Penalty:p,Atts⟩
⟨C1:CR | Rst:(read(r);rst) ,Levels:l⟩ ⟨Tbl:TBL |Addr:addr⟩
⟶
⟨Mp(C1):MP |mp:(k+p)⟩ ⟨Lev(1,C1):Cache |CM:LCi,Penalty:p,Atts⟩
⟨C1:CR | Rst:rst ,Levels:l⟩ ⟨Tbl:TBL |Addr:addr⟩
if validStatus(LCi,addr[ref(r)]) = true.

crl [PrRd2] :
⟨Lev(1,C1):Cache |CM:LCi, D:d ,Atts⟩
⟨C1:CR | Rst:(read(r);rst) ,Levels:l⟩ ⟨Tbl:TBL |Addr:addr⟩
⟶
⟨Lev(1,C1):Cache |CM:LCi, D:(d;fetch(addr[ref(r)])) ,Atts⟩
⟨C1:CR | Rst:(readBl(r);rst) ,Levels:l⟩ ⟨Tbl:TBL |Addr:addr⟩
if validStatus(LCi, addr[ref(r)]) = false.

crl [PrWr1] :
⟨Lev(1,C1):Cache |CM:LCi,D:d,Penalty:p,CacheSz:size,Assoc:mapPol,Lflag:flag⟩
⟨C1:CR | Rst:(write(r);rst) ,Levels:l⟩⟨Tbl:TBL |Addr:addr⟩⟨Mp(C1):MP |mp:k⟩
⟶
⟨Lev(1,C1):Cache|CM:LCi,D:d,Penalty:p,CacheSz:size,Assoc:mapPol,Lflag:flag⟩
⟨C1:CR | Rst:rst ,Levels:l⟩ ⟨Tbl:TBL |Addr:addr⟩ ⟨Mp(C1):MP |mp:(k+p)⟩
if selectStatus(LCi,addr[ref(r)])=mo.

crl [PrWr2] :
⟨Lev(1,C1):Cache | CM:LCi ,D:d,Penalty: p,Atts⟩
⟨C1:CR | Rst:(write(r);rst) ,Levels:l⟩ ⟨Tbl:TBL |Addr:addr⟩ ⟨Mp(C1):MP |mp:k⟩
⟶
⟨Lev(1,C1):Cache | CM:updateLine(LCi,...) ,D:d,Penalty:p,Atts⟩
⟨C1:CR | Rst:rst ,Levels:l⟩ ⟨Tbl:TBL |Addr:addr⟩ ⟨Mp(C1):MP |mp:(k+p)⟩
(broadcast RdX(addr[ref(r)]) from Lev(1,C1)) if selectStatus(LCi,addr[ref(r)])=sh .

crl [LC-Hit1] :
⟨Lev(i,C1):Cache | CM:LCi,D:(fetch(N);d) ,Penalty:p,Assoc:Assoc(t),Atts⟩
⟨Lev(j,C1):Cache | CM:LCj ,D:d',Penalty:p',Assoc:Assoc(t'),Atts'⟩⟨Mp(C1):MP |mp:k⟩
⟶
⟨Lev(i,C1):Cache | CM:SwapLinesi(...),D:d ,Penalty:p,Assoc:Assoc(t), Atts⟩
⟨Lev(j,C1):Cache | CM:SwapLinesj(...) ,D:d',Penalty:p',Assoc:Assoc(t'),Atts'⟩
⟨Mp(C1) :MP |mp:(k+p')⟩ if validStatus(LCj, N) and j = i+1 .

crl [LLC-Miss] :
⟨Clev:Cache |CM:LCi, D:(fetch(N);d) , Lflag: true,Atts⟩
⟶
⟨Clev:Cache |CM:LCi, D:(block(N);d) , Lflag: true,Atts ⟩
(broadcast Rd(N) from Clev) if validStatus(LCi, N) = false .
```

Figure 13: Rewriting rules in Maude for the local execution of tasks.

```
crl [Fetch1&2]:
⟨Lev(i,C1): Cache | CM:LCi,D:(Block(N);d) ,Penalty: p, Lflag: true,Atts⟩
⟨Me' : MM |M: mapSet, fetchCount:x',Penalty:p' ⟩ ⟨Mp(C1): MP |mp:k⟩
⟶
⟨Lev(i,C1): Cache | CM:fetchUpdateLine(LCi,...),D:d ,Penalty: p,Lflag: true,Atts⟩
⟨Me' : MM |M: mapSet,  fetchCount:x'+1,Penalty:p' ⟩ ⟨Mp(C1): MP |mp:(k+p')⟩
if allModified(cacheLineSet(LCi,...),...)=false and selectStatus(mapSet, N)= sh .

crl [Fetch3] :
⟨Me'  : MM |M: mapSet,fetchCount: x',Penalty: p'⟩
⟨Clev: Cache |CM: LCi, D:(Block(N);d) ,Lflag: true,Atts⟩
⟶
⟨Me' : MM |M: mapSet,fetchCount: x',Penalty: p'⟩
⟨Clev: Cache |CM: LCi, D:(flush(LineToFlush(selectStrategy(...)));Block(N);d) ,
Lflag: true,Atts⟩
if allModified(cacheLineSet(LCi,...),...)=true .
```

Figure 14: Subset of rewrite rules implement global semantics.

in `mo` state, so the core can perform the write operation. The penalty is incremented accordingly. Rule `PrWr2` describes a write operation when the first level cache has the required block in `sh` state. In this case, main memory and other caches may contain copies of the same block, so a `RdX` message is *broadcast* to invalidate all other copies before the `write` operation can proceed. The penalty is incremented according to the cost of fetching data from the first level cache. The function `updateLine` changes the status of a memory block from `sh` to `mo`. Message broadcast is discussed later in this section.

Rule `LC-Hit1` addresses the case when a cache at level `i` has to execute a `fetch` instruction for block `N` which is found with status `sh` or `mo` in the cache at the next level `j`. The block is transferred to level `i` by `SwapLinesi` and removed from level `j` by`SwapLinesj`. If a block needs to be evicted to give space to block `N`, the evicted block will be moved to the level cache `j` by `SwapLinesj`. The penalty is incremented according to the cost of fetching data from the next level cache. Rule `LLC-Miss` covers the case of a cache miss in last level cache, i.e., the required block is not found in any cache local in a core. In this case a `Rd` message is broadcast.

5.2.4. Global Semantics

The global semantics deals with message transmission and data transfer among caches and main memory. Since Maude only allows interleaving concurrency, parallel message passing is implemented by combining rules and equations: instantaneous broadcast communication can be captured in Maude by unfolding a broadcast message to transmit point-to-point messages using equations [32]. We focus on a representative set of rules, shown in Figure 14.

The rules `Fetch1&2` and `Fetch3` fetch a memory block `N` to the last level cache from main memory, with penalty. The predicate `allModified` is *true* if all cache lines in a particular set of the last level cache have status `mo`. Note that the cache line set and the corresponding size are decided by the replacement policy and associativity of the cache. Rule `Fetch1&2` applies when block `N` is in `sh` state and the cache line set

```
-- Equations for message passing
eq {(broadcast Rq from Clev) REST} =
   {(multicast Rq from Clev to objectIds(REST)) REST} .
eq multicast Rq from Clev to none = none.
eq multicast Rq from Lev(j, C1) to Lev(i,C1);RqSet =
   multicast Rq from Lev(j,C1) to ReSet.
eq multicast Re from Clev1 to Clev2;RqSet =
   (msg Rq from Clev1 to Clev2) (multicast Rq from Clev1 to RqSet) [owise].

ceq (msg RdX(N) from Clev1 to Clev2)
⟨Clev2: Cache |CM: Ca,D: d,Atts⟩ = ⟨Clev2: Cache |CM: updateLine(...),D: d,Atts⟩
if selectStatus(Ca,N) = sh.

ceq (msg RdX(N) from Clev1 to Clev2)
⟨Clev2: Cache |CM: Ca,Atts⟩ = ⟨Clev2: Cache |CM: Ca,Atts⟩
if selectStatus(Ca, N) ≠ sh.

eq (msg RdX(N) from C1 to Me )
⟨Me': MM |M: mapSet,fetchCount: x,Penalty: p⟩ =
if (selectStatus(mapSet, N) = sh)
then ⟨Me': MM |M: updateLine(mapSet, N),fetchCount: x,Penalty: p⟩
else ⟨Me': MM |M: mapSet,fetchCount: x,Penalty: p⟩ fi.

eq (msg Rd(N) from C1 to Me)
⟨Me': MM |M: mapSet,Atts⟩ = ⟨Me': MM |M: mapSet,Atts⟩ .

ceq (msg Rd(N) from Clev1 to Clev2)
⟨Clev2: Cache |CM: Ca,D: d,Atts⟩ = ⟨Clev2: Cache |CM: Ca,D: (flush(N);d),Atts⟩
if selectStatus(Ca,N) = mo and occurs(flush(N),d) = false.

ceq (msg Rd(N) from Clev1 to Clev2)
⟨Clev2: Cache |CM: Ca,Atts⟩ = ⟨Clev2: Cache |CM: Ca,Atts⟩
if selectStatus(Ca,N) ≠ mo .
```

Figure 15: Equations in Maude for instantaneous communication.

where `N` will be placed has at least one cache line that is not in `mo` state, i.e., function `allModified` returns *false*. Thus, the `fetch` instruction will be executed. Observe that after the execution, `fetchCount` is incremented, the penalty `mp` is increased, and the cache memory is updated with the fetched block using function `fetchUpdateLine`.

`Fetch3` applies when all lines in the cache line set where `N` is placed are modified (so `allModified` returns *true*) and one cache line needs to be flushed to main memory and evicted before fetching `N`. This is captured by inserting a `flush` instruction at the head of the data instruction list `D` after the transition, where the function `LineToFlush` selects the address of the line to be flushed (as determined by the function `selectStrategy`).

Instantaneous request broadcast is done by equations, see Figure 15. A broadcast is a term `broadcast Rq` **`from`** `Clev`, where `Rq` denotes the read or write request and `Clev` the identity of the sender. As requests are propagated to caches in other cores and to the main memory, a broadcast is recursively transformed into Maude messages **`msg`** `Rq` **`from`** `Clev1` **`to`** `Clev2`, where `Rq` is the request, `Clev1` the sender and `Clev2` the receiver. The function `objectIds` collects the identities of *all* caches in the configuration `REST` and returns a set of receivers `RqSet`, which excludes all the caches that are in the same core as the sender. Upon receiving a `RdX` message for a block, all caches and main memory will update the status of the block to `inv` if it exists in status `sh`, otherwise the message is discarded. Cache memory will respond to `Rd` message for

```
task T1{(read(r0);read(r5);write(r10);read(r15);write(r20);read(r25);read(r30);write(r35);
        read(r40);write(r45);read(r50);write(r55);write(r60);read(r65);write(r70);read(r75);
        write(r80);read(r85);write(r3);read(r8);write(r13);read(r18);write(r23);write(r28);
        write(r4);read(r9);write(r14);read(r19);write(r24);read(r29);read(r30);write(r85);
        read(r30);write(r40);read(r30);write(r40);write(r8);read(r3);write(r8);read(r3);
        write(r28); write(r23))*}

task T2{(read(r1);read(r6);read(r11);write(r16);read(r21);write(r26);read(r31);read(r36);
        write(r41);read(r46);write(r51);read(r56);read(r61);read(r66);write(r71);read(r76);
        write(r81);read(r86);read(r33);write(r38);read(r43);write(r48);write(r53);read(r58);
        read(r34);write(r39);read(r44);write(r49);read(r54);write(r59);read(r33);write(r38);
        read(r33);write(r38);write(r53);read(r58);read(r11);write(r16);read(r11);write(r16);
        write(r21);write(r26);read(r71);read(r66);write(r61);write(r16))*}

task T3{(read(r2);write(r7);read(r12);write(r17);read(r22);read(r27);write(r32);read(r37);
        write(r42);read(r47);read(r52);read(r57);read(r62);write(r67);read(r72);read(r77);
        write(r82);read(r87);write(r63);read(r68);write(r73);write(r78);read(r83);write(r88);
        write(r64);read(r69);write(r74);write(r79);read(r84);read(r89);write(r32);read(r37);
        write(r42);read(r47);read(r52);read(r57);read(r67);read(r62);read(r67);read(r62);
        read(r77);read(r82);read(r63);read(r47);read(r63);write(r87))*}

main{spawn(T1);spawn(T2);spawn(T3)}
```

Figure 16: An example of the data access patterns of a program.

a block by adding a `flush` instruction to the data instruction list `D` if the block is in modified state.

5.3. Example: Observing the Impact of Multilevel Caches and Data Layout

The impact of the number of caches and the data layout on data movement, as captured by weighted penalties associated with accessing data from memory other than the first level cache, can be observed with our proof-of-concept implementation.

Let us compare different deployment scenarios for a program represented by the data access patterns shown in Figure 16. We consider three architectures `Arch1`, `Arch2` and `Arch3` each with three cores `C1`, `C2` and `C3`, varying in the number of caches per core, where `C1`, `C2` and `C3` execute the tasks `T1`, `T2` and `T3`, respectively. For each of the three architectures, we consider three different data layouts, which gives nine scenarios

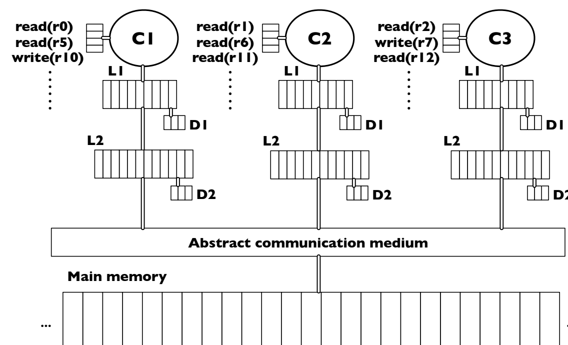


Figure 17: An example of a parallel architecture with 3 cores and 2 level caches.

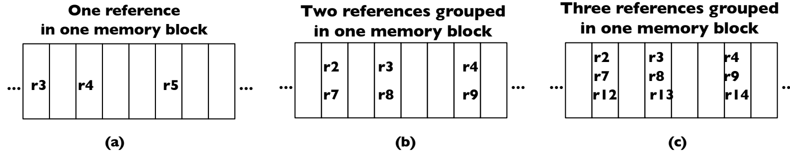


Figure 18: Different data layouts for main memory used in the example.

in total. For simplicity, we look at the results after running the loop of each task a finite number of times, say 20 iterations.

In the first architecture, each core has one single level cache `L1`; in the second, each core has two levels of cache `L1`, `L2`, as depicted in Figure 17; and in the third, each core has three levels of cache `L1`, `L2`, `L3`. The associativity of the caches has been configured as direct mapped, 2-way associativity and 3-way associativity for `L1`, `L2`, `L3`, respectively. To compare data accesses in the different cache levels and in main memory, we associate weighted penalties to accesses from different levels of memory. For simplicity in this example, we use order of magnitude differences and associate penalties 1, 10, 100 and 1000 to accesses from `L1`, `L2`, `L3` and main memory, respectively. We compare three different data layouts for main memory, depicted in Figure 18. In the first layout (Figure 18a) the tasks need to access different memory blocks for each reference, in the second (Figure 18b) two references are grouped together in one block, and in the third (Figure 18c) three references are grouped together.

Figure 19 summarises the results of executing the model with the nine considered scenarios. Observe that when the data is spread out (i.e., the references are placed in different memory blocks in main memory), the scenarios where cores have a single level cache need to perform many evictions and fetch operations to access data from main memory. This increases the access time, as reflected by the accumulated penalty. In the scenarios where cores have three levels of cache, the penalty is substantially lower, although the access patterns are the same as the scenarios of single level caches. This is because it requires fewer evictions and operations for the cores to fetch or flush data from or to main memory, although there are still penalties from swapping data between the different cache levels. Thus, the scenarios in the example confirm the expected behaviour of the model proposed in this paper, and we can observe the impact of data layout on data movement and the relation between data movement and the number of caches.

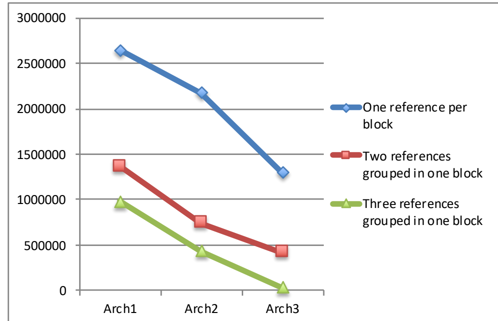


Figure 19: Accumulated penalties for the nine different scenarios of the example.

6. Related Work

We first compare our work to formal work on memory consistency models, and then to empirical work on performance analysis. Related work on the formal analysis of memory consistency models can be divided into work on weak memory models and on cache coherence protocols. Related work on performance analysis include general tools to improve program performance as well as specialized simulation tools for multicore architectures.

Both operational and axiomatic formal models have been used to capture the impact of parallel executions on shared memory under relaxed memory models. Approaches to the formalisation of relaxed memory models include abstract calculi [12], memory models for programming languages such as Java [22], and machine-level instruction sets for concrete processors such as POWER [27, 39] and x86 [40], and for programs executing under total store order (TSO) architectures [17, 41]. Verification techniques have also been developed to analyse programs executing on relaxed memory models. For instance, a monitor algorithm [9] has been proposed to guarantee TSO-safety by analyzing all possible executions in terms of traces; and automatic fences insertion [18] has been used to map the memory model at the programming language level onto a hardware memory model. This line of 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.

Programming models for heterogeneous systems with disjoint address spaces (such as combined CPU and GPU programs) have similarities to data access patterns and multicore memory systems as studied in this paper. These models typically expose data transfer to the programmer in terms of explicit instructions or annotations. For example, VectorPU [25] supports aggregated data containers, using access mode modifiers (e.g., read, write, read/write) on program variables. The correctness of these annotations for data consistency have recently been studied [21]. This work has similarities to our paper as it develops operational semantics for programs with multiple memories. It goes beyond our paper in defining abstract variables (for data containers) but considers neither space limitations of local memory nor the performance aspects of data transfer, such as our model of penalties.

Cache coherence protocols have been analysed in the setting of automata, and (parametrized) model checking (e.g., [14, 34, 37]) has been used to abstract from a specific number of cores when proving the correctness of the protocols (e.g., [15, 16]). For instance, Maude's model checker has recently been used to verify the correctness of configurations of the MSI and ESI protocols [29, 38]. This line of work focuses on proving the correctness of the message exchange of the cache coherence protocols (e.g., Figure 3a) and relates to our correctness proofs. Compared to this line of work, we take a programmer's perspective on the multicore memory system to consider cache coherent movement of data. We focus on formally capturing the movement of data as a consequence of the interaction between cores, caches and shared memory during the parallel data access from programs, rather than on protocol verification.

Empirical work to improve program performance may take into account the relation between data accesses and hardware architectures (e.g., [1, 44, 19, 45, 33]). In particular, tools for analyzing runtime program performance based on measurements,

like hardware counters, can aid developers to observe performance bottlenecks, e.g., synchronisation or communication. Such observations can be used later for optimisations purpose. For instance, [44] optimises cache energy consumption by means of context-sensitive profiling. Inspired by these techniques, our proof-of-concept tool in Maude uses the notion of penalty to quantify over the cost of data access for different choices of hardware configuration, cache associativity, and data layout.

Simulation tools for cache coherence protocols evaluate performance and efficiency on different architectures (e.g., gems [28] and gem5 [8]). These tools perform evaluations of, e.g., the cache hit/miss ratio and response time, by running benchmark programs written as low-level read and write instructions to memory. Advanced simulators such as Graphite [31] and Sniper [10] run programs on distributed clusters to simulate executions on multicore architectures with thousands of cores. Unlike our work, these simulators do not account for how communication and data movement among different components in the multicore memory architecture affect the simulation results. Our work complements these simulators by providing a formal model capturing the interactions that trigger data transfer between different components, and shows how different choices of data layout and the number of caches may impact such data movement. Our proof-of- concept implementation is more similar to the cache simulators in spirit, but it supports the executable exploration of design choices in the formal model rather than the simulation of large programs. Analysis techniques for worst-case response times of concurrent programs running on multicore architectures with shared caches have also been investigated [26]; this analysis differs from ours in its focus on response time rather than on data movement.

7. Conclusions and Future Work

Software is increasingly designed to run on multicore architectures, where parallel access to data triggering data movement between components in the architecture, crucially influence the performance of the parallel execution. Formal models that capture how tasks interact with shared memory systems may help software developers understand how data access influences the behaviour of parallel execution on multicore architecture. This paper develops a formal executable model of multicore architectures with multilevel caches from a program perspective rather than a hardware perspective, and addresses dynamically spawned data access patterns. The formal model is given as an operational semantics for data access patterns executing in parallel on different cores, and ensures data consistency by embodying a cache coherence protocol. The model is proven to guarantee correctness properties concerning data consistency, which shows that it correctly captures data movement triggered by the cache coherence protocol. We provide a proof-of-concept implementation, which allows the executable exploration of the formal model, and show by example how choices for a program's data layout in combination with the underlying hardware architecture affect data movement.

The work presented in this paper can be extended in several directions. Both data access patterns, the architectural model, the underlying operational model can be enriched, and the formalization and proofs can be mechanized in a theorem prover such as Isabelle or Coq. For example, a locking mechanism which allows atomic blocks and synchronisation between data access patterns to be modelled has been studied in [4],

which also discusses how data access patterns can be integrated in a software development context. Currently, we are implementing a more powerful simulation tool [3] based on the formal model presented in this paper. As for future work, we plan to enrich the data access patterns with data structures and dynamically allocated memory (e.g., object creation). This opens for extracting data access patterns from parallel object-oriented programming languages such as ABS [23]. Another interesting direction is to extend the architecture to support shared caches. Finally, models as developed in this paper could serve as a foundation to study the effects of program specific optimisations of data layout and scheduling.

Acknowledgment. We would like to thank the anonymous reviewers their careful reading of the manuscript, and for their insightful comments and constructive suggestions.

References

[1] Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N. R., Apr. 2010. HPCTOOLKIT: Tools for performance analysis of optimized parallel programs http://hpctoolkit.org. Concurr. Comput. : Pract. Exper. 22 (6), 685–701.

[2] Adve, S. V., Gharachorloo, K., 1996. Shared memory consistency models: A tutorial. IEEE Computer 29 (12), 66–76.

[3] Bezirgiannis, N., de Boer, F. S., Johnsen, E. B., Pun, K. I., Tapia Tarifa, S. L., 2019. Implementing SOS with active objects: A case study of a multicore memory system. In: Proceedings of the 22nd International Conference on Fundamental Approaches to Software Engineering (FASE 2019). Vol. 11424 of Lecture Notes in Computer Science. Springer, pp. 332–350.

[4] Bijo, S., Johnsen, E. B., Pun, K. I., Seidl, C., Tapia Tarifa, S. L., 2018. Deployment by construction for multicore architectures. In: Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Modeling (ISoLA 2018). Vol. 11244 of Lecture Notes in Computer Science. Springer, pp. 448–465.

[5] Bijo, S., Johnsen, E. B., Pun, K. I, Tapia Tarifa, S. L., 2016. A Maude framework for cache coherent multicore architectures. In: Proceedings of the 11th International Workshop on Rewriting Logic and Its Applications (WRLA). Vol. 9942 of Lecture Notes in Computer Science. Springer, pp. 47–63.

[6] Bijo, S., Johnsen, E. B., Pun, K. I, Tapia Tarifa, S. L., 2016. An operational semantics of cache coherent multicore architectures. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC). ACM, pp. 1219–1224.

[7] Bijo, S., Johnsen, E. B., Pun, K. I., Tapia Tarifa, S. L., 2017. A formal model of parallel execution on multicore architectures with multilevel caches. In: Proceedings of the 14th International Conference on Formal Aspects of Component Software (FACS 2017). Vol. 10487 of Lecture Notes in Computer Science. Springer, pp. 58–77.

[8] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., Wood, D. A., 2011. The gem5 simulator. SIGARCH Computer Architecture News 39 (2), 1–7.

[9] Burckhardt, S., Musuvathi, M., 2008. Effective program verification for relaxed memory models. In: Gupta, A., Malik, S. (Eds.), Proceedings of the 20th International Conference on Computer Aided Verification (CAV). Vol. 5123 of Lecture Notes in Computer Science. Springer, pp. 107–120.

[10] Carlson, T. E., Heirman, W., Eeckhout, L., 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC). ACM, pp. 52:1–52:12.
URL `http://doi.acm.org/10.1145/2063384.2063454`

[11] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. L. (Eds.), 2007. All About Maude — A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. Vol. 4350 of Lecture Notes in Computer Science. Springer.

[12] Crary, K., Sullivan, M. J., 2015. A calculus for relaxed memory. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). ACM, pp. 623–636.

[13] Culler, D. E., Gupta, A., Singh, J. P., 1997. Parallel Computer Architecture: A Hardware/Software Approach, 1st Edition. Morgan Kaufmann Publishers Inc.

[14] Delzanno, G., 2003. Constraint-based verification of parameterized cache coherence protocols. Formal Methods in System Design 23 (3), 257–301.
URL `http://dx.doi.org/10.1023/A:1026276129010`

[15] Dill, D. L., Drexler, A. J., Hu, A. J., Yang, C. H., 1992. Protocol verification as a hardware design aid. In: Proceedings of the IEEE International Conference on Computer Design on VLSI in Computer Processors (ICCD). IEEE Computer Society, pp. 522–525.
URL `http://dl.acm.org/citation.cfm?id=645461.654859`

[16] Dill, D. L., Park, S., Nowatzyk, A. G., 1993. Formal specification of abstract memory models. In: Proceedings of the Symposium on Research on Integrated Systems. MIT Press, pp. 38–52.

[17] Dongol, B., Travkin, O., Derrick, J., Wehrheim, H., 2013. A high-level semantics for program execution under total store order memory. In: Proceedings of the 10th International Colloquium on Theoretical Aspects of Computing (ICTAC). Vol. 8049 of Lecture Notes in Computer Science. Springer, pp. 177–194.

[18] Fang, X., Lee, J., Midkiff, S. P., 2003. Automatic fence insertion for shared memory multiprocessing. In: Proceedings of the 17th Annual International Conference on Supercomputing (ICS). ACM, pp. 285–294.

[19] Farshchi, F., Valsan, P. K., Mancuso, R., Yun, H., 2018. Deterministic memory abstraction and supporting multicore system architecture. In: Altmeyer, S. (Ed.), 30th Euromicro Conference on Real-Time Systems (ECRTS 2018). Vol. 106 of LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 1:1–1:25.

[20] Hennessy, J. L., Patterson, D. A., 2011. Computer Architecture: A Quantitative Approach, 5th Edition. Morgan Kaufmann Publishers Inc.

[21] Henrio, L., Kessler, C. W., Li, L., 2018. Ensuring memory consistency in heterogeneous systems based on access mode declarations. In: Proceedings of the International Conference on High Performance Computing & Simulation (HPCS 2018). IEEE, pp. 716–723.

[22] Jagadeesan, R., Pitcher, C., Riely, J., 2010. Generative operational semantics for relaxed memory models. In: Proceedings of the 19th European Conference on Programming Languages and Systems (ESOP). Vol. 6012 of Lecture Notes in Computer Science. Springer, pp. 307–326.

[23] Johnsen, E. B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M., 2011. ABS: A core language for abstract behavioral specification. In: Aichernig, B., de Boer, F. S., Bonsangue, M. M. (Eds.), Proceedings of the 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010). Vol. 6957 of Lecture Notes in Computer Science. Springer, pp. 142–164.

[24] Lamport, L., 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers 28 (9), 690–691.

[25] Li, L., Kessler, C. W., 2017. VectorPU: A generic and efficient data-container and component model for transparent data transfer on gpu-based heterogeneous systems. In: Proceedings of the 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM@HiPEAC 2017). ACM, pp. 7–12.

[26] Li, Y., Suhendra, V., Liang, Y., Mitra, T., Roychoudhury, A., 2009. Timing analysis of concurrent programs running on shared cache multi-cores. In: Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS). IEEE Computer Society, pp. 57–67.

[27] Mador-Haim, S., Maranget, L., Sarkar, S., Memarian, K., Alglave, J., Owens, S., Alur, R., Martin, M. M. K., Sewell, P., Williams, D., 2012. An axiomatic memory model for POWER multiprocessors. In: Proceedings of the 24th International Conference on Computer Aided Verification (CAV). Vol. 7358 of Lecture Notes in Computer Science. Springer, pp. 495–512.

[28] Martin, M. M. K., Sorin, D. J., Beckmann, B. M., Marty, M. R., Xu, M., Alameldeen, A. R., Moore, K. E., Hill, M. D., Wood, D. A., 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. SIGARCH Computer Architecture News 33 (4), 92–99.

[29] Martín, Ó., Verdejo, A., Martí-Oliet, N., 2014. Model checking TLR* guarantee formulas on infinite systems. In: Specification, Algebra, and Software – Essays Dedicated to Kokichi Futatsugi. Vol. 8373 of Lecture Notes in Computer Science. Springer, pp. 129–150.

[30] Meseguer, J., 1992. Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96 (1), 73–155.

[31] Miller, J. E., Kasture, H., Kurian, G., Gruenwald, C., Beckmann, N., Celio, C., Eastep, J., Agarwal, A., 2010. Graphite: A distributed parallel simulator for multicores. In: Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA). IEEE Computer Society, pp. 1–12.

[32] Ölveczky, P. C., 2018. Designing Reliable Distributed Systems – A Formal Methods Approach Based on Executable Modeling in Maude. Undergraduate Topics in Computer Science. Springer.

[33] Oprofile, 2018. Oprofile: A System Profiler for Linux. `http://oprofile.sourceforge.net/news/`, accessed: 2018-03-28.

[34] Pang, J., Fokkink, W., Hofman, R. F. H., Veldema, R., 2007. Model checking a cache coherence protocol of a Java DSM implementation. Journal of Logic and Algebraic Programming 71 (1), 1–43.
URL `http://dx.doi.org/10.1016/j.jlap.2006.08.007`

[35] Patterson, D. A., Hennessy, J. L., 2013. Computer Organization and Design: The Hardware/Software Interface, 5th Edition. Morgan Kaufmann Publishers Inc.

[36] Plotkin, G. D., 2004. A structural approach to operational semantics. Journal of Logic and Algebraic Programming 60-61, 17–139.

[37] Pong, F., Dubois, M., 1997. Verification techniques for cache coherence protocols. ACM Computing Surveys 29 (1), 82–126.

[38] Ramírez, S., Rocha, C., 2015. Formal verification of safety properties for a cache coherence protocol. In: Proceedings of the 10th Computing Colombian Conference (10CCC). IEEE Computer Society, pp. 9–16.

[39] Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D., 2011. Understanding POWER multiprocessors. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). ACM, pp. 175–186.

[40] Sewell, P., Sarkar, S., Owens, S., Nardelli, F. Z., Myreen, M. O., 2010. X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM 53 (7), 89–97.

[41] Smith, G., Derrick, J., Dongol, B., 2015. Admit your weakness: Verifying correctness on TSO architectures. In: Proceedings of the 11th International Symposium on Formal Aspects of Component Software (FACS). Vol. 8997 of Lecture Notes in Computer Science. Springer, pp. 364–383.

[42] Solihin, Y., 2015. Fundamentals of Parallel Multicore Architecture, 1st Edition. Chapman & Hall/CRC.

[43] Sorin, D. J., Hill, M. D., Wood, D. A., 2011. A Primer on Memory Consistency and Cache Coherence, 1st Edition. Morgan & Claypool Publishers.

[44] Vardhan, K. A., Srikant, Y. N., 2014. Exploiting critical data regions to reduce data cache energy consumption. In: Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems (SCOPES). ACM, pp. 69–78.

[45] Weaver, V. M., 2013. Linux perf_event features and overhead. In: Proceedings of the International Workshop on Performance Analysis of Workload Optimized Systems (FastPath 2013).

Appendix A. Proofs of Correctness Properties

A.1. Proof of Lemma 1: No Data Races

Let Ca_x be the cache $(caid_x \bullet M_x \bullet dst_x)$ and $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$ a reachable configuration. The following properties hold for $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$:

(a) $\forall n \in dom(M).(status(M,n) = inv \Leftrightarrow \exists\ Ca_i \in \overline{Ca}.\ status(M_i,n) = mo)$

(b) $\forall n \in dom(M).\ (status(M,n) = inv \Leftrightarrow (\exists\ Ca_i \in \overline{Ca}.\ status(M_i,n) = mo)$
$\quad \wedge \forall\ Ca_j \in \overline{Ca} \setminus \{Ca_i\}\ .\ status(M_j,n) \in \{inv, \perp\})$

(c) $\forall n \in dom(M).\ status(M,n) = sh \Leftrightarrow \forall\ Ca_i \in \overline{Ca}.\ status(M_i,n) \neq mo$

(d) $\forall\ Ca_i \in \overline{Ca}\ , \forall n \in dom(M_i).\ (status(M_i,n) = sh \Rightarrow status(M,n) = sh)$

Proof. An initial configuration $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$ satisfies the lemma since all the memory blocks in the main memory have the status *shared*, and all cores have empty caches and no data instructions or runtime statements.

Next we show the preservation of the invariant over transition steps:

$$M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H \xrightarrow{S} M' \circ \overline{T}' \circ \overline{Ca}' \circ \overline{CR}' : H' \tag{A.1}$$

where S is a set of send messages. Remember that the caches are exclusive in each core, and in order to apply SYNCH_1, S must contain at most one message for each block address n. In the following, the proof proceeds by case distinction on the rules for the transition steps. We simplify the proof by omitting the history annotations as they are irrelevant to this lemma. By induction, the configuration before the step, $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$, satisfies the lemma.

Case $S=\emptyset$. We first consider the case where $S=\emptyset$. Let $CR_i \in \overline{CR}$ where $CR_i = (cid \bullet rst) : h_i$ and $Ca_i, Ca_j \in \overline{Ca}$ where Ca_i and Ca_j are caches of same core. By rule ASYNCH in Figure 8, there are four possibilities of a reduction step when $S=\emptyset$: it can be (*i*) an internal transition between a core CR_i and its first local cache Ca_i (cf. rule PAR-INTERNAL-STEPS); or (*ii*) an internal transition between caches (cf. the rules PAR-CACHE); or (*iii*) a global step for CR_i to spawn a new task (cf. rule PAR-TASK-SPAWN), or to get a new task from the task queue (cf. rule PAR-TASK-SCHEDULER); or (*iv*) a global step for the communication between Ca_i and the main memory (cf. rule PAR-MEMORY-ACCESS).

- For transitions (*i*), after the decomposition with rule PAR-INTERNAL-STEPS, the relevant internal transitions between the core CR_i and its first level cache Ca_i are those un-labelled transitions in Figure 5. Those steps do not have any effect on the status of any memory block or main memory: either the status of the block remains unchanged, or is set to $\perp$ in the case where block is invalid or does not exist in the cache. Therefore by induction, the invariant still holds after the transition.

- It is analogous for transitions (*ii*), where the relevant transition between one or two caches are the un-labelled transitions in Figure 6. Most of those rules in Figure 6 do not affect any memory block and thus hold immediately. For cases LC-HIT$_1$ and LC-HIT$_2$ involving two caches that are at two consecutive levels, the cache $caid_i$ fetches the memory block with address n from the next level $caid_j$ without changing the status. By induction, the invariant holds before the step, and since all caches in a core are exclusive, block n will be removed from $caid_j$ after the transition. Thus, the invariant is maintained after the step. It is similar for cases LC-HIT$_2$, LC-BLOCK-FINISH$_1$ and LC-BLOCK-FINISH$_2$.

- Transitions (*iii*) hold immediately because the reduction steps do not change the status of any memory block.

- For transition (*iv*), decomposing the global configuration with rule PAR-MEMORY-ACCESS entails the application of one of the rules for fetching/flushing a block address n from/to the main memory in Figure 8. The proof proceeds by case distinction on those rules:

  Case FLUSH:
  $M \circ (caid_i \bullet M_i \bullet \mathbf{flush}(n); dst) \rightarrow M[n \mapsto \langle k, sh\rangle] \circ (caid_i \bullet M_i[n \mapsto \langle k, sh\rangle] \bullet dst)$
  We are further given $M_i(n) = \langle k, mo\rangle$. By induction, part (a) of the invariant gives $status(M,n) = inv$, and part (b) gives $\forall\ Ca_j \in \overline{Ca}\backslash\{Ca_i\}$ where $Ca_j = (caid_j \bullet M_j \bullet dst_j)$. $status(M_j,n) \in \{inv, \perp\}$ before the transition. Since main memory M and the cache M_i are updated to $M[n \mapsto \langle k, sh\rangle]$ and $M_i[n \mapsto \langle k, sh\rangle]$ after the step, which satisfies part (c) and part (d) of the lemma, and therefore concludes the case.

  Case FETCH$_2$:
  $M \circ (caid_i \bullet M_i \bullet \mathbf{fetchBl}(n); dst) \rightarrow M' \circ (caid_i \bullet M'_i \bullet dst)$

We are further given that $M = M'$, $M_i' = M_i[n' \mapsto \bot, n \mapsto \langle k, sh\rangle]$ where $select(M', n) = n'$ and $n' \neq n$, as well as $M(n) = \langle k, sh\rangle$, i.e., $status(M, n) = sh$. By induction, $status(M, n) = sh$ implies by part (c) of the invariant that $\forall\ Ca_i \in \overline{Ca}$. $status(M_i, n) \neq mo$ before the step. Since after the step $status(M', n) = status(M_i', n) = sh$, the invariant is maintained. It is analogous for case FETCH$_1$, which is a simpler case of FETCH$_2$.

The case for FETCH$_3$ holds immediately because the transition steps do not change the status of any memory location in either the main memory or cache local in a core.

Case $S \neq \emptyset$. Now we have to consider the set of sent messages S for a transition step in Equation (A.1) is *not empty*. The only applicable rules for the case where $S \neq \emptyset$ are GLOBAL-SYNCH and NODE-SYNCH in Figure 8, where updates will be done to the main memory and to each cache, respectively.

Assume Ca_i sends a message W for (exclusively) reading a block n in the following. Let us assume $S = S' \cup \{\xi\}$, where ξ can be either (*i*) $!Rd(n)$ or (*ii*) $!RdX(n)$. Note that the predicate $oneReqPerAddr(S)$ ensures that there is at most one message for each block address n in S. Due to this property, and to keep the proof clear, we further assume that $S' = \emptyset$, that is, no other cache sends any message apart from Ca_i. The proof is applicable to all other caches which send messages irrelevant to address n in parallel.

Case (i): $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} \xrightarrow{\{!Rd(n)\}} M' \circ \overline{T} \circ \overline{Ca}' \circ \overline{CR}'$
By rule SYNCH$_1$ in Figure 8, we are given that $R = dual(S)$, which is $\{?Rd(n)\}$ in this case. The premise $M \xrightarrow{\{?Rd(n)\}} M'$ corresponds to the transition for the main memory. By MAIN-MEMORY$_1$ and ONE-BLOCK-MAIN-MEMORY$_2$ in Figure 6, we have $M \xrightarrow{?Rd(n)} M'$, where $M' = M$.

Rule GLOBAL-SYNCH propagates the message to the cores $\overline{CR}$ and caches $\overline{Ca}$ with the premise $\overline{Ca} \circ \overline{CR} \xrightarrow{\{!Rd(n)\}} \overline{Ca}' \circ \overline{CR}'$, which is recursively decomposed by rule NODE-SYNCH. We then get

$$\overline{Ca_i} \circ CR_i \xrightarrow{\{!Rd(n)\}} \overline{Ca_i}' \circ CR_i' \tag{A.2}$$

where $belongs(\overline{Ca_i}, CR_i)$ ensuring the caches $\overline{Ca_i}$ belongs to the core CR_i, and

$$\overline{Ca}_j \circ CR_j \xrightarrow{\{?Rd(n)\}} \overline{Ca}_j' \circ CR_j' \quad \text{for all} \quad CR_j \in \overline{CR} \backslash \{CR_i\} \tag{A.3}$$

where $belongs(\overline{Ca}_j, \overline{CR}_j)$ ensuring the caches $\overline{Ca}_j$ belongs to exactly one core in $\overline{CR}_j$.

Consider equation (A.2), by rules RECEIVE-SEND-MESSAGE and SEND-MESSAGE$_2$ in Figure 7, we get $Ca_l \xrightarrow{!Rd(n)} Ca_l'$. The only relevant rule for this transition is rule LLC-MISS in Figure 6, which ensures Ca_l is the last level cache in a core. This rule guarantees that the block n either has status *inv* state or is undefined in the cache before the step, and is $\bot$ afterwards. Thus, the cache memory is not affected by the transition.

Consider the step in equation (A.3), rules RECEIVE-SEND-MESSAGE and RECEIVE-MESSAGE$_1$ and RECEIVE-MESSAGE$_2$ in Figure 7 give $Ca_j \xrightarrow{?Rd(n)} Ca_j'$, for all $Ca_j \in$

$\overline{Ca}_j$. The relevant rules, FLUSH-ONE-LINE and IGNORE-FLUSH-ONE-LINE in Figure 7, do not affect the status of block n. Since by induction, the configuration before the step satisfies the lemma, and the status of block address n is not affected in the main memory and in all cores after the labelled step, the global configuration after the transition $M' \circ \overline{T}' \circ \overline{Ca}' \circ \overline{CR}'$, also satisfies the lemma.

Case (ii): $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} \xrightarrow{\{!RdX(n)\}} M' \circ \overline{T} \circ \overline{Ca}' \circ \overline{CR}'$
By rule SYNCH$_1$ in Figure 8, we are given that $R = dual(S)$, which is $\{?RdX(n)\}$ in this case. The premise $M \xrightarrow{\{?RdX(n)\}} M'$ correspond to the transition in the main memory. Then, by rules MAIN-MEMORY$_1$ and ONE-BLOCK-MAIN-MEMORY$_1$ in Figure 7, we have $M \xrightarrow{?RdX(n)} M'$, where $M' = M[n \mapsto \langle_, inv\rangle]$.

Rule SYNCH$_1$ propagates the message to the cores $\overline{CR}$ and caches $\overline{Ca}$ with the premise $\overline{Ca} \circ \overline{CR} \xrightarrow{\{!RdX(n)\}} \overline{Ca}' \circ \overline{CR}'$, which is recursively decomposed by rule SYNCH$_2$. We then ultimately get

$$\overline{Ca}_i \circ CR_i \xrightarrow{\{!RdX(n)\}} \overline{Ca}'_i \circ CR'_i\ , \quad \text{and} \tag{A.4}$$

$$\overline{Ca}_j \circ CR_j \xrightarrow{\{?RdX(n)\}} \overline{Ca}'_j \circ CR'_j \quad \text{for all} \quad CR_j \in \overline{CR} \backslash \{CR_i\} \tag{A.5}$$

with $belongs(\overline{Ca}_i, CR_i)$ and $belongs(\overline{Ca}_j, \overline{CR}_j)$.

Consider equation (A.4), by rules RECEIVE-SEND-MESSAGE and SEND-MESSAGE$_1$ in Figure 7, we get $Ca_f \circ CR_i \xrightarrow{!RdX(n)} Ca'_f \circ CR'_i$. The relevant rules for this transition include PRWR$_2$ and PRWR$_5$ in Figure 5, where both ensure Ca_f is the first level cache of the core CR_i. We are further given in these two rules that the status of address n is *sh* before the step, which is updated to *mo* after the step.

For equation (A.5), rules RECEIVE-SEND-MESSAGE and RECEIVE-MESSAGE$_1$ and RECEIVE-MESSAGE$_2$ in Figure 7 give $Ca_j \xrightarrow{?RdX(n)} Ca'_j$ for all $Ca_j \in \overline{Ca}_j$. The applicable rules are INVALIDATE-ONE-LINE and IGNORE-INVALIDATE-ONE-LINE in Figure 7. The block address n either has status invalid or is undefined after the step for both rules. This together with, after the transition, $status(M', n) = inv$ and status of n is *mo* in Ca_f which is a cache in CR_i implies parts (a) and (b) of the lemma, and therefore concludes the case. □

A.2. Proof of Lemma 2: Consistent Shared Copies

Let $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$ be a reachable configuration where $status(M, n) = sh$. If $(caid_i \bullet M_i \bullet dst_i) \in \overline{Ca}$ such that $status(M_i, n) = sh$, then $version(M, n) = version(M_i, n)$.

Proof. An initial configuration $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$ satisfies the lemma since all the memory blocks in the main memory have the status *shared*, and all cores have empty caches and no data instructions or runtime statements. It is trivial that the invariant holds for the transition rules capturing two caches residing in the same core, or are local in the main memory because the transitions do not modify the version number of a block address.

We consider in the following the communication steps between main memory and a cache $Ca_i \in \overline{Ca}$, where the version number of the address n is involved. The rules responsible for such transition steps are those for fetching/flushing a memory block from/to the main memory by a cache in Figure 8. Let $Ca_i = caid_i \bullet M_i \bullet dst_i$ be the cache, and n be the block address.

Case FLUSH:
$M \circ (caid_i \bullet M_i \bullet \mathtt{flush}(n); dst'_i) \to M[n \mapsto \langle k, sh\rangle] \circ (caid_i \bullet M_i[n \mapsto \langle k, sh\rangle] \bullet dst'_i)$
We are further given $M_i(n) = \langle k, mo\rangle$ which implies $status(M,n) = inv$ by Lemma 1(a), and by Lemma 1(b), $\forall\, Ca_j \in \overline{Ca} \backslash \{Ca_i\}\ .\ status(M_j, n) \in \{inv, \perp\}$ where $Ca_j = (caid_j \bullet M_j \bullet dst_j)$. The case is concluded with $M[n \mapsto \langle k, sh\rangle]$ and $M_i[n \mapsto \langle k, sh\rangle]$ after the step.

Case FETCH$_2$:
$M \circ (caid_i \bullet M_i \bullet \mathtt{fetchBl}(n); dst'_i) \to M \circ (caid_i \bullet M_i[n' \mapsto \perp, n \mapsto \langle k, sh\rangle] \bullet dst'_i)$
We are further given that $M = M'$, $M'_i = M_i[n' \mapsto \perp, n \mapsto \langle k, sh\rangle]$ where $select(M', n) = n'$ and $n' \neq n$, as well as $M(n) = \langle k, sh\rangle$, i.e., $status(M,n) = sh$. By Lemma 1(c), $status(M,n) = sh$ implies $status(M_j, n) \neq mo$ for all $Ca_j \in \overline{Ca}$ where $Ca_j = (caid_j \bullet M_j \bullet dst_j)$. We just have to consider those caches $Ca_g \in \overline{Ca}$ where $Ca_g = caid_g \bullet M_g \bullet dst_g$ and $status(M_l, n) = sh$. (Note that the cases where $status(M_j, n) = inv$ or $n \in dom(M_j)$ are not relevant in this lemma.) By induction, $version(M,n) = version(M_g, n) = k$. Since we have after the step $M_i[n \mapsto \langle k, sh\rangle]$, and the status and version of address n in the main memory and in all other caches remain unchanged, the configuration satisfies the invariant. It is analogous for case FETCH$_1$, which is a simpler case of FETCH$_2$.

Case FETCH$_3$ holds immediately because the transition step does not change the status or version of any memory location in either the main memory.

For the transition steps of a core and its first level cache in Figure 5 where rules PRWR$_2$ and PRWR$_5$ increase the version number of a given block with address n by 1 for a successful write access, these two rules are irrelevant as the status of block n in M_i is updated to mo after the transition. □

A.3. *Proof of Lemma 3: Consistent Parallel Read Access*

Let CR_x be a core $(c_x \bullet rst_x)$ and Ca_y be a cache $(caid_y \bullet M_y \bullet dst_y)$. Assume further that $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$ is a reachable configuration where $CR_k, CR_l \in \overline{CR}$ and $Ca_k, Ca_l \in \overline{Ca}$. If

$$M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H \xrightarrow{S} M' \circ \overline{T}' \circ \overline{Ca}' \circ \overline{CR}' : H; \overline{ev}$$

such that $cid(caid_k) = c_k$ and $cid(caid_l) = c_l$ with $first(caid_k) = first(caid_l) = true$, then $\forall i, j, \forall n\ .\ R(c_i, n), R(c_j, n) \in \overline{ev}$, $version(M'_i, n) = version(M'_j, n)$.

Proof. If $S = \emptyset$, the transition step will only generate a write event $W(_,_)$ (see rules GLOBAL-SYNCH and NODE-SYNCH in Figure 10, and rules PRWR$_2$ and PRWR$_5$ in Figure 9), which is irrelevant to this lemma. Therefore, we focus on case $S \neq \emptyset$, which is captured by rule GLOBAL-ASYNCH in Figure 10.

In rule GLOBAL-ASYNCH, the reduction step that will generate events is the internal transitions between the core CR_i and its first level cache Ca_i, handled by rule PAR-

INTERNAL-STEPS in Figure 8. After the decomposition, the relevant rules are PRRD$_1$ and PRRD$_2$ in Figure 9 which capture the case where the step generates an event $R(c_i,n)$, meaning that core CR_i makes a successful read access to a block n by reading its first level cache. Assume there is another core $CR_j \in \overline{CR}$ that successfully reads block n from its first level cache, i.e., generate a event $R(c_j,n)$, in the same asynchronous step. Since the lemma trivially holds for the case $i = j$, we only discuss the case $i \neq j$ in the following.

Case PRRD$_1$:
$(caid_i \bullet M_i \bullet dst_i) \circ (c_i \bullet \mathtt{read}(r); rst_i') : h_i \to (caid_i \bullet M_i \bullet dst_i) \circ (c_i \bullet rst_i') : h_i; R(c_i,n)$
We are further given that $first(caid_i) = true$, $cid(caid_i) = c_i$ and $n = addr(r)$ as well as $status(M_i,n) \in \{sh, mo\}$. We have also the same conditions for $CR_j \circ Ca_j$.

If $status(M_i,n) = sh$, we get $status(M,n) = sh$ by Lemma 1(d), and consequently, Lemma 1(c) gives $status(M_j,n) = sh$. After the transition step, $R(c_i,n), R(c_j,n) \in \overline{ev}$ by rule GLOBAL-ASYNCH in Figure 10, and also both M_i and M_j are unchanged. Then by Lemma 2 we get $version(M_i,n) = version(M,n) = version(M_j,n)$, which satisfies the lemma.

If $status(M_i,n) = mo$, by Lemma 1(b), $status(M_j,n) = inv$, which contradicts the given condition $status(M_j,n) \in \{sh, mo\}$; thus, we conclude the case. It is analogous for case PRRD$_2$. ☐

A.4. Proof of Lemma 4: Preservation of Trace Semantics

If $(c \bullet rst) : \varepsilon \to^* (c \bullet rst') : h$, then $\{h; \tau \mid \tau \in [\![rst']\!]_c\} \subseteq [\![rst]\!]_c$.

Proof. The reduction $(c \bullet rst) : \varepsilon \to^* (c \bullet rst') : h$ captures a core c executing rst from an empty history ε, which reaches rst' with history h by making zero or more transition steps, where h is a sequence of successful read and write accesses generated during the execution local in the core . The proof proceeds by induction on the local transition steps for core c, as defined in Figure 5.

The invariant holds trivially for the initial configuration where the number of transition steps is *zero* and we then have $h = \varepsilon$ and $rst = rst'$. Next we are going to show the preservation of the local invariant over the transition steps, that is,

$$(c \bullet rst) : \varepsilon \to^z (c \bullet rst_1) : \varepsilon; h \to (c \bullet rst_2) : h'$$

for some h'. By induction, we have $(c \bullet rst) : \varepsilon \to^z (c \bullet rst_1) : \varepsilon; h$ for some h such that $\{h; \tau \mid \tau \in [\![rst_1]\!]_c\} \subseteq [\![rst]\!]_c$, where $z \geq 0$. We are going to show the lemma holds for the $z+1$ th step, that is,

$$(c \bullet rst_1) : \varepsilon; h \to (c \bullet rst_2) : h'$$

which may be labelled. The proof proceeds by case distinction on the rules for the local transition steps in Figure 5 (see Figure 9 for those steps where the history is extended after the transition).

Case PRRD$_1$: $(c \bullet \mathtt{read}(r); rst_2) : h \to (c \bullet rst_2) : h; R(c,n)$ where $n = addr(r)$.
In this case, $rst_1 = \mathtt{read}(r); rst_2$ and $h' = h; R(c,n)$. By induction, we have

$$\begin{aligned} & \{h; \tau \mid \tau \in [\![rst_1]\!]_c\} \subseteq [\![rst]\!]_c \\ = \quad & \{h; \tau \mid \tau \in [\![(\mathtt{read}(r); rst_2)]\!]_c\} \subseteq [\![rst]\!]_c \end{aligned} \tag{A.6}$$

By Definition 1, $[\![(\texttt{read}(r);rst_2)]\!]_c = \{\tau';\tau''|\tau' \in [\![\texttt{read}(r)]\!]_c, \tau'' \in [\![rst_2]\!]_c\}$, which gives traces of the form

$$R(c,n);\tau'' \in [\![\texttt{read}(r);rst_2]\!]_c \text{ where } R(c,n) \in [\![\texttt{read}(r)]\!]_c \text{ and } \tau'' \in [\![rst_2]\!]_c\ . \quad \text{(A.7)}$$

Defintion 1 gives $[\![\texttt{read}(r)]\!]_c = \{R(c,n)\}$, which is a singleton set. This together with equations (A.6) and (A.7), we get

$$\begin{array}{ll} & \{h;R(c,n);\tau'' \mid R(c,n);\tau'' \in [\![(\texttt{read}(r);rst_2)]\!]_c\} \subseteq [\![rst]\!]_c \\ = & \{h;R(c,n);\tau'' \mid \tau'' \in [\![(rst_2)]\!]_c\} \subseteq [\![rst]\!]_c \end{array} \quad \text{(A.8)}$$

which concludes the case. It is analogous for case PRRD$_3$.

Case PRRD$_2$: $(c \bullet \texttt{read}(r);rst_3) : h \rightarrow (c \bullet \texttt{readBl}(r);rst_3) : h$ where $n = addr(r)$. In this case, $rst_1 = \texttt{read}(r);rst_3$, $rst_2 = \texttt{readBl}(r);rst_3$ and $h' = h$. By induction and Definition 1, we have

$$\begin{array}{ll} & \{h;\tau \mid \tau \in [\![rst_1]\!]_c\} \subseteq [\![rst]\!]_c \\ = & \{h;\tau \mid \tau \in [\![(\texttt{read}(r);rst_3)]\!]_c\} \subseteq [\![rst]\!]_c \\ = & \{h;\tau \mid \tau \in \{\tau';\tau''|\tau' \in [\![\texttt{read}(r)]\!]_c, \tau'' \in [\![rst_3]\!]_c\}\} \subseteq [\![rst]\!]_c \end{array} \quad \text{(A.9)}$$

By Definition 1, $[\![\texttt{read}(r)]\!]_c = [\![\texttt{readBl}(r)]\!]_c$, which implies

$$\tau' \in [\![\texttt{readBl}(r)]\!]_c \text{ iff } \tau' \in [\![\texttt{read}(r)]\!]_c\ .$$

This together with equation (A.9) gives

$$\begin{array}{ll} & \{h;\tau \mid \tau \in \{\tau';\tau''|\tau' \in [\![\texttt{readBl}(r)]\!]_c, \tau'' \in [\![rst_3]\!]_c\}\} \subseteq [\![rst]\!]_c \\ = & \{h;\tau \mid \tau \in [\![(\texttt{readBl}(r);rst_3)]\!]_c\} \subseteq [\![rst]\!]_c \end{array} \quad \text{(A.10)}$$

which concludes the case. It is analogous for case PRRD$_4$.

Case PRWR$_1$: $(c \bullet \texttt{write}(r);rst_2) : h \rightarrow (c \bullet rst_2) : h;W(c,n)$ where $n = addr(r)$. In this case, $rst_1 = \texttt{write}(r);rst_2$ and $h' = W(c,n)$. By induction, we have

$$\begin{array}{ll} & \{h;\tau \mid \tau \in [\![rst_1]\!]_c\} \subseteq [\![rst]\!]_c \\ = & \{h;\tau \mid \tau \in [\![(\texttt{write}(r);rst_2)]\!]_c\} \subseteq [\![rst]\!]_c \end{array} \quad \text{(A.11)}$$

By Definition 1, $[\![(\texttt{write}(r);rst_2)]\!]_c = \{\tau';\tau''|\tau' \in [\![\texttt{write}(r)]\!]_c, \tau'' \in [\![rst_2]\!]_c\}$, which gives traces of the form

$$W(c,n);\tau'' \in [\![rst_1]\!]_c \text{ where } W(c,n) \in [\![\texttt{write}(r)]\!]_c \text{ and } \tau'' \in [\![rst_2]\!]_c\ . \quad \text{(A.12)}$$

By defintion 1, $[\![\texttt{write}(r)]\!]_c = \{W(c,n)\}$, which is a singleton set. This together with equations (A.11) and (A.12), we get

$$\begin{array}{ll} & \{h;W(c,n);\tau'' \mid W(c,n);\tau'' \in [\![(\texttt{write}(r);rst_2)]\!]_c\} \subseteq [\![rst]\!]_c \\ = & \{h;W(c,n);\tau'' \mid \tau'' \in [\![(rst_2)]\!]_c\} \subseteq [\![rst]\!]_c \end{array} \quad \text{(A.13)}$$

which concludes the case. It is analogous for cases PRWR$_2$, PRWR$_4$ and PRWR$_5$.

Case PRWR$_3$: $(c \bullet \mathtt{write}(r); rst_3) : h \to (c \bullet \mathtt{writeBl}(r); rst_3) : h$ where $n = addr(r)$. In this case, $rst_1 = \mathtt{write}(r); rst_3$, $rst_2 = \mathtt{writeBl}(r); rst_3$ and $h' = h$. By induction and Definition 1, we have

$$\begin{array}{rl} & \{h;\tau \mid \tau \in [\![rst_1]\!]_c\} \subseteq [\![rst]\!]_c \\ = & \{h;\tau \mid \tau \in [\![(\mathtt{write}(r); rst_3)]\!]_c\} \subseteq [\![rst]\!]_c \\ = & \{h;\tau | \tau \in \{\tau';\tau'' | \tau' \in [\![\mathtt{write}(r)]\!]_c, \tau'' \in [\![rst_3]\!]_c\}\} \subseteq [\![rst]\!]_c \end{array} \tag{A.14}$$

By Definition 1, $[\![\mathtt{write}(r)]\!]_c = [\![\mathtt{writeBl}(r)]\!]_c$, which implies

$$\tau' \in [\![\mathtt{writeBl}(r)]\!]_c \text{ iff } \tau' \in [\![\mathtt{write}(r)]\!]_c \; .$$

This together with equation (A.14) gives

$$\begin{array}{rl} & \{h;\tau \mid \tau \in \{\tau';\tau'' | \tau' \in [\![\mathtt{writeBl}(r)]\!]_c, \tau'' \in [\![rst_3]\!]_c\}\} \subseteq [\![rst]\!]_c \\ = & \{h;\tau \mid \tau \in [\![(\mathtt{writeBl}(r); rst_3)]\!]_c\} \subseteq [\![rst]\!]_c \end{array} \tag{A.15}$$

which concludes the case. Case PRWR$_6$ is analogous.

Case COMMIT: $(c \bullet \mathtt{commit}(r); rst_2) : h \to (c \bullet rst_2) : h$.
In this case, $rst_1 = \mathtt{commit}(r); rst_2$ and $h' = h$. By induction, we have

$$\begin{array}{rl} & \{h;\tau \mid \tau \in [\![rst_1]\!]_c\} \subseteq [\![rst]\!]_c \\ = & \{h;\tau \mid \tau \in [\![\mathtt{commit}(r); rst_2]\!]_c\} \subseteq [\![rst]\!]_c \end{array} \tag{A.16}$$

By Definition 1, we have $[\![\mathtt{commit}(r)]\!]_c = \varepsilon$ and $[\![(\mathtt{commit}(r); rst_2)]\!]_c = \{\varepsilon;\tau' | \varepsilon \in [\![\mathtt{commit}(r)]\!]_c, \tau' \in [\![rst_2]\!]_c\}$, which implies $\varepsilon;\tau' \in [\![\mathtt{commit}(r); rst_2]\!]_c$. This together with $\varepsilon;\tau' = \tau'$ by having $\varepsilon;h = h$ and equation (A.16) give

$$\begin{array}{rl} & \{h;\varepsilon;\tau' \mid \varepsilon;\tau' \in [\![\mathtt{commit}(r); rst_2]\!]_c\} \subseteq [\![rst]\!]_c \\ = & \{h;\tau' \mid \varepsilon;\tau' \in [\![\mathtt{commit}(r); rst_2]\!]_c\} \subseteq [\![rst]\!]_c \\ = & \{h;\tau' \mid \tau' \in [\![rst_2]\!]_c\} \subseteq [\![rst]\!]_c \end{array} \tag{A.17}$$

which concludes the case. Case COMMITALL is analogous.

Case CHOICE: $(c \bullet dap_1 \sqcap dap_2; rst_3) : h \to (c \bullet dap_1; rst_3) : h$.
In this case, $rst_1 = dap_1 \sqcap dap_2; rst_3$, $rst_2 = dap_1; rst_3$ and $h' = h$. By induction, we have

$$\begin{array}{rl} & \{h;\tau \mid \tau \in [\![rst_1]\!]_c\} \subseteq [\![rst]\!]_c \\ = & \{h;\tau \mid \tau \in [\![(dap_1 \sqcap dap_2); rst_3]\!]_c\} \subseteq [\![rst]\!]_c \end{array} \tag{A.18}$$

Then by Definition 1,

$$[\![(dap_1 \sqcap dap_2); rst_3]\!]_c = \{\tau';\tau'' | \tau' \in [\![dap_1]\!]_c \cup [\![dap_2]\!]_c, \tau'' \in [\![rst_3]\!]_c\} \; , \tag{A.19}$$

which is a superset of

$$[\![dap_1; rst_3]\!]_c = \{\tau''';\tau'' | \tau''' \in [\![dap_1]\!]_c, \tau'' \in [\![rst_3]\!]_c\} \; . \tag{A.20}$$

Equations (A.19) and (A.20) gives

$$\{h;\tau''';\tau'' \mid \tau''';\tau'' \in [\![dap_1; rst_3]\!]_c\} \subseteq \{h;\tau \mid \tau \in [\![(dap_1 \sqcap dap_2); rst_3]\!]_c\} \; . \tag{A.21}$$

Thus, equations (A.18) and (A.21), together with transitivity, implies

$$\{h;\tau''';\tau'' \mid \tau''';\tau'' \in [\![dap_1;rst_3]\!]_c\} \subseteq [\![rst]\!]_c \tag{A.22}$$

which concludes the case.

Case REPETITION: $(c \bullet dap^*;rst_3) : h \to (c \bullet (dap;dap^*) \sqcap \mathbf{skip};rst_3) : h.$
In this case, $rst_1 = dap^*;rst_3$, $rst_2 = (dap;dap^*) \sqcap \mathbf{skip};rst_3$ and $h' = h$. By induction and Definition 1, we have

$$\begin{array}{ll} & \{h;\tau \mid \tau \in [\![rst_1]\!]_c\} \subseteq [\![rst]\!]_c \\ = & \{h;\tau \mid \tau \in [\![dap^*;rst_3]\!]_c\} \subseteq [\![rst]\!]_c \\ = & \{h;\tau \mid \tau \in \{\tau';\tau'' | \tau' \in [\![dap^*]\!]_c, \tau'' \in [\![rst_3]\!]_c\}\} \subseteq [\![rst]\!]_c \end{array} \tag{A.23}$$

By Definition 1,

$$\begin{array}{rl} [\![dap^*]\!]_c = & [\![dap;dap^*]\!]_c \cup [\![\mathbf{skip}]\!]_c \\ = & [\![(dap;dap^*) \sqcap \mathbf{skip}]\!]_c \end{array} \tag{A.24}$$

which implies

$$\tau' \in [\![(dap;dap^*) \sqcap \mathbf{skip}]\!]_c \text{ iff } \tau' \in [\![dap^*]\!]_c \ .$$

This together with equation (A.23) gives

$$\begin{array}{ll} & \{h;\tau \mid \tau \in \{\tau';\tau'' | \tau' \in [\![(dap;dap^*) \sqcap \mathbf{skip}]\!]_c, \tau'' \in [\![rst_3]\!]_c\}\} \subseteq [\![rst]\!]_c \\ = & \{h;\tau \mid \tau \in [\![(dap;dap^*) \sqcap \mathbf{skip};rst_3]\!]_c\} \subseteq [\![rst]\!]_c \end{array} \tag{A.25}$$

which concludes the case. □

A.5. Proof of Theorem 2: No Access to Stale Data

Let $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$ be a reachable configuration such that $CR_i = (c_i \bullet rst_i)$ for $CR_i \in \overline{CR}$ with local history $h_i = H/c_i$, $Ca_i = (caid_i \bullet M_i \bullet dst_i)$ for $Ca_i \in \overline{Ca}$ and $belongs(Ca_i, CR_i)$. Consider a block address n and an event $e \in \{R(c_i,n), W(c_i,n)\}$.
If $Ca_i \circ CR_i : h_i \to Ca_i' \circ CR_i' : (h_i;e)$ or $Ca_i \circ CR_i : h_i \xrightarrow{!RdX(n)} Ca_i' \circ CR_i' : (h_i;e)$, then $M_i(n)$ has the most recent value.

Proof. For the core CR_i to generate an event $R(c_i,n)$ or $W(c_i,n)$, i.e., to make *successful* read or write accesses to block address n in its local cache, the relevant rules are those capturing the interactions between the core and its first level cache. The proof therefore proceeds by case distinction on these rules, namely PRRD$_1$, PRRD$_3$, PRWR$_1$, PRWR$_2$, PRWR$_4$ and PRWR$_5$ in Figure 9.

We first consider the cases concerning an un-labelled transition step, which involve rules PRRD$_1$, PRRD$_3$, PRWR$_1$ and PRWR$_4$.

Case PRRD$_1$:
$(caid_i \bullet M_i \bullet dst_i) \circ (c_i \bullet \mathbf{read}(r);rst_i') : h_i \to (caid_i \bullet M_i \bullet dst_i) \circ (c_i \bullet rst_i') : h_i;R(c_i,n)$
We are further given that $n = addr(r)$ and $status(M_i,n) = sh \vee mo$. Note that M_i remains unchanged after the transition. If $M_i(n) = \langle k, mo \rangle$, by induction, M_i has a version number of n greater than that in main memory or in other caches. The lemma then follows from Lemma 1(a) and (b) that $M_i(n)$ has the most recent copy according to

Definition 2(b). If $M_i(n) = \langle k, sh \rangle$, it follows from Lemma 1(d) that $status(M,n) = sh$, and consequently from Lemma 1 (c) that $\forall Ca_j \in \overline{Ca}.\ status(M_j,n) \neq mo$ where $Ca_j = (caid_j \bullet M_j \bullet dst_j)$. We only need to consider all caches $caid_g \bullet M_g \bullet dst_g \in \overline{Ca}$ where $status(M_g,n) = sh$. From Lemma 2, we get $version(M_i,n) = k = version(M,n) = version(M_g,n)$, which satisfies Definition 2(a). This concludes the case. The other three cases can be proven analogously.

The proof then proceeds by the cases concerning a labelled transition step, which involve rules PRWR_2 and PRWR_5.

Case PRWR_2:
$(caid_i \bullet M_i \bullet dst_i) \circ (c_i \bullet \mathbf{write}(r); rst'_i) : h_i \xrightarrow{!RdX(n)} (caid_i \bullet M'_i \bullet dst_i) \circ (c_i \bullet rst_i) : h_i; W(c_i,n)$
We are also given that $n = addr(r)$ and $M_i(n) = \langle k, sh \rangle$, which by induction gives $M(n) = \langle k, sh \rangle$ and $M_j(n) = \langle k, sh \rangle$ for all $(caid_j \bullet M_j \bullet dst_j) \in \overline{Ca} \backslash \{Ca_i\}$ as well as $version(M_i,n) = k = version(M,n) = version(M_j,n)$. We are further given after the transition that $M'_i = M_i[n \mapsto \langle k+1, mo \rangle]$. Then, by Lemma 1(a) and (b), we have $status(M,n) = inv$ and $status(M_j,n) = inv$, while the version number of n in main memory M and other caches M_j remain the same. Since $version(M'_i,n) = k+1 > k$, we conclude the case. It is analogous for case PRWR_5. □

A.6. Proof of Theorem 3: No mutually blocked nodes

Proof. The proof proceeds by contradiction. Assume a reachable configuration

$$M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H \tag{A.26}$$

in which there is a set of k mutually blocked nodes $\{CR_1 \circ \overline{Ca}_1, \ldots, CR_k \circ \overline{Ca}_k\}$, where $k \geq 2$. By rule LLC-MISS, a read request message $!Rd(n)$ is sent before a node becomes blocked. Thus, we can assume that all k nodes have made such a step to arrive at the reachable configuration in Equation (A.26). However, to apply rule LLC-MISS, it is necessary to first apply the global rules GLOBAL-SYNCH and NODE-SYNCH, such that the dual of the read request message $?Rd(n)$ from each of the k mutually blocked nodes is propagated to the main memory and to the caches that do not belong to the sender in *one single synchronisation step*. Upon receiving $?Rd(n)$, the corresponding cache in each of the mutually blocked nodes that has the requested address n in mo state puts a $\mathbf{flush}(n)$ at the front of its dst, which breaks Definition 3(b). Consequently, this violates Definition 4(a), which contradicts the assumption that the k nodes are mutually blocked in the reachable configuration in Equation (A.26). □