UNIVERSITY OF OSLO
Department of Informatics

Technical Report: Modelling Data Access Patterns with Atomic Sections for Multicore Architectures

Shiji Bijo,
Ka I Pun, and
S. Lizeth Tapia Tarifa

ISBN 978-82-7368-414-1
ISSN 0806-3036

July 2017

Modelling Data Access Patterns with Atomic Sections for Multicore Architectures *

Shiji Bijo, Ka I Pun, S. Lizeth Tapia Tarifa

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

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, parallel tasks accessing the same memory blocks may create hot spots in memory, when the same data is frequently transferred between caches in different cores. This negatively impacts software performance. In this paper, we study this problem in terms of a formal model of cache coherent data movement, in which tasks representing data access patterns with atomic sections execute on multicore architectures where cores with local caches communicate through an abstract communication medium. We develop an operational semantics that captures the interaction of dynamically spawned tasks with the underlying architecture and show the correctness of the model in terms of coherent access to shared data and exclusive access to atomic sections. This paper also presents a proof of concept implementation in rewriting logic which enables specifying and comparing data access patterns executing on models of different underlying architectures.

1 Introduction

Parallel computing enhances the performance of software applications by distributing the execution of tasks across multiple cores or processors. However, tasks running in parallel on different cores may need to access shared data simultaneously. Cache memory is typically used to provide quick access to recently used data, but it allows multiple copies of data to co-exist during execution. To fully benefit from multicore architectures, it is essential to understand how software applications interact with these architectures at runtime; i.e., 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 provide much guidance for software developers in understanding how data is transferred between main memory and caches during task execution. This is because (1) models of parallel programs generally abstract from caches in multicore architectures and only assume single copies of data in shared memory (i.e., they assume all threads have direct access to main memory), or (2) formal models of

* Supported by the EU project FP7-612985 *UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimisations* (www.upscale-project.eu) and the *SIRIUS Centre for Scalable Data Access* (www.sirius-labs.no).

hardware architecture and consistency protocols, such as cache coherence, focus on low-level correctness and completely abstract from the programming level, (i.e., how different task workflows compete for access to shared data). Integrating models of parallel execution and of hardware architecture enables reasoning about how data movement in the architecture is triggered by parallel access to data in shared memory.

As an example, consider the program code in Fig. 1. Here the method `worker` iterates `n` times to acquires a read access followed by a write access to the shared variable `sum`. In a parallel setting, such accesses could entail `2n` cache misses in the worst case, as the locally cached value of `sum` may be invalidated by other tasks after each read and write access. As an optimisation to reduce the number of cache misses, we can use atomic sections. These could be introduced by the compiler, derived by static analysis, or manually inserted by a programmer. In this particular case, an atomic section can be added to each iteration or it can contain the whole while-loop, depending on how heavy the computation of method `m` is.

```
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) { ... }
```

Fig. 1: A program repeatedly accessing a shared variable.

For the program above, we can model data movement using data access patterns such as $(\mathbf{read}(\mathtt{sum});\mathbf{write}(\mathtt{sum}))^n$, abstracting from the actual computation. We can use these patterns together with models of hardware architectures to understand how tasks running on different cores interact with memory. The purpose of such a model is not to evaluate the specifics of a concrete hardware architecture, but rather to formally describe a model of program execution in a setting with multiple and consistent copies of the same data in shared memory and in caches. Inspired by programming language semantics, we have developed a formal model of data access patterns and hardware architecture together with an operational semantics capturing cache coherent parallel execution [2–4]. This model abstracts from the actual communication medium (e.g., a bus or a ring), and uses the cache coherency protocol to orchestrate parallel executions on different cores, by restricting data accesses to the memory components of the architecture, and therefore ensuring consistency of data access.

As the main contribution of this paper, we study the behaviour of programs with atomic sections. We extend the formal model of data access patterns and its operational semantics to account for atomic sections protected by locks. Such extension add precision to our model, since it will approximate better the number of valid execution paths of programs e.g., blocking algorithms. The technical contributions of this paper are: (1) an operational model of execution on multicore architectures for tasks describing data access patterns with loops, choice, spawn and locks; (2) correctness properties of this formal model, expressed as invariants over an arbitrary number of cores: the preservation of the program order for the data access patterns, the absence of data races, the access to the most recent data value and the exclusive access of atomic sections; and (3) a proof of concept implementation of the model in the rewriting tool Maude [7].

Paper overview. Sec. 2 briefly reviews background concepts on atomic sections with binary locks in multicore architectures, Sec. 3 presents our model of multicore architectures executing tasks with atomic sections, Sec. 4 summarises the proven correctness

properties of the model, Sec. 5 presents a proof of concept implementation and an example, Sec. 6 discusses the related work, and Sec. 7 concludes the paper.

2 Multicore Architectures and Atomic Sections

In this section we briefly discuss basic background related to multicore architecture and to the implementation of mutual exclusion using locks, for further details see, e.g., [9, 10, 17, 21].

Multicore Architecture. Multicore architectures with shared memory use caches to give the cores rapid access to recently used data. Although it reduces traffic to main memory, it adds complexity to the architecture because multiple copies of the same data must coexist coherently. Shared data is stored in main memory as *words*, each with a unique reference. Multiple continuous words constitute a *block*, which has a distinct memory address. Cache memory is organised in cache lines, which store memory blocks. During program execution, cores access data in memory as a word using its reference, while the cache fetches the entire memory block containing the required word and stores it in a cache line. Blocks in cache lines may need to be evicted to free space for newly fetched blocks. The choice of which block to evict depends on the cache line organisation, the so-called *associativity*, and the *replacement policy*. 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. A *memory consistency model* [1] for cache-based architectures combines a (weak or strong) local memory model with a cache coherence protocol, which ensures the consistency of data between the caches of different cores. The local memory model and the cache coherence protocol are traditionally completely orthogonal: a weak memory model may be built on top of a cache coherence protocol which (normally) guarantees sequential consistency between caches. *Invalidation-based protocols* inform 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 the 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 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 in the communication medium. Following the 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 invalidating other copies of the same block in other caches.

Mutual Exclusion with Locks. Mutual exclusion can be enforced by using locks. The implementation of locks generally needs hardware support for some sort of atomic read-write instructions. *Test-and-set* is an atomic instruction that reads a value from a memory location and stores it in a local register, then it writes `1` to the memory location and returns the saved initial value. If a core performs a test-and-set and gets the return value as `0`, which means that it has got the lock; if the value is `1`, some other core has taken

the lock and the current core has to try again until it succeeds. Lock acquisition with test-and-set instruction is expensive compared to simple reads and writes because all cores that attempt to take the lock will write to the same memory location repeatedly until it succeeds, which can lead to both memory contention and cache invalidation overhead. To reduce such overhead, locks can be implemented by another instruction *test and test-and-set*, where a core that tries to acquire a lock will first repeatedly read the lock value from its local cache until it is invalidated, indicating that another core has modified the value of the lock, then the core will perform a test-and-set.

3 A Formal Model of Execution on Multicore Architectures with Atomic Sections

3.1 Abstractions in the Formal Model

Let us consider a model of multicore architectures with a communication medium that abstracts from concrete topologies, but ensures cache coherency using the MSI protocol. The architecture is illustrated in Fig. 2. In the model, each core has one private cache and may execute tasks that are scheduled from a shared pool, where tasks may contain one or more sections protected by at least one *binary lock*. A binary lock has two values: `0` refers to *unlocked* (or *free*), and `1` to *locked* (or *taken*). Cores communicate with each other by broadcasting request messages $!Rd(n)$ via the medium to read from a reference r in a memory block n, or $!RdX(n)$ to write to n or to manipulate a lock in n. Each cache has a queue D with fetch/flush instructions, to transfer blocks of data between caches and main memory.

To read data from a block n, the core first looks for n in its local cache. If we get a *cache miss*, the cache broadcasts a *read request* $!Rd(n)$ via the communication medium to the other caches and main memory. The cache *fetches* the block when it is available in main memory. Eviction is required if the cache is full. Writing to a block n is only allowed if it is in either a shared or modified state in the local cache, then an *invalidation request* $!RdX(n)$ is broadcast through the medium to obtain exclusive access. Let $?Rd(n)$ and $?RdX(n)$ respectively denote the reception of read and invalidation requests by a cache or by main memory. They are the duals of the broadcast requests mentioned above. If a cache receives a read request $?Rd(n)$ and it has the block in modified state, the cache *flushes* the block

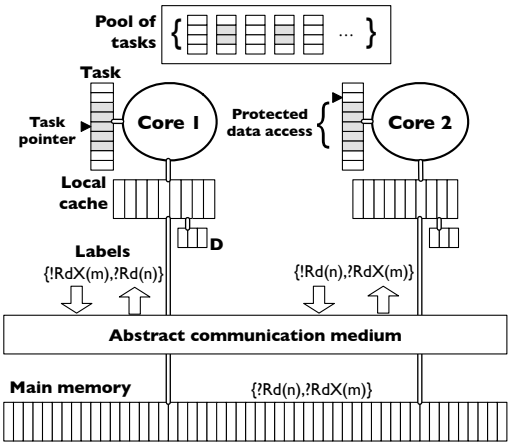

Fig. 2: Overview of the model. Where the data in both tasks is protected between a `lock`(r) and `unlock`(r) and r is in the block address n.

to main memory; if the cache receives an invalidation request $?RdX(n)$ and it has the block in shared state, the cache line will be *invalidated*; the requests are discarded otherwise. To take a lock residing in block n, the core first checks the block locally. If the lock value is `0` (i.e., free), it sends an invalidation messages $!RdX(n)$ to the other components in the architecture and takes the lock; if the value is `1` (i.e., take), it waits until its local copy is invalidated by receiving a $?RdX(n)$ message. The cache then fetches the lock from main memory and repeats the process until it succeeds. We model lock manipulations according to *test and test-and-set* instruction.
For simplicity, we abstract away the actual data stored in memory blocks and represent it with some symbolic value, except the value of locks. We also assume that locks must be released before they are taken again in the same protected section, that a cache line has the same size as a memory block, and that block is transferred between cores via the main memory. Read and invalidation requests in the communication medium are instantaneous in our model; this is justified because message transfer is an order of magnitude faster than data transfer. We can then match dual labels in a labelled transition system 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 in the proposed operational semantics.

3.2 The Syntax of the Runtime Configurations

The syntax of the formal model is shown in Fig. 3. A configuration *Config* consists of main memory M, shared among multiple cores $\overline{CR}$ with their own caches $\overline{Ca}$, a set of tasks $\overline{T}$ to be executed and a global history H. A core CR with identifier Cid executes runtime statements rst and a local history h. A cache Ca consists of a core identifier to which it belongs, a memory M, a sequence of data instructions dst to be performed and a global history H. A memory $M : n \rightharpoonup \langle k, val, st \rangle$ maps addresses n to a triple of a version number k, a stored value val and a status st. The version number k is part of the monitoring information used to prove properties such that cores never access stale data, see Sec. 4 for more details. The status tags *mo*, *sh*, and *inv* refer to the three states in the MSI protocol, respectively. Note that blocks in main memory can only be in *sh* or *inv* state.
The task table $Tb : T \rightharpoonup dap$ associates task identifiers T to *data access patterns* dap, which are sequences of basic operations $\texttt{read}(r)$ and $\texttt{write}(r)$ to read from and write to a memory reference r, $\texttt{commit}(r)$ to flush r to main memory. Access patterns also include a *locking block* which encapsulates any dap with a **`lock`** statement and an **`unlock`** statement. Note that a special memory block is reserved for each lock reference r, whose value is either 1 (*taken*) or 0 (*free*). The patterns further include control flow statements $dap_1 \sqcap dap_2$ to select either dap_1 or dap_2 for execution, dap^* to execute dap zero or many times, and $\texttt{Spawn}(T)$ to add T to the pool of tasks to be scheduled, where T is associated to a data access pattern dap in the task table Tb. To ensure data consistency, the statement **`commit`** is used at the end of each task to flush the entire cache after task execution. For simplicity, we assume the task table is statically given and is always available, and thus it is not explicitly shown in the configurations.

Syntactic categories.	***Definitions.***	
$Cid \in CoreId$	$Config$	$::= M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$
$n \in Address$	$CR \in Core$	$::= Cid \bullet rst : h$
$T \in TaskId$	$Ca \in Cache$	$::= Cid \bullet M \bullet dst$
$M \in Memory$	$M \in Memory$	$::= n \rightharpoonup \langle k, val, st \rangle$
$Tb \in TaskTable$	$st \in Status$	$::= \{mo, sh, inv\}$
	$Tb \in TaskTable$	$::= T \rightharpoonup dap$
	$rst \in RuntimeLang$	$::= dap \mid rst; rst \mid \texttt{readBl}(r) \mid \texttt{writeBl}(r)$
		$\texttt{lockBl}(r) \mid \texttt{unlockBl}(r)$
	$dap \in AccessPatterns$	$::= \varepsilon \mid dap; dap \mid \texttt{lock}(r); dap; \texttt{unlock}(r)$
		$\mid \texttt{read}(r) \mid \texttt{write}(r) \mid \texttt{commit}(r) \mid \texttt{commit}$
		$\mid dap \sqcap dap \mid dap^* \mid \texttt{skip} \mid \texttt{Spawn}(T)$
	$dst \in DataLang$	$::= \varepsilon \mid dst; dst \mid \texttt{fetch}(n) \mid \texttt{flush}(n)$
	$H \in GlobHistory$	$::= \varepsilon \mid H; \overline{ev}$
	$h \in LocHistory$	$::= \varepsilon \mid h; ev$
	$ev \in Events$	$::= W(Cid, n) \mid R(Cid, n)$

Fig. 3: Syntax for the abstract model of parallel architectures, where over-bar denotes sets (e.g., $\overline{CR}$), n represents memory addresses and r references.

The cores execute *runtime statements rst*, which extend *dap* with the additional control statements $\texttt{readBl}(r)$, $\texttt{writeBl}(r)$, $\texttt{lockBl}(r)$ and $\texttt{unlockBl}(r)$ to indicate that the core is blocked as a result of a cache miss. Each cache performs *data instructions dst*, which are sequences of $\texttt{fetch}(n)$ and $\texttt{flush}(n)$ to fetch a block n from main memory, and to flush the modified copy of n to the main memory, respectively.
The global history H records the concurrent executions in all cores $\overline{CR}$ in the architecture in terms of a sequence of sets of events $\overline{ev}$. Each event ev indicates a successful access to a block n by a core Cid, denoted by $R(Cid, n)$ for reading and $W(Cid, n)$ for writing. On the other hand, the local history h keeps track of the local execution of a single core in terms of a sequence of events. Thus, the local history h of a particular core is a projection over global history H with respect to that core. By abuse of notation, ε also represents empty history in both local and global levels. Same as the version numbers, histories are parts of the monitoring information to show correctness properties regarding program order and sequential consistency, see Sec. 4 for more details.

3.3 A Semantics of Parallel Execution

Local Semantics. We use structural operational semantics (SOS) [18] and transition labels to model parallel task execution and to synchronise read and invalidation requests from cores. The semantics is divided into local and global levels.
The *local semantics* captures task execution in each core as well as internal transitions in each cache and main memory in the configuration to ensure data consistency between different components. The *global semantics* captures transitions involving data transfer between the cache of each core and main memory, message broadcasting, task creation and scheduling, and enforces data consistency by following a global protocol and matching labels with composition rules. Multiple caches may request different

memory blocks at the same time with parallel instantaneous broadcast, using possibly empty sets of labels. The formal syntax for the label mechanism is as follows:

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

where S contains at most one label for each block address n.

Let $Config \xrightarrow{*} Config'$ denote an execution starting from *Config*, which reaches $Config'$, by recursively applying global transition rules, which in turn applies local transition rules for each component. In an *initial* configuration, all blocks in main memory M have status *sh* and default initial value, all locks are free, all cores are idle (i.e., *rst* is ε), all caches are empty and have no data instructions in *dst*, and the task pool in $\overline{T}$ names a single task, representing the main block of a program. A configuration $Config'$ is *reachable* if there is an execution $Config \xrightarrow{*} Config'$ starting from an initial configuration *Config*.

The local semantics captures the execution of statements in each core and the local state changes in each cache line according to the finite state controller enforcing the MSI protocal during the execution. The local transition rules are given in Figs. 4 and 5. A read access to a reference r by a core succeeds only if the memory block n containing r is available in its local cache in either shared or modified state as in rule PRRD_1, where a read event $R(c,n)$ is appended to history h after the transition. Otherwise, a read request message $!Rd(n)$ is broadcast in terms of a label as in rule PRRD_2 and the core is then blocked by a **readBl**(r) statement, and a **fetch**(n) instruction is appended to *dst*. Observe that no event is appended to h as reading from n has yet not succeeded. Once block n is fetched from main memory, execution may proceed as rule PRRD_3. However, in the parallel setting, the newly fetched cache line may get invalidated while the core is still blocked. Rule PRRD_4 captures this situation where the $!Rd(n)$ message is broadcast and a **fetch** instruction is added to the *dst* again.

Likewise, to write to a reference r, the memory block n containing r must be either in modified or shared state as in rules PRWR_1 and PRWR_2, where a write event $W(c,n)$ is appended to h. Note that if the status is shared, an invalidation message $!RdX(n)$ is broadcast to gain exclusive access to n as in rule PRWR_2. Similar to reading, if the requested block is invalid or not in the cache, the core first broadcasts a $!Rd(n)$ message and adds a **fetch** instruction to *dst* to fetch the block as in rule PRWR_3, while the execution is blocked by the statement **writeBl**(r) and no event is added. Once the block is fetched to the cache, rules PRWR_4 and PRWR_5 resume the execution according to the status of n. Rules COMMIT and COMMITALL flush one single cache line and entire cache, respectively by adding **flush**(n) to *dst*. Rule SKIP ignores the **skip**-statement and continues with the rest of the task. With the commutative $\sqcap$-operator, rule CHOICE non-deterministically selects either dap_1 or dap_2 for execution. Finally, rule REPETITION repeats the statement *dap* zero or many times.

Lock manipulations are defined in Fig. 5. A core can take a lock with reference r when it is free, i.e., *val* equals 0, while the status of block n where the lock resides can be either *sh* or *mo* as in rules LOCK_1 and LOCK_2. The value and the status of block n are updated to 1, i.e., *taken*, and to modified, respectively. Since taking or releasing a lock implies writing to lock references, exclusive access is required. Thus, if the status is

$$\frac{n = addr(r) \quad status(M,n) = sh \vee status(M,n) = mo}{(c \bullet M \bullet dst) \circ (c \bullet \texttt{read}(r); rst) : h \rightarrow (c \bullet M \bullet dst) \circ (c \bullet rst) : h; R(c,n)} \ (\textsc{PrRd}_1)$$

$$\frac{n = addr(r) \quad status(M,n) = inv \vee n \notin dom(M)}{(c \bullet M \bullet dst) \circ (c \bullet \texttt{read}(r); rst) : h \xrightarrow{!Rd(n)} (c \bullet M \setminus n \bullet dst; \texttt{fetch}(n)) \circ (c \bullet \texttt{readBl}(r); rst) : h} \ (\textsc{PrRd}_2)$$

$$\frac{n = addr(r) \quad status(M,n) = sh}{(c \bullet M \bullet dst) \circ (c \bullet \texttt{readBl}(r); rst) : h \rightarrow (c \bullet M \bullet dst) \circ (c \bullet rst) : h; R(c,n)} \ (\textsc{PrRd}_3)$$

$$\frac{n = addr(r) \quad status(M,n) = inv}{(c \bullet M \bullet dst) \circ (c \bullet \texttt{readBl}(r); rst) : h \xrightarrow{!Rd(n)} (c \bullet M \setminus n \bullet dst; \texttt{fetch}(n)) \circ (c \bullet \texttt{readBl}(r); rst) : h} \ (\textsc{PrRd}_4)$$

$$\frac{n = addr(r) \quad val' = value(M, \texttt{write}(r))}{(c \bullet M[n \mapsto \langle k, val, mo \rangle] \bullet dst) \circ (c \bullet \texttt{write}(r); rst) : h \rightarrow (c \bullet M[n \mapsto \langle k, val', mo \rangle] \bullet dst) \circ (c \bullet rst) : h; W(c,n)} \ (\textsc{PrWr}_1)$$

$$\frac{n = addr(r) \quad val' = value(M, \texttt{write}(r))}{(c \bullet M[n \mapsto \langle k, val, sh \rangle] \bullet dst) \circ (c \bullet \texttt{write}(r); rst) : h \xrightarrow{!RdX(n)} (c \bullet M[n \mapsto \langle k, val', mo \rangle] \bullet dst) \circ (c \bullet rst) : h; W(c,n)} \ (\textsc{PrWr}_2)$$

$$\frac{n = addr(r) \quad status(M,n) = inv \vee n \notin dom(M)}{(c \bullet M \bullet dst) \circ (c \bullet \texttt{write}(r); rst) : h \xrightarrow{!Rd(n)} (c \bullet M \setminus n \bullet dst; \texttt{fetch}(n)) \circ (c \bullet \texttt{writeBl}(r); rst) : h} \ (\textsc{PrWr}_3)$$

$$\frac{n = addr(r) \quad val' = value(M, \texttt{write}(r))}{(c \bullet M[n \mapsto \langle k, val, sh \rangle] \bullet dst) \circ (c \bullet \texttt{writeBl}(r); rst) : h \xrightarrow{!RdX(n)} (c \bullet M[n \mapsto \langle k, val', mo \rangle] \bullet dst) \circ (c \bullet rst) : h; W(c,n)} \ (\textsc{PrWr}_4)$$

$$\frac{n = addr(r) \quad status(M,n) = inv}{(c \bullet M \bullet dst) \circ (c \bullet \texttt{writeBl}(r); rst) : h \xrightarrow{!Rd(n)} (c \bullet M \setminus n \bullet dst; \texttt{fetch}(n)) \circ (c \bullet \texttt{writeBl}(r); rst) : h} \ (\textsc{PrWr}_5)$$

$$\frac{n = addr(r) \quad status(M,n) = mo}{(c \bullet M \bullet dst) \circ (c \bullet \texttt{commit}(r); rst) : h \rightarrow (c \bullet M \bullet dst; \texttt{flush}(n)) \circ (c \bullet rst) : h} \ (\textsc{Commit}_1)$$

$$\frac{n = addr(r) \quad status(M,n) \neq mo \vee n \notin dom(M)}{(c \bullet M \bullet dst) \circ (c \bullet \texttt{commit}(r); rst) : h \rightarrow (c \bullet M \bullet dst) \circ (c \bullet rst) : h} \ (\textsc{Commit}_2)$$

$$\frac{\exists n \in dom(M).status(M,n) = mo \quad \texttt{flush}(n) \notin dst}{(c \bullet M \bullet \texttt{commit}; dst) : h \rightarrow (c \bullet M \bullet \texttt{flush}(n); dst) : h} \ (\textsc{CommitAll}_1)$$

$$\frac{\forall n \in dom(M).status(M,n) \neq mo}{(c \bullet M \bullet \texttt{commit}; dst) : h \rightarrow (c \bullet M \bullet dst) : h} \ (\textsc{CommitAll}_2)$$

$$(c \bullet M \bullet dst) \circ (c \bullet \texttt{skip}; rst) : h \rightarrow (c \bullet M \bullet dst) \circ (c \bullet rst) : h \ (\textsc{Skip})$$

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

$$(c \bullet M \bullet dst) \circ (c \bullet dap^*; rst) : h \rightarrow (c \bullet M \bullet dst) \circ (c \bullet (dap; dap^*) \sqcap \texttt{skip}; rst) : h \ (\textsc{Repetition})$$

Fig. 4: Local semantics for task execution in cache coherent multicore architectures (Part I)

shared, the core broadcasts the message $!RdX(n)$ to have exclusive access to n, similar to rule $\textsc{PrWr}_2$. Also, a write event $R(c,n)$ is appended to h. If the requested lock is not available in the cache, similar to basic read and write operations, the core first needs

$$\frac{n = addr(r)}{(c \bullet M[n \mapsto \langle k,0,mo \rangle] \bullet dst) \circ (c \bullet \texttt{lock}(r); rst) : h \rightarrow (c \bullet M[n \mapsto \langle k,1,mo \rangle] \bullet dst) \circ (c \bullet rst) : h; W(c,n)} \text{(LOCK}_1\text{)}$$

$$\frac{n = addr(r)}{(c \bullet M[n \mapsto \langle k,0,sh \rangle] \bullet dst) \circ (c \bullet \texttt{lock}(r); rst) : h \xrightarrow{!RdX(n)} (c \bullet M[n \mapsto \langle k,1,mo \rangle] \bullet dst) \circ (c \bullet rst) : h; W(c,n)} \text{(LOCK}_2\text{)}$$

$$\frac{n = addr(r) \quad status(M,n) = inv \vee n \notin dom(M)}{(c \bullet M \bullet dst) \circ (c \bullet \texttt{lock}(r); rst) : h \xrightarrow{!Rd(n)} (c \bullet M \setminus n \bullet dst; \texttt{fetch}(n)) \circ (c \bullet \texttt{lockBl}(r); rst) : h} \text{(LOCKBLOCK}_1\text{)}$$

$$\frac{n = addr(r)}{(c \bullet M[n \mapsto \langle k,0,sh \rangle] \bullet dst) \circ (c \bullet \texttt{lockBl}(r); rst) : h \xrightarrow{!RdX(n)} (c \bullet M[n \mapsto \langle k,1,mo \rangle] \bullet dst) \circ (c \bullet rst) : h; W(c,n)} \text{(LOCKBLOCK}_2\text{)}$$

$$\frac{n = addr(r) \quad status(M,n) = inv}{(c \bullet M \bullet dst) \circ (c \bullet \texttt{lockBl}(r); rst) : h \xrightarrow{!Rd(n)} (c \bullet M \setminus n \bullet dst; \texttt{fetch}(n)) \circ (c \bullet \texttt{lockBl}(r); rst) : h} \text{(LOCKBLOCK}_3\text{)}$$

$$\frac{n = addr(r)}{(c \bullet M[n \mapsto \langle k,1,mo \rangle] \bullet dst) \circ (c \bullet \texttt{unlock}(r); rst) : h \rightarrow (c \bullet M[n \mapsto \langle k,0,mo \rangle] \bullet dst) \circ (c \bullet rst) : h; W(c,n)} \text{(UNLOCK}_1\text{)}$$

$$\frac{n = addr(r)}{(c \bullet M[n \mapsto \langle k,1,sh \rangle] \bullet dst) \circ (c \bullet \texttt{unlock}(r); rst) : h \xrightarrow{!RdX(n)} (c \bullet M[n \mapsto \langle k,0,mo \rangle] \bullet dst) \circ (c \bullet rst) : h; W(c,n)} \text{(UNLOCK}_2\text{)}$$

$$\frac{n = addr(r) \quad status(M,n) = inv \vee n \notin dom(M)}{(c \bullet M \bullet dst) \circ (c \bullet \texttt{unlock}(r); rst) : h \xrightarrow{!Rd(n)} (c \bullet M \setminus n \bullet dst; \texttt{fetch}(n)) \circ (c \bullet \texttt{unlockBl}(r); rst) : h} \text{(UNLOCKBLOCK}_1\text{)}$$

$$\frac{n = addr(r)}{(c \bullet M[n \mapsto \langle k,1,sh \rangle] \bullet dst) \circ (c \bullet \texttt{unlockBl}(r); rst) : h \xrightarrow{!RdX(n)} (c \bullet M[n \mapsto \langle k,0,mo \rangle] \bullet dst) \circ (c \bullet rst) : h; W(c,n)} \text{(UNLOCKBLOCK}_2\text{)}$$

Fig. 5: Local semantics for task execution in cache coherent multicore architectures (Part II)

to broadcast a $!Rd(n)$ message and appends a **`fetch`** instruction to fetch the block

as in rule LOCKBLOCK$_1$. The execution is then blocked by the statement **lockBl**(r) until the lock is available in the local cache as in rule LOCKBLOCK$_2$. A read request message $!Rd(n)$ is resent if the lock is invalidated after it is fetched when the core is blocked, see rule LOCKBLOCK$_3$. Releasing a lock is similar to locking, execpt that a core can only release a lock r if it has been taken, i.e., *val* equals 1, as in rules UNLOCK$_1$ and UNLOCK$_2$. Observe that our syntax restrains that only the core which owns the lock can release it. During the execution, block n may be removed from the cache to give space to another block, which can block unlocking. This is handled by rules UNLOCKBLOCK$_1$ and UNLOCKBLOCK$_2$.

Message receipts are manipulated by the labelled transition rules defined in Fig. 6. Rules MAIN-MEMORY 1 and 2 distribute the corresponding sets of labels. Invalidation requests to main memory are handled by updating the status of requested block to *inv* by MAIN-MEMORY-INVALIDATION, while read requests are ignored in MAIN-MEMORY-IGNORE.

Upon receiving a set of requests, each core decomposes the set by rules SEND-RECEIVE-MESSAGE that contains a W label for sending a request, RECEIVE-MESSAGE and REC-EMPTY. Note that receiving requests is prioritised over sending. When a core receives an exclusive request $?RdX(n)$ and if n is shared in the cache, the status of n will be invalidated; otherwise, the message will be ignored. When a core receives a read request $?Rd(n)$ and if n is modified in the cache, a **flush** instruction will be prepended to *dst* to ensure block n is first flushed to main memory before other blocks are read as well as to avoid deadlocks.

$$\frac{M \xrightarrow{R} M' \quad M' \xrightarrow{Q} M''}{M \xrightarrow{R\cup\{Q\}} M''}\ (\text{MAIN-MEMORY}_1) \qquad M \xrightarrow{\emptyset} M\ (\text{MAIN-MEMORY}_2) \qquad M \xrightarrow{?RdX(n)} M[n\mapsto\langle k,val,inv\rangle]\ (\text{MAIN-MEMORY-INVALIDATE})$$

$$M \xrightarrow{?Rd(n)} M\ (\text{MAIN-MEMORY-IGNORE}) \qquad \frac{Ca\circ CR \xrightarrow{R} Ca'\circ CR \quad Ca'\circ CR : h \xrightarrow{W} Ca''\circ CR' : h'}{Ca\circ CR : h \xrightarrow{R\cup\{W\}} Ca''\circ CR' : h'}\ (\text{SEND-RECEIVE-MESSAGE})$$

$$\frac{Ca \xrightarrow{Q} Ca' \quad Ca' \xrightarrow{R} Ca''}{Ca\circ CR \xrightarrow{\{Q\}\cup R} Ca''\circ CR}\ (\text{RECEIVE-MESSAGE})$$

$$Ca \xrightarrow{\emptyset} Ca\ (\text{REC-MSG-EMPTY}) \qquad c\bullet M[n\mapsto\langle k,val,sh\rangle]\bullet dst \xrightarrow{?RdX(n)} c\bullet M[n\mapsto\langle k,val,inv\rangle]\bullet dst\ (\text{INVALIDATE-ONE-LINE})$$

$$\frac{n\notin dom(M)\vee status(M,n)=inv}{c\bullet M\bullet dst \xrightarrow{?RdX(n)} c\bullet M\bullet dst}\ (\text{IGNORE-INVALIDATE})$$

$$\frac{status(M,n)=mo}{c\bullet M\bullet dst \xrightarrow{?Rd(n)} c\bullet M\bullet \mathbf{flush}(n);dst}\ (\text{FLUSH-ONE-LINE}) \qquad \frac{n\notin dom(M)\vee status(M,n)\neq mo}{c\bullet M\bullet dst \xrightarrow{?Rd(n)} c\bullet M\bullet dst}\ (\text{IGNORE-FLUSH})$$

Fig. 6: Local semantics for message passing in cache coherent multicore architectures

Global Semantics. The global semantics in Fig. 7 captures the interactions between different components in the configuration and ensures data coherency between caches

and main memory. To flush a modified block from a cache to main memory, rule FLUSH$_1$ replaces the block in main memory with the modified copy. The version number is incremented by one and is updated in the cache and main memory, where the status is set to *sh* in both. in both the cache and main memory, the status is set the version number is incremented by one The **`flush`** instruction is ignored in rule FLUSH$_2$ if the block is not in modified state in the cache. To get a block from main memory, a cache applies rule FETCH$_1$ if no eviction is required. In case eviction is needed, if the block chosen by the *select* function is not modified, the block is removed by rule FETCH$_2$; otherwise, the block should be flushed first by FETCH$_3$.
The rest of Fig. 7 deals with the synchronisation between components in the architecture. Rule TOP-SYNCH$_1$ defines the global synchronization for a non-empty set S of labels corresponding to broadcast requests. To apply the rule, S must contain at most one request per address, which is restricted by the predicate $\bigcap allAddrIn(S) = \emptyset$. The dual of S, the set of receiving messages R, is generated for updating main memory, while S is forwarded to the cores for synchronisation. The set S is then recursively decomposed by rule TOP-SYNCH$_2$ into sets of sending and receiving messages, which are distributed over two set of cores $\overline{CR}_1$ and $\overline{CR}_2$, such that each set eventually contains at most one W label. The decomposition repeats until the dual set R of S is propagated to each individual core. The *belongs* function relates cores to their own caches. The rule guarantees that the sender of a message W does not receive its dual Q. If the transitions generate any events, they will be merged into a set that is appended to the global history H.
Rule TOP-ASYNCH decomposes the asynchronous steps that does not involve message communications, including transitions local to individual cores with their caches which are further handled by rule PAR-INTERNAL-STEPS, task creation and allocation by rules PAR-TASK-SPAWN and PAR-TASK-SCHEDULER and parallel memory access by PAR-MEMORY-ACCESS. Observe that in PAR-TASK-SCHEDULER a **`commit`** statement is always added to the end of the scheduled task to ensure all modified data will be flushed before the next task is allocated to the same core.

4 Correctness of the Model

For the proposed model, we consider standard correctness properties for data consistency and cache coherency, based on the literature [9, 22], including the preservation of program order in each core, absence of data races and no access to stale data. We also show mutual exclusion of data access patterns protected by locks. The preservation of these properties by our semantics ensures that the model correctly captures cache coherent data movement as triggered by the underlying parallel architecture with any number of cores and caches, using our formalization of the MSI protocol for data consistency. The detailed proofs can be found in the appendix of this report.
We now formalize the denotational meaning of the *rst*-statements of our syntax, in terms of sets of local histories. Let $\preceq$ the reflexive prefix relation on histories.

Definition 1. *Let c be a core identifier and let $addr(r) = n$. The denotational meaning $[\![rst]\!]_c$ of a task rst is defined inductively as follows:*

$$\frac{M'(n) = \langle k, val', mo\rangle \quad k' = k+1}{M \circ (c \bullet M' \bullet \mathtt{flush}(n); dst) \rightarrow M[n \mapsto \langle k', val', sh\rangle] \circ (c \bullet M'[n \mapsto \langle k', val', sh\rangle] \bullet dst)} \text{(FLUSH}_1\text{)}$$

$$\frac{status(M', n) \neq mo \vee n \notin dom(M')}{M \circ (c \bullet M' \bullet \mathtt{flush}(n); dst) \rightarrow M \circ (c \bullet M' \bullet dst)} \text{(FLUSH}_2\text{)}$$

$$\frac{select(M', n) = n \quad M(n) = \langle k, val, sh\rangle}{M \circ (c \bullet M' \bullet \mathtt{fetch}(n); dst) \rightarrow M \circ (c \bullet M'[n \mapsto \langle k, val, sh\rangle] \bullet dst)} \text{(FETCH}_1\text{)}$$

$$\frac{select(M', n) = n' \quad n' \neq n \quad status(M', n') \neq mo \quad M(n) = \langle k, val, sh\rangle}{M \circ (c \bullet M' \bullet \mathtt{fetch}(n); dst) \rightarrow M \circ (c \bullet M'[n \mapsto \langle k, val, sh\rangle] \setminus n' \bullet dst)} \text{(FETCH}_2\text{)}$$

$$\frac{select(M', n) = n' \quad n' \neq n \quad status(M', n') = mo}{M \circ (c \bullet M' \bullet \mathtt{fetch}(n); dst) \rightarrow M \circ (c \bullet M' \bullet \mathtt{flush}(n'); \mathtt{fetch}(n); dst)} \text{(FETCH}_3\text{)}$$

$$\frac{S \neq \emptyset \quad \bigcap allAddrIn(S) = \emptyset \quad R = dual(S) \quad M \xrightarrow{R} M' \quad \overline{Ca} \circ \overline{CR} : H \xrightarrow{S} \overline{Ca'} \circ \overline{CR'} : H'}{M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H \xrightarrow{S} M' \circ \overline{T} \circ \overline{Ca'} \circ \overline{CR'} : H'} \text{(TOP-SYNCH}_1\text{)}$$

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

$$\frac{\overline{CR} = \overline{CR}_1 \uplus \overline{CR}_2 \uplus \overline{CR}_3 \quad \overline{Ca} = \overline{Ca}_1 \uplus \overline{Ca}_2 \uplus \overline{Ca}_3 \quad H' = H; \overline{ev} \quad belongs(\overline{Ca}_1, \overline{CR}_1) \quad \overline{Ca}_1 \circ \overline{CR}_1 : H/Cid(\overline{CR}_1) \rightarrow \overline{Ca'}_1 \circ \overline{CR'}_1 : H/Cid(\overline{CR}_1); \overline{ev} \quad \overline{T} \circ \overline{CR}_2 \rightarrow \overline{T'} \circ \overline{CR'}_2 \quad M \circ \overline{Ca}_2 \rightarrow M' \circ \overline{Ca'}_2 \quad \overline{CR'} = \overline{CR'}_1 \cup \overline{CR'}_2 \cup \overline{CR}_3 \quad \overline{Ca'} = \overline{Ca'}_1 \cup \overline{Ca'}_2 \cup \overline{Ca}_3}{M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H \xrightarrow{\emptyset} M' \circ \overline{T'} \circ \overline{Ca'} \circ \overline{CR'} : H'} \text{(TOP-ASYNCH)}$$

$$\frac{M[n \mapsto \alpha] \circ Ca \rightarrow M[n \mapsto \alpha'] \circ Ca' \quad M' = M \setminus n \quad M' \circ \overline{Ca} \rightarrow M'' \circ \overline{Ca'}}{M[n \mapsto \alpha] \circ Ca \circ \overline{Ca} \rightarrow M''[n \mapsto \alpha'] \circ Ca' \circ \overline{Ca'}} \text{(PAR-MEMORY-ACCESS)}$$

$$\frac{\begin{array}{c} belongs(Ca_1, CR_1) \quad H' = H; (\{ev_1\} \cup \overline{ev}_2) \\ Ca_1 \circ CR_1 : H/Cid(CR_1) \rightarrow Ca'_1 \circ CR'_1 : H/Cid(CR_1); ev_1 \\ \overline{Ca}_2 \circ \overline{CR}_2 : H/Cid(\overline{CR}_2) \rightarrow \overline{Ca'}_2 \circ \overline{CR'}_2 : H'/Cid(\overline{CR}); \overline{ev}_2 \end{array}}{Ca_1 \circ CR_1 \circ \overline{Ca}_2 \circ \overline{CR}_2 : H \rightarrow Ca'_1 \circ CR'_1 \circ \overline{Ca'}_2 \circ \overline{CR'}_2 : H'} \text{(PAR-INTERNAL-STEPS)}$$

$$\frac{\overline{T'} = \overline{T} \cup T \quad \overline{T'} \circ \overline{CR} \rightarrow \overline{T''} \circ \overline{CR'}}{\overline{T} \circ \overline{CR} \circ (Cid \bullet \mathtt{Spawn}(T); dap) \rightarrow \overline{T''} \circ \overline{CR'} \circ (Cid \bullet dap)} \text{(PAR-TASK-SPAWN)}$$

$$\frac{\overline{T'} = \overline{T} \setminus T \quad dap = Tb(T) \quad \overline{T'} \circ \overline{CR} \rightarrow \overline{T''} \circ \overline{CR'}}{\overline{T} \circ \overline{CR} \circ (Cid \bullet \varepsilon) \rightarrow \overline{T''} \circ \overline{CR'} \circ (Cid \bullet dap; \mathtt{commit})} \text{(PAR-TASK-SCHEDULER)}$$

Fig. 7: Global semantics for cache coherent multicore architectures

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

Intuitively, $[\![rst]\!]_c$ reflects the possible program orders 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 1. *If* $(c \bullet rst) : \varepsilon \rightarrow^* (c \bullet rst') : h$, *then* $\{h;\tau \mid \tau \in [\![rst']\!]_c\} \subseteq [\![rst]\!]_c$.

Corollary 1 (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* $h \in [\![rst]\!]_c$.

Corollary 1 establishes the local program order of the operations of each individual core. Hence, the the model's formalization of the MSI protocol preserves *sequential consistency* [13] in the sense that the result of any execution on the proposed model of multicore architectures is equivalent to the result of executing the operations of all cores in some sequential order. The next lemma captures the absence of data races when accessing a block from main memory.

Lemma 2 (No data races). *Let* Ca_x *denote the cache* $(c_x \bullet M_x \bullet dst_x)$. *The conjunction of the following properties hold for all reachable configurations* $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$ *:*

(a) $\forall n \in dom(M).(status(M,n) = inv \Leftrightarrow \exists\, Ca_i \in \overline{Ca}.\ status(M_i,n) = mo)$
(b) $\forall n \in dom(M).\ (status(M,n) = inv \Leftrightarrow (\exists\, Ca_i \in \overline{Ca}.\ status(M_i,n) = mo)$
$\wedge \forall\, Ca_j \in \overline{Ca} \backslash Ca_i\ .\ (status(M_j,n) = inv \vee n \notin dom(M_j)))$
(c) $\forall n \in dom(M).\ status(M,n) = sh \Leftrightarrow \forall\, Ca_i \in \overline{Ca}.\ status(M_i,n) \neq mo$
(d) $\forall\, Ca_i \in \overline{Ca}\ , \forall n \in dom(M_i).\ (status(M_i,n) = sh \Rightarrow status(M,n) = sh)$

Lemma 2 ensures that there is at most one modified copy of a memory block among the cores. This guarantees single write access and parallel read accesses to memory blocks. The next lemma shows that shared copies of a memory block in different cores always have the same version number. Let function $version(M,n)$ return the version number of block address n in M.

Lemma 3 (Consistent shared copies). *Let* $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$ *be a reachable configuration and assume that* $status(M,n) = sh$. *If* $(c_i \bullet M_i \bullet dst_i) \in \overline{Ca}$ *such that* $status(M_i,n) = sh$ *for any cache, then* $value(M,n) = value(M_i,n)$ *and* $version(M,n) = version(M_i,n)$.

To show that cores in our formal model never access stale values in a memory block, we first 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* $Ca_i = (c_i \bullet M_i \bullet dst_i)$. *Then* $M_i(n)$ *has* the most recent value *if the following holds:*

(a) If $M_i(n) = \langle k, val, sh\rangle$, *then* $M(n) = \langle k, val, sh\rangle$
and $\forall\, (c_j \bullet M_j \bullet dst_j) \in \overline{Ca} \backslash Ca_i.\ status(M_j,n) = sh \Rightarrow M_j(n) = \langle k, val, sh\rangle$.
(b) If $status(M_i,n) = mo$, *then* $status(M,n) = inv$
and $\forall\, (c_j \bullet M_j \bullet dst_j) \in \overline{Ca} \backslash Ca_i.\ status(M_j,n) = inv$.

With Lemma 3 and Definition 2, we can show that if a core succeeds to access a memory block, it will always get the most recent value.

Lemma 4 (No access to stale data). *Let $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$ be a reachable configuration such that $CR_i = (c_i \bullet rst_i) : h_i$ for $CR_i \in \overline{CR}$, $Ca_i = (c_i \bullet M_i \bullet dst_i)$ 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.*

To ensure mutually exclusive access to operations protected by a lock reference r, we first provide the definition of a lock being taken by a core.

Definition 3 (Taken lock). *Let $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$ be a global configuration, $CR_i \in \overline{CR}$, $Ca_i \in \overline{Ca}$ a cache such that $Ca_i = (c_i \bullet M_i \bullet dst_i)$ and $belongs(Ca_i, CR_i)$. Then a lock reference r with address n is considered to be taken by a core if and only if either*

(a) $value(M,n) = 1$ and $status(M,n) = sh$; or
(b) $\exists Ca_i \in \overline{Ca}$ such that $value(M_i,n) = 1$ and
i. $status(M_i,n) = sh$; or
ii. $status(M_i,n) = mo$.

The following lemma shows that once a lock is taken, no other locking step for the same lock is allowed until it has been unlocked.

Lemma 5 (Mutual exclusion). *Let $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$ be a reachable configuration such that $CR_i = (c_i \bullet rst_i) : h_i$ for $CR_i \in \overline{CR}$, $Ca_i = (c_i \bullet M_i \bullet dst_i)$ for $Ca_i \in \overline{Ca}$ and $belongs(Ca_i, CR_i)$. Consider a lock reference r which is taken by a core where $addr(r) = n$. If $(c_i \bullet M_i \bullet dst_i) \circ (c_i \bullet rst_i) : h_i \rightarrow (c_i \bullet M_i' \bullet dst_i') \circ (c_i \bullet rst_i') : h_i'$ where for any rst_i'', $rst_i \neq$ **`unlock`**$(r); rst_i''$ or $rst_i \neq$ **`unlockBl`**$(r); rst_i''$, then*

(a) $n \notin dom(M_i)$; or
(b) $value(M_i,n) = value(M_i',n)$.

Observe that the syntax of our language ensures that no core other than the owner of the lock can release the lock.

5 Proof of Concept Implementation

We have extended the proof of concept implementation[1] of the model presented in previous work [2, 3] with binary locks using the rewriting logic system Maude [7]. The Maude framework allows executable implementation of the operational semantics where transition rules are conditional rewrite rules of the form $[label]: \ t \longrightarrow t'$ if $cond$, which transforms term matching pattern t into corresponding term of pattern t', and as conditional equations of the form $t = t'$ if $cond$, for modelling the instantaneous communication of the label mechanism and the implementation of different auxiliary functions.

[1] The proof of concept extension in Maude and the complete example scenarios can be downloaded from http://folk.uio.no/shijib/Locks.zip.

The main differences and challenges between our operational semantics and the Maude implementation of the model are rather technical: while the former is not explicit with respect to parameters (e.g., the number of cores and caches, the size of caches, cache associativity and replacement policies), the latter requires them to be explicit to define specific parallel architectures, and the number of parallel tasks to be executed and its behaviour to be observed. Another challenge is to implement the true concurrency captured by the label mechanism of operational semantics. Maude allows only interleaving executions. Therefore, one parallel and global step in the semantics will be translated into multiple interleaving steps in the Maude proof of concept implementation.
This proof of concept implementation is complementary to the proposed semantics as it makes the semantics executable, which allow us to simulate and compare different configurations. In this paper, we show an example about how parallel access to shared data triggered by mutually exclusive execution of tasks can influence cache hits.

Example: Observing Task Execution with/without Atomic Sections

```
task T1{(read(r0);write(r0);write(r1);write(r2);write(r3);read(r4);write(r4);write(r5))}
task T2{(lock(r13);read(r0);write(r0);...;write(r3);read(r4);write(r4);write(r5);unlock(r13))}
```

Fig. 8: An example of a data access pattern with and without locks.

Consider a task which has been abstracted into its data access pattern, as `T1` shown in Fig. 8. Let `T2` have the same data access patterns as `T1`, but in addition, `T2` should acquire a lock before starting its execution and release the lock after it finishes. We are going to compare the percentage of cache hits in a configuration with two cores `C1` and `C2` each with a private cache, where both cores are executing either `T1` or `T2` at the same time. In addition, we have three different data layout of main memory to obtain six scenarios. In the first layout (Fig. 9a) the task needs to access different memory blocks for each reference, in the second (Fig. 9b) we group two references together in one block, and in the third (Fig. 9c) we group three references together.

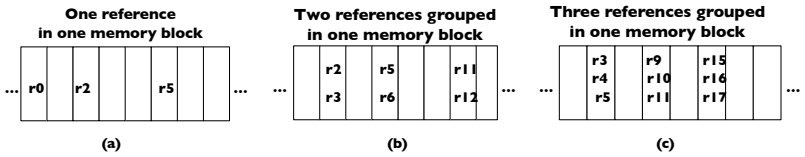

Fig. 9: Different data layouts to be setup in the example.

Figure 10 summarises the results of executing the model in the proof of concept implementation in Maude for these six scenarios. Observe that in the scenarios where cores run the same lock-free task in parallel, the simulations show more mutual invalidations and result in more access to main memory. For the same memory layout, the scenarios with mutually exclusive data access prevent repeated invalidations and therefore

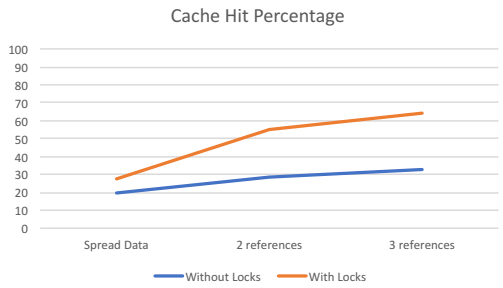

Fig. 10: Hit percentage of six scenarios.

the percentage of cache hit is higher compared to the lock-free task. Observe also that the cache hit increases when the references are closely packed. The reason behind this is that a single fetch will bring references that are used later. Thus, data locality also affects the percentage of cache hit.

6 Related Work

For the analysis of multicore architectures, work typically includes simulation tools and formal techniques. Simulation tools such as gem5 [5] 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 simulation tools such as Graphite [16] and Sniper [6] run programs on distributed clusters to simulate executions on multicore architectures with thousands of cores. Compared to our work, these simulation tools lack formalisation, and the main reason of the Maude implementation in our paper is to show the potential of our formalisation by making the semantics executable. Parallel executions on shared memory under relaxed memory models have been studied, including abstract calculi [8], memory models for programming languages such as Java [11], and machine-level instruction sets for concrete processors such POWER [14] and x86 [19], and for programs executing under total store order (TSO) architectures [20]. This work on weak memory models abstracts from caches, and is as such largely orthogonal to our work that does not consider the reordering of source-level syntax.
Cache coherence protocols have also been analysed and proven for correctness, for instance Maude's model checker has been used to verify the correctness of configurations of the MSI protocols [15]. In contrast, our work, which also considers cache coherent movement of data, focuses on formally capturing the movement of data as a consequence of the interaction between cores, caches and shared memory during the parallel execution of programs, rather than on protocol verification.

7 Conclusions and Future Work

This paper integrates a formal executable model of multicore architectures with binary locks to understand the execution of mutually exclusive tasks from the perspective of

a program instead of hardware. The model consists of an operational semantics for data access patterns executing in parallel on different cores and moving data between shared memory and local caches. The model ensures data consistency by embodying the MSI cache coherence protocol. We have shown that the model guarantees correctness properties concerning data consistency and mutual exclusion. We also provide a proof of concept implementation of the model, and show by example how mutually exclusive execution of tasks can affect data movement.
As for future work, we plan to enrich the data access patterns with data structures and dynamically allocated memory (e.g., object creation). We are also considering the extraction of data access patterns from models in ABS [12]. Other 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.

References

1. S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.
2. S. Bijo, E. B. Johnsen, K. I Pun, and S. L. Tapia Tarifa. A Maude framework for cache coherent multicore architectures. In *Proceedings of the 11th International Workshop on Rewriting Logic and Its Applications (WRLA)*, volume 9942 of *Lecture Notes in Computer Science*, pages 47–63. Springer, 2016.
3. S. Bijo, E. B. Johnsen, K. I Pun, and S. L. Tapia Tarifa. An operational semantics of cache coherent multicore architectures. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC)*, pages 1219–1224. ACM, 2016.
4. S. Bijo, E. B. Johnsen, K. I Pun, and S. L. Tapia Tarifa. A formal model of parallel execution on multicore architectures with multilevel caches. In *Formal Aspects of Component Software - 14th International Conference, FACS 2017, Braga, Portugal, October 10-13, 2017, Proceedings*, volume 10487 of *Lecture Notes in Computer Science*, pages 58–77. Springer, 2017.
5. N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, 2011.
6. T. E. Carlson, W. Heirman, and L. Eeckhout. 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)*, pages 52:1–52:12. ACM, 2011.
7. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
8. K. Crary and M. J. Sullivan. A calculus for relaxed memory. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 623–636. ACM, 2015.
9. D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., 1st edition, 1997.

10. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 5th edition, 2011.
11. R. Jagadeesan, C. Pitcher, and J. Riely. Generative operational semantics for relaxed memory models. In *Proceedings of the 19th European Conference on Programming Languages and Systems (ESOP)*, volume 6012 of *Lecture Notes in Computer Science*, pages 307–326. Springer, 2010.
12. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011.
13. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.
14. S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams. An axiomatic memory model for POWER multiprocessors. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*, volume 7358 of *Lecture Notes in Computer Science*, pages 495–512. Springer, 2012.
15. Ó. Martín, A. Verdejo, and N. Martí-Oliet. Model checking TLR* guarantee formulas on infinite systems. In *Specification, Algebra, and Software – Essays Dedicated to Kokichi Futatsugi*, volume 8373 of *Lecture Notes in Computer Science*, pages 129–150. Springer, 2014.
16. J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12. IEEE Computer Society, 2010.
17. D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/-Software Interface*. Morgan Kaufmann Publishers Inc., 5th edition, 2013.
18. G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.
19. P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.
20. G. Smith, J. Derrick, and B. Dongol. Admit your weakness: Verifying correctness on TSO architectures. In *Proceedings of the 11th International Symposium on Formal Aspects of Component Software (FACS)*, volume 8997 of *Lecture Notes in Computer Science*, pages 364–383. Springer, 2015.
21. Y. Solihin. *Fundamentals of Parallel Multicore Architecture*. Chapman & Hall/CRC, 1st edition, 2015.
22. D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

A Supplementary proofs

A.1 Proof of Lemma 1

Proof. It is trivial for the initial case 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. By induction, we have $(c \bullet rst) : \varepsilon \rightarrow^k (c \bullet rst_1) : \varepsilon; h$ for some h such that $\{h; \tau \mid \tau \in [\![rst_1]\!]_c\} \subseteq [\![rst]\!]_c$. Note that k here refers to k transition steps where $k > 0$.
Next we have to show the lemma still holds for the $k+1$ steps, that is,

$$(c \bullet rst) : \varepsilon \rightarrow^k (c \bullet rst_1) : \varepsilon; h \rightarrow (c \bullet rst_2) : h'$$

for some h'. Note that the $k+1$ th step may be labelled. The proof proceeds by case distinction on the rules for the local transition steps from Figures 5 and 4 for the $k+1$ th step, which is

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

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

$$\begin{array}{l} \{h; \tau \mid \tau \in [\![rst_1]\!]_c\} \subseteq [\![rst]\!]_c \\ = \{h; \tau \mid \tau \in [\![(\mathtt{read}(r); rst_2)]\!]_c\} \subseteq [\![rst]\!]_c \end{array} \tag{1}$$

Then by Definition 1, $[\![(\mathtt{read}(r); rst_2)]\!]_c = \{\tau'; \tau'' | \tau' \in [\![\mathtt{read}(r)]\!]_c, \tau'' \in [\![rst_2]\!]_c\}$, which gives

$$R(c,n); \tau'' \in [\![rst_1]\!]_c \text{ where } R(c,n) \in [\![\mathtt{read}(r)]\!]_c \text{ and } \tau'' \in [\![rst_2]\!]_c \ . \tag{2}$$

Defintion 1 gives $[\![\mathtt{read}(r)]\!]_c = \{R(c,n)\}$, which is a singleton set. Thus this together with equations (1) and (2), we get

$$\begin{array}{l} \{h; R(c,n); \tau'' \mid R(c,n); \tau'' \in [\![(\mathtt{read}(r); rst_2)]\!]_c\} \subseteq [\![rst]\!]_c \\ = \{h; R(c,n); \tau'' \mid \tau'' \in [\![(rst_2)]\!]_c\} \subseteq [\![rst]\!]_c \end{array} \tag{3}$$

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

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

$$\begin{array}{l} \{h; \tau \mid \tau \in [\![rst_1]\!]_c\} \subseteq [\![rst]\!]_c \\ = \{h; \tau \mid \tau \in [\![(\mathtt{read}(r); rst_3)]\!]_c\} \subseteq [\![rst]\!]_c \\ = \{h; \tau \mid \tau \in \{\tau'; \tau'' | \tau' \in [\![\mathtt{read}(r)]\!]_c, \tau'' \in [\![rst_3]\!]_c\}\} \subseteq [\![rst]\!]_c \end{array} \tag{4}$$

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

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

This together with equation (4) gives

$$\begin{array}{l}\{h;\tau \mid \tau \in \{\tau';\tau'' | \tau' \in [\![\mathtt{readBl}(r)]\!]_c, \tau'' \in [\![rst_3]\!]_c\}\} \subseteq [\![rst]\!]_c \\ = \{h;\tau \mid \tau \in [\![(\mathtt{readBl}(r);rst_3)]\!]_c\} \subseteq [\![rst]\!]_c\end{array} \tag{5}$$

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

Case PRWR$_1$: $(c \bullet \mathtt{write}(r);rst_2) : h \to (c \bullet rst_2) : h;W(c,n)$ where $n = addr(r)$.
By the rule PRWR$_1$, $rst_1 = \mathtt{write}(r);rst_2$ and $h' = W(c,n)$. By induction, we have

$$\begin{array}{l}\{h;\tau \mid \tau \in [\![rst_1]\!]_c\} \subseteq [\![rst]\!]_c \\ = \{h;\tau \mid \tau \in [\![(\mathtt{write}(r);rst_2)]\!]_c\} \subseteq [\![rst]\!]_c\end{array} \tag{6}$$

Then by Definition 1, $[\![(\mathtt{write}(r);rst_2)]\!]_c = \{\tau';\tau'' | \tau' \in [\![\mathtt{write}(r)]\!]_c, \tau'' \in [\![rst_2]\!]_c\}$, which gives

$$W(c,n);\tau'' \in [\![rst_1]\!]_c \text{ where } W(c,n) \in [\![\mathtt{write}(r)]\!]_c \text{ and } \tau'' \in [\![rst_2]\!]_c\ . \tag{7}$$

By defintion 1, $[\![\mathtt{write}(r)]\!]_c = \{W(c,n)\}$, which is a singleton set. Thus this together with equations (6) and (7), we get

$$\begin{array}{l}\{h;W(c,n);\tau'' \mid W(c,n);\tau'' \in [\![(\mathtt{write}(r);rst_2)]\!]_c\} \subseteq [\![rst]\!]_c \\ = \{h;W(c,n);\tau'' \mid \tau'' \in [\![(rst_2)]\!]_c\} \subseteq [\![rst]\!]_c\end{array} \tag{8}$$

which concludes the case. It is analogous for the cases of PRWR$_2$, PRWR$_4$, PRWR$_5$, LOCK$_1$, LOCK$_2$, LOCKBLOCK$_2$, UNLOCK$_1$ and UNLOCK$_2$.

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

$$\begin{array}{l}\{h;\tau \mid \tau \in [\![rst_1]\!]_c\} \subseteq [\![rst]\!]_c \\ = \{h;\tau \mid \tau \in [\![(\mathtt{write}(r);rst_3)]\!]_c\} \subseteq [\![rst]\!]_c \\ = \{h;\tau | \tau \in \{\tau';\tau'' | \tau' \in [\![\mathtt{write}(r)]\!]_c, \tau'' \in [\![rst_3]\!]_c\}\} \subseteq [\![rst]\!]_c\end{array} \tag{9}$$

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 (9) gives

$$\begin{array}{l}\{h;\tau \mid \tau \in \{\tau';\tau'' | \tau' \in [\![\mathtt{writeBl}(r)]\!]_c, \tau'' \in [\![rst_3]\!]_c\}\} \subseteq [\![rst]\!]_c \\ = \{h;\tau \mid \tau \in [\![(\mathtt{writeBl}(r);rst_3)]\!]_c\} \subseteq [\![rst]\!]_c\end{array} \tag{10}$$

which concludes the case. The cases of PRWR$_6$, LOCKBLOCK$_1$ and LOCKBLOCK$_3$ can be proven similarly.

Case COMMIT$_1$: $(c \bullet \mathtt{commit}(r);rst_2) : h \to (c \bullet rst_2) : h$.
By the rule COMMIT, $rst_1 = \mathtt{commit}(r);rst_2$ and $h' = h$. By induction, we have

$$\begin{array}{l}\{h;\tau \mid \tau \in [\![rst_1]\!]_c\} \subseteq [\![rst]\!]_c \\ = \{h;\tau \mid \tau \in [\![\mathtt{commit}(r);rst_2]\!]_c\} \subseteq [\![rst]\!]_c\end{array} \tag{11}$$

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 (11) give

$$\begin{array}{l} \{h; \varepsilon; \tau' \mid \varepsilon; \tau' \in [\![\mathtt{commit}(r); rst_2]\!]_c\} \subseteq [\![rst]\!]_c \\ = \{h; \tau' \mid \varepsilon; \tau' \in [\![\mathtt{commit}(r); rst_2]\!]_c\} \subseteq [\![rst]\!]_c \\ = \{h; \tau' \mid \tau' \in [\![rst_2]\!]_c\} \subseteq [\![rst]\!]_c \end{array} \tag{12}$$

which concludes the case. The other cases of commit, including $\textsc{Commit}_2$, $\textsc{CommitAll}_1$ and $\textsc{CommitAll}_2$ work in a similar way.

Case Choice: $(c \bullet dap_1 \sqcap dap_2; rst_3) : h \to (c \bullet dap_1; rst_3) : h$.
By the rule Choice, $rst_1 = dap_1 \sqcap dap_2; rst_3$, $rst_2 = dap_1; rst_3$ and $h' = h$. By induction, we have

$$\begin{array}{l} \{h; \tau \mid \tau \in [\![rst_1]\!]_c\} \subseteq [\![rst]\!]_c \\ = \{h; \tau \mid \tau \in [\![(dap_1 \sqcap dap_2); rst_3]\!]_c\} \subseteq [\![rst]\!]_c \end{array} \tag{13}$$

Then by Definition 1,

$$[\![(dap_1 \sqcap dap_2); rst_3]\!]_c = \{\tau'; \tau'' | \tau' \in [\![dap_1]\!]_c \cup [\![dap_2]\!]_c, \tau'' \in [\![rst_3]\!]_c\} \ , \tag{14}$$

which is a superset of

$$[\![dap_1; rst_3]\!]_c = \{\tau'''; \tau'' | \tau''' \in [\![dap_1]\!]_c, \tau'' \in [\![rst_3]\!]_c\} \ . \tag{15}$$

Equations (14) and (15) gives

$$\{h; \tau'''; \tau'' \mid \tau'''; \tau'' \in [\![dap_1; rst_3]\!]_c\} \subseteq \{h; \tau \mid \tau \in [\![(dap_1 \sqcap dap_2); rst_3]\!]_c\} \ . \tag{16}$$

Thus, equations (13) and (16), together with transitivity, implies

$$\{h; \tau'''; \tau'' \mid \tau'''; \tau'' \in [\![dap_1; rst_3]\!]_c\} \subseteq [\![rst]\!]_c \tag{17}$$

which concludes the case.

Case Repetition: $(c \bullet dap^*; rst_3) : h \to (c \bullet (dap; dap^*) \sqcap \mathtt{skip}; rst_3) : h$.
By the rule Repetiotion, $rst_1 = dap^*; rst_3$, $rst_2 = (dap; dap^*) \sqcap \mathtt{skip}; rst_3$ and $h' = h$. By induction and Definition 1, we have

$$\begin{array}{l} \{h; \tau \mid \tau \in [\![rst_1]\!]_c\} \subseteq [\![rst]\!]_c \\ = \{h; \tau \mid \tau \in [\![dap^*; rst_3]\!]_c\} \subseteq [\![rst]\!]_c \\ = \{h; \tau \mid \tau \in \{\tau'; \tau'' | \tau' \in [\![dap^*]\!]_c, \tau'' \in [\![rst_3]\!]_c\}\} \subseteq [\![rst]\!]_c \end{array} \tag{18}$$

By Definition 1,

$$\begin{array}{rl} [\![dap^*]\!]_c = & [\![dap; dap^*]\!]_c \cup [\![\mathtt{skip}]\!]_c \\ = & [\![(dap; dap^*) \sqcap \mathtt{skip}]\!]_c \end{array} \tag{19}$$

which implies

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

This together with equation (18) gives

$$\begin{array}{l} \{h; \tau \mid \tau \in \{\tau'; \tau'' | \tau' \in [\![(dap; dap^*) \sqcap \mathtt{skip}]\!]_c, \tau'' \in [\![rst_3]\!]_c\}\} \subseteq [\![rst]\!]_c \\ = \{h; \tau \mid \tau \in [\![(dap; dap^*) \sqcap \mathtt{skip}; rst_3]\!]_c\} \subseteq [\![rst]\!]_c \end{array} \tag{20}$$

which concludes the case. □

A.2 Proof of Corollary 1

Proof. Since h_2 is the sequence of events produced by the transition step(s) from *rst* to *rst'*, we can then get $\{h_2;\tau \mid \tau \in [\![rst']\!]_c\} \subseteq [\![rst]\!]_c$ by Lemma 1. Thus, $h_2 \preceq h$ for some $h \in [\![rst]\!]_c$. □

A.3 Proof of Lemma 2

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 are going to show the preservation of the invariant over all transition steps:

$$M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H \xrightarrow{S} M' \circ \overline{T}' \circ \overline{Ca}' \circ \overline{CR}' : H' \tag{21}$$

where S is a set of sending messages. Remember that S contains 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. By induction, the configuration $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$ satisfies the lemma.
We first consider the case where $S = \emptyset$. Let $CR_i \in \overline{CR}$ where $CR_i = (Cid \bullet rst) : h_i$ and $Ca_i \in \overline{Ca}$ where Ca_i is the cache belonging to CR_i. By rule TOP-ASYNCH in Figure 7, there are three possibilities of a reduction step when $S = \emptyset$: it can be (1) an internal transition between a core CR_i and its local cache Ca_i (cf. rule PAR-INTERNAL-STEPS); or (2) 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 (3) a global step for the communication between Ca_i and the main memory (cf. rule PAR-MEMORY-ACCESS).
For case (1), after the decomposition with rule PAR-INTERNAL-STEPS, the relevant internal transitions between the core CR_i and its local cache Ca_i are those un-labelled transitions in Figures 5 and 4. Those steps do not have any effect on the status of any memory block or main memory. Therefore by induction, the invariant still holds after the transition.
Case (2) holds immediately because the transition steps do not change the status of any memory block.
For case (3), 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 7.

Case FLUSH$_1$:
$M \circ (c_i \bullet M_i' \bullet \texttt{flush}(n); dst_i) \rightarrow$
$M[n \mapsto \langle k', val', sh\rangle] \circ (c_i \bullet M_i'[n \mapsto \langle k', val', sh\rangle] \bullet dst_i)$
We are further given $M'(n) = \langle k, val', mo\rangle$ and $k' = k+1$. Since by induction, the lemma holds for the configuration before the transition step, by part (a) of the invariant from the induction gives $status(M,n) = inv$, and part (b) gives $\forall\, Ca_j \in \overline{Ca} \backslash Ca_i\ .\ (status(M_j,n) = inv \ \vee n \notin dom(M_j))$ before the transition. Since the main memory M and the cache M_i are updated to $M'[n \mapsto \langle k+1, val', sh\rangle]$ and $M_i'[n \mapsto \langle k', val', sh\rangle]$ after the step, which

satisfies part (c) and part (d) of the lemma, and therefore concludes the case. The case of FLUSH$_2$ trivially holds.

Case FETCH$_1$:
$M \circ (c_i \bullet M_i \bullet \texttt{fetch}(n); dst_i) \rightarrow M' \circ (c_i \bullet M_i'[n' \mapsto \bot, n \mapsto \langle k, val, sh\rangle] \bullet dst_i)$
By induction, the invariant holds for the configuration before the step, and therefore $status(M,n) = sh$ implies by part (c) of the invariant that $\forall\ Ca_j \in \overline{Ca}.\ status(M_j,n) \neq mo$ before the step. Since after the step $status(M',n) = status(M_i',n) = sh$, the invariant is maintained. It is analogous for the case 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.
Now we have to consider the set of sent messages S for a transition step in Equation (21) is *not empty*. The only applicable rules for the case where $S \neq \emptyset$ is TOP-SYNCH$_1$ and TOP-SYNCH$_2$ in Figure 7, where updates will be done to the main memory and to each cache, respectively.
Consider a block address n and assume $Ca_i \circ CR_i$ sends a message W for (exclusively) reading n in the following. Let us assume $S = S' \cup \{W\}$, where W can be either (i) $!Rd(n)$ or (ii) $!RdX(n)$. Note that there is at most one message for each block address n in S. Because of this property, and to keep the proof clear, we further assume that $S' = \emptyset$; that is, no other core sends any message apart from Ca_i. The proof is applicable to all other caches in parallel which send messages irrelevant to address n. We further simplify the proof for this case by omitting both the global and local histories in the configurations because they do not affect the correctness of the proof.

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 TOP-SYNCH$_1$ in Figure 7, we are given that $R = dual(S)$. In this case, we have $R = \{?Rd(n)\}$.
The premise $M \xrightarrow{\{?Rd(n)\}} M'$ corresponds to the transition for the main memory. By MAIN-MEMORY$_1$ and MAIN-MEMORY-IGNORE in Figure 6, we have $M \xrightarrow{?Rd(n)} M'$, where $M' = M$.
Rule TOP-SYNCH$_1$ 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 TOP-SYNCH$_2$. We then get

$$Ca_i \circ CR_i \xrightarrow{\{!Rd(n)\}} Ca_i' \circ CR_i' \tag{22}$$

where $belongs(Ca_i, CR_i)$, and

$$Ca_j \circ CR_j \xrightarrow{\{?Rd(n)\}} Ca_j' \circ CR_j' \quad \text{for all} \quad CR_j \in \overline{CR} \backslash CR_i \tag{23}$$

and $belongs(Ca_j, CR_j)$. Consider equation (22), by rule RECEIVE-SEND-MESSAGE in Figure 6, we get $Ca_i \circ CR_i \xrightarrow{!Rd(n)} Ca_i' \circ CR_i'$. Consider the relevant rules, namely, PRRD$_2$, PRRD$_4$, PRWR$_3$ and PRWR$_5$ in Figure 4, as well as LOCKBLOCK$_1$ and LOCKBLOCK$_3$ in Figure 5. The address n is either not in the local cache or its status is *inv* before the step, and is removed from the local cache in all these rules after the transition.

Consider the step in equation (23), rules RECEIVE-SEND-MESSAGE and RECEIVE-MESSAGE in Figure 6 give $Ca_j \xrightarrow{?Rd(n)} Ca'_j$. The relevant rules, FLUSH-ONE-LINE and IGNORE-FLUSH in Figure 6, do not affect the status of the address 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 TOP-SYNCH$_1$ in Figure 7, we are given that $R = dual(S)$. In this case, we have $R = \{?RdX(n)\}$.
The premise $M \xrightarrow{\{?RdX(n)\}} M'$ correspond to the transition in the main memory. Then, by rules MAIN-MEMORY$_1$ and MAIN-MEMORY-INVALIDATE in Figure 6, we have $M \xrightarrow{?RdX(n)} M'$, where $M' = M[n \mapsto \langle k, val, inv \rangle]$.
Rule TOP-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 TOP-SYNCH$_2$. We then get

$$Ca_i \circ CR_i \xrightarrow{\{!RdX(n)\}} Ca'_i \circ CR'_i \tag{24}$$

where $belongs(Ca_i, CR_i)$, and

$$Ca_j \circ CR_j \xrightarrow{\{?Rd(n)\}} Ca'_j \circ CR'_j \quad \text{for all} \quad CR_j \in \overline{CR} \backslash CR_i \tag{25}$$

and $belongs(Ca_j, CR_j)$. Consider equation (24), RECEIVE-SEND-MESSAGE in Figure 6, we get $Ca_i \circ CR_i \xrightarrow{!Rd(n)} Ca'_i \circ CR'_i$. The relevant rules for this transition includes PRWR$_2$ and PRWR$_4$ in Figure 4, as well as LOCK$_2$, LOCKBLOCK$_2$ and UNLOCK$_2$ in Figure 5. We are further given in these rules that the status of address n is *sh* before the step, which is updated to *mo* after the step.
For equation (25), rules RECEIVE-SEND-MESSAGE and RECEIVE-MESSAGE in Figure 6 give $Ca_j \xrightarrow{?RdX(n)} Ca'_j$. Consider the relevant rules, INVALIDATE-ONE- LINE and IGNORE-INVALIDATE in Figure 6. In the configuration after the step for both rules, the block address n either is in an invalid state or does not exist in the cache. This together with, after the transition, $status(M', n) = inv$ and status of address n is *mo* in Ca_i which is the local cache in CR_i, implies parts (a) and (b) of the lemma, and therefore concludes the case. □

A.4 Proof of Lemma 3

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. For the transition rules which are either local in a core and its cache (Figures 4 and 5), or local in the main memory or the cache (Figure 6), the steps do not affect the version number. Regarding the values, these rules have either no effect, or the values in cache memory are changed and the status is

updated to *mo* (see e.g., PRWR_2 in Figure 4 and LOCK_2 in Figure 5). Thus, the lemma holds after the transition steps for these cases.

We consider in the following the cases where transitions are those communication steps between main memory and a cache $Ca_i \in \overline{Ca}$. The rules relevant to such communication steps are those for fetching/flushing a memory block from/to the main memory by a cache in Figure 7. Let $Ca_i = c_i \bullet M_i \bullet dst_i$ be the cache, and n be the block address.

Case FLUSH_1:
$M \circ (c_i \bullet M_i \bullet \texttt{flush}(n); dst_i') \rightarrow M[n \mapsto \langle k', val, sh\rangle] \circ (c_i \bullet M_i[n \mapsto \langle k', val, sh\rangle] \bullet dst_i')$
where $k' = k+1$. We are further given $M_i(n) = \langle k, val, mo\rangle$ which implies $status(M,n) = inv$ by Lemma 2(a). Then, by Lemma 2(b), we have $\forall Ca_j \in \overline{Ca} \backslash Ca_i.(status(M_j,n) = inv \ \vee n \notin dom(M_j))$ where $Ca_j = (c_j \bullet M_j \bullet dst_j)$, which is maintained after the transition. Together with $M[n \mapsto \langle k', val, sh\rangle]$ and $M_i[n \mapsto \langle k', val, sh\rangle]$ where $k' = k+1$ after the step, we conclude the case. The case for FLUSH_2 holds immediately.

Case FETCH_1:
$M \circ (c_i \bullet M_i \bullet \texttt{fetch}(n); dst_i') \rightarrow M' \circ (c_i \bullet M_i[n \mapsto \langle k, val, sh\rangle] \bullet dst_i')$
We are further given $M(n) = \langle k, val, sh\rangle$, i.e., $status(M,n) = sh$. By Lemma 2(c), $status(M,n) = sh$ implies $status(M_j,n) \neq mo$ for all $Ca_j \in \overline{Ca}$ where $Ca_j = (c_j \bullet M_j \bullet dst_j)$. We just have to consider those caches $Ca_g \in \overline{Ca}$ where $Ca_g = c_g \bullet M_g \bullet dst_g$ and $status(M_g,n) = sh$. (Note that the cases where $status(M_j,n) = inv$ or $n \notin dom(M_j)$ are irrelevant in this lemma.) By induction, $version(M,n) = version(M_g,n) = k$ and $value(M,n) = value(M_g,n)$. While keeping the content of address n in the main memory and in all other caches unchanged, FETCH_1 gives after the step $M_i[n \mapsto \langle k, val, sh\rangle]$ for the cache Ca_i. Thus, the configuration after the transition satisfies the invariant. It is analogous for rule FETCH_2. The cases FETCH_3 holds immediately because the transition step does not change the status or version of any memory location in either the main memory. □

A.5 Proof of Lemma 4

Proof. For the core CR_i to make *successful* read or write accesses to block address n in its local cache (i.e., to generate an event $R(c_i,n)$ or $W(c_i,n)$), CR_i applies the rules which capture the interactions between the core and its local cache, including PRRD_1, PRRD_3, PRWR_1, PRWR_2 and PRWR_4 in Figure 4, and LOCK_1, LOCK_2, LOCKBLOCK_2, UNLOCK_1 and UNLOCK_2 in Figure 5. The proof therefore proceeds by cases for these rules.

Consider, for example, the rule PRRD_1, where the status of n is either *mo* or *sh*. If $status(M_i,n) = mo$, it follows from Lemma 2 (a) and (b) that $M_i(n)$ has the most recent copy according to Definition 2 (b). If $M_i(n) = \langle k, val, sh\rangle$, it follows from Lemma 2 (d) that $status(M,n) = sh$, and consequently from Lemma 2 (c) that $\forall Ca_j \in \overline{Ca}.\ status(M_j,n) \neq mo$ where $Ca_j = (c_j \bullet M_j \bullet dst_j)$. Then we need to consider all caches $c_g \bullet M_g \bullet dst_g \in \overline{Ca}$ where $status(M_g,n) = sh$. From Lemma 3, we get $version(M_i,n) = k = version(M,n) = version(M_g,n)$ and $value(M_i,n) = k = value(M,n) = value(M_g,n)$, which satisfies Definition 2 (a). This concludes the case. The rest of rules mentioned above can be proven analogously. □

A.6 Proof of Lemma 5

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 *sh*, that is, no lock has been taken. For a core *CR* to change the value of a lock residing in its own cache, *CR* applies the rules which allow the core to take the lock, including LOCK_1, LOCK_2, LOCKBLOCK_1, LOCKBLOCK_2 and LOCKBLOCK_3 in Figure 5, while the lemma trivially holds for all other local semantics that do not manipulate locking steps. We do not consider the unlocking rules because they are irrelevant to this lemma.
By Definition 3, there are two cases where a lock is considered to be taken. For the case (a), by Lemma 2(c), $\forall\ Ca_j \in \overline{Ca}.\ status(M_j,n) \neq mo$, and thus rule LOCK_1 is not applicable. Then, by Lemma 3, $\forall Ca_k \in \overline{Ca}$ where $status(M_k,n) = sh$ and $belongs(Ca_k,CR_k)$ such that $value(M,n) = 1 = value(M_k,n)$, rules LOCK_2 and LOCKBLOCK_2 require the value of the lock to be *zero* to make the lock step and thus irrelevant. Finally, $\forall Ca_l \in \overline{Ca}$ and $belongs(Ca_l,CR_l)$ such that $status(M,n) = inv$ or $n \notin dom(M_l)$, rules LOCKBLOCK_1 and LOCKBLCOK_3 remove the lock with address *n* from the cache and have no effect on the value of the lock *r* for the transition step, which satisfies part (a) of the lemma.
For the case (b) of Definition 3, we are given that $value(M_i,n) = 1$ for some $Ca_i \in \overline{Ca}$ and $belongs(Ca_i,CR_i)$ where $CR_i \in \overline{CR}$. If $status(M_i,n) = sh$, then by Lemma 2(d), we get $status(M,n) = sh$ and by Lemma 3, $value(M,n) = value(M_i,n) = 1$, which follows similarly as case (a). If $status(M_i,n) = mo$, then by Lemma 2(a) and (b), $\forall Ca_j \in \overline{Ca} \backslash Ca_i.(status(M_j,n) = inv\ \vee n \notin dom(M_j))$. Rule LOCK_1 is not applicable to CR_i as it requires the value of the lock to be *zero*. Rules LOCK_2 and LOCKBLOCK_2 are irrelevant for all cores, while rules LOCKBLOCK_1 and LOCKBLOCK_3 can be applied to CR_j where $belongs(Ca_j,CR_j)$ where the block *n* is removed from the cache and thus satisfies part (a) of the lemma. □