UNIVERSITY OF OSLO
Department of Informatics

An Operational Semantics of Cache Coherent Multicore Architectures

Research Report 449

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

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

December 2015
An Operational Semantics of
Cache Coherent Multicore Architectures

Shiji Bijo, Einar Broch Johnsen, Ka I Pun, and S. Lizeth Tapia Tarifa
University of Oslo, Norway
{shijib, einarj, violet, sltarifa}@ifi.uio.no

ABSTRACT
This paper presents a formal semantics of multicore architectures with private cache, shared memory, and instantaneous inter-core communications. The purpose of the semantics is to provide an operational understanding of how low-level read and write operations interact with caches and main memory. The semantics is based on an abstract model of cache coherence and allows formal reasoning over parallel programs that execute on any given number of cores. We prove correctness properties expressed as invariants for the preservation of program order, data-race free execution of low-level operations, and no access to stale data. A prototype implementation is provided as a proof of concept interpreter for the formal model.

Keywords
Formal semantics, multicore architectures, memory consistency, cache coherence, correctness properties, observable behaviour.

1. INTRODUCTION
Multicore architectures dominate today’s hardware design. In these architectures, cache memory is used to accelerate program execution by providing quick access to recently used data, but allowing multiple copies of data to co-exist during execution. Cache coherence protocols ensure that cores do not access stale data. With the dominating position of multicore architectures, system developers can benefit from a better understanding and ability to reason about interactions between programs, caches and main memory. For this purpose we need clear and precise operational models which allow us to reason about such interactions.

In this paper, we propose a formalization of an abstract model of cache coherent multicore architectures, directly connecting the parallel execution of programs on different cores to the movement of data between caches and main memory. Similar to formal semantics for programming languages, we develop an operational semantics of parallel computations on cache coherent multicore architectures. Our purpose is not to evaluate the specifics of a concrete cache coherence protocol, but rather to capture program execution on shared data at locations with coherent caches in a formal way. Consequently, we integrate the basic MSI protocol directly into the operational semantics of our formal model, while abstracting from the concrete communication medium (which could be, e.g., a bus or a ring), and from the specifics of cache associativity and replacement policies. We show that this abstract model of cache coherent architectures guarantees desirable properties for the programmer such as program order, absence of data races, and that cores always access the most recent value of data. The technical contributions of this paper are (1) a formal, operational model of executions on cache coherent multicore architectures and (2) correctness properties for the formal model expressed as invariants over any given number of cores.

Related work. Approaches to the analysis of multicore architectures include on the one hand simulators for efficiency and on the other hand formal techniques for proving the correctness of specific cache coherence protocols. We are not aware of work on abstract models of execution on cache coherent multicore architectures and their formalization, as presented in this paper.

Simulation tools allow cache coherence protocols to be specified to evaluate their performance on different architectures (e.g., gems [17] and gem5 [1]). These tools run benchmark programs written as low-level read and write instructions to memory and perform measurements, e.g., the cache hit/miss ratio. Advanced simulators such as Graphite [18] and Sniper [3] can handle multicore architectures with thousands of cores by running on distributed clusters. A framework, proposed in [15], statically estimates the worst-case response times for concurrent applications running on multiple cores with shared cache.

Both operational and axiomatic formal models have been used to describe the effect of parallel executions on shared memory under relaxed memory models, including abstract calculi [5], memory models for programming languages such as Java [13], and machine-level instruction sets for concrete processors such as POWER [16, 22] and x86 [23]. The behavior of programs executing under total store order (TSO) architectures is studied in [10, 24]. However, work on weak memory models abstracts from caches, and is as such largely orthogonal to our work that does not consider reordering of source-level syntax. Cache coherence protocols can be formally specified as automata and verified by (parametrized) model checking (e.g., [7, 11, 19, 21]), or in terms of operational formalizations which abstract from the specific number of cores to prove the correctness of the protocols (e.g., [8, 9, 25]). In contrast to these approaches, our model allows the explicit representation of programs executing on caches. In this sense, our approach is more similar to the unformalized work on simulation tools discussed above.

Paper overview. Sect. 2 briefly reviews background concepts on multicore architectures, Sect. 3 presents our abstract model of cache coherent multicore architectures, Sect. 4 details the operational semantics for this model, and Sect. 5 the associated correctness properties. Sect. 6 discusses the prototype implementation, and Sect. 7 concludes the paper.
2. MULTICORE ARCHITECTURES

Modern multicore architectures consist of components such as independent processing units or cores, small and fast memory units or caches associated to one or more cores, and main memory. Cores execute program instructions and interact with main memory to load and store data. Cores use caches to speed up their execution. The current market offers different designs for integrating these components. Cache memory keeps the most recently used multiple continuous words in main memory form a block, which has a unique memory address.

An attempt to access data from the cache is called a hit if the data is found in the cache and a miss otherwise. In the case of a miss, the block containing the requested data must be fetched from a lower level in the memory hierarchy (e.g., main memory). Since caches are small compared to main memory, a fetch instruction may require the eviction of an existing cache line. In this case, the selection of which cache line to evict depends on how the cache lines are organized, the so-called cache associativity, and on the replacement policy. In k-way set associative caches, the caches are grouped as sets with k cache lines and the memory block can go anywhere in a particular set. For direct mapped caches, associativity is one and the cache is organized in single-line groups. In fully associative caches, the entire cache is considered as a single set and memory blocks can be placed anywhere in the cache. Replacement policies determine the line to evict from a full cache set when a new block is fetched into that set. Typical policies are random, FIFO, and LRU (Least Recently Used).

Multicore architectures use cache coherence protocols to keep the data stored in different local caches and in main memory consistent. In particular, invalidation-based protocols are characterized by broadcasting invalidation messages when a particular core requires write access to a memory address. Examples of invalidation protocols are MSI and its extensions (e.g., MESI and MOESI). An invalidation-based coherence protocol integrates a finite state controller in each core, and connects the cores and memory using a broadcast medium (a bus, ring, or other topology). The controller responds to requests from its core and from other cores via the medium. In the MSI protocol, a cache line can be in one of the three states: modified, shared, invalid. For a line in a cache, a modified state indicates that it is the most updated copy, and that all other copies in the system are invalid, while a shared state indicates that the copy is shared among one or more caches, and the main memory and that all copies are consistent. When a core attempts to access a line which is either invalid or does not exist in the cache, i.e., a cache miss, it will broadcast a read request. Upon receiving this message, the core which has a modified copy of the requested cache line will flush it to the main memory and change the state of the cache line to shared in both the core and the main memory. For write operations, the cache line must be in either shared or modified state. An attempt to write to a cache line in shared state will broadcast an invalidation message to the other cores and the main memory. The state of the cache line will be updated to modified if the attempt succeeds. Upon the receipt of an invalidation message, a core will invalidate its copy only if the state is shared. For more details on variations of multicore architectures, coherence protocols, and memory consistency, the reader may consult, e.g., [6, 12, 20].

3. THE ABSTRACT MODEL

This section describes our abstract model of execution on architectures with shared memory, inter-core communications, and where cores have a private one-level cache. Figure 1 depicts one such architecture. The cores in our model execute low-level statements, given as tasks and scheduled by a task queue, reflecting the read and write operations of a program. These statements interact with local caches and may trigger the movement of data between the caches and main memory, reflected as fetch and flush data instructions. The exchange of messages between caches and main memory is captured by an abstract communication medium, abstracting from different concrete topologies such as bus, ring, or mesh. Communication in this medium appears to be instantaneous and is captured by labels. If a core needs to access a block of memory with address n, which is not available with the right permissions in its local cache, it will broadcast a !Rd(n) or ?RdX(n) message to all other components in the configuration to obtain read or read exclusive permissions to n, respectively, and it will proceed to fetch the data, if needed. Observe that a read exclusive message will invalidate all other copies of that memory block in other caches, so the sender can perform a write operation. Consequently, the consistency of copies of data in different caches in this abstract model will be maintained by an abstracted version of the basic coherence protocol MSI. Technically, we let synchronization of dual labels on parallel transitions capture the instantaneous exchange of messages, as common in process algebra. This mechanism is used to model the abstract communication medium; a component which sends a message generates a label and the other components will instantaneously receive the dual labels ?Rd(n) and !RdX(n) that the medium automatically generates (e.g., as in Figure 1). For simplicity, the data contained in memory blocks and cache lines is ignored, a cache line has the same size as a memory block, and data does not move from one cache to another directly, but indirectly via the main memory. The model guarantees sequential consistency [14].

Figure 2 contains the syntax of the runtime structure in the model. The input language consists of tasks with source-level statements sst. These statements are PrRd(r) for reading from a memory reference r, PrWr(r) for writing to r and commit(r) for flushing r. Cores execute runtime statements sst, including sst, by interacting with the local cache. The extra statements are explained as follows. A core may be blocked during the execution due to a cache miss; PrRdBl(r) and PrWrBl(r) represent the corresponding waiting states while data is being fetched from main memory, and commit forces the flushing of all modified data contained in a cache into

Figure 1: Abstract model of multi-core architecture (illustration).
4. OPERATIONAL SEMANTICS

We develop a structural operational semantics (SOS) for our abstract model of cache coherent multicore architectures. In an initial configuration, all memory blocks in the main memory MM have status sh and version number 0, the task queue Qu contains a set of tasks written in the source-language syntax, each core in CR has an empty cache, and no data instructions as well as no runtime events. Executions start from an initial configuration by applying global transition rules, which in turn apply local transition rules. Let Config → Config′ denote an execution starting from Config and reaching configuration Config′ by applying zero or more global transition rules, in which case we call Config′ reachable.

Global steps capture the abstract communication medium with interactions to flush and fetch data to and from main memory, schedule tasks and follow a global protocol to guarantee data consistency. The communication medium, using labels for instantaneous communication, allows many cores to request and access different memory blocks in parallel. Therefore, there may in general be many interactions occurring at the same time and synchronization of labels on transitions is over a possibly empty sets of labels. We formally define the syntax for the label mechanism as follows:

\[
\begin{align*}
W & ::= \top \mid \bot \mid \text{?Rd(n)} \mid \text{?RdX(n)} \\
S & ::= :\emptyset \mid \{W\} \mid S \cup S
\end{align*}
\]

where S contains at most one label per block address n. Sect. 4.1 details the global rules.

Local steps capture the local transitions in main memory, the local executions of statements in each core and the local actions derived from the global protocol to keep local data coherent with respect to the other components. Sect. 4.2 details the local rules.

4.1 Global Transition Rules

The global steps of the operational semantics are given in Figure 3. These transition rules describe the interactions and communications between the different components in the configuration, and ensure data synchronization between cores and main memory. Rule TOP-SYNCH captures the global synchronization for handling a non-empty set S of labels corresponding to broadcast messages. In this rule, R is the set of receiving labels dual to S. The configuration is updated in two steps: the main memory must accept the set R and the cores must accept the set S.

Rule CORE-COMMUNICATION recursively decomposes the label set S into sets of sending and receiving labels distributed over the cores CR, such that each set eventually contains at most one W label. Each set of cores must accept the associated set of labels in the premises of the rule. The decomposition ensures that only receiving messages are shared between the transitions. The rule ensures that a core which does not send a message W will receive the dual message Q. The decomposition also applies to the global history which projects to the sets CR1 and CR2, respectively. If the transitions generate events, these are merged into a set of events which extends the global history H.

The rule TOP-ASYNCH captures parallel transitions in different components when the set of labels is empty. These transitions can be local to individual cores, parallel memory accesses or scheduling of new tasks. There are four cases: CR performs local transitions without labels, CR3 access main memory, CR4 get new tasks from the task queue, and CR5 are idle. The decomposition for local transitions, memory accesses, and scheduling of new tasks is respectively handled by the rules PAR-INTERNAL-STEPS, PAR-MEMORY-ACCESS, and PAR-TASK-SCHEDULER. Let the predicate disjoint(CR1, CR2, CR3, CR4) express that the sets of cores involved in the parallel transitions are disjoint to each other.

Data transfer between a cache and main memory is described in rules of the form MM(MM); (MLoc) → MM(MM'); (MLoc'). They capture the execution of data instructions fetch and flush. Here the function select(M, ~); n, used in the rules for fetching a data block with address n, returns the address of the cache line that needs to be evicted to give space to the data block being fetched. If no eviction is needed, the select function returns n (cf. rule FETCH1). Rule FETCH2 describes the case where we need to evict a non-modified cache line m; rule FETCH3 refers to the case where the
cache line \( m \) to be evicted has status \( m0 \), so it needs to be flushed before cache line \( n \) can be loaded. Rules \textsc{Fetch1} and \textsc{Fetch2} check that the cache line has status \( sh \) in main memory, otherwise the instruction is blocked until the data is flushed from another cache.

Rule \textsc{Flush1} stores a cache line in main memory, incrementing the version number and setting its status to \( sh \) both in the cache and main memory. Rule \textsc{Flush2} discards the \textit{flush}(\textit{n}) instruction if the cache line is no longer modified (or has been evicted).

### 4.2 Local Transition Rules

The rules in this section capture local transitions in either the cores or the main memory, and are given in Figure 4. Local rules in the cores reflect the statements being executed and the local finite state controller enforcing the MSI protocol. Let \textit{addr}(\textit{r})\textit{ denote the block address that contains the reference \( r \), and status(\textit{M};\textit{n}) the status of cache line \( n \) in the map \( \textit{M} \).

In the main memory, the controller sets the status of a block to \textit{inv} in rule \textsc{One-LINE-Main-Memory}; if exclusive access has been requested. Rule \textsc{One-LINE-Main-Memory2} will always accept a shared read request. Rules \textsc{Main-Memory1} and \textsc{Main-Memory2} are distribution rules for sets of labels.

Label sets are decomposed by the \textsc{Send-Receive-MESSAGE} (which has only one \textit{W} label), \textsc{Receive-Empty}, and \textsc{Receive-MESSAGE} rules in order to feed the finite state controller. A core can only receive an exclusive request \textit{\&RdX}(\textit{n}) for a cache line that is not modified. Rule \textsc{INVALIDATE-ONE-LINE} sets the status of cache line \( n \) to \textit{in} if the cache line has status \( sh \) when the core receives a \textit{\&RdX}(\textit{n}) message. Rule \textsc{IGNORE-INVALIDATE-ONE-LINE} ignores any read exclusive message for an invalid cache line, or for a block which is not in the cache. For messages \textit{\&Rd}(\textit{n}), if the cache line \( n \) has status \( m0 \), rule \textsc{Flush-ONE-LINE} adds a \textit{flush} to the header of the data instructions \( d \) (to avoid deadlock), otherwise rule \textsc{Ignore-Flush-ONE-LINE} ignores the message.

Read statements succeed if the cache line containing the requested reference \( r \) is available in the cache, applying rule \textsc{PrRd1}. Otherwise, a \textit{fetch}(\textit{n}) instruction is added to the tail of the data instructions \( d \) in rule \textsc{PrRd2}. In this case, execution is blocked by the statement \textsc{PrRdBlock1}(\textit{r}). Execution may proceed once the block \( n \) has been copied into the cache, captured by rule \textsc{PrRdBlock1}. In the parallel setting, the cache line may get invalidated while the core is still blocked after \textit{fetch}. Rule \textsc{PrRdBlock2} captures this situation and broadcasts the \textit{\&Rd}(\textit{n}) message again.

Rule \textsc{PrWr1} expresses that a write statement \textit{PrWrite}(\textit{r}) succeeds when the memory block has \( m0 \) status in cache memory. If the cache line is shared, the core needs to get exclusive access, captured by rule \textsc{PrWr2}. If the cache line is invalid (or the block is not in the cache, the core first needs to request the cache line in rule \textsc{PrWr3}. Similar to the case for read, we use a statement \textsc{PrWrite1}(\textit{r}) and the rule \textsc{PrWriteBlock1} to block repeated read requests. Once the cache line has status \( sh \), rule \textsc{PrWriteBlock2} requests exclusive access, as in rule \textsc{PrWr2}.

The statements \textit{commit}(\textit{r}) and \textit{commit} are respectively used to force flushing of a single modified cache line and of the entire cache. Rules \textsc{Commit1} and \textsc{Commit2} capture the single cache line for modified and non-modified cache lines, respectively. Rules \textsc{Commit-All1} and \textsc{Commit-All2} reduce a \textit{commit} statement to a sequence of \textit{flush}-instructions. In order to ensure data consistency among main memory and individual caches, the final statement in a task should be \textit{commit} (see rule \textsc{Par-Task-Scheduler} in Figure 3), in this way all modified data in the cache will be flushed before another task is assigned to the core.

### 5. Correctness

We consider correctness properties for the proposed model, including the preservation of program order in core requests, the absence of data races, and successful accesses to memory locations always retrieve the most recent value\textsuperscript{1}. We first define a function which translates statements into event histories:

**Definition 1.** Let \( addr(r) = n \). Define \textit{rst} \( r \textit{c} \) inductively over \textit{rst}:

\[
\begin{align*}
    (PrRd(r); \textit{rst}) &\text{ c } \leftarrow \textit{R}(C; n); \textit{rst} \textit{c} \\
    (PrRdBlock1(r); \textit{rst}) &\text{ c } \leftarrow \textit{R}(C; n); \textit{rst} \textit{c} \\
    (PrWrite(r); \textit{rst}) &\text{ c } \leftarrow \textit{W}(C; n); \textit{rst} \textit{c} \\
    (commit(r); \textit{rst}) &\text{ c } \leftarrow \textit{rst} \textit{c} \\
    \textit{commit} &\text{ c } \leftarrow \textit{e} \\
\end{align*}
\]

\textsuperscript{1}The proofs of the lemmas in this section can be found in Appendix A.
Intuitively, $rst \triangleq_C$ reflects the expected program order of read and write accesses when executing $rst$ directly on main memory. Note that $h; h = h$. We show that execution with local cache preserves this program order:

**LEMMA 1 (PROGRAM ORDER).**
If $C(M, \sim \vdash rst) \models h$, then $h; h \triangleq_C r st = rst \triangleq_C$.

The next lemma states properties about data races when accessing a memory block from main memory.

**LEMMA 2 (NO DATA RACES).** The following properties hold for all reachable configurations $MM(M) \triangleright_{\Pi} (Q) \triangleright_{\Pi} MM(M)$:

\begin{enumerate}
  \item \hspace{1cm} For all $n \in \text{dom}(M)$, $\text{status}(M, n) = \text{inv} \Rightarrow \exists C_i, \forall C_{i'} (C_i \triangleright C_{i'} \Rightarrow \text{status}(M, n) = \text{inv})$
  \item \hspace{1cm} For all $n \in \text{dom}(M)$, $\text{status}(M, n) = \text{inv} \Rightarrow \forall C_i, C_{i'} \in C \triangleright C_i \triangleright C_{i'} \Rightarrow \text{status}(M, n) = \text{inv}$
  \item \hspace{1cm} $\forall n \in \text{dom}(M), \text{status}(M, n) = \text{inv}$
  \item \hspace{1cm} $\forall n \in \text{dom}(M), \text{status}(M, n) = \text{commit}$
\end{enumerate}

Lemma 2 guarantees that there is at most one modified copy of a memory block among the cores. This ensures single write access and parallel read access to the memory blocks.

The following lemma shows that the shared copies of a memory block $n$ in different cores always have the same version number. Let the function $\text{version}(M, n)$ return the version number of block address $n$ in $M$.

**LEMMA 3 (CONSISTENT SHARED COPIES).** Given a reachable configuration $MM(M) \triangleright_{\Pi} (Q) \triangleright_{\Pi} (Q) \triangleright_{\Pi} MM(M)$ and $n \in \text{dom}(M)$: If $\text{status}(M, n) = \text{sh}$, and for any $C_i (C_i \triangleright \text{commit}, j_i \vdash rst) : h_j \in C \triangleright_C$ such that $\text{status}(M, n) = \text{sh}$, then $\text{version}(M, n) = \text{version}(M, n)$.

We define the most recent value of a memory block as follows:

**DEFINITION 2 (MOST RECENT VALUE).** For a global configuration $MM(M) \triangleright_{\Pi} (Q) \triangleright_{\Pi} (Q) \triangleright_{\Pi} MM(M)$ and $n \in \text{dom}(M)$:

\begin{enumerate}
  \item \hspace{1cm} If $\text{status}(M, n) = \text{sh}$, and for any $C_i (C_i \triangleright \text{commit}, j \vdash rst) : h_j \in C \triangleright_C$ such that $\text{status}(M, n) = \text{sh}$, then $\text{version}(M, n) = \text{version}(M, n)$.
  \item \hspace{1cm} If $\text{status}(M, n) = \text{sh}$, and for any $C_i (C_i \triangleright \text{commit}, j \vdash rst) : h_j \in C \triangleright_C$ such that $\text{status}(M, n) = \text{sh}$, then $\text{version}(M, n) = \text{version}(M, n)$.
\end{enumerate}

With Lemma 3 and Definition 2, the following lemma shows 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 $MM(M) \triangleright_{\Pi} (Q) \triangleright_{\Pi} (Q) \triangleright_{\Pi} MM(M)$ be a reachable configuration such that $C_i \triangleright C_i \triangleright (Q) \triangleright_{\Pi} (Q) \triangleright_{\Pi} MM(M)$ and $n \in \text{dom}(M)$. Given a block address $n$ and an event $e \in \{C, W, N\}$, then $\text{version}(M, n) = \text{version}(M, n)$.
we have that: if $CR_i : h \rightarrow CR'_i ; h; e$ or $CR_i : h \mapsto CR'_i ; h; e$, then $M_i(n)$ has the most recent value.

6. PROOF OF CONCEPT INTERPRETER

The proposed operational semantics has been implemented in Maude [4], a high-level declarative language for executable specifications based on rewriting logic, which supports different analysis strategies. Maude has been proposed as a semantic framework for concurrent and distributed systems; a parallel system can be axiomatized using equational theory to model the states and rewrite rules can be used to model the system’s dynamic behaviour. Maude specifications can be executed using rewrite commands which implement top-down rule fair and depth-first position fair strategies. Maude also supports searching through all reachable states for states in which invariants are violated such that we can verify invariant properties for all execution paths of concrete examples.

To implement the semantics as an executable specification\(^2\), configurations are defined as multi-sets of Maude objects where each object has a unique identity and the communications between objects are done by message passing. To make the operational semantics executable, we need to concretize the address-function and the associativity and replacement policy of the caches. This is now a configurable parameter to the cache objects in the implementation. Local transition rules, for the execution of both low-level statements and data instructions, are specified as rewrite rules. Messages between the cores are identified operationally and not declared for the global transitions. The main challenge in the implementation was the CORE-COMMUNICATION rule, because Maude does not support label synchronization on transitions as in the operational semantics. Instead we implemented a broadcast operation using a sink to collect messages from different cores, which then multicasts the messages to the other cores using equational logic. The version numbers for cache lines and the histories in the semantics have been omitted in Maude. Instead, we have added counters for cache hits and cache misses per cache, to collect information about the cache hit/miss ratio during simulations.

6.1 Simulations

This section looks closer at the Maude implementation of the operational semantics, discussed above. We need to make the abstract associativity function \( \sim \) executable. In the implementation, the function $\text{Associative}(k)$ is used to store the associativity of the cache memory where \( k \) is the number of cache lines in each cache set. For direct mapped caches, \( k = 1 \) while it is equal to the cache size for fully associative caches. Each cache set has an associated index which helps to find the particular cache set at the time of fetching. Even though the implementation evicts cache lines randomly, first and second priority will be given for invalid and shared cache lines, respectively.

Example. Fig. 5 shows an initial configuration $\text{conf0}$ of the Maude implementation of the operational semantics. Each core is modelled by a Maude object with an identifier $c_i$ and a group of attributes such as $\text{MLoc}$ for the memory map $M$, $\text{CacheSize}$ for total number of cache lines in the private cache, \( \sim \) for the cache associativity, $\text{Rst}$ for executing the task ($\text{rst}$), $\text{D}$ for data instruction queue, $\text{Misses}$ for counting the local cache misses, and $\text{Hits}$ for counting the local cache hits.

The $\text{TBL}$ object is used to concretize the address-function. Here, the $\text{Addr}$ attribute stores a map of bindings $\text{ref}(r) \rightarrow n$ which defines the $\text{address}(r)$ function to find memory block where a given reference is stored.

The $\text{M}$ object implements the main memory. Here, the $\text{attr}$ attribute stores a map of bindings $n \rightarrow \text{status}$. The attribute $\text{missCount}$ counts the number of $\text{fetch}$ operations which have taken place in the configuration during the execution.

The $\text{Sync}$ object implements the label set synchronisation during message passing. Its $\text{Data}$ attribute collects broadcast messages without causing any data race to the same memory address. When a consistent set of messages has been collected, they are delivered to the system.

The $\text{sch}$ object is the scheduler which allocates tasks (i.e., statement sequences) to each core when its executing task is empty. There is no $\text{Version}$ number in this Maude model.

The Maude model is executed by the command $\text{frew} \{ \text{conf0} \}$. The output from a simulation shows one of the possible final object configurations, as in Fig. 6. In this final configuration, all cores have completed their task queue and committed all modified copies to the main memory.

The $\text{Maude}$ $\text{search}$ command allows all possible states from an initial configuration $\{ \text{conf0} \}$ to be explored. The search command is written $\text{search} \{ \text{conf0} \} \rightarrow \text{C: Configuration}$ where $\text{C}$ is a variable of sort $\text{Configuration}$ which matches with all global configurations. We can also use $\text{search}$ command to prove safety properties by testing a predicate on the matches. For example, the search condition in Fig. 7 detects violations of the in-

\(^2\)The Maude prototype is available at http://folk.uio.no/shijib/musepat2016maude.zip
variant in Lemma 2(a), if any. Since no solutions are found, the search shows that the invariant was never violated from this initial state.

\[
\text{search } \{\text{conf0}\} \rightarrow \{\text{C:Configuration} \}
\]

\[
\{\text{A: OID : CR | MLOC: (x : Int \rightarrow (y : Int \rightarrow stA: Status)), Atts }\} \{\text{B: OID : CR | MLOC: (x : Int \rightarrow (y : Int \rightarrow stB: Status)), Atts }\} \{\text{O: OID : MN | M: (y: Int \rightarrow stM: Status, DataSet), Atts }\} \{\text{tbl: OID : TBL | Atts }\} \{\text{sy: OID : Sync | Atts }\} \{\text{sch: OID : Qu | Atts }\} \]

\[
\text{such that } (stM: Status = inv) \text{ and } ((stA: Status = mo \text{ and stB: Status = mo}) \text{ or } (stA: Status \neq mo \text{ and stB: Status \neq mo})).
\]

No solution.

Figure 7: Example for search.

7. CONCLUSIONS

Slogans such as “move the processes closer to the data” reflect how data location is becoming increasingly important in parallel computing. To study how computations and data locations interfere, formal models which account for the location of data and the penalties associated with data access may help the system developer. This paper proposes a basis for such formal models in terms of an operational semantics of execution on cache coherent multicore architectures. The proposed model also opens for reasoning about the proximity of processes and data using techniques from programming languages research such as subject reduction proofs.

In this paper, the semantics incorporates the MSI cache coherence protocol. In the semantics, version numbers and histories are only needed for correctness proofs, and have been omitted from the prototype implementation. Obvious extensions to our work include dynamic task creation, loops, and choice in the statement language, which all extend the operational semantics with standard rules. Going beyond these language extensions, we plan to use the presented semantics in future work to study data layout in main memory, as well as synchronization mechanisms between tasks and scheduling policies to improve the cache hit/miss ratio. The long term goal is to feed such analyses into a compiler for a high-level programming language which targets multicore architectures, such as Encore [2].

8. REFERENCES


APPENDIX

A. SUPPLEMENTARY PROOFS

A.1 Proof of Lemma 1

PROOF. It is trivial to see that the lemma holds for the initial case where the number of transition step is zero, i.e., \( e ; \text{rst} \downarrow^s \text{ci} = \text{rst} \downarrow^s \text{ci} \). Next we are going to show the preservation of the local invariant over the transition steps. By induction hypothesis, we have \( C(M, \sim, e ; \text{rst} \downarrow^s) : h' \rightarrow C(M', \sim, d' ; \text{rst}') : h' \downarrow^s \text{ci} \). Note that \( k \) here refers to \( k \) transition steps. Next we have to show the lemma still holds for the \( k + 1 \) th step.

Assume \( C(M', \sim, d' ; \text{rst}') : h' \rightarrow C(M'', \sim, d'' ; \text{rst}'') : h'' \downarrow^s \text{ci} \), and the step may be labelled. The proof proceeds by case distinction on the rules for the transition steps from Figure 4.

Case \( \text{PrRd}1 \): \( C(M', \sim, d' ; \text{PrRd}(r); rst') : h' \rightarrow C(M'', \sim, d'' ; \text{PrRd}(r); rst'') : h'' \downarrow^s \text{ci} \)

We are further given in the case that \( n = \text{addr}(r) \). Then by induction and Definition 1, we get \( h':(\text{PrRd}(r); rst') \downarrow^s \text{ci} = h':R(C(n); rst') \downarrow^s \text{ci} \), which concludes the case. It is analogous for the case of \( \text{PrRd} \) BLOCK.

Case \( \text{PrRd}2 \): \( C(M', \sim, d' ; \text{PrRd}(r); rst') : h' \rightarrow C(M'', \sim, d'' ; \text{PrRd}(r); rst'') : h'' \downarrow^s \text{ci} \)

We are further given in the case that \( n = \text{addr}(r) \). Then by induction and Definition 1, we have \( h':(\text{PrRd}(r); rst') \downarrow^s \text{ci} = h':R(C(n); rst') \downarrow^s \text{ci} \). For the configuration after the step, we have \( h':(\text{PrRd}(r); rst') \downarrow^s \text{ci} = h':R(C(n); rst') \downarrow^s \text{ci} \) by Definition 1, which concludes the case. The case of \( \text{PrRd} \) BLOCK works similarly.

Case \( \text{PrW}1 \): \( C(M', \sim, d' ; \text{PrW}(r); rst') : h' \rightarrow C(M'', \sim, d'' ; \text{PrW}(r); rst'') : h'' \downarrow^s \text{ci} \)

We are further given in the case that \( n = \text{addr}(r) \). By induction and Definition 1, we have \( h':(\text{PrW}(r); rst') \downarrow^s \text{ci} = h':W(C(n); rst') \downarrow^s \text{ci} \), which concludes the case. The invariant holds for the cases of \( \text{PrRd} \) and \( \text{PrW} \) BLOCK in the similar way.

Case \( \text{PrRd} \) BLOCK: \( C(M', \sim, d' ; \text{PrRd}(r); rst') : h' \rightarrow C(M'', \sim, d'' ; \text{PrRd}(r); rst'') : h'' \downarrow^s \text{ci} \)

We are further given in the case that \( n = \text{addr}(r) \). Then by induction and by Definition 1, we have \( h':(\text{PrRd}(r); rst') \downarrow^s \text{ci} = h':W(C(n); rst') \downarrow^s \text{ci} \). For the configuration after the step, we have \( h':(\text{PrW}(r); rst') \downarrow^s \text{ci} = h':W(C(n); rst') \downarrow^s \text{ci} \) by Definition 1, which concludes the case.

Case \( \text{PrW} \) BLOCK: \( C(M', \sim, d' ; \text{PrW}(r); rst') : h' \rightarrow C(M'', \sim, d'' ; \text{PrW}(r); rst'') : h'' \downarrow^s \text{ci} \)

We know by induction that the configuration before the step satisfies the invariant. Since the given step does not change either \( h' \) or the runtime statements, the invariant holds for the configuration after the step.

Case COMMIT1: \( C(M', \sim, d' ; \text{commit}(r); rst') : h' \rightarrow C(M', \sim, d'' ; \text{rst}') : h' \downarrow^s \text{ci} \)

We are further given in the case that \( n = \text{addr}(r) \). Then by induction and by Definition 1, we have \( h':(\text{commit}(r); rst') \downarrow^s \text{ci} = h':\text{rst} \downarrow^s \text{ci} \), where the lemma holds. The other three commit cases work analogously.

The rest of the rules in Figure 4 is not applicable because they do not affect either the runtime statements or the local history.

A.2 Proof of Lemma 2

PROOF. An initial configuration \( MM(M) \ Qu(\text{rst}) \ CR : h \) satisfies the lemma since all the memory blocks in the main memory have the status \( \text{shared} \), 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:

\[ MM(M) \ Qu(\text{rst}) \ CR : h \xrightarrow{S} MM(M') \ Qu(\text{rst'}) \ CR' : h' \] (1)

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 \( MM(M) \ Qu(\text{rst}) \ CR : h \) satisfies the lemma.

We first consider the case where \( S = \emptyset \). Let \( CR_i \in CR \) where \( CR_i = C_i(M_i; \sim_i, d_i; \text{rst}) : h_i \). By rule TOP-ASYNCH in Figure 3, there are three possibilities of a reduction step when \( S = \emptyset \): (1) it can either be an internal transition in \( CR_i \) (cf. rule PAR-INTERNAL-STEPS); or (2) a global step for the communication between \( CR_i \) and the main memory (cf. rule PAR-MEMORY-ACCESS); or (3) a global step for \( CR_i \) to get a new task from the task queue (cf. rule PAR-TASK-SCHEDULER).

For case (1), after the decomposition with rule PAR-INTERNAL-STEPS, the relevant internal transitions local in \( CR_i \) include: PRD1, PRD BLOCK1, PRW1, and the four commit rules in Figure 4. These steps do not have any effect on the status of any memory block address in either the local cache or the main memory. By induction, the configuration before the step satisfies the invariant. Hence, the invariant still holds after the transition, which concludes this case.

For case (2), decomposing the global configuration with rule PAR-MEMORY-ACCESS will ultimately lead to the application of one of the rules for fetching/flushing a block address \( n \) from/to the main memory in Figure 3.

Case FETCH1:

\[ MM(M); (M_i; \sim_i, \text{fetch}(n); d_i) \rightarrow MM(M_i); (M_i'; \sim_i, d_i') \]

We are further given in this case that \( M(n) = (k, sh) \), i.e., \( \text{status}(M, n) = \text{sh} \) and \( M_i(n) = (k, sh, n') \) where \( n' = \text{select}(M_i, \sim_i, n) \) and \( n' = n \). By induction, the invariant holds for the configuration before the step, and therefore \( \text{status}(M, n) = \text{sh} \) implies by part (c) of the invariant that \( \forall C_j(M_j; \sim_j, d_j; \text{rst}) : h_j \in CR \). \text{status}(M_j, n) \neq \text{mo} \) before the step. Note that there exists a core \( CR_j \) such that \( CR_j = CR_i \). Since after the step \( \text{status}(M_i', n) = \text{status}(M, n) = \text{sh} \), the invariant is maintained. It is analogous for the case FETCH2.

Case FLUSH1:

\[ MM(M); (C_i(M_i; \sim_i, \text{flush}(n); d_i)) \rightarrow MM(M_i); (C_i(M_i'; \sim_i, d_i')) \]

In addition, we have in this case \( M_i(n) = (k, mo) \). Since by induction, the lemma holds for the configuration before the transition step, by part (a) of the invariant from the induction it follows \( \text{status}(M, n) = \text{inv} \), and part (b) gives \( \forall C_j(M_j; \sim_j, d_j; \text{rst}) : h_j \in \text{mo} \).
The premise \( MM(M) \stackrel{RdX(n)}{\rightarrow} MM(M') \) correspond to the transition in the main memory. Then, by rules \( \text{MAIN-MEMORY}_1 \) and \( \text{ONE-LINE-MAIN-MEMORY}_1 \) in Figure 4, we have \( MM(M) \stackrel{RdX(n)}{\rightarrow} MM(M') \), where \( M' = M[n\rightarrow(k, \text{inv})] \).

For the cores \( \text{CR} \), rule \( \text{TOP-SYNCH} \) gives \( \text{CR} \stackrel{RdX(n)}{\rightarrow} \text{CR} \). Then by recursive decomposition with rule \( \text{CORE-COMMUNICATION} \), we get \( \text{CR}_i \stackrel{RdX(n)}{\rightarrow} \text{CR}'_i \) for some \( \text{CR}'_i \), and \( \text{CR}_j \stackrel{RdX(n)}{\rightarrow} \text{CR}'_j \) for all \( \text{CR}_i \in \text{CR} \), and for some \( \text{CR}'_j \). For the step \( \text{CR}_i \stackrel{RdX(n)}{\rightarrow} \text{CR}'_i \), by rule \( \text{SEND-RECEIVE-MESSAGE} \) in Figure 4, we get \( \text{CR}_i \stackrel{RdX(n)}{\rightarrow} \text{CR}'_i \), which is a step local in \( \text{CR}_i \). Consider the relevant rules in Figure 4, namely \( \text{PrRW}_2 \) and \( \text{PrWRBlock}_2 \). The status of the address \( n \) in the cache local in \( \text{CR}_i \) is updated to \( \text{mo} \) after the step.

For the step \( \text{CR}_j \stackrel{RdX(n)}{\rightarrow} \text{CR}'_j \), by rule \( \text{RECEIVE-MESSAGE} \) in Figure 4, we get \( \text{CR}_j \stackrel{RdX(n)}{\rightarrow} \text{CR}'_j \). Considering the relevant rules, \( \text{INVALIDATE-ONE-LINE} \) and \( \text{IGNORE-INVALIDATE-ONE-LINE} \) in Figure 4, we have after the step the status of address \( n \) in all \( \text{CR}_j \) either \( \text{inv} \) or \( \text{no} \) does not exist in the domain. This together with having \( \text{status}(M', n) = \text{inv} \) and \( \text{status}(M_i, n) = \text{mo} \) after the transition shows that part (a) and part (b) in the lemma hold, and therefore concludes the case.

\[ \square \]

A.3 Proof of Lemma 3

**Proof.** An initial configuration \( MM(M) \) satisfies the lemma since all the memory blocks in the main memory have the status \( \text{shared} \) and all cores have empty caches and no data instructions or runtime statements. It is trivial that the invariant holds for the transition rules which are local to either the main memory or a core because the transitions do not modify the version number of a block address.

We consider in the following the cases where transitions are communication steps between main memory and a core \( \text{CR}_R \in \text{CR} \). The rules responsible for such communication steps are those for fetching/flushing a memory block from/to the main memory in Figure 3. Let \( \text{CR}_i = C_i[M_j, \sim_i, d_i, \sim_j \rightarrow \text{rst}_j] \) and \( n \) be the block address.

**Case FETCH_1**

We are further given in this case that \( M_i(n) = (k, \text{sh}) \), i.e. \( \text{status}(M, n) = \text{sh} \) and \( M_j(n) = (k, \text{sh}) \) where \( n' = \text{select}(M_j, \sim_i, n) \) and \( n' = n \). By Lemma 2(a), \( \text{status}(M, n) = \text{sh} \) implies \( \text{status}(M_j, n) = \text{sh} \). By Lemma 2(b), \( \text{status}(M_j, n) = \text{inv} \) or \( n \notin \text{dom}(M_j) \) are not relevant in this lemma. By induction, \( \text{version}(M, n) = \text{version}(M_i, n) = k \).

Since \( M_i = M_i[n \rightarrow (k, \text{sh})] \), and the status of the address \( n \) in the main memory and in all other cores remain unchanged, the configuration after the step satisfies the invariant. It is analogous for case FETCH_2.

**Case FLUSH_1**

We have in this case \( M_i(n) = (k, \text{mo}) \) and \( M_j(n) = (k, \text{mo}) \) implies \( M_j(n) = (k, \text{inv}) \) by Lemma 2(a), and by Lemma 2(b), \( \text{status}(M_j, n) = \text{inv} \) or \( n \notin \text{dom}(M_j) \). The case is concluded by having \( M_i = M_i[n \rightarrow (i + 1, \text{sh})] \) after the step.

The cases FETCH_3 and FLUSH_2 hold immediately because the transition steps do not change the status or version of any memory location in either the main memory or cache local in a core. \[ \square \]
A.4 Proof of Lemma 4

PROOF. In order for the core \( CR_i \) to read from (or write to) a block address \( n \) successfully in its local cache, that is, to generate the event \( R(C_i, n) \) (or \( W(C_i, n) \)), the relevant reduction rules include \( \text{PrRD}_1, \text{PrRD}_{\text{BLOCK}}_1, \text{PrWR}_1, \text{PrWR}_2 \) and \( \text{PrWR}_{\text{BLOCK}}_2 \) in Figure 4.

Consider the rule \( \text{PrRD}_1 \) where \( \text{status}(M_i, n) \neq \text{inv} \), that is, the status of \( n \) is either \( \text{mo} \) or \( \text{sh} \). We first consider the case where \( \text{status}(M_i, n) = \text{mo} \), it is trivial by Lemma 2(a) and (b) that \( M_i(n) \) has the most recent copy according to Definition 2(b). For the case where \( M_i(n) = (k, \text{sh}) \), it implies by Lemma 2(d) that \( \text{status}(M, n) = \text{sh} \) which gives by Lemma 2(c), that \( \forall CR_j \in CR \) where \( CR_j = C_j(M_j, \sim_j, d_j) \). \( \text{status}(M_j, n) \neq \text{mo} \). Similar to the proof of Lemma 3, we just have to consider those cores \( C_i(M_i, \sim_i, d_i) \vdash \text{rst}_l \) : \( h_l \in CR \) where \( \text{status}(M_l, n) = \text{sh} \). Then by Lemma 3, we get \( \text{version}(M_l, n) = k = \text{version}(M_i, n) = \text{version}(M_l, n) \), which satisfies Definition 2(a) conclude the case. The rest of rules mentioned above can be proven analogously. \( \Box \)