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.

CCS Concepts

•Computer systems organization → Multicore architectures; •Software and its engineering → Formal language definitions; Semantics;

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 multicore 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 [4] 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 cor-

*Supported by the EU project FP7-612985 *UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimisations (http://www.upscale-project.eu).*

SAC 2016, April 04 - 08, 2016, Pisa, Italy

ACM ISBN 978-1-4503-3739-7/16/04…$15.00

DOI: http://dx.doi.org/10.1145/2851613.2851718

rectness 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 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 executions. The current market offers different designs for integrating these components. Cache memory keeps the most recently used data accessed by the core available for quick reading or writing. A core reads or writes data as *words*. The cache is organized in cache lines. Each cache line contains several words, such that a specific word can be accessed by the core using a memory reference. 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].

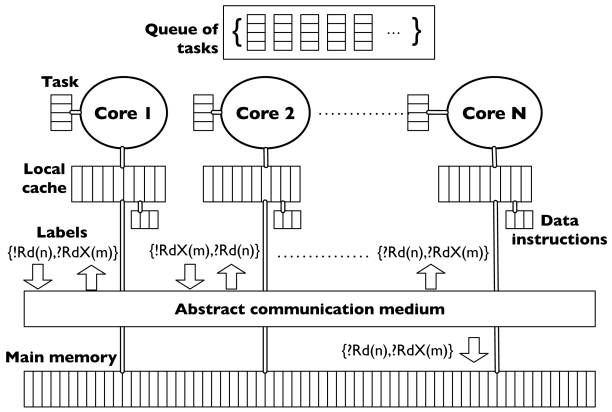

Figure 1: Abstract model of multi-core architecture (illustration).

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**

Syntactic categories.	*Definitions.*
MM, C, Qu *in id*	$Config ::= MM(M)\ Qu(\overline{sst})\ \overline{CR} : H$
n, m *in address*	$CR ::= C(MLoc \vdash rst) : h$
r *in reference*	$MLoc ::= M, \sim, d$
k *in version*	$M ::= \emptyset \mid M[n \mapsto \langle k, status \rangle]$
α *in* $\langle k, status \rangle$	$status ::= mo \mid sh \mid inv$
	$d ::= \varepsilon \mid d;d \mid \texttt{fetch}(n) \mid \texttt{flush}(n)$
	$sst ::= \varepsilon \mid sst;sst \mid \texttt{PrRd}(r) \mid \texttt{PrWr}(r) \mid \texttt{commit}(r)$
	$rst ::= sst \mid rst;rst \mid \texttt{PrRdBl}(r) \mid \texttt{PrWrBl}(r)$
	$\mid \texttt{commit}$
	$H ::= \varepsilon \mid H;\overline{ev}$
	$h ::= \varepsilon \mid h;ev$
	$ev ::= W(C,n) \mid R(C,n)$

Figure 2: Syntax for the abstract model of parallel architectures, where over-bar denotes sets (e.g., $\overline{CR}$) and where $\sim$ represents the associativity and replacement policy.

sst. These statements are $\texttt{PrRd}(r)$ for reading from a memory reference r, $\texttt{PrWr}(r)$ for writing to r and $\texttt{commit}(r)$ for flushing r. Cores execute **runtime statements** *rst*, 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; $\texttt{PrRdBl}(r)$ and $\texttt{PrWrBl}(r)$ represent the corresponding waiting states while data is being fetched from main memory, and $\texttt{commit}$ forces the flushing of all modified data contained in a cache into main memory, used at the end of a task to guarantee that all data is flushed before another task is assigned to the core.

A **runtime configuration** *Config* expresses that the multicore architecture has reached a given state after observing a trace H of events. The configuration consists of a main memory $MM(M)$, a set $Qu(\overline{sst})$ of tasks to be scheduled and executed in a core, multiple cores $\overline{CR}$, and a global history H of events. Memory maps M bind memory addresses n to pairs $\langle k, status \rangle$, representing both memory blocks in main memory and cache lines in cache memory. Each core CR includes a local cache memory *MLoc*, a sequence *rst* of statements to be executed, and a local history h. A **cache memory** *MLoc* consists of a map M, a function $\sim$ expressing its cache associativity and replacement policy, and a sequence of data instructions d. In MSI, a cache line can be *modified, shared* or *invalid*; this is captured in our formal syntax by a *status mo*, *sh* or *inv*, respectively. (Since data is first modified in a cache, blocks in main memory only have status *sh* or *inv*, see Sect. 4 below.) We abstract from the actual data contained in M, but keep the version number k of the data such that the highest version number together with a *sh* or *mo* status represents the most updated copy of data. The lookup function $M(n)$ returns the corresponding pair $\langle k, status \rangle$ if n is in the domain of M. Otherwise it returns $\perp$, indicating n does not exist in M. **Data instructions** $\texttt{fetch}(n)$ load memory blocks into the cache when there is a cache miss and $\texttt{flush}(n)$ store cache lines in main memory.

Logs. The global history H logs the concurrent global execution of statements in the cores $\overline{CR}$, such that a successfully completed read or write by a core to a reference r is reflected by an event ev appended to H. Since many cores execute in parallel, multiple events may be appended at the same time. In events $R(C,n)$ for reading and $W(C,n)$ for writing, C is the *id* of the core and n the block address in which r is located. The local history h logs the execution of a core; it appends single events ev whenever a read or write operation succeeds in the core. Thus, a local history h is a projection over the global history H with respect to a given core.

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-level language *sst*, each core in $\overline{CR}$ has an empty cache, and no data instructions as well as no runtime statements. Executions start from an initial configuration by applying global transition rules, which in turn apply local transition rules. Let $Config \xrightarrow{*} Config'$ 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:

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

where S contains at most one label 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 $\overline{CR}$, 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 $\overline{CR}_1$ and $\overline{CR}_2$, 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: $\overline{CR}_1$ perform local transitions without labels, $\overline{CR}_2$ access main memory, $\overline{CR}_3$ get new tasks from the task queue, and $\overline{CR}_4$ 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 $disjoint(\overline{CR}_1, \overline{CR}_2, \overline{CR}_3, \overline{CR}_4)$ express that the sets of cores involved in the parallel transitions are disjoint to each other.

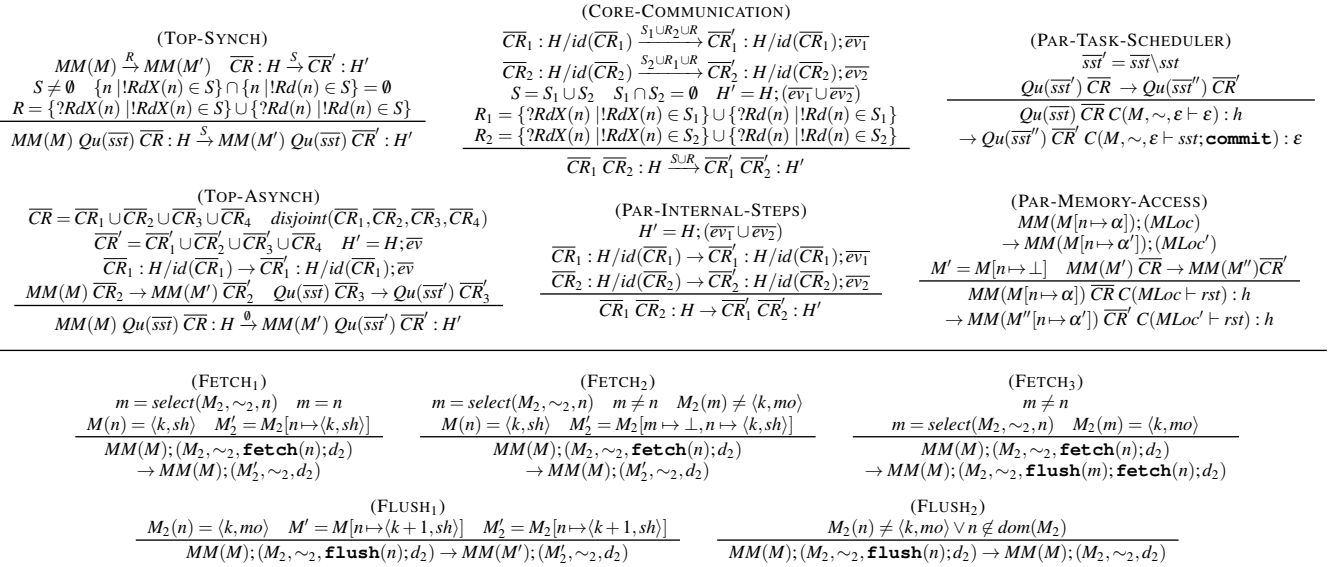

Figure 3: Global semantics for cache coherent multicore architectures

Data transfer between a cache and main memory is described in rules of the form $MM(M);(MLoc) \rightarrow MM(M');(MLoc')$. They capture the execution of data instructions **fetch** and **flush**. Here the function $select(M,\sim,n)$, 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 FETCH$_1$). Rule FETCH$_2$ describes the case where we need to evict a non-modified cache line m; rule FETCH$_3$ refers to the case where the cache line m to be evicted has status *mo*, so it needs to be flushed before cache line n can be loaded. Rules FETCH$_1$ and FETCH$_2$ 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 FLUSH$_1$ 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 FLUSH$_2$ discards the **flush**(n) instruction if the cache line n 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 $addr(r)$ denote the block address that contains the reference r, and $status(M,n)$ the status of cache line n in the map M.

In the main memory, the controller sets the status of a block to *inv* in rule ONE-LINE-MAIN-MEMORY$_1$ if exclusive access has been requested. Rule ONE-LINE-MAIN-MEMORY$_2$ will always accept a shared read request. Rules MAIN-MEMORY$_1$ and MAIN-MEMORY$_2$ are distribution rules for sets of labels.

Label sets are decomposed by the SEND-RECEIVE-MESSAGE (which has only one W label), RECEIVE-EMPTY, and RECEIVE-MESSAGE rules in order to feed the finite state controller. A core can only receive an exclusive request $?RdX(n)$ for a cache line that is not modified. Rule INVALIDATE-ONE-LINE sets the status of cache line n to *inv* if the cache line has status *sh* when the core receives a $?RdX(n)$ message. Rule 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 $?Rd(n)$, if the cache line n has status *mo*, rule FLUSH-ONE-LINE adds a **flush** to the head of the data instructions d (to avoid deadlock), otherwise rule IGNORE-FLUSH-ONE-LINE ignores the message.

Read statements succeed if the cache line n containing the requested reference r is available in the cache, applying rule PRRD$_1$. Otherwise, a **fetch**(n) instruction is added to the tail of the data instructions d in rule PRRD$_2$. In this case, execution is blocked by the statement **PrRdBl**(r). Execution may proceed once the block n has been copied into the cache, captured by rule PRRDBLOCK$_1$. In the parallel setting, the cache line may get invalidated while the core is still blocked after **fetch**. Rule PRRDBLOCK$_2$ captures this situation and broadcasts the $!Rd(n)$ message again.

Rule PRWR$_1$ expresses that a write statement **PrWr**(r) succeeds when the memory block has *mo* status in cache memory. If the cache line is shared, the core needs to get exclusive access, captured by rule PRWR$_2$. 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 PRWR$_3$. Similar to the case for read, we use a statement **PrWrBl**(r) and the rule PRWRBLOCK$_1$ to block repeated read requests. Once the cache line has status *sh*, rule PRWRBLOCK$_2$ requests exclusive access, as in rule PRWR$_2$.

The statements **commit**(r) and **commit** are respectively used to force flushing of a single modified cache line and of the entire cache. Rules COMMIT$_1$ and COMMIT$_2$ capture the single cache line for modified and non-modified cache lines, respectively. Rules COMMIT-ALL$_1$ and COMMIT-ALL$_2$ reduce a **commit** statement to a sequence of **flush**-instructions. In order to ensure data consistency among main memory and individual caches, the final statement in a task should be **commit** (see rule 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 cores, the absence of data races, and successful accesses to memory locations always retrieve the most recent value. We first define a function which translates statements into event histories:

$$\frac{MM(M) \xrightarrow{R} MM(M') \quad MM(M') \xrightarrow{Q} MM(M'')}{MM(M) \xrightarrow{R\cup\{Q\}} MM(M'')}\ (\textsc{Main-Memory}_1) \qquad MM(M) \xrightarrow{\emptyset} MM(M)\ (\textsc{Main-Memory}_2)$$

$$\frac{M' = M[n \mapsto \langle k, inv\rangle]}{MM(M) \xrightarrow{?RdX(n)} MM(M')}\ (\textsc{One-Line-Main-Memory}_1) \qquad MM(M) \xrightarrow{?Rd(n)} MM(M)\ (\textsc{One-Line-Main-Memory}_2)$$

$$\frac{C(MLoc \vdash rst)\ : h \xrightarrow{R} C(MLoc' \vdash rst)\ : h \quad C(MLoc' \vdash rst)\ : h \xrightarrow{W} C(MLoc'' \vdash rst')\ : h'}{C(MLoc \vdash rst)\ : h \xrightarrow{R\cup\{W\}} C(MLoc'' \vdash rst')\ : h'}\ (\textsc{Send-Receive-Message})$$

$$C(MLoc \vdash rst)\ : h \xrightarrow{\emptyset} C(MLoc \vdash rst)\ : h\ (\textsc{Receive-empty})$$

$$\frac{C(MLoc \vdash rst)\ : h \xrightarrow{R} C(MLoc' \vdash rst)\ : h \quad C(MLoc' \vdash rst)\ : h \xrightarrow{Q} C(MLoc'' \vdash rst)\ : h}{C(MLoc \vdash rst)\ : h \xrightarrow{R\cup\{Q\}} C(MLoc'' \vdash rst)\ : h}\ (\textsc{Receive-Message})$$

$$\frac{M(n) = \langle k, sh\rangle \quad M' = M[n \mapsto \langle k, inv\rangle]}{C(M, \sim, d \vdash rst)\ : h \xrightarrow{?RdX(n)} C(M', \sim, d \vdash rst)\ : h}\ (\textsc{Invalidate-One-Line})$$

$$\frac{n \notin dom(M) \vee status(M,n) = inv}{C(M, \sim, d \vdash rst)\ : h \xrightarrow{?RdX(n)} C(M, \sim, d \vdash rst)\ : h}\ (\textsc{Ignore-Invalidate-One-Line})$$

$$\frac{n \in dom(M) \quad status(M,n) = mo \quad d' = \texttt{flush}(n); d}{C(M, \sim, d \vdash rst)\ : h \xrightarrow{?Rd(n)} C(M, \sim, d' \vdash rst)\ : h}\ (\textsc{Flush-One-Line})$$

$$\frac{n \notin dom(M) \vee status(M,n) \neq mo}{C(M, \sim, d \vdash rst)\ : h \xrightarrow{?Rd(n)} C(M, \sim, d \vdash rst)\ : h}\ (\textsc{Ignore-Flush-One-Line})$$

$$\frac{n = addr(r) \quad status(M,n) \neq inv}{C(M, \sim, d \vdash \texttt{PrRd}(r); rst)\ : h \to C(M, \sim, d \vdash rst)\ : h; R(C,n)}\ (\textsc{PrRd}_1)$$

$$\frac{n = addr(r) \quad status(M,n) = inv \vee n \notin dom(M) \quad d' = d; \texttt{fetch}(n) \quad M' = M[n \mapsto \bot]}{C(M, \sim, d \vdash \texttt{PrRd}(r); rst)\ : h \xrightarrow{!Rd(n)} C(M', \sim, d' \vdash \texttt{PrRdBl}(r); rst)\ : h}\ (\textsc{PrRd}_2)$$

$$\frac{n = addr(r) \quad status(M,n) = sh}{C(M, \sim, d \vdash \texttt{PrRdBl}(r); rst)\ : h \to C(M, \sim, d \vdash rst)\ : h; R(C,n)}\ (\textsc{PrRdBlock}_1)$$

$$\frac{n = addr(r) \quad status(M,n) = inv \quad d' = d; \texttt{fetch}(n) \quad M' = M[n \mapsto \bot]}{C(M, \sim, d \vdash \texttt{PrRdBl}(r); rst)\ : h \xrightarrow{!Rd(n)} C(M', \sim, d' \vdash \texttt{PrRdBl}(r); rst)\ : h}\ (\textsc{PrRdBlock}_2)$$

$$\frac{n = addr(r) \quad status(M,n) = mo}{C(M, \sim, d \vdash \texttt{PrWr}(r); rst)\ : h \to C(M, \sim, d \vdash rst)\ : h; W(C,n)}\ (\textsc{PrWr}_1)$$

$$\frac{n = addr(r) \quad M(n) = \langle k, sh\rangle \quad M' = M[n \mapsto \langle k, mo\rangle]}{C(M, \sim, d \vdash \texttt{PrWr}(r); rst)\ : h \xrightarrow{!RdX(n)} C(M', \sim, d \vdash rst)\ : h; W(C,n)}\ (\textsc{PrWr}_2)$$

$$\frac{n = addr(r) \quad status(M,n) = inv \vee n \notin dom(M) \quad d' = d; \texttt{fetch}(n) \quad M' = M[n \mapsto \bot]}{C(M, \sim, d \vdash \texttt{PrWr}(r); rst)\ : h \xrightarrow{!Rd(n)} C(M', \sim, d' \vdash \texttt{PrWrBl}(r); rst)\ : h}\ (\textsc{PrWr}_3)$$

$$\frac{n = addr(r) \quad status(M,n) = inv \quad d' = d; \texttt{fetch}(n) \quad M' = M[n \mapsto \bot]}{C(M, \sim, d \vdash \texttt{PrWrBl}(r); rst)\ : h \xrightarrow{!Rd(n)} C(M', \sim, d' \vdash \texttt{PrWrBl}(r); rst)\ : h}\ (\textsc{PrWrBlock}_1)$$

$$\frac{n = addr(r) \quad M(n) = \langle k, sh\rangle \quad M' = M[n \mapsto \langle k, mo\rangle]}{C(M, \sim, d \vdash \texttt{PrWrBl}(r); rst)\ : h \xrightarrow{!RdX(n)} C(M', \sim, d \vdash rst)\ : h; W(C,n)}\ (\textsc{PrWrBlock}_2)$$

$$\frac{n = addr(r) \quad status(M,n) = mo \quad d' = d; \texttt{flush}(n)}{C(M, \sim, d \vdash \texttt{commit}(r); rst)\ : h \to C(M, \sim, d' \vdash rst)\ : h}\ (\textsc{Commit}_1)$$

$$\frac{n = addr(r) \quad status(M,n) \neq mo \vee n \notin dom(M)}{C(M, \sim, d \vdash \texttt{commit}(r); rst)\ : h \to C(M, \sim, d \vdash rst)\ : h}\ (\textsc{Commit}_2)$$

$$\frac{status(M,n) = mo \quad \texttt{flush}(n) \notin d \quad d' = d; \texttt{flush}(n)}{C(M, \sim, d \vdash \texttt{commit})\ : h \to C(M, \sim, d' \vdash \texttt{commit})\ : h}\ (\textsc{Commit-All}_1)$$

$$\frac{\forall n \in dom(M).status(M,n) \neq mo}{C(M, \sim, d \vdash \texttt{commit})\ : h \to C(M, \sim, d \vdash \varepsilon)\ : h}\ (\textsc{Commit-All}_2)$$

Figure 4: Local semantics for cache coherent multicore architectures

DEFINITION 1. *Let $addr(r) = n$. Define $rst\downarrow_C$ inductively over rst:*

$$\begin{aligned}
(\textbf{\textit{PrRd}}(r); rst)\downarrow_C &= R(C,n);\ rst\downarrow_C \\
(\textbf{\textit{PrRdBl}}(r); rst)\downarrow_C &= R(C,n);\ rst\downarrow_C \\
(\textbf{\textit{PrWr}}(r); rst)\downarrow_C &= W(C,n);\ rst\downarrow_C \\
(\textbf{\textit{PrWrBl}}(r); rst)\downarrow_C &= W(C,n);\ rst\downarrow_C \\
(\textbf{\textit{commit}}(r); rst)\downarrow_C &= rst\downarrow_C \\
\textbf{\textit{commit}}\downarrow_C &= \varepsilon \\
\varepsilon\downarrow_C &= \varepsilon
\end{aligned}$$

Intuitively, $rst\downarrow_C$ reflects the expected program order of read and write accesses when executing *rst* directly on main memory. Note that $\varepsilon; h = h$. We show that execution with local cache preserves this program order:

LEMMA 1 (PROGRAM ORDER).
If $C(M, \sim, \varepsilon \vdash rst) : \varepsilon \to^* C(M', \sim, d \vdash rst') : h$, then $h; rst'\downarrow_C = rst\downarrow_C$.

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 $MM(M)\ Qu(\overline{sst})\ \overline{CR} : H$:*

(a) $\forall n \in dom(M).(status(M,n) = inv \Leftrightarrow \exists\, C_i(M_i, \sim_i, d_i \vdash rst_i) : h_i \in \overline{CR}.\ status(M_i,n) = mo)$

(b) $\forall n \in dom(M).\ (status(M,n) = inv \Leftrightarrow (\exists\, CR_i \in \overline{CR}\ where\ CR_i = C_i(M_i, \sim_i, d_i \vdash rst_i) : h_i.\ status(M_i,n) = mo) \wedge \forall\, C_j(M_j, \sim_j, d_j \vdash rst_j) : h_j \in \overline{CR} \backslash CR_i.\ (status(M_j,n) = inv \vee n \notin dom(M_j)))$

(c) $\forall n \in dom(M).\ (status(M,n) = sh \Leftrightarrow (\forall\, C_i(M_i, \sim_i, d_i \vdash rst_i) : h_i \in \overline{CR}.\ status(M_i,n) \neq mo))$

(d) $\forall\, C_i(M_i, \sim_i, d_i \vdash rst_i) : h_i \in \overline{CR}, \forall n \in dom(M_i).\ (status(M_i,n) = sh \Rightarrow status(M,n) = sh)$

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 $version(M,n)$ return the version number of block address n in M.

LEMMA 3 (CONSISTENT SHARED COPIES). *Given a reachable configuration $MM(M)\ Qu(\overline{sst})\ \overline{CR}$ and $n \in dom(M)$: If $status(M,n) = sh$, and if for any $C_j(M_j, \sim_j, d_j \vdash rst_j) : h_j \in \overline{CR}$ such that $status(M_j,n) = sh$, then $version(M,n) = version(M_j,n)$.*

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

DEFINITION 2 (MOST RECENT VALUE). *For a global configuration $MM(M)Qu(\overline{sst})\ \overline{CR}$, a memory location n, and a core $CR_i \in \overline{CR}$, where $CR_i = C_i(M_i, \sim_i, d_i \vdash rst_i) : h_i$. $M_i(n)$ has* the most recent value *if the following holds.*

(a) *If $M_i(n) = \langle k, sh\rangle$, and $\forall\, C_j(M_j, \sim_j, d_j \vdash rst_i) : h_j \in \overline{CR} \backslash CR_i$. $status(M_j,n) = sh$, then $M_j(n) = \langle k, sh\rangle$, and $M(n) = \langle k, sh\rangle$.*

(b) *If $status(M_i,n) = mo$, then $status(M,n) = inv$, and $\forall\, C_j(M_j, \sim_j, d_j \vdash rst_i) : h_j \in \overline{CR} \backslash CR_i.\ status(M_j,n) = inv$.*

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 $MM(M)$ $Qu(\overline{sst})$ $\overline{CR}$ be a reachable configuration such that $CR_i = C_i(M_i, \sim_i, d_i \vdash rst_i) : h_i$ for $CR_i \in \overline{CR}$. Given a block address n and an event $e \in \{R(C_i,n), W(C_i,n)\}$, we have that: if $CR_i : h_i \rightarrow CR_i' : h_i; e$ or $CR_i : h \xrightarrow{!RdX(n)} CR_i' : h_i; e$, then $M_i(n)$ has the most recent value.*

6. 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. Obvious extensions to our work include dynamic task creation, loops, and choice in the 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].

For further details of this work, including supplementary proofs for Sect. 5 and the description of a prototype interpreter, see [3].

7. REFERENCES

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

[2] S. Brandauer, E. Castegren, D. Clarke, K. Fernandez-Reyes, E. B. Johnsen, K. I Pun, S. L. Tapia Tarifa, T. Wrigstad, and A. M. Yang. Parallel objects for multicores: A glimpse at the parallel language Encore. In *Formal Methods for Multicore Programming*, *LNCS* 9104, pages 1–56. Springer, 2015.

[3] S. Bijo, E. B. Johnsen, K. I Pun, and S. L. Tapia Tarifa. An Operational Semantics of Cache Coherent Multicore Architectures. *Res. Rep. 449*, Dept. of Informatics, Univ. of Oslo, Dec 2015.

[4] T. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC'11)*, pages 1–12, ACM 2011.

[5] K. Crary and M. J. Sullivan. A calculus for relaxed memory. In *Proc. POPL*, pages 623–636. ACM, 2015.

[6] D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1997.

[7] G. Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.

[8] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proc. Intl. Conf. on Computer Design (ICCD'92)*, pages 522–525. IEEE, 1992.

[9] D. L. Dill, S. Park, and A. G. Nowatzyk. Formal specification of abstract memory models. In *Proc. Symp. on Research on Integrated Systems*, pages 38–52. MIT Press, 1993.

[10] B. Dongol, O. Travkin, J. Derrick, and H. Wehrheim. A high-level semantics for program execution under total store order memory. In *Theoretical Aspects of Computing (ICTAC 2013)*, *LNCS* 8049, pages 177–194. Springer, 2013.

[11] E. A. Emerson, and V. Kahlon. Rapid parameterized model checking of snoopy cache coherence protocols. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, *LNCS* 2619, pages 144–159. Springer, 2003.

[12] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2011.

[13] R. Jagadeesan, C. Pitcher, and J. Riely. Generative operational semantics for relaxed memory models. In *Proc. ESOP*, *LNCS* 6012, pages 307–326. Springer, 2010.

[14] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[15] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *Real-Time Systems Symp. (RTSS 2009)*, pages 57–67. IEEE Press, Dec 2009.

[16] S. Mador-Haim, L. Maranget, S. Sarkar, K. Memarian, J. Alglave, S. Owens, R. Alur, M. M. K. Martin, P. Sewell, and D. Williams. An axiomatic memory model for POWER multiprocessors. In *Proc. CAV*, *LNCS* 7358, pages 495–512. Springer, 2012.

[17] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, Nov. 2005.

[18] J. E. Miller, H. Kasture, G. Kurian, C. G. III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *High-Performance Computer Architecture (HPCA-16)*, pages 1–12. IEEE, 2010.

[19] J. Pang, W. Fokkink, R. F. H. Hofman, and R. Veldema. Model checking a cache coherence protocol of a Java DSM implementation. *J. Log. and Alg. Prog.*, 71(1):1–43, 2007.

[20] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2013.

[21] F. Pong and M. Dubois. Verification techniques for cache coherence protocols. *ACM Comp. Surv.*, 29(1):82–126, 1997.

[22] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *Proc. PLDI*, pages 175–186. ACM, 2011.

[23] 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. *CACM*, 53(7):89–97, 2010.

[24] G. Smith, J. Derrick, and B. Dongol. Admit your weakness: Verifying correctness on TSO architectures. In *11th Intl. Symp. on Formal Aspects of Component Softwares*, *LNCS* 8997, pages 364–383. Springer, 2014.

[25] X. Yu, M. Vijayaraghavan, and S. Devadas. A proof of correctness for the Tardis cache coherence protocol. *arXiv preprint*, arXiv:1505.06459, 2015.