A Formal Model of Parallel Execution on Multicore Architectures with Multilevel Caches *

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

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

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

1 Introduction

Multicore architectures enhance the performance of software applications by executing programs in parallel on multiple cores, and by exploiting a hierarchy of cache memory which allows quick access to recently used data, but comes at a price of managing multiple co-existing copies of the same data. The cost of accessing data from a core depends on where the data is located and how it is used by tasks executing on other cores. 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. For example, the benefits from developing lock-free algorithms may be severely reduced by bad data locality and unexpected cache misses [18].

Software developers targeting multicore architectures need to answer questions about data locality, data access, and data movement: Is the data organized in the most convenient way to allow efficient data access for the application? Are the organization and ordering of tasks optimal with respect to a given data layout? How does a given data layout fit with the target cache hierarchy? These questions are important for software

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

quality, but they are difficult to answer in an intuitive and straightforward way. Formal models of program execution may help answer such questions, but models of parallel programs today generally abstract from caches in multicore architectures and only assume single copies of data in shared memory [9, 14, 16, 34] (i.e., they assume that threads have direct access to memory). On the other hand, formal models of hardware architecture and consistency protocols, such as cache coherence, focus on low-level correctness, but completely abstract from the programming level [11, 12, 21, 32, 33]. Consequently, neither programming nor hardware models provide much guidance for software developers in making efficient use of cache memory. In the context of shared memory multicore computing, it seems interesting to integrate models of parallel program execution with models of hardware architecture. Such an integration opens for reasoning about data movement when parallel applications access data from shared memory multicore architectures.

To address the problem described above, our aim for this paper is to develop a formal model of parallel programs executing on *shared* memory multicore architectures with multilevel caches. For simplicity, we focus on programs specified in terms of their *data access patterns*, rather than on the programs themselves. These data access patterns describe how tasks running on a core interact with memory in terms of read and write accesses. The formalization is inspired by programming language semantics; we develop an operational semantics of parallel computation for these data access patterns, which accounts for data movement and data consistency in an architecture with many cores and associated multilevel caches.

The purpose of this work is not to evaluate the specifics of a concrete hardware architecture, but rather to formally describe program execution in a setting with multiple and consistent copies of the same data in shared memory and in caches. Consequently, we integrate a cache coherence protocol directly into the operational semantics of our formal model, while abstracting from the concrete communication medium (e.g., a bus or a ring). This protocol acts as an orchestrator between parallel executions on different cores, by restricting data access to the memory components of the shared memory architecture to ensure consistency. Cache coherence is orthogonal to weak memory models and associated program reordering [1]; in fact, most cache coherence protocols guarantee sequential consistency. Whereas work on weak memory models (e.g., [2]) focus on the possible values of program variables, our work completely abstracts from the data being manipulated. The presented model of multicore architecture with multilevel caches guarantees desirable properties for the program, such as the preservation of the program order for the data access patterns, the absence of data races, and that cores always access the most recent data value. The technical contributions of this paper are

1. a formal, operational model of execution on multicore architectures with multilevel caches for tasks describing data access patterns with loops, choice and spawn;
2. correctness properties for this formal model, expressed as invariants over an arbitrary number of cores and an arbitrary number of multilevel caches; and
3. a proof of concept implementation of the model in the rewriting tool Maude [8].

This work is part of a line of research by the authors. Whereas previous work [3, 4] studied the much simpler setting of statically given, purely sequential data access pat-

terns and single-level caches, this paper addresses data access patterns with dynamically spawned tasks with loops and branching, and multilevel caches.

Paper overview. Sect. 2 briefly reviews background concepts on shared memory multicore architectures, Sect. 3 presents our abstract formal model of multicore architecture with multilevel caches and shared memory, Sect. 4 summarizes the proven correctness properties of this model, Sect. 5 presents a proof of concept implementation and an example, Sect. 6 discusses the related work, and Sect. 7 concludes the paper.

2 An Overview of Shared Memory Multicore Architectures

We briefly discuss basic concepts of multicore architectures, for further details see, e.g., [10, 15, 28, 35]. The components of multicore architectures are parallel processing units called *cores* for executing tasks, a main memory for data storage, and memory units called *caches* which give the cores rapid access to recently used data. Each core has a hierarchically structured memory system, organized in terms of size, speed, and distance: the L_1 cache is the smallest, fastest, and closest to the core and the L_m cache is the slowest, largest, and furthest away. The memory systems of the cores are connected via a communication medium for inter-core communication with a given topology such bus, ring, or mesh. A *cache hit* expresses that data required by the core is found in its caches, a *cache miss* that the data needs to be fetched from main memory. The hierarchy can be generalized to architectures in which caches may be shared between cores.

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 organized in cache lines, which store memory blocks. During program execution, cores access data in memory as a word using its reference, but 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 give space for newly fetched blocks. The choice of which block to evict depends on the cache line organization, the so-called *associativity*, and the *replacement policy*. In *k-way* set associative caches, cache memory is organized as sets of k-cache lines and a memory block can go anywhere in a particular set. *Fully associative* caches treat the entire cache memory as a single set. A *direct mapped* cache consists of singleton sets; thus, a particular block can only go to one specific cache line. If the set in which a new block should be placed is full, a block is evicted to free space using a replacement policy such as random, FIFO or LRU (Least Recently Used).

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

A *memory consistency model* [1] for cache-based architectures combines a (weak or strong) local memory model with a cache coherence protocol. Cache coherence protocols ensure 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. The cost of sequential consistency is that writing to non-exclusive cache lines need to be broadcast. *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 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.

3 A Formal Model of Execution on Shared Memory Multicore Architectures

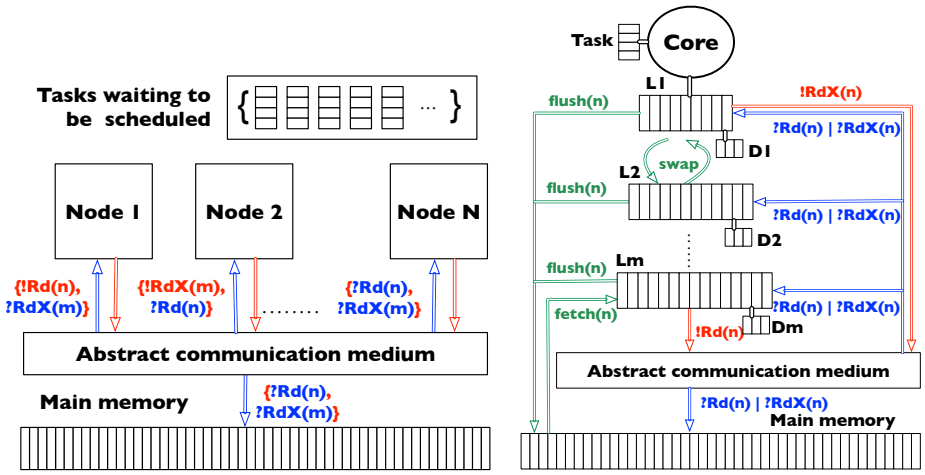
This section presents our formal model of program execution on shared memory multicore architectures with multilevel caches. We first discuss the abstractions introduced in the model, then its syntax, and finally the operational semantics of the formal model.

3.1 Abstractions in the Formal Model

We consider a model of multicore architectures with a communication medium that abstracts from concrete topologies but ensures cache coherency using the MSI protocol. The architecture is illustrated in Fig. 1a. A *node* consists of a core and its hierarchy of private caches. Each core in the model executes tasks scheduled from a shared task pool, which can easily be extended to a more advanced scheduler. To communicate with the other components, the node broadcasts request messages $!Rd(n)$ and $!RdX(n)$ via the medium to read or write to a block with address n, respectively.

The structure of a single node comprises a core with multiple levels of exclusive caches $L_1, L_2, \cdots, L_m$, as illustrated by Fig. 1b. Each cache L_i has a data instruction queue D_i for *flush* and *fetch* instructions, which move blocks of data from or to main memory or between caches. Red lines capture messages broadcast by the node to the others via the medium, blue lines capture messages received by main memory and by components in each node, and green lines capture data transfer between components.

To read data from a block n, the core looks for n by traversing its local caches in the hierarchical order (i.e., from the first level L_1 to the last level, here L_m). If we get a *cache miss*, the last-level cache broadcasts a *read request* $!Rd(n)$ via the communication medium to the other nodes and main memory. The last-level cache *fetches* the block when it is available in main memory. Eviction is required if the last-level cache is full. From a cache L_i, block n is propagated to the first level L_1 through intermediate caches. The block is transferred from L_i to L_{i-1} if the cache has free space; otherwise a block is selected from L_{i-1} and *swapped* with n in L_i. Writing to a block n is only allowed if it is



(a) Overview of the model.

(b) Intra-node interactions.

Fig. 1: The structure of the formal model of multicore architectures.

in shared or modified state in the first-level cache, which then broadcasts an *invalidation request* $!RdX(m)$ through the medium to all other nodes, to obtain exclusive access.

Let $?Rd(n)$ and $?RdX(n)$ denote the reception of read and invalidation requests by a cache or by main memory (the blue lines in Fig. 1b); they are the duals of the broadcast requests discussed above. If a cache receives a read request $?Rd(n)$ and it has the block in modified state, the cache *flushes* the block to main memory (the green lines in Fig. 1b); 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. For simplicity, we abstract from the actual data stored in memory blocks, let blocks transfer between nodes via the main memory, and without compromising the validity of the model, we assume that a cache line has the same size as a memory block. We model read and invalidation requests in the communication medium to be instantaneous; this is justified by message transfer being an order of magnitude faster than data transfer, and by the focus of the work is on data movement. 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 Data Access Patterns and Runtime Configurations

The syntax of the formal model is shown in Fig. 2. A configuration *Config* consists of a main memory M, shared between multiple nodes with cores $\overline{CR}$ and caches $\overline{Ca}$, and a set of tasks $\overline{T}$ to be executed. A core CR with identifier Cid has runtime statements rst to be executed. A cache Ca has a memory M, an identifier Lev, and a sequence of data instructions dst to be performed. In a cache identifier $lev(Cid, Lid, flag)$, Cid indicates to which core the cache belongs, $Lid \in \mathbb{N}^+$ the level of the cache, and $flag$ whether it is

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

Fig. 2: Syntax for the formal model of multicore architectures, where over-bar denotes sets (e.g., $\overline{CR}$), n represents memory addresses and r references.

at the last level in the node. *Lid* is unique within each node. A memory $M : n \rightharpoonup st$ maps addresses n to status tags st. 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 $Tb : T \rightharpoonup dap$ maps task identifiers T to *data access patterns* *dap*. These 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, and control flow statements $dap_1 \sqcap dap_2$ to select either dap_1 or dap_2 for execution, dap^* to repeat the execution of *dap* zero or many times, and $\texttt{spawn}(T)$ to add *dap* to the pool of tasks to be scheduled, where *dap* is the data access pattern corresponding to task T in the task table. To ensure data consistency, the statement `commit` is used at the end of each tasks to flush the entire cache after task execution. Since the task table is statically given, we assume it is always available and not represent it explicitly in the configurations.

The cores execute *runtime statements rst*, which extend *dap* with the additional control statements $\texttt{readBl}(r)$ and $\texttt{writeBl}(r)$ to indicate that the core is blocked due to a cache miss. Each cache performs *data instructions dst*, which are sequences of $\texttt{fetch}(n)$ to fetch a block n from the next level cache or from main memory, $\texttt{flush}(n)$ to flush the modified copy of n to the main memory, and `flush` to flush all modified copies in the cache. The instruction $\texttt{fetch}(n)$ is replaced by $\texttt{fetchBl}(n)$ when the cache is suspended, waiting for block n to arrive in the next level cache.

3.3 An Operational Semantics of Parallel Execution on Multicore Architectures

We define a parallel model of task execution, which expresses true concurrency in the multicore setting, by means of a structural operational semantics (SOS) [29], and use labels on transitions to synchronize read and invalidation requests. The semantics consists of a local and a global level. The *local level* captures local transitions in main memory, task execution in each core and intra-node communications to ensure data consistency between different components. The *global level* captures transitions involving data transfer between caches and main memory, broadcasting of messages, scheduling of tasks, and enforces data consistency by restricting how labels match in the composition rules. Multiple nodes may request different memory blocks at the same time by

$$\frac{first(Lev) = true \quad Cid(Lev) = c \quad n = addr(r) \quad status(M,n) = sh \vee status(M,n) = mo}{(Lev \bullet M \bullet dst) \circ (c \bullet \texttt{read}(r); rst) \rightarrow (Lev \bullet M \bullet dst) \circ (c \bullet rst)} \; (\text{PRRD}_1)$$

$$\frac{first(Lev) = true \quad Cid(Lev) = c \quad n = addr(r) \quad status(M,n) = inv \vee n \notin dom(M)}{(Lev \bullet M \bullet dst) \circ (c \bullet \texttt{read}(r); rst) \rightarrow (Lev \bullet M[n \mapsto \bot] \bullet dst; \texttt{fetch}(n)) \circ (c \bullet \texttt{readBl}(r); rst)} \; (\text{PRRD}_2)$$

$$\frac{first(Lev) = true \quad Cid(Lev) = c \quad n = addr(r) \quad status(M,n) = sh}{(Lev \bullet M \bullet dst) \circ (c \bullet \texttt{write}(r); rst) \xrightarrow{!RdX(n)} (Lev \bullet M[n \mapsto mo] \bullet dst) \circ (c \bullet rst)} \; (\text{PRWR}_2)$$

Fig. 3: Local semantics of task execution in a core and the first level cache

parallel instantaneous broadcast, using possibly empty sets of labels. The formal syntax for the label mechanism is as follows:

$$\begin{array}{ll} W ::= !Rd(n) \mid !RdX(n) & Q ::= ?Rd(n) \mid ?RdX(n) \\ S ::= \emptyset \mid \{W\} \mid S \cup S & R ::= \emptyset \mid \{Q\} \mid R \cup R \end{array}$$

where S and R represent possibly empty sets of sent and received requests, respectively.

Let $Config \xrightarrow{*} Config'$ denote an execution starting from $Config$ which produces $Config'$ by repeatedly applying rules at the global level, which in turn apply rules at the local level for each component. In an *initial configuration*, all blocks in main memory M have status tag sh, 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 $Config$. For brevity, we do not discuss the full operational semantics here, but focus on a representative subset of the rules; the full semantics can be found in the accompanying technical report [5].

Local Semantics. The local semantics reflects the execution of statements, the interactions between caches in a node, and how the local state changes in each cache line by following the finite state controller that enforces the MSI protocol during the execution. A representative selection of local transition rules for a node and for main memory is given in Figs. 3 and 4. Let the function $addr(r)$ return the block address n containing reference r, the predicate $first(Lev)$ is true when Lev is the first level cache, $last(Lev)$ is true when Lev is at the last level, and the function $status(M,n)$ returns the status of block n in map M.

Fig. 3 shows a representative selection of transition steps involving a core and its first level cache. Reading reference r succeeds in rule PRRD$_1$ if the block containing r is available in the first-level cache. Otherwise, rule PRRD$_2$ adds a $\texttt{fetch}(n)$ instruction to the end of the data instructions dst of the first level cache and blocks further execution of the core with the statement $\texttt{readBl}(r)$. Execution may proceed once the block n is

$$\frac{\begin{array}{c}Cid(Lev_i) = Cid(Lev_j) \quad status(M_j,n) = s_j \quad s_j \in \{sh,mo\} \\ Lid(Lev_j) = Lid(Lev_i)+1 \quad select(M_i,n) = n_i \quad status(M_i,n_i) = s_i\end{array}}{\begin{array}{c}(Lev_i \bullet M_i \bullet \texttt{fetch}(n);dst_i) \circ (Lev_j \bullet M_j \bullet dst_j) \rightarrow \\ (Lev_i \bullet M_i[n_i \mapsto \bot, n \mapsto s_j] \bullet dst_i) \circ (Lev_j \bullet M_j[n \mapsto \bot, n_i \mapsto s_i] \bullet dst_j)\end{array}} \text{(LC-HIT}_1\text{)}$$

$$\frac{status(M_j,n) = inv \vee n \notin dom(M_j) \quad Lid(Lev_j) = Lid(Lev_i)+1 \quad Cid(Lev_i) = Cid(Lev_j)}{\begin{array}{c}(Lev_i \bullet M_i \bullet \texttt{fetch}(n);dst_i) \circ (Lev_j \bullet M_j \bullet dst_j) \rightarrow \\ (Lev_i \bullet M_i \bullet \texttt{fetchBl}(n);dst_i) \circ (Lev_j \bullet M_j[n \mapsto \bot] \bullet dst_j;\texttt{fetch}(n))\end{array}} \text{(LC-MISS}_1\text{)}$$

$$\frac{last(Lev) = true \quad status(M,n) = inv \vee n \notin dom(M)}{(Lev \bullet M \bullet \texttt{fetch}(n);dst) \xrightarrow{!Rd(n)} (Lev \bullet M[n \mapsto \bot] \bullet \texttt{fetchBl}(n);dst)} \text{(LLC-MISS)}$$

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

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

Fig. 4: Local semantics between caches in a core

copied into the cache with status *sh*. Repeated invalidation may occur if the cache line gets invalidated by another core while the core is still blocked, which entails reapplying rule PRRD$_2$. Writing to reference r succeeds if the associated memory block has *mo* status in the first-level cache. If the cache line is in shared state, the core broadcasts $!RdX(n)$ request, which appears as a label in rule PRWR$_2$, to get exclusive access. If the cache line is invalid (or the block is not in the cache), the core needs to fetch the block from main memory and execution is blocked by the statement $\texttt{writeBl}(r)$, similar to reading in rule PRRD$_2$. The rules for the other *rst* statements are standard.

Fig. 4 shows rules which are local to the cache hierarchy: rules LC-HIT$_1$ and LC-MISS$_1$ capture the interactions between two adjacent levels of caches, while the rest describes the transition steps local to a cache. Rule LC-HIT$_1$ captures the case where cache Lev_i needs to fetch block n and finds it in *sh* or *mo* state in the next level cache. The function $select(M_i,n)$ determines the address where the block should be placed, based on a cache associativity and a replacement policy. If eviction is needed, block n from Lev_j will be swapped with the selected block in Lev_i in rule LC-HIT$_1$. Otherwise, n is transferred to Lev_i and removed from Lev_j since the model considers exclusive caches. Setting a block n to $\bot$ in memory M, denoted as $M[n \mapsto \bot]$, means that n is removed from M. Rule LC-MISS$_1$ shows how fetch instructions are propagated to lower levels in the cache hierarchy. If the block cannot be found in any local cache, we have a *cache miss*: execution is blocked by the instruction $\texttt{fetchBl}(n)$, and a read request $!Rd(n)$ will be broadcast, represented by a label in rule LLC-MISS.

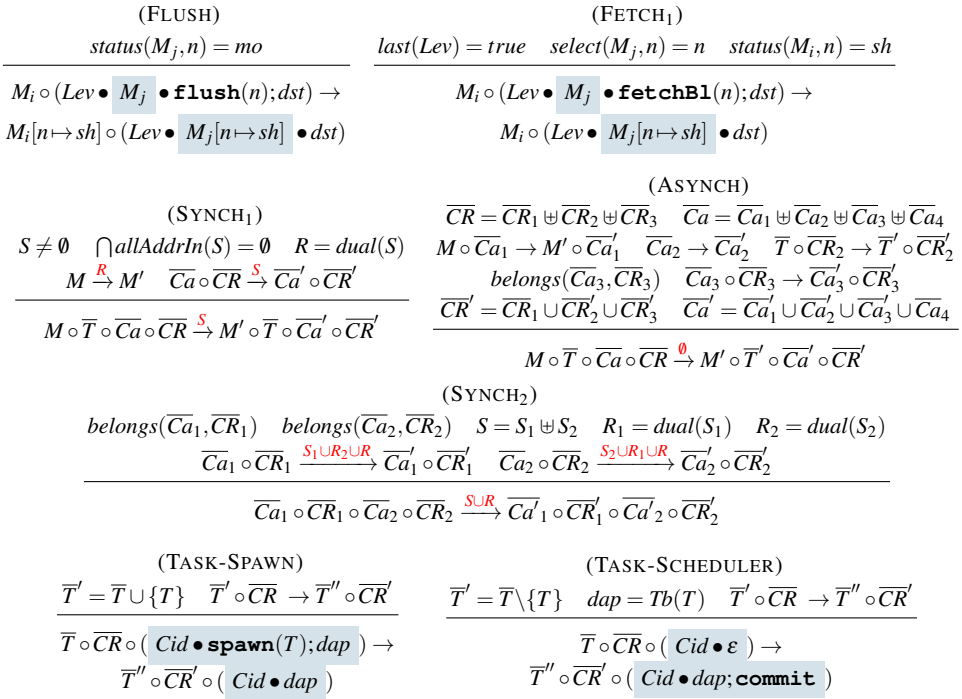


Fig. 5: Global semantics for cache coherent multicore architectures. The disjoint union operator $\uplus$ is defined as $X_1 \uplus X_2 = X_1 \cup X_2$ such that $X_1 \cap X_2 = \emptyset$.

If a cache receives an invalidation request $?RdX(n)$ for a block n and has this block with status *sh*, the cache changes the status to *inv* in rule INV-ONE-LINE. If a cache receives a read request $?Rd(n)$ and has block n with status *mo*, rule FLUSH-ONE-LINE appends a **flush**-instruction to *dst* to prioritize the flushing of the modified copy (to avoid deadlock caused by cyclic waiting for modified data to be flushed to main memory). The received messages are ignored in all other cases. The main memory ignores read requests, but responds to invalidate requests by changing the status of a block to *inv* as in the rule INV-MAIN-MEMORY, defined as $M \xrightarrow{?RdX(n)} M[n \mapsto inv]$.

Global Semantics. The global semantics represents the abstract communication medium: it captures interactions between different components in the configuration and ensures data coherency between caches and main memory. A representative selection of global transition rules is given in Fig. 5. Rules FLUSH and FETCH$_1$ capture the data movement between a cache and the main memory. A cache at any level can flush data to main memory. Rule FLUSH updates a block in main memory with the modified copy in the cache and sets the status to *sh* both in the cache and main memory. However, only the last-level cache can fetch data from main memory. Rule FETCH$_1$ copies the data to the cache if no eviction is required. If eviction is needed and the block chosen by the *select* function has status *mo*, it will be flushed before the requested block can be fetched.

The remaining rules in Fig. 5 handle interactions between different components in the architecture. Rule SYNCH$_1$ captures global synchronization for a non-empty set S. In this rule, different read and invalidation requests are being broadcast, and to maintain data consistency, the different components must process these requests at the same time. Note that to apply rule SYNCH$_1$, S must contain at most one request per address, which is ensured by the predicate $\bigcap allAddrIn(S) = \emptyset$, and the set of receiving labels R is generated as the *dual* of S. For synchronization, the transition is decomposed into a premise for main memory with labels R, and another premise for the cores with labels S. Rule SYNCH-2 distributes labels over cores by recursively decomposing S into sets of sending and receiving labels for sets of cores $\overline{CR}_1$ and $\overline{CR}_2$, such that each set eventually contains at most one W label (either $!Rd(n)$ or $!RdX(n)$) to match transitions in the local rules. The predicate $belongs(\overline{Ca}, \overline{CR})$ expresses that any cache in $\overline{Ca}$ belongs to exactly one of the cores in $\overline{CR}$. The recursive decomposition of S repeats until the dual labels have been generated for each single node. The rule ensures that the sender of a message W does not receive its dual Q. Rule ASYNCH captures parallel transitions when the label set is empty. These transitions can be local to individual nodes, parallel memory accesses, or scheduling of new tasks. TASK-SPAWN adds a new task identifier to the task queue and TASK-SCHEDULER looks up in the task table with the task identifier T and schedules the corresponding task to a core. Adding the statement **`commit`** to the end of the scheduled task ensures that all modified data is flushed before the next task is executed on the same core. Note that parallel spawning/scheduling in one transition is allowed by rules TASK-SPAWN and TASK-SCHEDULER.

4 Correctness of the Model

For the proposed model, we consider standard correctness properties for data consistency and cache coherency, based on the literature [10, 36], including the preservation of program order in each core, absence of data races and no access to stale data. The preservation of these properties by our semantics ensures that the model correctly captures cache coherent data movement triggered by the underlying parallel architecture with any number of cores and caches, using our formalization of the MSI protocol for data consistency. For brevity, the full proofs have been omitted in this paper, and can be found in the accompanying technical report [5].

To formulate and prove these properties, we extend the syntax of Sect. 3.2 with monitoring information. For *data consistency*, the memory mapping M is extended with version numbers k, therefore $M : n \rightharpoonup \langle k, st \rangle$ such that k is incremented every time there is a **`flush`** operation in n. For *program order*, we add local histories h to the cores, which log all successful read and write operations executed so far by the current task; therefore, the syntax of a core is modified to $(c \bullet rst) : h$, expressing that the core with identifier c is executing the task rst starting after history h. The history h is extended in the semantics by the rules which correspond to successful operations. The syntax of the global configuration is also extended with global history H as $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR} : H$, where H records the concurrent executions in all cores $\overline{CR}$ in the architecture in terms of a sequence of sets of successful operations. The monitoring extensions do not influence

execution in the operational semantics; i.e., the applicability of the rules is not affected by these extensions (the details are omitted for brevity, see [5]).

We now formalize the denotational meaning of the *rst*-statements of our syntax, in terms of sets of local histories. Let ε denote the empty history, ";" the concatenation operator, and $\preceq$ the reflexive prefix relation on histories. Let $R(c,n)$ and $W(c,n)$ denote successful read and write operations to address n by core c, respectively.

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:*

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

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* $\{h;\tau \mid \tau \in [\![rst']\!]_c\} \subseteq [\![rst]\!]_c$.

Proof (sketch). Starting with an empty history ε, $(c \bullet rst) : \varepsilon \rightarrow^* (c \bullet rst') : h$ describes a core c executing *rst* reaches *rst'* with history h by making zero or more transition steps, where h is a sequence of successful read and write access generated during the execution. The proof is by induction on the transition steps local in a core, partially captured in Fig. 3. □

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* $h \in [\![rst]\!]_c$.

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

Corollary 1 establishes the local program order of the operations of each individual core. Hence, the model's formalization of the MSI protocol preserves *sequential consistency* [19] 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 *be the cache* $(Lev_x \bullet M_x \bullet dst_x)$. *The conjunction of the following properties holds 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) $\forall\ Ca_i \in \overline{Ca}\ , \forall n \in dom(M_i).\ (status(M_i,n) = sh \Rightarrow status(M,n) = sh)$

Proof (sketch). The lemma can be proven by showing that these properties are invariants preserved by all transition steps:

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

where S is a set of sending messages, handled by ASYNCH, SYNCH_1 and SYNCH_2 in Fig. 5. Remember that the caches are exclusive in each core, and in order to apply SYNCH_1, S must contain at most one message for each block address n. The proof proceeds by case distinction on the rules for the transition steps. □

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 $(Lev_i \bullet M_i \bullet dst_i) \in \overline{Ca}$ such that $status(M_i,n) = sh$ for any cache, then $version(M,n) = version(M_i,n)$.*

Proof (sketch). The invariant trivially holds for transition rules that are for two caches residing in the same core, or local in either a single cache or the main memory as the transitions do not modify the version number of a block address. The proof then proceeds by cases for the transition steps dealing with fetching/flushing a memory block from/to the main memory by a cache, e.g., the rules FLUSH and FETCH_1 in Fig. 5. □

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 $Ca_i = (Lev_i \bullet M_i \bullet dst_i)$. Then $M_i(n)$ has* the most recent value *if the following holds:*

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

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 $CR_i \in \overline{CR}$, $Ca_i = (Lev_i \bullet M_i \bullet dst_i)$ for $Ca_i \in \overline{Ca}$ and $belongs(Ca_i, CR_i)$. Consider a block address n and an event $e \in \{R(c_i,n), W(c_i,n)\}$. If $Ca_i \circ CR_i : h_i \rightarrow Ca_i' \circ CR_i' : (h_i; e)$ or $Ca_i \circ CR_i : h_i \xrightarrow{!RdX(n)} Ca_i' \circ CR_i' : (h_i; e)$, then $M_i(n)$ has the most recent value.*

Proof (sketch). 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 first level cache. The proof therefore proceeds by cases for these rules, a subset of which is shown in Fig. 3.

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 $M_i(n) = \langle k, sh \rangle$, 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 $Ca_j = (Lev_j \bullet M_j \bullet dst_j)$. Then we need to consider all caches $Lev_g \bullet M_g \bullet dst_g \in \overline{Ca}$ where $status(M_g,n) = sh$. From Lemma 3, we get $version(M_i,n) = k = version(M,n) = version(M_g,n)$, which satisfies Definition 2 (a). This concludes the case. The other rules can be proven analogously. □

5 Proof of Concept Implementation

To show the proposed model executable and to observe the behavior of different configurations, we have developed a proof of concept implementation[1] in Maude [8], a rewriting logic system. The Maude framework allows us to build an executable implementation of the operational semantics where transition rules are implemented as conditional rewrite rules of the form `crl [label]: t => t' if cond`, which transforms a term which matches a pattern t into a term of the corresponding pattern t', and as conditional equations of the form `ceq t = t' if cond` for modelling the instantaneous communication of the label mechanism and the implementation of different auxiliary functions. 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, replacement policies and memory layout), the latter requires them to be explicit such that the model with a particular configuration, containing an explicit parallel architecture and a number of parallel tasks that are specified by the user, can be executed. This enables behavior of various configurations to be observed and compared. Another important difference is that while our semantics captures true concurrency by using the label mechanism, the Maude framework only allows interleaving. Therefore, one parallel and global step in the semantics will be translated into one or more interleaving steps in the Maude proof of concept implementation. Such translation does not affect the properties discussed in Section 4.

The proof of concept implementation in Maude is complementary to the proposed semantics because it allows specifying and comparing configurations in which the design choices for the underlying hardware architecture are different, such as the number of cores, cache levels, the data layout in main memory, the cache associativity and replacement policy. Exploring such design decisions is beneficial for the development of software for multicore systems, where hardware features and data layout influence data movement, and consequently the performance of an application. Using a simple example, we illustrate how to observe the impact of the number of caches and the data layout

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

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

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

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

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

Fig. 6: An example of the data access patterns of a program.

on data movement, captured by weighted penalties associated with accessing data from memory other than the first level cache.

Example: Observing the Impact of Multilevel Caches and Data Layout

Consider a program that has been abstracted into the data access patterns shown in Fig. 6. We want to compare different scenarios for running this program, using our proof of concept implementation. We consider three architectures with three cores C1, C2 and C3, varying in the number of caches per core. We are going to observe a parallel execution where C1, C2 and C3 execute the tasks T1, T2 and T3, respectively, and consider nine scenarios in which for each of the three architectures, there are three different data layouts. For simplicity, we here consider the results after running the loop of each task a finite number of times, in this case, 20.

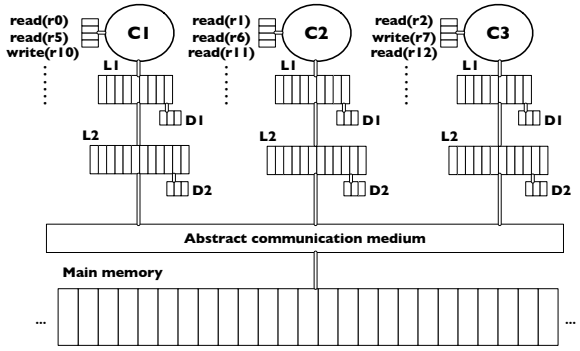


Fig. 7: An example of a parallel architecture with 3 cores and 2 level caches.

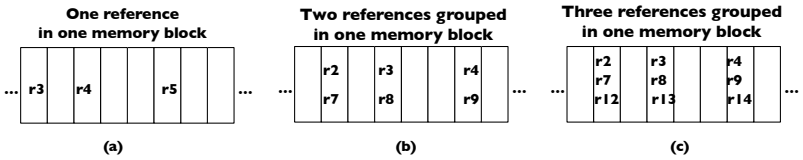


Fig. 8: Different data layouts to be setup in the example.

In the first architecture, we have a single level cache L1 in each core; in the second architecture, we have two levels of cache L1, L2, as depicted in Fig. 7; and in the third architecture, we have three levels of cache L1, L2,L3. To easily compare the data access in the different levels of cache and in main memory, we associate weighted penalties to accesses from different levels of memory. For simplicity in this example, we use order of magnitude differences and associate penalties 1,10,100, and 1000 with accesses from L1, L2, L3 and main memory, respectively. Cache associativity has been set up as direct mapped, 2-way associativity and 3-way associativity for L1, L2, L3, respectively. We additionally consider three different data layouts, depicted in Fig. 8. In the first layout (Fig. 8a) the tasks need to access different memory blocks for each reference, in the second (Fig. 8b) we group two references together in one block, and in the third (Fig. 8c) we group three references together.

Figure 9 summarises the results of executing the model in the Maude proof of concept implementation, for the nine considered scenarios. Observe that when we have spread data, that is, the different references reside in different memory blocks, the scenarios where cores have a single level of cache need to perform many evictions and fetch operations to access data from main memory. This increases the access time, as reflected by the accumulated the penalty. In the scenarios where cores have three levels of cache, the penalty is substantially lower, although the access patterns are the same as the scenarios of single level caches. This is because it requires fewer evictions and operations for the cores to fetch or flush data from or to main memory, although there are still penalties from swapping data between the different cache levels. Thus, the scenarios in the example confirm the expected behavior of the model proposed in this paper, and we can observe the impact of data layout on data movement and the relation between data movement and the number of caches.

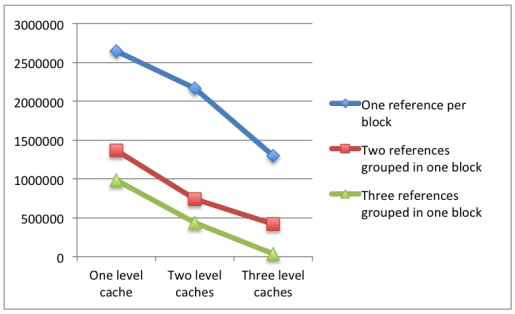


Fig. 9: Accumulated penalties of the nine different scenarios.

6 Related Work

Work on analysis of multicore architectures typically include simulation of cache coherence protocols and formal techniques analyzing their correctness. Simulation tools

for cache coherence protocols evaluate their performance and efficiency on different architectures (e.g., gems [22] and gem5 [6]). These tools perform evaluations of, e.g., the cache hit/miss ratio and response time, by running benchmark programs written as low-level read and write instructions to memory. Advanced simulators such as Graphite [25] and Sniper [7] run programs on distributed clusters to simulate executions on multicore architectures with thousands of cores, where the simulations do not consider the data movements in the architecture. In contrast, our work provides a formal model capturing the interactions that trigger data transfer between different components, and shows the potential impacts on such movement with respect to data layout and the number of caches by a proof of concept implementation of the model in Maude. The worst-case response times of concurrent programs running on multicore architectures with shared caches can also be analyzed [20].

Both operational and axiomatic formal models have been used to capture the impact of parallel executions on shared memory under relaxed memory models. They include abstract calculi [9], memory models for programming languages such as Java [16], and machine-level instruction sets for concrete processors such as POWER [21, 32] and x86 [33], and for programs executing under total store order (TSO) architectures [14, 34]. This work on weak memory models abstracts from caches, and is as such largely orthogonal to our work which does not consider the reordering of source-level syntax.

Cache coherence protocols have also been analyzed in the setting of automata, and (parametrized) model checking (e.g., [11, 27, 30]) has been used to abstract from a specific number of cores when proving the correctness of the protocols (e.g., [12, 13, 37]). For instance, Maude's model checker has recently been used to verify the correctness of configurations of the MSI and ESI protocols [23, 31]. 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

Software is increasingly designed to run on multicore architectures, where data locality, data access, and data movement crucially influence the performance of the parallel execution. We believe that formal models that capture how parallel programs interact with memory, may help software developers understand how data access influences the behavior of parallel tasks executing on multicore architectures with shared memory, and thereby improve data locality and better avoid expensive cache misses. For this purpose, we combine abstract models of parallel program execution with models of shared memory multicore architecture, to capture data movement when parallel programs access data on such architectures. This paper develops a formal executable model of multicore architectures with multilevel caches from a program perspective rather than a hardware perspective, and addresses dynamically spawned data access patterns. The formal model is given as an operational semantics for data access patterns executing in parallel on different cores, and ensures data consistency by embodying the MSI cache coherence protocol. We have shown that the model guarantees correctness properties concerning data consistency, to ensure that we correctly capture data movement triggered by the

cache coherence protocol. We provide a proof of concept implementation of the model, and show by example how choices for a program's data layout in combination with the underlying hardware architecture affect data movement.

This work opens several interesting directions for future work, including extensions required for richer programming languages. For data structures and dynamically allocated memory (e.g., object creation), the model could be extended with type layouts and alloc (e.g., [26]). We are currently considering the extraction of data access patterns from models in ABS [17] and we are implementing a more powerful simulation tool. Other directions of future work include shared caches and locking mechanisms which allow atomic blocks and synchronization between data access patterns to be modeled. Finally, models as developed in this paper could serve as a foundation to study the effects of program specific optimizations 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. J. Alglave, L. Maranget, and M. Tautschnig. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.*, 2014.
3. S. Bijo, E. B. Johnsen, K. I Pun, and S. L. Tapia Tarifa. A Maude framework for cache coherent multicore architectures. In *Proc. Workshop on Rewriting Logic and Its Applications (WRLA)*, LNCS 9942, pages 47–63. Springer, 2016.
4. S. Bijo, E. B. Johnsen, K. I Pun, and S. L. Tapia Tarifa. An operational semantics of cache coherent multicore architectures. In *Proc. Symp. Applied Computing (SAC)*. ACM, 2016.
5. S. Bijo, E. B. Johnsen, K. I Pun, and S. L. Tapia Tarifa. A formal model of parallel execution in multicore architectures with multilevel caches (long version). Res. rep., Dept. of Informatics, Univ. of Oslo, 2017. `http://violet.at.ifi.uio.no/papers/mc-rr.pdf`.
6. N. Binkert, *et al.* The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2), 2011.
7. T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proc. High Performance Computing, Networking, Storage and Analysis (SC)*, pages 52:1–52:12. ACM, 2011.
8. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, LNCS 4350. Springer, 2007.
9. K. Crary and M. J. Sullivan. A calculus for relaxed memory. In *Proc. Principles of Programming Languages (POPL)*, pages 623–636. ACM, 2015.
10. D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1997.
11. G. Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.
12. D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proc. Computer Design on VLSI in Computer Processors (ICCD)*. IEEE, 1992.
13. D. L. Dill, S. Park, and A. G. Nowatzyk. Formal specification of abstract memory models. In *Proc. Symp. Research on Integrated Systems*, pages 38–52. MIT Press, 1993.
14. B. Dongol, O. Travkin, J. Derrick, and H. Wehrheim. A high-level semantics for program execution under total store order memory. In *Proc. Theoretical Aspects of Computing (ICTAC)*, LNCS 8049, pages 177–194. Springer, 2013.
15. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2011.

16. R. Jagadeesan, C. Pitcher, and J. Riely. Generative operational semantics for relaxed memory models. In *Proc. Eur. Conf. Programming Languages and Systems (ESOP)*, LNCS 6012, pages 307–326. Springer, 2010.
17. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Proc. Formal Methods for Components and Objects (FMCO 2010)*, LNCS 6957, pages 142–164. Springer, 2011.
18. M. Kandemir, *et al.* Improving locality using loop and data transformations in an integrated framework. In *Proc. ACM/IEEE International Symposium on Microarchitecture*,1998.
19. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.
20. Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *Proc. Real-Time Systems Symp. (RTSS)*, pages 57–67. IEEE, 2009.
21. S. Mador-Haim, *et al.* An axiomatic memory model for POWER multiprocessors. In *Proc. Computer Aided Verification (CAV)*, LNCS 7358, pages 495–512. Springer, 2012.
22. M. M. K. Martin, *et al.* Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.
23. 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*, LNCS 8373, pages 129–150. Springer, 2014.
24. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
25. J. E. Miller, *et al.* Graphite: A distributed parallel simulator for multicores. In *Proc. High-Performance Computer Architecture (HPCA)*, pages 1–12. IEEE, 2010.
26. M. Nita, D. Grossman, and C. Chambers. A theory of platform-dependent low-level software. In *Proc. Principles of Prog. Languages (POPL)*, pages 209–220. ACM, 2008.
27. 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 Algeb. Prog.*, 71(1):1–43, 2007.
28. D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/-Software Interface*. Morgan Kaufmann, 2013.
29. G. D. Plotkin. A structural approach to operational semantics. *J. Log. and Algeb. Prog.*, 60-61:17–139, 2004.
30. F. Pong and M. Dubois. Verification techniques for cache coherence protocols. *ACM Computing Surveys*, 29(1):82–126, 1997.
31. S. Ramírez and C. Rocha. Formal verification of safety properties for a cache coherence protocol. In *Proc. Colombian Computing Conf. (10CCC)*, pages 9–16. IEEE, 2015.
32. S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *Proc. PLDI*, pages 175–186. ACM, 2011.
33. 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.
34. G. Smith, J. Derrick, and B. Dongol. Admit your weakness: Verifying correctness on TSO architectures. In *Proc. Formal Aspects of Component Software (FACS)*, LNCS 8997, pages 364–383. Springer, 2015.
35. Y. Solihin. *Fundamentals of Parallel Multicore Architecture*. Chapman & Hall/CRC, 2015.
36. D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool, 2011.
37. X. Yu, M. Vijayaraghavan, and S. Devadas. A proof of correctness for the Tardis cache coherence protocol. *CoRR*, abs/1505.06459, 2015.