Deployment by Construction for Multicore Architectures [⋆]

Shiji Bijo[1], Einar Broch Johnsen[1], Ka I Pun[1,2], Christoph Seidl[3], and S. Lizeth Tapia Tarifa[1]

[1] Department of Informatics, University of Oslo, Oslo, Norway
{shijib,einarj,violet,sltarifa}@ifi.uio.no
[2] Western Norway University of Applied Sciences, Bergen, Norway
[3] Technical University of Braunschweig, Braunschweig, Germany
c.seidl@tu-bs.de

Abstract. In stepwise program development, abstract specifications can be transformed into (parallel) programs which preserve functional correctness. Although tackling bad performance after a program's deployment may require a costly redesign, deployment decisions are usually made very late in program development. This paper argues for the introduction of deployment decisions as an integrated part of a development-by-construction process: Deployment decisions should be expressed as part of a program's high-level model and evaluated by how they affect program performance, using metrics at an appropriate level of abstraction. To illustrate such a deployment-by-construction process, we sketch how deployment decisions may be modelled and evaluated, concerning data layout in shared memory for parallel programs targeting shared-memory multicore architectures with caches. For simplicity, we use an abstract metric of data access penalties and simulate data accesses on a memory system which internally ensures data coherency between cores.

1 Introduction

Software development following the correctness-by-construction approach introduces transformation steps to gradually turn abstract, declarative specifications into concrete constructive programs, such that each transformation step preserves the functional correctness of the original specification [23]. This line of work has deep roots in computer science, going back to Dijkstra's Guarded Command language, programming from specifications [1, 26], as well as to work on refinement [6, 25]. While transformations originally focused on strategies such as *divide and conquer*, to introduce *branching* and *recursion*, other transformation steps were developed to introduce *concurrency*, e.g., in the context of Action systems [4, 5] and Unity [13, 14].

A computer system includes not only its (functionally correct) program code but also the *deployment* of this code on, e.g., multicore or distributed hardware.

[⋆] Supported by *SIRIUS – Centre for Scalable Data Access* (www.sirius-labs.no).

Although the careful planning of deployment is critical, deployment decisions are generally made after design, implementation, and validation (e.g., [20, 30]). Deployment decisions affect both data-driven applications and service-oriented scenarios with traffic fluctuations and peaks, typical for parallel software running on distributed cloud or multicore HPC architectures. While "bad deployment" need not affect the functional correctness of software, it may be crucial for its perceived quality, for example in terms of increased runtimes. Tackling performance problems may require considerable changes in design and even impact the requirements level [7]. For example, a program may be tweaked to run more efficiently on parallel on distributed machines by replacing synchronized code with carefully hand-crafted lock-free procedures, just to see its runtime severely increased by unfortunate cache misses. These could be avoided by a different data layout in memory or by reintroducing locks to protect regions of memory.

In this paper, we approach this problem in the context of parallel architectures with shared memory and caches, by providing a formalism to abstractly represent and analyze a program's memory accesses with regard to their impact on runtime behavior, given a set of deployment decisions. We further provide a model-based proof of concept implementation that allows to specify programs in a high-level programming language as well as custom memory layouts for deployment, which are compiled to our formalism. With these contributions, it is possible to model, simulate, analyze and optimize potential interactions of a program and the memory of a chosen deployment architecture *before* deploying (and possibly even writing) the program, which makes the procedure part of the software construction process. Hence, in this paper, we coin the term *Deployment-by-Construction (D-b-C)*.

In summary, the contributions of the paper are as follows:

- We motivate and discuss deployment decisions in the context of D-b-C;
- we illustrate how deploying a parallel program on a shared memory multicore machine with caches could work, seeing the operational model of this machine as a black box from the developer's perspective;
- we extend our framework for data accesses with locks to control exclusive access to data, preserving atomic regions from the source program, and show the correctness for the operational model inside the black box; and
- we provide a proof of concept implementation that allows reducing artifacts of a high-level programming language and a resource model describing memory layouts to our formalization for analysis, simulation and optimization.

2 Deployment-by-Construction: An Overview

This section outlines our D-b-C approach. Consider a toolbox that receives inputs from a software developer (including a high-level *program* describing the functional behavior of the targeted system, a *resource model* specifying the deployment configuration on which the program executes) and returns a performance indicator for the program in terms of *data access penalties*, i.e., a metric to compare the quality of different deployment configurations for a particular program. Figure 1 depicts the abstract model of the D-b-C approach.

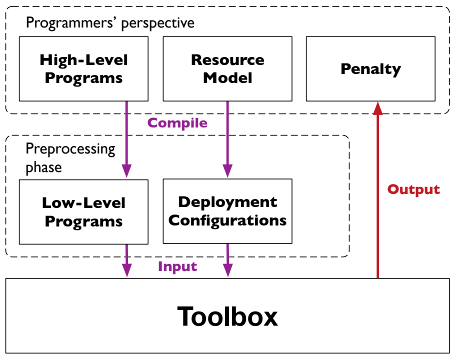

Fig. 1: The components of the deployment-by-construction approach.

To focus on the data access aspect of a given program, we abstract from other aspects of its behavior. In a first phase, each program written in a high-level language, provided by the programmer, is automatically translated into a low-level program, which, together with the deployment configuration of a multicore architecture generated from the provided resource model, serve as inputs to the toolbox. Note that the developer can also provide a low-level program as an input to the toolbox. The developer need not worry about formalizing aspects of deployment which are provided in the actual architecture. In particular, the toolbox handles the communications between different cores and caches in the multicore architecture specified in the generated deployment configuration to ensure consistency of the concurrently existing data. In a second phase, the low-level program is executed and the toolbox returns the corresponding penalties with respect to data access. In the following, we briefly present each component in our D-b-C approach[4]

High-Level Programming Language (HLP) is a Turing-complete synthetic programming language featuring constructs for variables, assignments, expressions, control flow branching, loops and synchronized execution of parallel calculations. Its purpose is to illustrate basic features of a high-level programming language, which are compiled to LLP (see below) to simulate the effects of program deployment. Details regarding its syntax and semantics are given in Sec. 3.1. In a real-world scenario, languages such as Java or C# could be substituted for HLP without the need to change any of the other components.

Low-Level Programming Language (LLP) is an abstraction from concrete programming instructions to only represent memory accesses. It serves as basis for simulation and analysis but does not have to be written by software developers directly. Details regarding its syntax and semantics are given in Sec. 3.2. Artifacts of LLP are compiled to the toolbox (see below).

Resource Model (RM) is an input format, enabling developers to provide information regarding the number and type of cores in the multicore architecture along with their respective memory layouts. The information contained in RM artifacts is used when compiling to the toolbox (see below) as values for configuration parameters. Details regarding RM are given in Sec. 3.3.

[4] The full approach with the different components can be download from: `https://github.com/ShijiBijo84/DbC`.

```
HLP ::= "Program" <String> ["Variables" VariableDeclaration
  {"," VariableDeclaration} "."] {Task} [Schedule] "End" ".";
Schedule ::= "Schedule" "(" <String> {"," <String>} ")" ".";
Task ::= "Task" <String> "Variables" VariableDeclaration
  {"," VariableDeclaration} ".")? Block "End" ".";
VariableDeclaration ::= Variable [":=" Value];
Variable ::= <String>
Block ::= {Statement};

Statement ::= Assignment | If | For | Synchronized | ExpressionStatement;
Assignment ::= Variable ":=" Expression ".";
If ::= "If" "(" Condition ")" "Then" Block ["Else" Block] "End" ".";
Condition ::= Expression ("=" | "!=" | "<" | "<=" | ">=" | ">") Expression;
For ::= "For" Variable ":=" Value ("To" | "Down To") Value "Do" Block "End" ".";
Synchronized ::= "Synchronized" "(" Variable {"," Variable} ")" Block "End" ".";

Expression ::= UnaryExpression | BinaryExpression | ParenthesisExpression;
UnaryExpression ::= "-" Expression;
BinaryExpression ::= Expression ("+" | "-" | "*" | "/") Expression;
ParenthesisExpression ::= "(" Expression ")";

Value ::= <Integer>;
```

Lst. 1.1: Syntax of the High-Level Programming Language (HLP).

Toolbox contains a formalism and operational semantics of LLP that respect the memory configuration provided in the resource model. With the toolbox, it is possible to simulate and compare different deployments of a program or different variations of a program with similar/different deployment decisions to choose the more beneficial configuration. An inside view of the toolbox is provided in Sec. 4.

3 Using the Toolbox: A Practical Modeling Environment for Deployment-by-Construction

A user of our toolbox need not know the details of its construction. To demonstrate this transparency, we now focus on the artifacts relevant for applying D-b-C. (The construction of the toolbox is detailed in Sec. 4.)

3.1 HLP: An Example Programming Language with Parallelism

A plethora of languages may be used to implement functionality for a parallelizable application, also within the C-b-C framework. The artifacts of these languages may be subject to the effects of bad deployment with respect to non-functional properties and negatively impact the execution times as we investigate with our toolbox. We deliberately decided to not limit our approach to a particular programming language, but instead make it suitable for a variety of languages. To demonstrate the use of our toolbox, we have devised a synthetic programming language simply called *High-Level Programming Language (HLP)*, which serves as placeholder for other languages, e.g., Java or C#.

Listing 1.1 provides an overview of the language concepts of HLP in terms of syntax in Extended Backus-Naur Form (EBNF). A high-level program `HLP` has a name, may declare variables and consists of a sequence of tasks and a **`Schedule`** instruction that may be used to instantiate those tasks. A **`Task`** is an individual unit, which may be run in parallel to other tasks. It has a name, may declare local variables and provides an implementation consisting of a sequence of program statements. Note that, depending on where they are declared, variables may be either local to a task or global to the program, i.e., available for all tasks and, thus, subject to potential *concurrent accesses*.

HLP provides fairly standard program statements: An **`Assignment`** associates a variable with a (calculated) value. An **`If`** statement performs conditional control flow branching with an optional else block. A **`For`** loop increments/decrements a loop variable to repeat a specific number of times. To deal with concurrency (as in multicore or distributed systems), apart from tasks, HLP offers a **`Synchronized`** statement similar to that of Java, which allows *intrinsic locking* over a set of variables for a particular block of statements to prohibit concurrent access to these variables. To perform calculations, HLP contains **`Expressions`**, e.g., for arithmetic operations. For simplicity, all variables in HLP are implicitly of type integer and, thus, **`Value`** literals may only consist of integers.

Listing 1.2 contains a code snippet covering most aspects of the HLP language. The code consists of three tasks which access shared variables, protected by the **`Synchronized`** statement. From the functional point of view, there is no race conditions in this example. However, as we will see in Sec. 3.3, an inopportune memory layout may cause race conditions due to the *false sharing* phenomenon [18], in which various variables are allocated in the same memory block as commonly done when allocating arrays.

In a practical application of the toolbox, HLP should be replaced by the respective programming language(s) used to implement functionality (e.g., Java or C#). Due to the structuring of our toolbox, the concepts of the respective programming language just have to be abstracted to data access patterns when compiling to LLP to use the full capabilities of the toolbox.

3.2 LLP: An Abstraction Model for Data Accesses

To get an insight into the impact of deployment decisions for a particular program, it is of utmost importance to analyze the interaction of the program logic with the existing memory structures. To abstract from unnecessary details and specifics of programming languages such as Java or C#, we introduce *Low-Level Programming Language (LLP)* to focus on *data access patterns* of programs in terms of their possible read and write accesses to memory rather than their computation. LLP is generally the target of compilation from HLP (or any other suitable design or programming language) and serves as input for our further analyses and simulation.

Listing 1.3 presents the syntax of LLP in EBNF. A low-level program **`LLP`** consists of a set of tasks and a main **`Block`**. Each **`Task`** has a name and a **`Block`** with a sequence of data access patterns. A data access pattern **`DAP`** may take multiple forms: For input/output operations, it is possible to **`Read`** and **`Write`** a memory reference. Memory allocation for local variables is done using **`Malloc`**.

```
Program Example
  Variables x1, x2, x3, y1, y2, y3, z1, z2, z3.

  Task T1 Variables i1.
    For i1 := 1 To 20 Do
      Synchronized(x3) x1 := x3 + 1. End.
      Synchronized(y1) y1 := 6. End.
      Synchronized(z1, z3) z1 := z3 + 4. End.
    End.
  End.

  Task T2 Variables i2.
    For i2 := 1 To 30 Do
      Synchronized(x3) x2 := x3 + 2. End.
      Synchronized(y1) y2 := y1. End.
      x2 := 7.
    End.
  End.

  Task T3 Variables i3.
    For i3 := 1 To 50 Do
      Synchronized(x3) x3 := 5. End.
      y3 := 1.
      Synchronized(z1, z3) z3 := z1 + 1. End.
    End.
  End.

  Schedule(T1, T2, T3).
End.
```

Lst. 1.2: Running example: Code snippet in the High-Level Programming Language (HLP).

For cached access, `Commit` flushes the contents of the cache line with the given reference, or the entire cache if no reference is given, to main memory. For mutual exclusion, `Lock` and `Unlock` take and release a lock[5], respectively. For control flow, there are instructions for `ControlFlowBranching` (perceived as non-deterministic choice at the abstraction level of LLP), a `Repetition` with a specified number of loops and a standard `Skip` instruction. Finally, there is the option for nesting with `Parentheses` and dynamic task creation with `Spawn`, which instantiates and executes a specified task. A `Reference` to memory is given via a descriptive name similar to a variable name.

Listing 1.4 shows the LLP representation of the HLP program in Listing 1.2, obtained via an automatic translation of the HLP language concepts to the respective LLP concepts by considering the operational semantics of HLP. For example, a variable in HLP is automatically initialized when declared, which means that there is a write access for every variable declaration. Of particular note for the example is the translation of the `For` statement of HLP: While the body of the for-loop is repeated the specified number of times via an LLP repetition, there are also additional memory accesses to consider. Before entering the LLP repetition, the HLP loop variable is initialized with the lower bound

[5] Note that a special memory block is reserved for each lock reference, whose value is either 0 or 1, indicating whether a lock is taken.

```
LLP ::= {Task} "main" "{" Block "}";

Task ::= "task" <String> "{" Block "}";
Block ::= [DAP {";" DAP}];

DAP ::= Read | Write | Malloc | Commit | Lock | Unlock | ControlFlowBranching |
        Repetition | Skip | Parentheses | Spawn;

Read ::= "read" "(" Reference ")";
Write ::= "write" "(" Reference ")";
Malloc ::= "malloc" "(" Reference {"," Reference } ")";
Commit ::= "commit" ["(" Reference ")"];
Lock ::= "lock" "(" Reference ")";
Unlock ::= "unlock" "(" Reference ")";
ControlFlowBranching ::= "(" Block ")" "||" "(" Block ")";
Repetition ::= "(" Block ")" "*" <Integer>;
Skip ::= "skip";
Parentheses ::= "(" Block ")";
Spawn ::= "spawn" "(" <String> ")";

Reference ::= <String>;
```

Lst. 1.3: Syntax of the Low-Level Programming Language (LLP).

of the HLP **For** statement resulting in a write access. At the beginning of each repetition, the HLP loop variable is checked for whether it has reached the upper bound resulting in a read access. Finally, at the end of each LLP repetition, the loop variable is incremented resulting in a read and a write access. During the translation procedure, a HPL assignment, e.g., **x1:=x3+1** (Listing 1.2, l. 6) will be translated to read accesses to variables in the right hand side, e.g., **x3**, followed by a write access to the variable in the left hand side, e.g., **x1** (Listing 1.4, l. 4). Also, the HLP **Synchronized** statements over variables are translated to LLP **lock** . . . **unlock** sequences where each unique constellation of HLP variables is translated to a specific LLP lock variable, e.g., the HLP **Synchronized** statement over variables **z1** and **z3** (Listing 1.2, l. 23) is translated to a lock variable **l3** (Listing 1.4, l. 24).

By focusing on memory accesses in LLP and abstracting from unnecessary details of more high-level programming languages, it is possible to analyze the interaction of deployment decisions and program execution. While LLP artifacts are usually generated as part of compilation so that they do not have to be written manually, LLP code may also be written manually to give very fine-grained control over memory accesses to analyze the resulting impact on execution. LLP programs are used as input to the toolbox to perform simulation and analysis.

3.3 RM: A Modeling Notation for Deployment Configurations

Potential deployment configurations may differ in memory availability, distribution, sharing and layout. We propose the *Resource Model (RM)* to specify the deployment configuration relevant for a particular deployment setup.

A resource model **RM** consists of a shared **Memory** and possibly multiple **Devices** operating on that memory. The size of the **Memory** may be config-

```
task T1 {malloc(i1);
  write(i1); write(i1);              //Initialize variables and loop variable.
  (read(i1);                         //Check loop condition.
  lock(l1); read(x3); write(x1); unlock(l1);
  lock(l2); write(y1); unlock(l2);
  lock(l3); read(z3); write(z1); unlock(l3);
  read(i1); write(i1)) * 20            //Increment loop variable and repeat.
}

task T2 {malloc(i2);
  write(i2); write(i2);              //Initialize variables and loop variable.
  (read(i2);                         //Check loop condition.
  lock(l1); read(x3); write(x2); unlock(l1);
  lock(l2); read(y1); write(y2); unlock(l2);
  write(x2);
  read(i2); write(i2)) * 30            //Increment loop variable and repeat.
}

task T3 {malloc(i3);
  write(i3); write(i3);              //Initialize variables and loop variable.
  (read(i3);                         //Check loop condition.
  lock(l1); write(x3); unlock(l1);
  write(y3);
  lock(l3); read(z1); write(z3); unlock(l3);
  read(i3); write(i3)) * 50            //Increment loop variable and repeat.
}

main {
  write(x1); write(x2); write(x3); write(y1); write(y2); write(y3);
  write(z1); write(z2); write(z3);        //Initialize variables.
  spawn(T1); spawn(T2); spawn(T3)         //Spawning tasks.
}
```

Lst. 1.4: Running example: Automatically generated code in the Low-Level Programming Language (LLP).

ured and an initial layout may be defined via references associating variable names with memory locations. Each **`Device`** conceptually consists of a *core*, doing calculations, with an associated *cache* of configurable size that can buffer data from the *shared memory* to improve access times. Additionally, each device may declare a range of blocks from the shared memory as being local, which automatically makes the remaining memory remote. Thus, the model supports architectures with non-uniform memory access; remote memory has a more severe penalty for access. The RM artifact is used as input for the compilation process to the toolbox to perform simulation and analysis, where it is reflected by specific configuration values.

Listing 1.6 shows a possible resource model for the running example, in which variables **`x1,x2,x3`** have been allocated in the same memory block, which is local to the first device. Similarly, **`y1,y2,y3`** and **`z1,z2,z3`** have been allocated local to the second and third device, respectively. In addition, the local variables have been allocated in different memory blocks. In particular, we assume that lock variables are always taking one memory block, which is justified by the common practice of padding blocks to isolate certain data. We add an extra

```
RM ::= Memory Device+;
Memory ::= "memory" "{" "size" ":" <Integer> ";"
  ["references" "{" { <String> ":" <Integer> ";"} "}" ] "}";
Device ::= "device" "{" "cacheSize" ":" <Integer> ";"
  [ "localMemory" ":" <Integer> "-" <Integer> ";"] "}";
```

Lst. 1.5: Syntax of the Resource Model (RM).

```
memory {
  size: 20;

  references {
    x1 :  1; x2 :  1; x3 :  1; l1 :  3;
    y1 :  7; y2 :  7; y3 :  7; l2 :  9;
    z1 : 13; z2 : 13; z3 : 13; l3 : 15; l4 : 18;
  }
}

device {
  cacheSize: 5;
  localMemory: 0 - 5;
}

device {
  cacheSize: 5;
  localMemory: 6 - 11;
}

device {
  cacheSize: 5;
  localMemory: 12 - 17;
}
```

Lst. 1.6: Resource model of the running example.

lock `l4`, which will be used later to avoid the race conditions introduced by this particular memory layout, see Sec. 3.4 for further details.

3.4 Running the Example in the Toolbox

Recall from Fig. 1 that the toolbox takes as input the LLP program from Lst. 1.2 and the compiled version of the resource model from Lst. 1.6. We can now simulate the example. The toolbox takes care of coherent accesses to caches and shared memory while running the simulations. To measure the quality of the simulation, the tool introduces penalties for accesses to the memory system as a metric. Inspired by a real NUMA system [27], we let accesses to the local cache, local shared memory and remote shared memory differ by an order of magnitude and have penalties of 10, 100 and 1,000, respectively. Out of 1000 simulations, 175 are inconclusive after a timeout of six seconds. (Observe that inconclusive simulations within the time bound may be explained by ping pong effects betweens cores repeatedly requesting exclusive access to a memory block and invalidating each other without making progress.) We observed that the minimum, maximum and average penalty for the given inputs are 208 205, 479 605 and 351 705 respectively.

For comparison, we run a variant of the example that contains an extra lock surrounding the previous contents of the block of each task as shown in Lst. 1.7. This lock aims to remove the race conditions introduced by the allocator in the memory layout and to reduce the overall penalty. For these inputs, all 1000 simulations terminate, and we observed obviously better results with the minimum, maximum and average penalty of 108 311, 265 311 and 202 606, respectively.

```
task T1 {
  lock(l4); /* Previous T1 */ unlock(l4);
}

task T2 {
  lock(l4); /* Previous T2 */ unlock(l4);
}

task T3 {
  lock(l4); /* Previous T3 */ unlock(l4);
}

main {
  //Previous main
}
```

Lst. 1.7: Running example with additional locks in LLP, aiming to reduce race conditions due to the false sharing in the memory layout.

4 A Peek Inside the Toolbox: A Formal Model of Distributed Computations and Data Accesses

This section details the construction of the toolbox, realized as a proof-of concept implementation of a formal model in the rewriting logic system Maude [17]. The formal model captures a multicore architecture consisting of cores with a private cache and main memory with a NUMA design, as shown in Fig. 2.

Tasks, expressed in terms of data accesses, are scheduled for execution on available cores from a shared pool. Task execution in a core requires memory blocks to be transferred from main memory to the corresponding cache. The main memory where the needed blocks are residing, can be either local to the core or remote. Each cache has a queue of fetch/flush instructions to move memory blocks between cache and main memory. To ensure consistency between concurrently existing copies of data in caches and main memory, the model implements the MSI cache coherence protocol (e.g., [28]), a standard protocol from in the literature. In MSI, a cache line can be in one of the three states: **m**odified, **s**hared or **i**nvalid. A *modified* state indicates that the block in that cache line has the most recent value 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). This protocol broadcasts messages between the cores. We abstract from the specific shape of this broadcast (e.g., a mesh or a ring) in terms of a communication medium. Following standard nomenclature, *Rd* messages request read access to a memory block and *RdX* messages request exclusive read access to a memory block (for writing purposes), thereby invalidating other copies of the same block.

To read data from block n, the core first looks for n in its local cache. If n is not in the cache, then a read request $!Rd(n)$ is instantaneously broadcast to other caches and main memory. The cache *fetches* the block when it is available in main memory. Eviction is required if the cache is full. Writing to block n requires that n is in either shared or modified state in the local cache, an *invalidation request* $!RdX(n)$ is then instantaneously broadcast to obtain exclusive access. We use $?Rd(n)$ and $?RdX(n)$ to denote the reception of read and invalidation requests, respectively. 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; 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. A full formal description of this model can be found in [8].

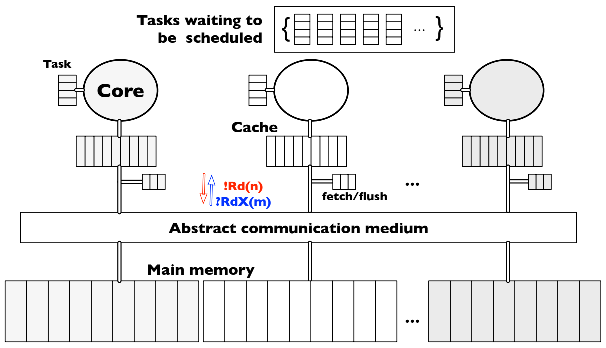

Fig. 2: A distributed layer of computations with coherent data accesses.

As a technical contribution of this paper, we have extended the model to support tasks with atomic sections implemented by *binary locks*. We model lock manipulations according to *test and test-and-set* instruction [3]: to take a lock located in block n, the core first checks the block locally in the cache. If the lock value is `0` (lock is free), it sends an invalidation message $!RdX(n)$ to the other components in the architecture and takes the lock; if the value is `1` (lock is taken), it waits until its local copy is invalidated to fetch the lock from main memory and repeats the process until it succeeds.

For simplicity, this model abstracts the actual data in memory blocks to symbolic values, except for the binary value of the locks. We assume that locks must be released before they are taken again. We also assume that a cache line has the same size as a memory block, and that blocks are transferred between cores via this main memory. Read and invalidation requests in the communication medium are broadcast instantaneously in our model, which captures *true concurrency* for an arbitrary number of cores in the proposed semantics.

4.1 Operational Semantics for Tasks with Binary Locks and Coherent Data Accesses

We extend previous work [8, 9] with binary locks. While the previous work describes a memory system with multilevel caches, this paper only considers single-level caches. This simplification is orthogonal to the extension.

Runtime syntax. A configuration *Config* consists of main memory M, shared among cores $\overline{CR}$ with their own caches $\overline{Ca}$, and a set of tasks $\overline{T}$ to be executed. A core $(Cid \bullet rst)$ has identifier *Cid* and executes runtime statements *rst*. A cache $(Cid \bullet M \bullet dst)$ consists of a core identifier to which it belongs, a local memory M and a sequence of data instructions *dst* with $\texttt{fetch}(n)$ and $\texttt{flush}(n)$ instructions to be performed. A memory $M : n \rightharpoonup \langle val, st\rangle$ maps address n to a pair of a stored value *val* and a status *st*. The status tags *mo*, *sh*, and *inv* refer to the three states of the MSI protocol. Blocks in main memory are in *sh* or *inv* state.

The task table $Tb : T \rightharpoonup dap$ associates task identifiers T to *data access patterns dap*, which are sequences of the basic operations described in Sec. 3. To ensure data consistency, a statement **`commit`** is added at the end of each task to flush the entire cache after execution. We assume that the task table is statically given and is always available. Cores execute *runtime statements rst*, which extend *dap* with additional statements $\texttt{readBl}(r)$, $\texttt{writeBl}(r)$, $\texttt{lockBl}(r)$ and $\texttt{unlockBl}(r)$ to indicate that the core is blocked, waiting for data to be fetched.

Semantics. The semantics is divided into local and global levels. The local semantics captures the execution of statements in each core and local state changes in each cache line according to the finite state controller enforcing the MSI protocol during execution. The global semantics captures the synchronization and coordination between the different components in a configuration. In an *initial* configuration *Config*, all blocks in main memory M have a default initial value with status *sh*, all locks are free, all cores are idle, all caches are empty, and the task pool in $\overline{T}$ has a single task representing the main block of a program. Let $Config \xrightarrow{*} Config'$ denote an execution starting from *Config* and reaching $Config'$ by recursively applying global transition rules, which in turn apply local transition rules for each component. To give a feel for the operational semantics within the available space, we focus on the subset of local rules extending previous work; the full semantics can be found in an accompanying technical report [10].

The local transition rules for lock manipulations are given in Fig. 3, where the function $addr(r)$ returns the address n of the block containing reference r and $status(M, n)$ the status of n in memory M. A task running in a core can take a lock with reference r when its value in the local cache is 0 (i.e., free) and the status of block n, where $n = addr(r)$, is either *sh* or *mo*, see rules $\textsc{Lock}_1$ and $\textsc{Lock}_2$. After a lock is taken, the value and status of block n in the local cache are updated to 1 (taken) and *mo* (modified), respectively. Taking or releasing a lock implies writing to a reference and getting exclusive access to the block. If the status in the local cache is shared, the core broadcasts message $!RdX(n)$ to get exclusive access to n, see $\textsc{Lock}_2$. If the requested lock is not available in the cache, the core first broadcasts a $!Rd(n)$ message and appends a **`fetch`** instruction to

$$\frac{n = addr(r)}{\begin{array}{c}(c \bullet M[n \mapsto \langle k, 0, mo\rangle] \bullet dst) \circ (c \bullet \texttt{lock}(r); rst) \\ \to (c \bullet M[n \mapsto \langle k, 1, mo\rangle] \bullet dst) \circ (c \bullet rst)\end{array}} \quad (\textsc{Lock}_1)$$

$$\frac{n = addr(r)}{\begin{array}{c}(c \bullet M[n \mapsto \langle k, 0, sh\rangle] \bullet dst) \circ (c \bullet \texttt{lock}(r); rst) \\ \xrightarrow{!RdX(n)} (c \bullet M[n \mapsto \langle k, 1, mo\rangle] \bullet dst) \circ (c \bullet rst)\end{array}} \quad (\textsc{Lock}_2)$$

$$\frac{n = addr(r) \qquad status(M, n) = inv \vee n \notin dom(M)}{\begin{array}{c}(c \bullet M \bullet dst) \circ (c \bullet \texttt{lock}(r); rst) \\ \xrightarrow{!Rd(n)} (c \bullet M \setminus n \bullet dst; \texttt{fetch}(n)) \circ (c \bullet \texttt{lockBl}(r); rst)\end{array}} \quad (\textsc{LockBlock}_1)$$

$$\frac{n = addr(r)}{\begin{array}{c}(c \bullet M[n \mapsto \langle k, 0, sh\rangle] \bullet dst) \circ (c \bullet \texttt{lockBl}(r); rst) \\ \xrightarrow{!RdX(n)} (c \bullet M[n \mapsto \langle k, 1, mo\rangle] \bullet dst) \circ (c \bullet rst)\end{array}} \quad (\textsc{LockBlock}_2)$$

$$\frac{n = addr(r) \qquad status(M, n) = inv}{\begin{array}{c}(c \bullet M \bullet dst) \circ (c \bullet \texttt{lockBl}(r); rst) \\ \xrightarrow{!Rd(n)} (c \bullet M \setminus n \bullet dst; \texttt{fetch}(n)) \circ (c \bullet \texttt{lockBl}(r); rst)\end{array}} \quad (\textsc{LockBlock}_3)$$

$$\frac{n = addr(r)}{\begin{array}{c}(c \bullet M[n \mapsto \langle k, 1, mo\rangle] \bullet dst) \circ (c \bullet \texttt{unlock}(r); rst) \\ \to (c \bullet M[n \mapsto \langle k, 0, mo\rangle] \bullet dst) \circ (c \bullet rst)\end{array}} \quad (\textsc{Unlock}_1)$$

$$\frac{n = addr(r)}{\begin{array}{c}(c \bullet M[n \mapsto \langle k, 1, sh\rangle] \bullet dst) \circ (c \bullet \texttt{unlock}(r); rst) \\ \xrightarrow{!RdX(n)} (c \bullet M[n \mapsto \langle k, 0, mo\rangle] \bullet dst) \circ (c \bullet rst)\end{array}} \quad (\textsc{Unlock}_2)$$

$$\frac{n = addr(r) \qquad status(M, n) = inv \vee n \notin dom(M)}{\begin{array}{c}(c \bullet M \bullet dst) \circ (c \bullet \texttt{unlock}(r); rst) \\ \xrightarrow{!Rd(n)} (c \bullet M \setminus n \bullet dst; \texttt{fetch}(n)) \circ (c \bullet \texttt{unlockBl}(r); rst)\end{array}} \quad (\textsc{UnLockBlock}_1)$$

$$\frac{n = addr(r)}{\begin{array}{c}(c \bullet M[n \mapsto \langle k, 1, sh\rangle] \bullet dst) \circ (c \bullet \texttt{unlockBl}(r); rst) \\ \xrightarrow{!RdX(n)} (c \bullet M[n \mapsto \langle k, 0, mo\rangle] \bullet dst) \circ (c \bullet rst)\end{array}} \quad (\textsc{UnLockBlock}_2)$$

Fig. 3: Local semantics for taking and releasing locks in cache coherent multicore architectures.

fetch the block in $\textsc{LockBlock}_1$. The execution is then blocked by the statement **lockBl**(r) until the lock is available in the local cache, see $\textsc{LockBlock}_2$. A new read request message $!Rd(n)$ occurs if the lock has been invalidated after it has been fetched while the core was blocked, see $\textsc{LockBlock}_3$. Releasing a lock is similar to locking, except that a core can only release a lock r that it owns (i.e., *val* equals 1 in the local cache), see UNLOCK$_1$ and UNLOCK$_2$. Observe that only the core that owns the lock can release it. During execution, a block n may be evicted from the cache to give space to another block, and may therefore need to be fetched again. This is handled by UNLOCKBLOCK$_1$ and UNLOCKBLOCK$_2$.

4.2 Correctness of the Model

We evaluated the correctness of the model with respect to standard correctness properties for data consistency and cache coherence [18, 29] including: (1) the result of any execution of the global system is equivalent to interleaving the results of the data access from each core in some serial order, (2) task execution preserves program order, and (3) for all memory blocks and for any synchronous or asynchronous parallel global step, cores cannot access stale data. The formal formulation and full proof of these properties can be found in the accompanying technical report [10]. These properties show that the formal model correctly captures a coherent multicore memory system and are following the same idea of the proofs shown in previous work by the authors [8, 9].

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

Definition 1 (Taken lock). *Let $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR}$ be a global configuration, $CR_i \in \overline{CR}$, $Ca_i \in \overline{Ca}$ a cache such that $Ca_i = (c_i \bullet M_i \bullet dst_i)$ and $belongs(Ca_i, CR_i)$. Then, a lock with address n is considered to be taken by a core if and only if either*

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

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

Lemma 1 (Mutual exclusion). *Let a configuration $M \circ \overline{T} \circ \overline{Ca} \circ \overline{CR}$ be reachable from an initial configuration. For $CR_i \in \overline{CR}$, let $CR_i = (c_i \bullet rst_i)$ and for $Ca_i \in \overline{Ca}$, $Ca_i = (c_i \bullet M_i \bullet dst_i)$. Consider a lock reference r with $addr(r) = n$.*

*If $(c_i \bullet M_i \bullet dst_i) \circ (c_i \bullet rst_i) \rightarrow (c_i \bullet M'_i \bullet dst'_i) \circ (c_i \bullet rst'_i)$ such that $rst_i \neq$ **`unlock`**$(r); rst''_i$ or $rst_i \neq$ **`unlockBl`**$(r); rst''_i$ for any rst''_i, then (a) $n \notin dom(M_i)$ or (b) $value(M_i, n) = value(M'_i, n)$.*

Proof (Sketch). We show that the property holds for initial configurations and proceed by induction on the transition rules, see [10]. □

5 Related Work

Taylor et al. [30] point out that the deployment view of a software system's architecture can be critical in assessing whether the system will be able to satisfy its requirements. In order to model deployment decisions, we need modelling abstractions that capture relevant aspects of the system structure. Programming languages like ArchJava [2] and Koala [31] take steps in this direction by integrating architectural concepts such as components and connectors into the design, allowing high-level logical structure to be expressed inside programs. To support deployment decisions, similar concepts are needed to express properties of physical or virtual devices. For cloud-deployed software, the modelling of resource management strategies can be done in terms of *deployment components* with associated resources [22], which are first-class modelling concepts in ABS [21]. For cyber-physical systems there is a similarly recognized need capture that computation and communication take time, which can be addressed by platform models to support design space exploration [19]. Our work in this paper is motivated by this need to express deployment decisions early in program construction. Rather than directly addressing the timing aspects of communication, we have opted for a more abstract approach, using a metric of penalties for data access patterns reflecting the program deployed with a given data layout on *shared memory multiprocessor architectures.*

Methods for model-based performance prediction which build custom performance models can generally be classified as being based on queuing networks, process algebra, Petri nets, simulations or stochastic models [7]. For multicore systems, simulation based approaches are most common. For example, a recent model-driven engineering methodology for deployment optimisation of task allocations for multicore embedded systems at design time [15, 16] is based on the UML MARTE profile for Modeling and Analysis of Real-Time and Embedded systems. This methodology uses model transformation techniques to optimise task allocation by refining the model of task allocations according to the simulation results. This work is complementary to our work by their focus on optimizing task allocation, which is non-deterministic in our model. Furthermore, in contrast to our work they do not focus on data access or caches.

Many simulation tools have been developed to analyse the performance of parallel programs running on multicore architectures. For instance gem5 [11] performs 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. Tools such as Graphite [24] and Sniper [12] run programs on distributed clusters to simulate executions on multicore architectures with thousands of cores. Compared to our work, these simulation tools take as their starting point the finished program and do not expose how data accesses affect the performance of a parallel program for a particular choice of data layout.

6 Conclusion and Future Work

This paper argues that deployment decisions should be modelled and analyzed as part of by-construction program development. Although common practice tackles deployment after validation, resulting performance problems may in the worst case require considerable changes in design and even impact the requirements level. To move deployment decisions earlier, models need to express platform artefacts and how the logical structure of the program maps to these artefacts.

To illustrate deployment modelling and decisions, this paper presents a toolbox for simulation, comparison and analysis of the effects of deployment decision for a particular parallel program when interacting with existing memory structures. We make this toolbox available to users by providing two programming languages at different levels of abstraction as well as a notation for resource models to specify memory availability, distribution, sharing and layout. Although the content of the toolbox can be ignored from the developer's perspective, it can be formalizes as an operational semantics for data access patterns executing in parallel on different cores and moving data between shared memory and local caches, and embodies a cache coherence protocol for data consistency. We show that the model guarantees correctness properties concerning data consistency and protected access to atomic sections.

As a next step, we plan to enrich the data access patterns and data layout to support more complex data structures and their dynamic allocation in memory (e.g., object creation). This opens for extracting data access patterns from richer high-level languages such as parallel object-oriented languages. Another interesting direction is to extend the architecture to support shared caches. It is also interesting to relate our high-level language (HLP) to existing correctness by construction formalisms such as Unity [13, 14]. 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. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. J. Aldrich, C. Chambers, and D. Notkin. ArchJava: connecting software architecture to implementation. In W. Tracz, M. Young, and J. Magee, editors, *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*, pages 187–197. ACM, 2002.
3. G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison Wesley, 2000.
4. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In R. L. Probert, N. A. Lynch, and N. Santoro, editors, *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 131–142. ACM, 1983.
5. R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Sci. Comput. Program.*, 13(1):133–180, 1989.
6. R.-J. J. Back and J. V. Wright. *Refinement Calculus: A Systematic Introduction.* Springer, 1st edition, 1998.
7. S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: a survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, May 2004.
8. 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). Research report, Department of Informatics, University of Oslo, 2017. URL `http://violet.at.ifi.uio.no/papers/mc-rr.pdf`.
9. S. Bijo, E. B. Johnsen, K. I Pun, and S. L. Tapia Tarifa. A formal model of parallel execution on multicore architectures with multilevel caches. In *Proc. 14th*

Intl. Conf. on Formal Aspects of Component Software (FACS 2017), volume 10487 of *Lecture Notes in Computer Science*, pages 58–77. Springer, 2017.

10. S. Bijo, K. I Pun, and S. L. Tapia Tarifa. Modelling data access patterns with atomic sections for multicore architectures (long version). Research report, Department of Informatics, University of Oslo, 2017. URL `http://violet.at.ifi.uio.no/papers/mc-lock-rr.pdf`.
11. N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, 2011.
12. T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 52:1–52:12. ACM, 2011.
13. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.
14. K. M. Chandy and S. Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett, 1991.
15. F. Ciccozzi, D. Corcoran, T. Seceleanu, and D. Scholle. Smartcore: Boosting model-driven engineering of embedded systems for multicore. In *Proceedings of the 12th International Conference on Information Technology – New Generations*, pages 89–94. IEEE Computer Society, 2015.
16. F. Ciccozzi, J. Feljan, J. Carlson, and I. Crnković. Architecture optimization: speed or accuracy? both! *Software Quality Journal*, Nov 2016.
17. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
18. D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers Inc., 1st edition, 1997.
19. P. Derler, E. A. Lee, and A. L. Sangiovanni-Vincentelli. Modeling cyber-physical systems. *Proceedings of the IEEE*, 100(1):13–28, 2012.
20. R. Hähnle and E. B. Johnsen. Designing resource-aware cloud applications. *IEEE Computer*, 48(6):72–75, 2015.
21. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011.
22. E. B. Johnsen, R. Schlatte, and S. L. T. Tarifa. Integrating deployment architectures and resource consumption in timed object-oriented models. *J. Log. Algebr. Meth. Program.*, 84(1):67–91, 2015.
23. D. G. Kourie and B. W. Watson. *The Correctness-by-Construction Approach to Programming.* Springer, 2012.
24. J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12. IEEE Computer Society, 2010.
25. C. Morgan. *Programming from specifications, 2nd Edition.* Prentice Hall International Series in computer science. Prentice Hall, 1994.

26. C. Morgan. *Programming from Specifications.* Prentice-Hall International, 1998.
27. D. Schmidl, A. Vesterkjær, and M. S. Müller. Evaluating OpenMP performance on thousands of cores on the Numascale architecture. In *PARCO*, volume 27 of *Advances in Parallel Computing*, pages 83–92. IOS Press, 2015.
28. Y. Solihin. *Fundamentals of Parallel Multicore Architecture.* Chapman & Hall/CRC, 1st edition, 2015.
29. D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence.* Morgan & Claypool Publishers, 1st edition, 2011.
30. R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice.* Wiley Publishing, 2009.
31. R. C. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, 2000.