Data Dependency based Parallel Simulation of Wireless Networks

Mirko Stoffers*, Torsten Sehy*, James Gross‡, Klaus Wehrle*
*Communication and Distributed Systems, RWTH Aachen University
‡School of Electrical Engineering, KTH Royal Institute of Technology
stoffers@comsys.rwth-aachen.de

ABSTRACT

Simulation of wireless systems is highly complex and can only be efficient if the simulation is executed in parallel. To this end, independent events have to be identified to enable their simultaneous execution. Hence, the number of events identified as independent needs to be maximized in order to increase the level of parallelism. Traditionally, dependencies are determined only by time and location of events: If two events take place on the same simulation entity, they must be simulated in timestamp order. Our approach to overcome this limitation is to also investigate data-dependencies between events. This enables event reordering and parallelization even for events at the same simulation entity. To this end, we design the simulation language PSimLa, which aids this process. In this paper, we discuss the PSimLa design and compiler as well as our data-dependency analysis approach in detail and present case studies of wireless network models, speeded up by a factor of 10 on 12 cores where time-based parallelization only achieves a 1.6x speedup.

Categories and Subject Descriptors

I.6.2 [**Simulation and Modeling**]: Simulation Languages

General Terms

Languages, Performance, Algorithms

Keywords

Parallel simulation; Static code analysis; Data dependencies

1. INTRODUCTION

Simulation is an essential methodology in the design and development of wireless communication systems. However, wireless simulation models are particularly complex, as they need to reflect sophisticated physical processes in software. Hence, parallelization is necessary to retrieve results in time.

All traditional parallelization approaches base on the local causality constraint, fulfilled "if and only if each Logical Process (LP) processes events in nondecreasing timestamp order" [9, p. 32]. However, we argue that this constraint is not a necessary condition, but two events can be independent even if their re-ordering poses a causal violation. By analyzing data-dependencies at compile time it can be determined if the two events do actually not access a data item in a conflicting manner. Since the only existing approach by Chen et al. [4, 5] does not incorporate the challenging yet essential part of analyzing data access by pointers or references we propose a novel approach to data-dependency analysis for parallelizing network simulation. We suggest [18] to develop a Domain-Specific Language (DSL) for data-dependency based parallel simulation, but keep it close to C++ to maintain a flat learning curve for model developers. However, to avoid problems rendering data access tracking infeasible, like the unsolved problem of pointer analysis [2, 10], our language differs from C++, but does not remove a feature without providing proper alternatives. We allow using C++ and our language in a single model to enable smooth transition for existing code. In particular, we make the following contributions in this paper:

1. We discuss PSimLa, a language similar and compatible to C++, but replacing features of C++ by alternative concepts to increase static analyzability of model code.
2. We discuss the details of our data-dependency analysis approach. This shows, that in fact structured languages can be analyzed by tracking data access, increasing the amount of events recognized as independent and enhancing the degree of parallelism of otherwise hard-to-parallelize simulation models.

We integrate our approaches into OMNeT++ and the parallelization framework Horizon [13, 14]. OMNeT++ is one of the most commonly used open-source simulation frameworks. The shared-memory architecture of Horizon maximizes the applicability of the analysis results due to the possibility to efficiently determine the events currently being processed. We perform a detailed evaluation of this approach and discuss general applicability. Our evaluation shows that certain previously hard-to-parallelize wireless simulation models achieve almost linear speedup when data-dependency information is exploited. On a 12-core machine our approach is more than 3 times faster than time-based parallelization for a wireless mesh case study, and more than 6 times for an LTE simulation.

The remainder of this paper is structured as follows: After analyzing the problem (Sec. 2), we discuss the design of PSimLa (Sec. 3) in more detail. We show the feasibility of static analysis by introducing a data-dependency analysis algorithm (Sec. 4). We discuss important issues (Sec. 5) before we analyze evaluation results (Sec. 6). Finally, we compare our approach to existing simulation languages and analysis approaches (Sec. 7) before our conclusion (Sec. 8).

MSWiM'15, November 2–6, 2015, Cancún, Mexico.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3762-5/15/11 ...$15.00.
DOI: http://dx.doi.org/10.1145/2811587.2811593.

2. PROBLEM ANALYSIS

Static code analysis has been commonly used to investigate different properties of computer programs in many languages [2]. However, certain properties of the languages render the analysis easier, harder, or infeasible. The absence of side effects in functional programs, e. g., enables trivial detection of function in- and output. On the other hand, pointer analysis is still considered unsolved [2, 10], hence reliably tracking data access in pointer-based languages like C++ has to be considered infeasible. Though smart pointers and unique pointers mitigate the problems, this still holds for recent C++ standards, especially because raw pointers are still available and necessary. Since most wireless simulation models are written in structured languages and many model developers are not familiar with functional programming, limiting data-dependency based parallel simulation to functional languages is not an option. Instead, we investigate the applicability on structured programming languages.

The infeasibility of pointer analysis excludes languages which heavily rely on pointers. Global variables are considered harmful in parallel programs: in shared-memory systems where they might induce race conditions and in distributed systems, whose entities cannot share the state of a global variable. Since today's wide-spread languages do not fulfill those requirements, we decided to modify such a language to support data-dependency analysis. Many model developers are familiar with C++ as the most commonly used network simulators OMNeT++ and ns-3 base on C++. However, to support data-dependency analysis, global variables and pointer support have to be removed and a proper alternative for pointers needs to be provided to avoid rendering the language useless.

Data-dependency analysis has been extensively studied in the compiler construction domain to enable automatic parallelization of arbitrary programs. However, these approaches do not incorporate the notion of simulation time and discrete events into their analysis, hence they would not exploit the full potential in this scope. Instead, we develop an approach that applies similar analyses, but as well accounts for the peculiarities of discrete event simulation.

3. THE PSimLa LANGUAGE

PSimLa is designed according to the goals elaborated [18]. It is a Turing complete language with particular emphasize on static analyzability for data-dependencies. Similarity and compatibility to C++ ensures that existing models can be smoothly translated and the learning curve is kept as flat as possible. In the following, we describe the compilation process and introduce the building blocks of PSimLa in detail.

3.1 Compilation Process

PSimLa is based on C++ and the simulation elements of OMNeT++. Code-to-code translation to C++ enables flexible linking of PSimLa modules with both C++ code and C++-compatible libraries. The compilation runs in 5 steps (cf. Fig. 1). The developer (1) creates code in PSimLa and/or C++ and the configuration in the OMNeT++ NED-, MSG-, and INI-format. Our parser which checks the syntax of the PSimLa code and creates an internal representation similar to an abstract syntax tree (2). The static code analysis (see Sec. 4 for more details) can then use this representation to identify event dependencies and independencies (3). The abstract syntax tree is then serialized into C++ code

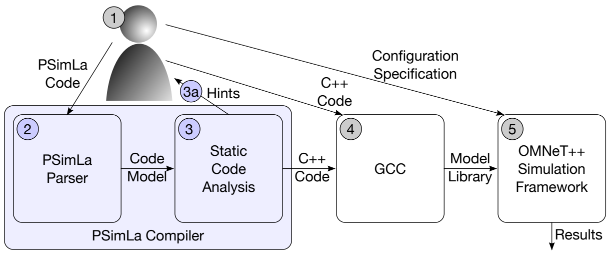


Figure 1: The PSimLa compilation process. The developer implements the model (1), triggers translation of PSimLa code into C++ code (2, 3), triggers compilation of C++ code (4), and executes the simulation using OMNeT++ (5).

effectively resulting in a code-to-code translation of PSimLa into C++. Additionally, the analysis yields C++ code representing the gained dependency information that can be used during runtime for parallelization. If it detects event dependencies, which the developer might be able to resolve manually, it provides hints to the modeler (3a) that can be used to improve upon the model in a next iteration. The final result of step 3 together with the C++ code and configuration specification of the developer poses a simulation model complying with the OMNeT++ specification. This model can be compiled (4) and executed by OMNeT++ (5). A modified version of OMNeT++ (see Sec. 4.6) uses the results of the static analysis to improve the parallelization.

3.2 PSimLa Building Blocks

In [18], we describe the basic building blocks of PSimLa, which correspond to those of OMNeT++. A PSimLa *Module*, corresponding to an OMNeT++ Simple Module, can be specified either in PSimLa or – to maintain compatibility with existing code – in C++. Modules can be connected into Compound Modules or a Network by OMNeT++ NED-files. Data can only be exchanged between Modules by Messages (OMNeT++ MSG-format). Modelers cannot create global variables. Classes can be used to define complex data types and are specified in PSimLa or C++. In the following, we discuss each building block in more detail.

Modules: A Module is defined similar to a C++ class, but with the keyword `module`. In the body, developers include both behavior implementation and parameter and gate specification. The syntax of parameter and gate specification is adopted from OMNeT++. Like a Simple Module in OMNeT++ a PSimLa Module needs to implement an event handler and can implement initialization and teardown functions. From the Module code, the compiler generates an OMNeT++ NED-file as well as C++ code and header files.

Data Types: Like C++, we provide primitive data types and enumerations, and developers can create classes. We also provide container formats from the C++ standard library as language built-ins to increase analyzability due to the possibility to exploit the container semantics in the analysis algorithm. We include the containers `vector` (dynamically resized array) and `queue` (FIFO queue) with the option to include more into the compiler implementation.

Memory Management: To increase the level of analyzability, the memory management is handled by the compiler rather than the developer. Hence, the developer is only provided references to objects. Like Java, we hand object references to functions (call-by-reference). To monitor an object's life cycle, we use reference counting and delete objects as soon as the reference count reaches zero, i. e., no more references to the object persist. Additionally, only one reference must exist to data items included in a message to ensure that no two independent events can access the same data item concurrently. To avoid that reference loops prevent correct cleanup, we adopt the weak pointer concept.

Our prototype implementation of the PSimLa compiler chooses the storage location (dynamic or automatic memory, usually implemented as heap and stack) similar to Java. Local variables of primitive data types are placed in automatic memory, objects and arrays are dynamically allocated. This decreases performance compared to optimized C++ code (see Sec. 6.1), hence future compiler implementations should improve the algorithm to select the best location. However, this does not affect the question how well data-dependency analysis is suited for speeding up parallel simulation.

Libraries: Due to our compatibility with C++, existing libraries with C++ bindings (like the C++ standard library) can be linked with PSimLa models. Nevertheless, PSimLa built-in data types (like FIFO queues) should be preferred over external libraries to increase the code analyzability.

Code Elements: The syntax of PSimLa functions is basically similar to the C++ syntax, except that – as discussed above – the memory is not managed by the user, hence the user cannot use pointers. We provide the well-known `include` directive to include C++ header files to bind existing C++ code. Inside any PSimLa function, inline C++ code blocks can be created. To ease model development, we provide language built-ins for simulation related operations like message transmission or random number generation.

4. ANALYSIS TECHNIQUES

In this section we show the feasibility of static code analysis of PSimLa by discussing our data-dependency analysis approach in detail. We sketched the idea shortly in [18].

To this end, our analysis approach targets three goals:

Maximize Recognition Rate: Our primary goal is to successfully detect as many independent events as possible since this increases the parallelization gain.

Minimize Runtime Overhead: While our static code analysis determines the event dependencies at compile time, the gained knowledge has to be applied during runtime. To this end, it is necessary to determine whether it is safe to execute events in parallel by querying the dependency information. While we cannot completely avoid runtime overhead, we aim at preparing as much as possible at compile time.

Maintain Correctness: The analysis must not recognize two dependent events as independent to avoid false simulation results. Our approach must compute the same results as sequential execution of the code. However, we allow a relaxation to this constraint that can be activated or deactivated by the user: If two events draw a random number from the same Random Number Generator (RNG) stream, there is in fact a dependence between the two events. By executing those events in reverse order the random numbers provided to the events are swapped. Nevertheless, both numbers stem from the same distribution. Hence, if the model only expects a random number following a certain distribution (rather than the next random number from a certain RNG stream), the events could be executed out-of-order (or in parallel if the RNG source is thread-safe or locked). If we waive the guarantee of assigning the same random number to the same event in every execution, we still obtain valid results, but lose repeatability. Consequently, this relaxation switch allows the user to trade repeatability for performance.

In the following we first introduce the analysis approach in general before we discuss the details of each part.

4.1 General Approach

To elaborate promising analysis approaches, we post a simple formalism for parallel simulation. The focus hereby is to determine whether two events are independent or not. We assume that the simulation can be decomposed into a set of partitions which communicate with each other only by message passing. These partitions comply with LPs in distributed simulations. In PSimLa, every Module can be a partition since Modules can only communicate via messages.

We define an event as a 3-tuple, such that the set of all possible events is $E := (T \times P \times D)$. T is the time domain, P the set of partitions, and elements in D denote the data access pattern of an event, i. e., which data element is accessed how. The exact definition of D is provided in Sec. 4.3. Let $s \subseteq (E \times E)$ be the scheduling relation with $(e_1, e_2) \in s$ iff e_2 is created by e_1. The reflexive, transitive closure s^* yields all directly and transitively created events and the event itself.

To formalize independence, we define the operator $\perp \subseteq (E \times E)$ with $e_1 \perp e_2$ denoting that e_1 and e_2 are independent.

In traditional Parallel Discrete Event Simulation (PDES) the approaches differ in the way they predict whether two events on *different* partitions eventually induce a dependency. However, for two events on the *same* partition they all check the local causality constraint [9], i. e., they are always assumed to depend on each other. Hence, for two events $e_1 = (t_1, p_1, d_1) \in E, e_2 = (t_2, p_2, d_2) \in E$ traditional PDES assumes: $p_1 = p_2 \Rightarrow e_1 \not\perp e_2$.

However, while this assumption is safe, it is not true in general. In fact, the events can still be independent if the effects of e_1 do not influence e_2 and vice versa. We introduce the operator $\succ \subseteq (D \times E)$ with $d \succ e$ indicating that the behavior of e is influenced by a write operation in d.

We can then derive the following additional method to determine independence of e_1 and e_2. Without loss of generality we assume $t_1 \leq t_2$. If first, e_2 is neither affected by e_1 nor any event created by e_1, and second, none of those events (e_1 and the events created by e_1) is influenced by e_2, then e_1 and e_2 are in fact independent: $(\forall (e_1, e') = (e_1, (t', \_, d')) \in s^* : t' \leq t_2 \wedge d' \not\succ e_2 \wedge d_2 \not\succ e') \Rightarrow e_1 \perp e_2$.

Hence, to determine the independence of two events it is crucial to analyze not only time and location of each event, but also its scheduling behavior and data access pattern.

To this end, our approach is to analyze the event handler code in order to categorize event types and identify conflicts with other types based on the data accessed. In the following, we discuss the five steps of our analysis in detail.

4.2 Identifying Event Types

In a first step, at compile time we identify different types of events in the provided simulation model. This allows us to specify dependency rules on an event type basis. At runtime we then only need to determine the type of an event in order to be able to investigate the dependency rules.

The common programming model in OMNeT++ is to provide a single message handler per module and branch upon different attributes of the message delivered to that handler. For example, an event handler at a router might branch upon the message kind (routing message, data packet) and one or more of the branches might branch again upon the receiving

```
1 void handleMessage(Message msg) {
2   if(msg.getKind()==1)
3     myInt=0;
4   if(msg.getKind()==2)
5     myInt=myFn(myInt);
6   sendDelayed(msg,0,"myOutputGate");
7 }
```

Figure 2: Event Handler of Example PSimLa Module `MyMod`.

interface (i. e., distinguish packets from the upper layer from packets from the lower layer).

Our event type classification algorithm starts with a single event type per module. We then analyze the event handler code of each module for any branch solely based on a message attribute. The first conditional encountered results in splitting the event type into two different types, one with the condition being fulfilled and one for the opposite. Similarly, a switch statement results in multiple event types, one for each case of the switch statement. For each branch we repeat this procedure possibly splitting the corresponding event type again until no more message attribute based branches are detected. Hence, the branches form a tree and each leaf of that tree corresponds to exactly one event type.

We extend our formalism by the notion of event types. We define Θ as the set of all detected event types and the function $\tau : E \to \Theta$ assigning each event instance its type.

There are three types identified for each instance of the module MyMod in the example in Fig. 2: One for message kind 1, one for message kind 2, and one for message kind being neither 1 nor 2. Note that at compile time we do not know how many instances will be created of each module. Hence, the set Θ is rather a theoretic set that is not exactly known to the analysis algorithm. Instead the analysis only stores the criteria to distinguish event types. For the example, we assume that there is only one instance of MyMod, called mymod, and the output of that instance is connected to its input. Hence, we only have three event types in the simulation. In the following, we refer to those types as $\vartheta_{\text{mymod1}}$ to $\vartheta_{\text{mymod3}}$, respectively.

4.3 Tracking Data Access

For every event type, we first determine which code blocks the program flow passes if an event of this type is executed. This includes any code outside of conditionals, as well as either of the two (potentially empty) blocks of a conditional. For conditionals not branching upon message attributes and therefore not resulting in different event types, we consider both blocks, i. e., we assume the worst case here. This is necessary since not every condition can be evaluated at compile time. Hence, we apply a conservative simplification here. Similarly, we assume that loops are always executed at least once even though the loop termination condition might already be false before the loop is entered for the first time.

In the example, $\vartheta_{\text{mymod1}}$ passes lines 3 and 6, $\vartheta_{\text{mymod2}}$ passes lines 5 and 6, and $\vartheta_{\text{mymod3}}$ passes line 6 only.

For every code block passed by the program flow of each event type, we determine which data items are accessed. Since PSimLa does not provide global variables, data items can only be accessed by two means: First, they can be an element of the current module or (directly or indirectly) referred to by a reference in the module. Second, data items can be an element of the message. Local variables cannot cause conflicts with concurrently running events since they cannot be accessed by the concurrent event. Since messages must not reference to data items that are still referenced from somewhere else, all items of a message can be treated as local variables. Hence, we only need to investigate items accessed via module members. Additionally, if a reference is copied into a local variable, we have to treat this local variable like the original reference.

For each event type $\vartheta \in \Theta$, we can now create a list L_ϑ of accessed data items by parsing the code line by line and performing the following actions for certain statements.

- On a *function call* we retrieve the code of the called function and parse that code as well.
- On *creating a local reference variable* R_l, we create an initially empty list $\pi(R_\text{l})$ of possible locations this variable might point to. Although during program execution a reference can only point to a single location at any point in time, we need a list of potential locations since we cannot evaluate every conditional, hence there might be more than one option.
- On *copying a reference* R_m referred to by a *module variable* to a local reference R_l, we add R_m to $\pi(R_\text{l})$.
- On *copying a local reference* R_1 to another local reference R_2 we append $\pi(R_1)$ to $\pi(R_2)$.
- If a *local variable of primitive data type* is accessed, we do nothing since this can never cause a conflict.
- If an *object referenced by a local reference variable* R_l is accessed, we append $\pi(R_\text{l})$ to L_ϑ. Note that references are also special cases of objects. Reading or modifying a member variable of a module is as well a relevant operation if this member variable is a reference.
- If an *object referenced by the module* is accessed, we append this object to L_ϑ.

During this procedure we do not only track which objects are accessed, but also how they are accessed. To this end, we distinguish between references, primitive data types, arrays, and certain special data items for language built-ins.

For *references* we track the following two operations:

Dereferencing: If a reference is dereferenced in order to find an object, the reference itself is read.

Update: If a reference is modified, this poses a write operation on the reference itself.

For *primitive data items*, we distinguish between:

Read: The variable value is read, but not modified.

Write: The value is modified in any way but the following.

Increment: The value is incremented or decremented by one of the operators `++`, `--`, `+=`, or `-=`. Such accesses are modifications, but can be re-ordered without changing the final result if performed in a thread-safe manner.

Since it is not generally possible to determine the index used to access an element of an *array* (as well as similar index-accessible containers) during compile time, we do not treat each item of an array separately, but treat the whole array as a single item. Hence, we distinguish between read and write operations on the array.

Our analysis recognizes two *language built-ins* that are handled as special data items: RNGs and queues. This list can be extended by adding further objects as language built-ins to PSimLa and implementing the corresponding analysis passes according to the semantics of these objects. For *RNGs* there is only a single operation: drawing a random number. This is treated according to the relaxation switch discussed in the analysis goals. With strict RNG ordering, RNG accesses need to be treated like write operations. With relaxed ordering, the OMNeT++ RNGs are locked and accesses are handled like read-only operations.

For *queues*, we distinguish between:

Enqueue: Appending at the end of the queue.

Dequeue: Retrieving and removing the first item.

Query Size: Determine the size of the queue. This also includes special cases like emptiness checks (size=0).

Now, for each event type we maintain a list of objects accessed and for each access the operation performed.

We now specify the data access pattern set D previously introduced in Sec. 4.1 as $D := \mathcal{P}(\Delta \times A)$ with Δ being the set of all data items in the simulation and A the set of all access operations discussed above. While d is the actual data access pattern of $e = (\_, \_, d) \in E$, we introduce d'_ϑ as a conservative over-estimation of data accesses by events of type ϑ. This reflects the inability of exactly predicting every data access in a Turing-complete programming language at compile time. For every event type ϑ our analysis then determines a set d'_ϑ such that $\forall e = (\_, \_, d) \in E : (\tau(e) = \vartheta \Rightarrow d'_\vartheta \supseteq d)$.

For convenience, we define $r \mathrel{\widehat{=}}$ "read primitive data type" and $w \mathrel{\widehat{=}}$ "write primitive data type". For the provided example our analysis determines $d'_{\vartheta_{\text{mymod1}}} := \{(\text{myInt}, w)\}$, $d'_{\vartheta_{\text{mymod2}}} := \{(\text{myInt}, r), (\text{myInt}, w), (\text{myParam}, r)\}$, and for the third event type $d'_{\vartheta_{\text{mymod3}}} := \emptyset$.

4.4 Determining Scheduling Relations

For the independence criterion derived in Sec. 4.1 we need to additionally determine the scheduling relation s. Like d, we cannot determine s exactly due to the inability to analyze every property of any Turing-computable function. Instead, we define the relation $s' \subseteq (\Theta \times \Theta)$ with the requirement $(e_1, e_2) \in s \Rightarrow (\tau(e_1), \tau(e_2)) \in s'$. However, the opposite does not necessarily need to be true. This means, s' is a conservative over-estimation of s on event type basis.

In order to derive the scheduling relations, we follow a similar procedure as to derive the data access patterns (see Sec. 4.3). For every event type, we search the reachable code blocks for calls to scheduling built-ins of PSimLa. On occurance of such a built-in, we need to determine the type of the newly created event. Since this type depends on the attributes of the message (see Sec. 4.2), we need to investigate which attributes might be set. This can be an easy task if the attributes are set closely to the scheduling without conditionals that cannot be evaluated during compile time. However, it is not always possible to exactly determine the value of a message property at compile time in a Turing-complete language. Hence, we allow wildcards for message properties. If the message handler of the receiving module branches on a property that contains a wildcard, we have to consider both event types as potential options.

In our simple example, all three event types create a new event by the code in line 6. Since the incoming message is sent unmodified, the type of the newly created event is identical to the type of the incoming event. Hence, $s' = \{(\vartheta_{\text{mymod1}}, \vartheta_{\text{mymod1}}), (\vartheta_{\text{mymod2}}, \vartheta_{\text{mymod2}}), (\vartheta_{\text{mymod3}}, \vartheta_{\text{mymod3}})\}$.

4.5 Inferring Event Dependencies

This step combines the information gathered during the previous steps into a set of dependency information. We defined the relation $\succ$ for this purpose in Sec. 4.1. Again, we define a conservative over-estimation on event type basis $\succ' \subseteq (\Theta \times \Theta)$, such that for every two events $e_1 = (t_1, p_1, d_1)$, $e_2 = (t_2, p_2, d_2) \in E$ with $t_1 \leq t_2$ the following condition must be fulfilled: $d_1 \succ e_2 \Rightarrow \tau(e_1) \succ' \tau(e_2)$.

Additionally, we define the relation $\succ_{\text{A}} \subseteq (A \times A)$, such that $a_1 \succ_{\text{A}} a_2$ denotes that if in sequential execution the operation a_1 is executed prior to a_2 on the same data item, those operations must not be re-ordered. In particular, this relation holds for the following items:

References: If the data item is a reference and either or both of the operations are update operations, the operations must not be re-ordered. Hence, $a_1 \succ_{\text{A}} a_2$ if either a_1 or a_2 is a reference update operation.

Primitive Data Types: If either or both of the operations are write operations, they depend on each other. If both operations are increment operations, we rewrite these accesses by using atomic operations such that they can still be executed independently of each other. However, a read operation and an increment operation yield a dependency since the result of the read operation is changed by the incrementation.

Arrays: If either or both of the operations are write operations, they depend on each other.

RNGs: As discussed in Sec. 3 the user can trade repeatability for performance by allowing re-ordering of RNG calls. Hence, depending on the user's choice, for $a \mathrel{\widehat{=}}$ "draw random number" $a \succ_{\text{A}} a$ or $a \not\succ_{\text{A}} a$.

Queues: In general, only read-only operations on a queue can be parallelized. However, we apply special handling of queues to increase the level of parallelism: To every element enqueued, we assign the timestamp of the event that performed the enqueuing operation. This allows us to correct an out-of-order enqueuing operation, by not enqueuing the element to the queue's end, but shifting it to a position, such that the queue elements are ordered according to their timestamps.
A size query then still depends on a previous enqueuing operation: If the size is queried by an event $e_2 = (5\,\text{s}, \_, \_)$, but e_2 is executed prior to $e_1 = (4\,\text{s}, \_, \_)$ and e_1 potentially includes an enqueuing operation, the size query cannot determine how many elements the queue will contain at $t = 5\,\text{s}$ since this depends on the behavior of e_1. However, an enqueuing operation does not depend on a previous size query: An enqueuing operation by e_2 can be executed before a size query by e_1, and e_1 can still determine the size at $t = 4\,\text{s}$ by ignoring any element in the queue with a timestamp greater than $4\,\text{s}$.
Nevertheless, a dequeuing operation must not be reordered with neither operation since by removing an element from the queue we lose the necessary information. This could only be solved by keeping information of deleted elements. Since this causes additional overhead, it is not part of our analysis approach.

We use this relation $\succ_{\text{A}}$ and the sets d'_ϑ determined in Sec. 4.3 to determine $\succ'$. A pair of event types is in this relation if we find a conflicting data access: $\exists \alpha_1 = (\delta_1, a_1) \in d'_{\vartheta_1}, \alpha_2 = (\delta_2, a_2) \in d'_{\vartheta_2}$ such that $\delta_1 = \delta_2$ and $a_1 \succ_{\text{A}} a_2$, then $\vartheta_1 \succ' \vartheta_2$. To reduce the runtime overhead, we store for every event type a list C_ϑ with "conflicting" event types, i. e., $\vartheta' \in C_\vartheta \Leftrightarrow \vartheta' \succ' \vartheta$.

For the provided example, we determine the following sets: $C_{\vartheta_{\text{mymod1}}} = \{\vartheta_{\text{mymod1}}, \vartheta_{\text{mymod2}}\}$ for the first event type, $C_{\vartheta_{\text{mymod2}}} = \{\vartheta_{\text{mymod1}}, \vartheta_{\text{mymod2}}\}$ for the second type, and for the third type $C_{\vartheta_{\text{mymod3}}} = \emptyset$.

4.6 Runtime Component

The static code analysis provides the event dependencies in a representation that enables efficient dependency lookups during runtime. We integrate a runtime component into Horizon[13, 14] which exploits this information.

When the conservative synchronization algorithm of Horizon prevents offloading the next event e_{N} for parallelization, we hand e_{N} to the runtime component. This first determines the type $\tau(e_{\mathrm{N}})$ and derives the dependency list $C_{\tau(e_{\mathrm{N}})}$. We apply a shortcut for the case that this list is empty. In this situation, e_{N} does not depend on any other event in the simulation, hence we can offload it immediately for parallel execution without any further investigation.

However, if $C_{\tau(e_{\mathrm{N}})}$ is not empty, we need to determine whether the type of one of the events that are currently executed by a worker thread occurs in $C_{\tau(e_{\mathrm{N}})}$. We name the set of the currently executed events $O \subseteq E$. If $\exists e \in O : \tau(e) \in C_{\tau(e_{\mathrm{N}})}$, then e_{N} depends on e and cannot be executed in parallel with e. Hence, we need to wait for e to complete, before we can check for independence again.

If such an e does not exist, we need to investigate the transitive closure of the scheduling relation of every event in O. To this end, let s'^{+} be the transitive closure of s'. During runtime, we establish the set $s'_{\mathrm{O}} := \{\vartheta | s'^{+}(\tau(e), \vartheta), e \in O\}$ and search for a $\vartheta' \in s'_{\mathrm{O}}$. If we find such an element, e_{N} cannot safely be executed immediately. If we determine that $s'_{\mathrm{O}} \cap C_{\tau(e_{\mathrm{N}})} = \emptyset$, then we can safely offload e_{N} immediately to a worker thread for parallel execution.

5. DISCUSSION

We discuss aspects regarding the investment required by modelers to benefit from our solution as well as the underlying assumptions and general applicability of our approach.

5.1 Manual Effort

With the introduction of a new simulation language we pose a rather high entry barrier for modelers to adapt our solution. To lower this barrier, we decided to 1. keep the syntax as close to C++ as possible to reduce the learning effort, and 2. maintain compatibility with C++-based OMNeT++ models to allow partial translations.

Hence, for existing models we recommend a smooth transition by translating one module after the other. By translating the modules which pose the most severe parallelization bottlenecks, modelers can already expect some speedup from our analysis. Further translation can then be performed at a later point in time or omitted if the gain already suffices.

New simulation models can be implemented from scratch in PSimLa. The similarity to C++ reduces the effort to understand the concepts. Furthermore, existing libraries can still be used within PSimLa models.

We argue that while we cannot completely eliminate the entry barrier, we kept it as low as possible.

5.2 General Applicability

Our analysis bases on two assumptions: 1. Each data item is only accessible from a single module at a time. 2. Global knowledge about the current simulation state exists.

An assumption similar to the first one has to be posed for every parallel simulation approach since multiple processing units cannot access the same data item at the same time. While we decided to scope this on a module level, our approaches can be easily adapted to fit other scopes as well.

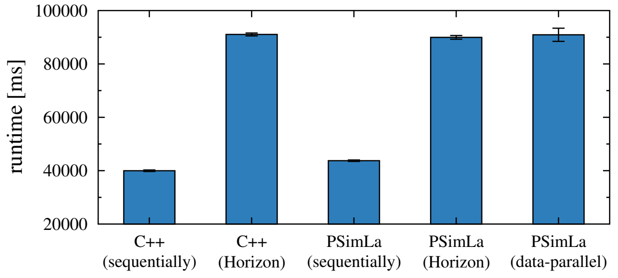


Figure 3: Overhead of the different configurations measured by the Null-Model. Less is better.

However, the second assumption limits applicability of our analysis approach to distributed simulation. An empty condition list still allows immediate execution of any event of such type in distributed simulation as the state of a remote LP does not matter. If, however, the event interferes with other events which might arrive from a remote LP later, the runtime component cannot work in a distributed simulation like described above. Hence, the runtime component has to be adapted by applying different means to determine which event types might arrive at the LP. This needs to be investigated in more detail in future research efforts. Nevertheless, we argue that the design of PSimLa – aiming at a high level of analyzability – as well supports analysis approaches better suited for distributed simulation.

Alternatively, by simply ignoring the state of remote LPs events can be speculatively executed on the local LP. The data-dependency analysis then reduces the number of rollbacks by determining if an event arriving out-of-order does actually not violate the correctness of the simulation.

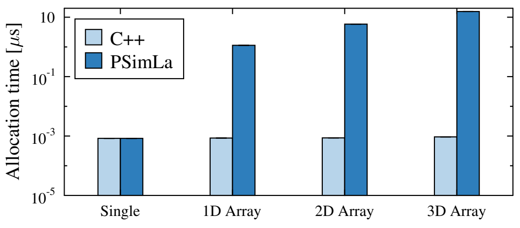
6. EVALUATION

We evaluate our approaches in terms of both overhead and speedup. Additionally, we conduct two case studies to quantify the impact on existing simulation models. All measurements are conducted on a compute server with two 6-core Opteron CPUs and a total of 32 GB of RAM running Ubuntu 12.04. We repeat every experiment at least 5 times, the plots depict the average and the 95 % confidence interval for each configuration. Note, that some confidence intervals are particularly small, thus hard to recognize.

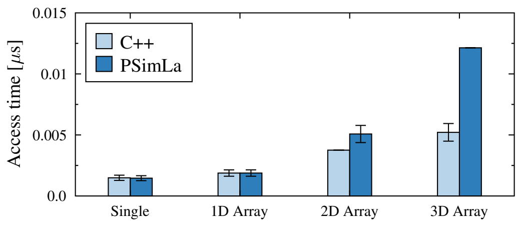
6.1 Overhead Evaluation

There are two sources of overhead in our approaches: First, suboptimal translation of PSimLa into C++ code can decrease performance. Second, the runtime component of the analysis tool introduces overhead to yield decisions.

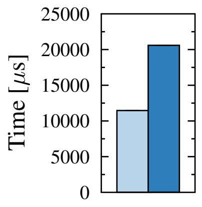
To quantify this overhead, we use three different models: First, we verify that the overhead is negligible if we use a "Null-Model" [14], i. e., a model that only handles events without actually performing computations. In this case, the only overhead stems from the fact that the OMNeT++ message pointer that is handled to the event handler, needs to be cast into a smart pointer to activate reference counting. Second, we estimate the code-translation overhead by analyzing the overhead of the most severe data type in our prototype implementation of the PSimLa compiler: multidimensional arrays. Third, we measure the decision overhead of the runtime component by feeding it with decision problems of varying levels of complexity.



(a) Time for memory allocation for different data types.

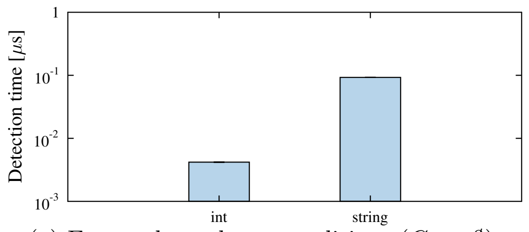


(b) Time to access variables.

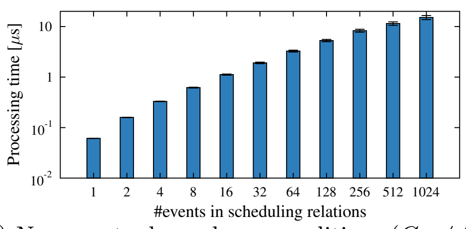


(c) Real-world example: fading model [20].

Figure 4: Code translation overhead. Time needed for different operations. Less is better.



(a) Empty dependency conditions ($C_\vartheta = \emptyset$).



(b) Non-empty dependency conditions ($C_\vartheta \neq \emptyset$).

Figure 5: Decision overhead. Time needed for yielding an offloading decision. Less is better.

Null-Model.

The Null-Model [14] only evaluates overhead by not performing any computations. It consists of 110 modules with an event handler only repetitively rescheduling an event.

We implemented this model in C++ as well as PSimLa to compare the overhead of the two languages. We ran the model 1. sequentially, 2. with the parallelization architecture Horizon [13, 14], and 3. with additional activation of our analysis approaches. The third option can only be applied on the PSimLa model. Fig. 3 depicts the runtimes of these 5 different configurations. Comparing the sequential executions, we observe a slight but negligible additional overhead of PSimLa caused by pointer casts and reference counting. Activating the parallelization architecture Horizon significantly raises the overhead as discussed in [14]. In this case the overhead introduced by the translation of PSimLa is no longer observable. Activating the analysis does not increase the overhead either. We conclude that the overhead of PSimLa and our analysis is negligible if the event handlers of the model are simple. Any additional overhead stems from suboptimal translation of event handler code and complicated offloading decisions as discussed in the following.

Code-Translation Overhead.

As discussed in Sec. 3.2, our prototype implementation does not automatically choose the best storage location for arrays, but always allocates them dynamically. While C++ developers can decide to place an array in automatic memory with virtually no allocation time, our PSimLa compiler allocates dynamic memory even if the array is only required locally. Since this is the primary source for code translation overhead, we decided to quantify this overhead as a worst-case example. We investigated primitive data types as well as 1-, 2-, and 3-dimensional arrays. We measured performance of allocation as well as element access by performing only such an operation in a loop.

The results are displayed in Fig. 4. While *allocation* in automatic memory consumes virtually no time, dynamic allocation in PSimLa is considerable and grows with the number of dimensions. This is not a surprising result and confirms that future efforts need to investigate better placement strategies. However, *access* to array elements only poses a small amount of overhead caused by better cache-locality in automatic memory. We also observe that this overhead super-linearly grows with the number of dimensions.

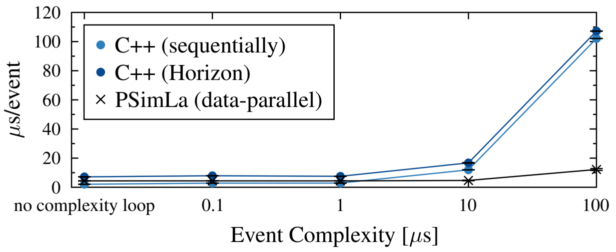
We as well considered a real world fading model [20] heavily using multi-dimensional arrays. For this example, we observe a factor 2 slowdown in the fading computation function due to allocation and access overhead. We hence recommend to use the opportunity of PSimLa to implement parts of the model in C++ for such kind of code and only realize the remaining parts in PSimLa. Nevertheless, the PSimLa compiler needs to deal with this issue in future versions.

When the runtime component has to decide whether to offload the next event immediately for parallel execution, it determines the event type and then determines the scheduling relations only if the next event generally depends on other events. Hence, for the overhead evaluation we investigate two questions: 1. How long does it take to identify the event type ϑ, determine that there are no dependencies ($C_\vartheta = \emptyset$), and return the result? 2. How long does it take to yield a decision if the scheduling relations need to be investigated (since $C_\vartheta \neq \emptyset$)?

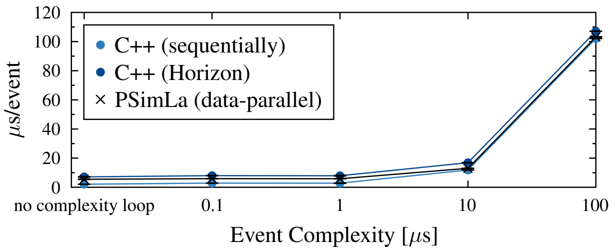
Decision Overhead.

For the first question, we measure the decision time for events whose event type is classified by either an integer or a string (14 characters on average). Hence, determining the event type is either an integer-based branch or a string comparison. The results are depicted in Fig. 5(a). The integer-based type classification and decision yielding takes only a few processor cycles (4 ns), the string comparison takes about 100 ns. Hence, on event types with no data-dependencies the decision is yielded highly efficiently, especially if the classification does not involve string comparisons.

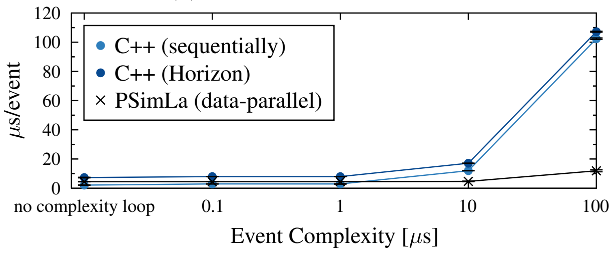
If the event has dependencies on other events, we need to determine the scheduling relation for the currently offloaded events s'_O and search it for conflicts. We therefore investigate the overhead for different sizes of s'_O (see Fig. 5(b)). For $|s'_O| = 1$ we need 60 ns to determine the event types, con-



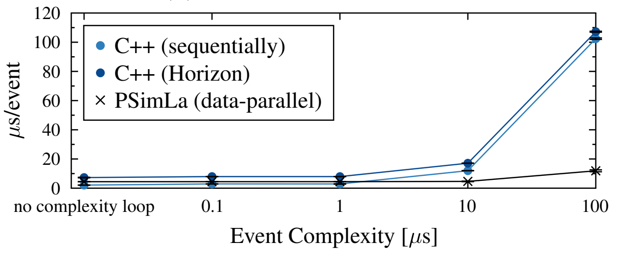
(a) Access operation: read



(b) Access operation: write



(c) Access operation: increment



(d) Access operation: enqueue

Figure 6: Runtime of events with basic operations and a complexity loop. Less is better.

struct s'_O, and yield the decision. With increasing size of s'_O the decision overhead grows almost linearly up to 15 µs for 1024 event types. Hence, the runtime component is more efficient if the scheduling relations are smaller. However, since the number of different event types in a simulation is typically rather low, we expect also the scheduling relation set to be rather small in typical simulations.

6.2 Speedup Evaluation

We evaluate the gain of our analysis approaches in simple cases. We only use one of the basic operations that our analysis should detect (see Sec. 4.3). We create events that only use this basic operation followed by a loop simulating different workloads of the event. Due to the overhead introduced by the parallelization engine the loop needs to maintain a certain complexity until parallelization can pay off. In our experiments, we vary the complexity between 1 µs and 100 µs and run an additional experiment without the loop. We investigate four operations: reading of a primitive data type, writing to a primitive data type, incrementing a primitive data type, enqueuing to a queue.

Fig. 6 depicts the results. Observing the results with no or low complexity again shows the overhead introduced by the parallelization engine Horizon (which cannot run events in parallel since the timestamps do not yield independent events). Our analysis already improves upon this situation by yielding positive parallelization decisions which allow parallelizing the overhead. For a complexity of 10 µs we observe that the parallelization gains significant boosts for read, increment, and enqueuing. We also observe that there is no gain for write operations. This means that the analysis correctly identifies conflicting write operations and prevents them from being executed in parallel. Finally, we observe close-to-linear speedup for high event complexities when the analysis can identify independent events. Additionally, we observe that there is no severe impact of neither the increased complexity by using atomic operations for incrementations nor the sorting of the queue.

From these results we derive that the benefit in general not only depends on the simulation model but also on the model implementation. If an event handler issues a write to a variable used by another event, the two events cannot be parallelized. In a worst case scenario (i. e., the affected event types yield by far the most complex events in the model), this eliminates the benefit of our approach. However, our analysis tool provides information on the determined event dependencies. If the gained speedup does not fulfill the expectations, this information can be used to track down and eliminate the bottleneck. For the implementation of wireless models this means that complex channel computations should be separated from modifying the channel state to avoid introducing unnecessary dependencies.

6.3 Case Studies

We conduct two case studies to evaluate the practical impact of our approaches in wireless network simulation, namely a Wireless Mesh Network (WMN) and an LTE model. We implemented both models completely in C++ as well as PSimLa. We also exploit the opportunity to combine both languages in order to mitigate the drawbacks of poor translation in certain cases (see Sec. 3.1).

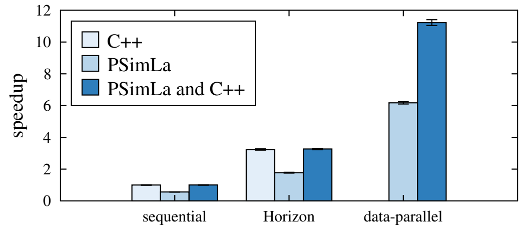
We execute all three implementations (PSimLa, C++, and combined) 1. sequentially and 2. parallelized by Horizon. Additionally, we execute the two implementations containing PSimLa-code parallelized by exploiting the data-dependency knowledge of the code analysis. This yields eight configurations. We measure the speedup of each configuration compared with sequential execution of the C++ model, i. e., the speedup of this case is by definition 1.

For both case studies we verified that in all configurations the simulation results are identical to the results of a sequential execution. We discuss the results in the following.

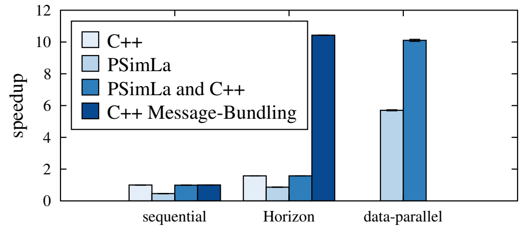
Wireless Mesh Network.

The first model simulates a 57 node mesh network where data is transmitted from node to node over a wireless channel simulated with an accurate OFDM fading model [20].

The results are depicted in Fig. 7(a). As the overhead evaluation of the OFDM fading code shows (cf. Fig. 4(c)), the fading code is poorly translated from PSimLa to C++ by our prototype compiler. For this reason, sequential execution of a full implementation in PSimLa yields a considerable



(a) Wireless Mesh Network.



(b) LTE Network.

Figure 7: Case studies: Speedup over sequential execution of the C++ model. More is better.

slowdown over C++ code. Hence, we decided to implement the fading in C++, which yields performance similar to the C++ model if executed sequentially or by Horizon. By additionally exploiting the analysis results the model execution achieves close-to-linear speedup while the C++ model can only be speeded up by a factor of 3.2.

LTE Simulation Model.

Our second case study simulates a 48 cell LTE network with an abstract LTE model as described in [17]. Initially, this model creates separate events if multiple data packets are transmitted in a single Time to Transmit Interval (TTI). Since the parallelizability of these events is not encountered by traditional techniques, this configuration parallelizes poorly (cf. Fig. 7(b)). However, the data-dependency analysis determines the actual independence of those events and achieves close-to-linear speedup.

The results of our code analysis disclosed this bottleneck in the C++ model. We fixed the bottleneck in the C++ model by bundling the events together and achieved a comparable performance. This shows that the analysis has the potential to eliminate performance bottlenecks automatically and avoid the need for manual model adaptations.

7. RELATED WORK

We compare our work against related efforts in the area of DSLs designed for the simulation context as well as approaches to static code analysis of simulation code.

7.1 Simulation Languages

DSLs for simulation have a long tradition. Simula [6] and GPSS [16] have been developed in the 60s and 70s respectively to incorporate the special requirements of simulation in the language design. However, these languages were not designed for PDES. Apostle [22] and Parsec [1] are the most prominent examples of DSLs designed for parallel simulation. Though these languages have been tailored to ease the development of models that can efficiently be executed in parallel, the underlying parallelization concepts only rely on event timestamps in order to derive event dependencies. The languages are not designed with a focus on analyzability.

A more modern approach in this field is Pose [21]. Though Pose is rather a simulation framework than a simulation language, it bases on the DSL Charm++ [11], which is designed for parallel programming in general. The aim of Pose is to make parallelization transparent to the model developer by applying the virtualization scheme of Charm++ providing virtual objects. The framework can map those virtual objects to processors, such that model developers no longer need to cope with efficient partitioning. However, Charm++ is designed to allow developers to create efficient parallel programs, but does not put particular emphasis on analyzability to automatically detect independencies.

Recent efforts include ErlangTW [19], a simulation framework based on the functional programming language Erlang. We already discussed the pros and cons of functional programming languages in Sec. 2. Although we expect static analysis to be easier for functional programs, to the best of our knowledge no approach exists analyzing data-dependencies in ErlangTW. We believe that our analysis approach could be as well applied on ErlangTW with less effort for data access tracking due to the absence of side effects. In this paper, however, we demonstrate the feasibility of data-dependency analysis even for a structured language.

Further modern DSLs include SESSL [7] and DEVS-Ruby [8]. While SESSL focuses on a language to describe simulation experiments, DEVS-Ruby focuses on a specification for models following the DEVS formalism [23]. Neither of those languages incorporates the ability to analyze model code for data-dependencies into the language design. We conclude that a language that actually incorporates this feature can increase the parallelism in simulation models.

7.2 Static Code Analysis

Chen et al. [4, 5] propose a similar approach to analyze data-dependencies for parallel simulation. To this end, they focus on Electronic System Level (ESL) design. As opposed to our approach, the authors did not create a dedicated language, but base their analysis on the language SpecC, commonly used in ESL design. Like C, SpecC supports pointers to address data items in dynamic memory. Similar to our approach, the analysis of Chen et al. checks the model code for conflicts between data accesses. However, since the approach is unable to reliably detect conflicts when data is accessed via a pointer, the authors decided to terminate the analysis on any occurance of a pointer access, and handle the pointer access like a conflicting data access. Hence, only if all items accessed by an event handler are either objects in automatic memory or members of the local module, this analysis can improve performance. While this might be a feasible limitation in the ESL domain, we argue that for a wide area of simulation models, especially wireless networks, this is not a suitable simplification. In order to investigate whether data-dependency analysis is generally feasible for simulation models implemented in a structured programming language, it is necessary to support the basic features of the selected language. In particular, this includes access to data items which are not within the scope of the function, but can only be accessed via a reference or pointer. To circumvent the unsolved problem of pointer analysis [2, 10] without facing this limitation, we decided to create the pointer-less language PSimLa and base our analysis on PSimLa code. This shows the feasibility of detecting data-dependencies even if references are used within the analyzed code.

Further approaches to static code analysis in the area of simulation focus on performance prediction or model testing rather than automatic performance improvements. Cavitt et al. [3] and Kappler et al. [12] use static code analysis to investigate the model and derive performance prediction models. Overstreet [15] showed the usability of static code analysis to verify correctness of simulation models.

We conclude that, while there is an approach using static code analysis to performance improvement of ESL models, there is no general approach to increase the level of parallelism usable in wireless and other simulation models. In this paper, we demonstrate the feasibility of data-dependency analysis for a structured programming language that includes all features necessary for model development.

8. CONCLUSION

This paper discusses the simulation language PSimLa, which is specifically designed for analyzability, as well as an approach exploiting this feature to analyze event dependencies and use this information to increase the level of parallelism. A model can be implemented completely or partially in PSimLa, which is similar to C++. This enables a smooth transition for existing C++ code. The translated parts can then be analyzed at compile time, such that the execution is speeded up by exploiting those information. Our evaluation shows promising results indicating that certain wireless simulation models, which are not well parallelized by traditional approaches, can strongly benefit from this analysis: In two case studies our approach reaches close-to-linear speedup for models which are hardly parallelized by traditional techniques.

Future efforts address three dimensions: language syntax, translation quality, and analysis approaches. The features of PSimLa supported by our prototype implementation of the compiler are sufficient to implement any model. Nevertheless, the feature set needs to be extended to provide all elements of modern high-level languages. The evaluation of the code translation quality has shown certain bottlenecks that need to be eliminated. Especially the allocation of arrays can be improved by smarter concepts to decide whether to allocate dynamic memory or use automatic memory instead. Additional analysis approaches can further improve the degree of parallelism. These should address more sophisticated methods to analyze more complex code, potentially detecting more independent events in situations where our approaches assume dependency due to a too conservative analysis. Furthermore, independence information in a shape better applicable for distributed simulation can improve performance for simulations executed on multiple machines.

Acknowledgments

This work has been co-funded by the German Research Foundation (DFG) within the Collaborative Research Center (CRC) 1053 – MAKI.

9. REFERENCES

[1] R. Bagrodia, R. Meyer, M. Takai, Y.-A. Chen, X. Zeng, J. Martin, and H. Y. Song. Parsec: A Parallel Simulation Environment for Complex Systems. *IEEE Computer*, 31(10):77–85, 1998.

[2] D. Binkley. Source Code Analysis: A Road Map. In *Future of Software Engineering*, 104–119, 2007.

[3] D. B. Cavitt, C. M. Overstreet, and K. J. Maly. A Performance Analysis Model for Distributed Simulations. In *Proc. of the 28th Winter Sim. Conf.*, 629–636, 1996.

[4] W. Chen and R. Dömer. Optimized Out-of-Order Parallel Discrete Event Simulation Using Predictions. In *Proc. of the 2013 Conf. on Design, Autom. & Test in Eur.*, 2013.

[5] W. Chen, X. Han, and R. Dömer. Out-of-Order Parallel Simulation for ESL Design. In *Proc. of the 2012 Conf. on Design, Autom. & Test in Eur.*, 141–146, 2012.

[6] O.-J. Dahl and K. Nygaard. SIMULA: An ALGOL-based Simulation Language. *Communications of the ACM*, 9(9):671–678, 1966.

[7] R. Ewald and A. M. Uhrmacher. SESSL: A Domain-specific Language for Simulation Experiments. *ACM Trans. on Modeling and Computer Sim.*, 24(2):11:1–11:25, 2014.

[8] R. Franceschini, P. Bisgambiglia, and D. Hill. DEVS-Ruby: a Domain Specific Language for DEVS Modeling and Simulation (WIP). In *Proc. of the 2014 Symp. On Theory of Modeling and Sim.*, 15:1–15:6, 2014.

[9] R. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[10] M. Hind. Pointer Analysis: Haven't We Solved This Problem Yet? In *Proc. of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 54–61, 2001.

[11] L. V. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proc. of the 8th Conf. on Object-oriented Programming Systems, Languages, and Applications*, 91–108, 1993.

[12] T. Kappler, H. Koziolek, K. Krogmann, and R. Reussner. Towards Automatic Construction of Reusable Prediction Models for Component-Based Performance Engineering. *GI LNI: Software Engineering*, 121:140–154, 2008.

[13] G. Kunz, O. Landsiedel, J. Gross, S. Götz, F. Naghibi, and K. Wehrle. Expanding the Event Horizon in Parallelized Network Simulations. In *Proc. of the 18th Symposium on Modeling, Analysis and Sim. of Computer and Telecommunication Systems*, 172–181, 2010.

[14] G. Kunz, M. Stoffers, J. Gross, and K. Wehrle. Runtime Efficient Event Scheduling in Multi-threaded Network Simulation. In *Proc. of the 4th Conf. on Sim. Tools and Techniques*, 359–366, 2011.

[15] C. Overstreet. Model Testing: Is it only a Special Case of Software Testing? In *Proc. of the 34th Winter Sim. Conf.*, 641–647, 2002.

[16] T. J. Schriber. Simulation using GPSS. Technical report, DTIC Document, 1974.

[17] M. Stoffers, S. Schmerling, G. Kunz, J. Gross, and K. Wehrle. Large-Scale Network Simulation: Leveraging the Strengths of Modern SMP-based Compute Clusters. In *Proc. of the 7th Conf. on Sim. Tools and Techniques*, 2014.

[18] M. Stoffers, T. Sehy, J. Gross, and K. Wehrle. Analyzing Data Dependencies for Increased Parallelism in Discrete Event Simulation. In *Proc. of the 29th ACM SIGSIM Conf. on Principles of Advanced Discrete Sim.*, 2015.

[19] L. Toscano, G. D'Angelo, and M. Marzolla. Parallel Discrete Event Simulation with Erlang. In *Proc. of the 1st ACM SIGPLAN Workshop on Functional High-performance Computing*, 83–92, 2012.

[20] C. Wang, M. Pätzold, and Q. Yao. Stochastic Modeling and Simulation of Frequency-Correlated Wideband Fading Channels. *IEEE Trans. on Vehicular Technology*, 56(3):1050–1063, 2007.

[21] T. Wilmarth and L. Kale. Pose: getting over grainsize in parallel discrete event simulation. In *Proc. of the 2004 Intl. Conf. on Parallel Processing*, 12–19, 2004.

[22] P. Wonnacott and D. Bruce. The APOSTLE Simulation Language: Granularity Control and Performance Data. In *Proc. of the 10th Workshop on Parallel and Distributed Sim.*, 114–123, 1996.

[23] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems.* Academic Press, Waltham, MA, U. S., 2000.