Chapter 1

Introduction

1.1 Context

A *schedule* is a ternary relation amongst time, resources, and tasks. That is, a schedule is an allocation of time and resources to tasks. We represent time as all non-negative, real-valued timestamps. The collection of all related tasks forms an *application.* An application can run indefinitely, which may happen due to repetitive tasks. We implicitly allow tasks to admit subtasks, although this hierarchical structure is irrelevant for the work presented in this thesis. The number of resources is generally finite, which introduces scarcity.

A *protocol* is a constraint on schedules that determines a set of valid schedules. For example, a protocol may demand that each resource is allocated to at most one task at any given time. A protocol may also constrain the execution times of tasks to follow a particular partial order. An application is sequential if its timing constraint forms a total order, and the application is concurrent, otherwise. Typically, timing constraints of an application specify only the relative order of the start and completion times of its tasks. Real-time applications involve stricter timing constraints that may specify exact (i.e., absolute) start times, completion times, and/or durations for its tasks. Typically, the timing constraints of tasks are not specified explicitly; instead, they are implied by the ordering of actions of a task and their durations.

Scheduling is the activity of selecting a valid schedule (i.e., that respects the constraints of the protocol of the application) that optimizes some given criterion such as total execution time, throughput, latency, or resource utilization. The selected schedule can have a significant impact on the quality of the application. For example, in mission critical applications like a pacemaker or a flight controller it is imperative to avoid catastrophic failure by producing reliable, predictable output.

Applications can exhibit two types of non-deterministic behavior. Internal non-determinism is under our control and is resolved by scheduling. External non-determinism (such as user input) is uncontrolled and requires flexible schedules. However, the ternary relations defined above are deterministic and leave no room for uncertainty. Therefore, a scheduler generally determines a *scheduling strategy*, which is a collection of schedules. At run time, a scheduler can follow any such schedule that is consistent with the given external non-determinism.

The simple and intuitive definitions of schedules and protocols above have significant consequences that we exploit in the work presented in this thesis. The above definitions show that scheduling strategies and protocols are essentially the same (both being sets of schedules), and that as such, protocols and scheduling strategies are convertible into each other. We exploit this convertibility property in Chapter 8 to improve the performance of the operating system scheduler for a cyclo-static dataflow program.

Although both protocols and scheduling strategies are essentially identical, they serve different purposes. Roughly speaking, a protocol primarily expresses the *correctness constraints* of a concurrent application, whereas a scheduling strategy primarily expresses the *efficiency constraints* under which an application or a set of applications execute efficiently, as determined by some objective function that optimizes various concerns such as execution time, throughput, resource utilization, etc. Just as the boundary between functional versus quality of service properties of an application is not always sharply defined, the distinction between correctness constraints (i.e., protocols) from efficiency constraints (i.e., scheduling strategies) is not always crisp, either. Importantly, protocols often contain a wealth of useful information, essential for optimal scheduling. We consider protocols in Part I and scheduling strategies in Part III.

Although we define protocols as constraints on schedules, in practice, protocols are never literally specified as constraints on schedules (i.e., constraints on ternary relations). Instead, traditional models of concurrency and concurrent programming constructs specify a protocol only as a derived by-product of the timing constraints of the tasks that comprise a software. In contrast to traditional models of concurrency, exogenous coordination[1] languages, such as Reo [Arb04], offer first-class constructs for expressing concurrency/coordination protocols as concrete, identifiable, reusable software modules, independent of the computation carried out by an application [Arb11, Arb16].

1.2 Problem

1.2.1 Resource virtualization

Scheduling is generally easy in case of *non-interference*, that is, no two tasks require a shared resource at the same time. Indeed, non-interference assures that we can allocate resources on demand. A trivial example of non-interference is a set of tasks that do not share any resources. However, such task sets are uncommon, because cooperative tasks require shared resources for communication. A more common example of non-interference is a set of tasks that consists of coroutines. Only one of the tasks is active at any given time, which implies non-interference of processors. Other examples of non-interference are a read-only shared variable, or an antenna that can broadcast at different frequencies simultaneously.

[1]Coordination models and languages offer higher-level constructs for specification of protocols that coordinate the tasks that comprise a concurrent application [Arb98].

Non-interference allows for *resource virtualization*, which duplicates a physical resource into multiple virtual resources that can be used by multiple independent applications. Virtualization minimizes the number of otherwise-necessary physical resources, which reduces cost. For example, coroutines can run on dedicated processors. However, since only one of these processors is active at any given time, these dedicated processors can be virtual processors that map to a single physical processor. Similarly, tasks that require antennas to broadcast on different frequency bandwidths can of course use a dedicated physical antenna, each. Non-interference of their bandwidths, however, allows a virtualization that maps all dedicated antennas of these tasks as virtual antennas onto a single physical antenna. Another example of virtualization is virtual memory, which divides memory into pages. Since tasks often work on a single page at time, these memory pages can be virtual memory pages that map to a single physical memory page. Of course, we need additional cheap memory to store unused pages.

In general, resource virtualization requires *interference resolution.* For example, we can virtualize a printer by means of a queue that ensures non-interference. The queue plays a dual role. On the one hand, the queue implements a mutual exclusion protocol that regulates access to the printer. On the other hand, the queue implements the first-come-first-served scheduling strategy. Hence, we observe a first glimpse of a duality between protocols and schedules. Later on, we exploit this duality by transforming a schedule into a protocol.

Software developers generally outsource part of the scheduling by virtualization of frequently used resources. The prime example is the virtualization of processors by the operating system, which creates virtual duplicates of the physical processors in the form of processes and threads. The operating system allows for context-switching by reserving some memory to save the state of each virtual processor. By assigning to each task a dedicated virtual processor, the scheduling problem becomes trivial from the perspective of the software developer.

1.2.2 Conflicting objectives

Resource virtualization usually resolves interference via a generic scheduling strategy that aims at a fair distribution of the physical resource over the virtual ones. Hence, the objective of generic schedulers generally differs from the objectives of applications that use these resources. For example, an operating system scheduler implements a variant of round-robin scheduling that allocates processing time to non-blocked virtual processors. Virtual processors are blocked whenever their associated task accesses another resource (like an I/O device) that is already in use.

However, fair distribution of the physical resources over the virtual ones does not necessarily optimize the desired scheduling objective. For example, consider two tasks that produce ice creams and a single consuming task. Assume that production requires the same amount of work as consumption. Fair distribution of resources yields a production rate that is twice the consumption rate. As a result, most ice creams melt before being consumed. Ideally, the production and consumption rates should be equal to yield maximal throughput.

To optimize the schedule of an application, we must inform the default scheduler in virtual resources about relevant application-specific scheduling information. This brings us to the main question of this thesis:

How to optimize scheduling in the presence of virtual resources?

There are two ways to inform the generic scheduler in a virtual resource. The first and most straightforward approach is to hard-code this information by replacing the scheduler in the virtual resource with an application-specific scheduler. However, an application-specific scheduler is often unacceptable, as one of the goals of virtualization is to provide seamless resource sharing by multiple independent applications. Hence, the application-specific scheduler is possible only for systems dedicated to a single application, such as embedded devices with a real-time operating system.

The second and less intrusive approach is to change the application (and its constituent tasks), so that the generic default scheduler behaves optimally (or as close to it as possible). This approach is obviously more complex than changing the mapping of virtual resources. In Part III, we develop a technique that alters the application to optimize the scheduling. Our approach is based on the duality between scheduling strategies and protocols mentioned in Section 1.1.

1.3 Related work

The literature on scheduling offers partial solutions to our main question, and we broadly classify them in three categories, namely online scheduling, low-level offline scheduling and high-level offline scheduling.

Online scheduling The OS scheduler uses a variant of generic round-robin scheduling strategy, which schedules fairly by assigning a fixed amount of time (a time quantum) per task, and cycles through the tasks. If a tasks performs a blocking operation, such as read from/write to memory or acquisition of a lock, then the task enters a waiting state. The OS detects when a process is waiting, and does not assign time quanta to that process.

Round-robin scheduling does not require any a priori knowledge of the running task. The OS scheduler cannot predict which tasks it will encounter in the future. Consequently, its performance is suboptimal. One can still provide guarantees on the quality of a given scheduling algorithm, A, by considering its *competitive ratio*, which is defined as the smallest[2] factor C such that

$$\mathrm{cost}_A(x) \leq C \cdot \mathrm{cost}_{\mathrm{optimal}}(x), \qquad \text{for every input } x$$

where $\mathrm{cost}_A(x)$ is the cost of the given online scheduler A that cannot predict the future, and $\mathrm{cost}_{\mathrm{optimal}}(x)$ is the cost of a clairvoyant scheduler that knows all future tasks. For example, if we consider the cost of the schedule length (or makespan), then list scheduling has a competitive ratio of $2 - 1/n$, with $n \geq 1$ the number of machines [Gra66]. Of course, any competitive ratio is at least 1, which corresponds to the ideal situation wherein the scheduler has perfect information on the behavior of the application.

[2]Or the greatest lower bound if no smallest C exists.

Low-Level offline scheduling Round-robin scheduling leaves plenty of room for optimization. To squeeze even more performance out of a given software, we turn our attention to application-specific scheduling.

One generic approach in improving the performance of software is to perform *instruction scheduling.* The GNU Compiler Collection is a C compiler that performs instruction scheduling when using the `-march` or `-mtune` flags. Instruction scheduling is a compiler optimization that reorders independent statements in a basic block to optimize the total execution time. A basic block is a sequence of instructions without branching, which constitutes a single node in the control flow graph of the application. Two instructions are independent and can be reordered, when neither instruction writes to a memory location that is accessed by the other. For example, two instructions that do not access the same memory location are independent, and two instructions that both read from the same memory location are independent.

Since all instructions and their dependencies are known, it is possible to produce an optimal instruction order. However, finding the optimal schedule is computationally intensive. Therefore, one often pretends that instruction scheduling is an online scheduling problem, and uses the list scheduling that we discussed above. The resulting execution time at most doubles, because the competitive ratio of list scheduling is $2 - 1/n$, for n machines.

High-Level offline scheduling Instruction scheduling improves the software execution by considering very low-level information on the software. As a result, instruction scheduling alone is generally insufficient, as it does not consider high-level information, like the way different parts of the software interact.

For specific application domains, such as digital signal processing (DSP), this extra performance gain is necessary. To this end, one develops an application in a domain specific language (DSL) that is amenable to the extraction of relevant scheduling information. Examples of such DSLs include CSDF graphs and Kahn process networks.

However, most software is written in a general-purpose language, such as Python, C, or Java[3]. Scheduling techniques developed for software written in DSLs for specific application domains do not readily apply to software written in these general-purpose languages, because the extraction of relevant scheduling information is much more difficult for software written in general-purpose languages than for software written in these DSLs. The primary reason is that the protocol of software written in languages that use traditional models of concurrency is implicit, nebulous, and diffused among the large number of lines of code that comprise the software [Arb11]. Without an explicit protocol, we cannot know in advance which schedules will be valid at run time.

1.4 Contributions

This thesis is structured into three parts, namely coordination, compilation, and scheduling.

[3]`https://www.tiobe.com/tiobe-index/`

1.4.1 Coordination

Scheduling requires detailed information on the protocol of an application. For software written in general-purpose programming languages, such information is generally not available. Therefore, we turn in Part I our attention to coordination languages, which simplify design and development of concurrent systems. Particularly, exogenous coordination languages, like BIP [BBS06, BS07] and Reo [Arb04, Arb11], enable system designers to express the interactions among components in a system explicitly. A formal relation between exogenous coordination languages comprises the basis for a solid comparison and consolidation of their fundamental concepts.

In Chapter 2, we establish a formal relation between BI(P) (i.e., BIP without the priority layer) and Reo, by defining transformations between their semantic models. We show that these transformations preserve all properties expressible in a common semantics. We use these transformations to define data-sensitive BIP architectures and their composition.

In Chapter 3, we address the discrepancy between Reo and BIP regarding priority by revising *soft constraint automata*, wherein transitions are weighted and each action has an associated preference value [AS12, KAT16, KAT17]. Soft constraint automata can be used as a semantics for Reo connectors. These preferences can be used to guide the selection of an interaction that is enabled by an (unpredictable) environment.

We revise soft constraint automata in three ways. First, we relax the underlying algebraic structure to allow bipolar preferences, which allow one to express positive preferences (I like X) as well as negative preferences (I do not like X). Next, we equip automata with memory locations, that is, with an internal state to remember and update information from transition to transition. Finally, we revise automata operators, such as composition and hiding, providing examples on how such memory locations interact with preferences.

We show the utility of our revised soft constraint automata by encoding context-sensitive behavior in terms of these soft constraint automata.

1.4.2 Compilation

The exogenous coordination languages from Part I offer an explicit protocol specification that governs the interactions amongst a set of tasks. However, such a protocol specification does not contain all information necessary for our scheduling purposes. The main shortcoming is that the protocol specifications from Part I do not specify the *workload* of each task, which is the amount of processing time required by that task. Furthermore, the implementation of a protocol specification may require one or more auxiliary (protocol) tasks.

In Part II, we provide all necessary ingredients for a complete protocol specification by developing a compiler that translates these protocol specifications into executable code. The original tasks, together with the auxiliary tasks, completely define the scheduling problem. Although it is certainly possible to derive or approximate the workload of each task by analyzing the code generated by our compiler (including the target architecture), we leave this as future work.

Recent benchmarks show that compiling high-level Reo specifications produces executable code that can compete with or even beat the performance of handcrafted programs written in languages such as C or Java using conventional concurrency constructs [JHA14, JA15, JA16b, JA16a, JA18].

The original declarative graphical syntax of Reo does not support intuitive constructs for parameter passing, iteration, recursion, or conditional specification [ECT]. This shortcoming hinders Reo's uptake in large-scale practical applications. Therefore, a number of Reo-inspired syntax alternatives have appeared in the past, such as RSL [Klü12] and FOCAML [Jon16]. However, none of them follow the primary design principles of Reo: a) declarative specification; b) user-defined channels and channel types; and c) channels compose via shared nodes.

In Chapter 4, we offer a textual syntax for Reo (called Treo) that respects these principles and supports flexible parameter passing, iteration, recursion, and conditional specification. The Treo language is extensible and allows the creation of new semantics for components, such as the soft constraint automaton semantics from Chapter 3. In on-going work, we use this textual syntax to compile Reo into target languages such as Java, Promela, and Maude.

The state-of-the-art Reo compiler (Lykos) by Jongmans represents protocols as constraint automata [Jon16], which specify protocols as labeled transition systems that preserve synchronization under composition. Besides compilers, constraint automata have been used as a basis for other tools like model checkers [Klü12]. Unfortunately, composition of transition systems suffers from state space and transition space explosions, which limits scalability of the tools based on constraint automata.

Clarke et al. avoid these problems by representing protocols as constraints [CPLA11]. These constraints determine, at every point in time, which combination of actions constitute an interaction. Constraints are composed via conjunction, which means that the size of the representation is potentially linear in the number of components. However, the approach by Clarke et al. relies on a constraint solver, which incurs a significant overhead in the run time.

In Chapter 5, we avoid the overhead of the constraint solver by transforming constraints directly into executable code, without relying on a constraint solver. More precisely, we apply a commandification technique developed by Jongmans that generates low-level executable code for a constraint that does not contain any disjunction [JA15]. We extend this commandification to arbitrary constraints by rewriting them into a *rule-based form.* We provide sufficient conditions under which our approach avoids transition space explosions. As a result, we can now compile protocols that could not be compiled by Lykos.

Constraint-based representations of a Reo protocol are inherently sequential. Therefore, straightforward compilation of constraints produces sequential code that does not utilize all available computational resources.

In Chapter 6, we make concurrency explicit by reformulating our ideas for constraints in terms of (multilabeled) Petri nets. Similar to the constraint-based approach, our novel composition of Petri nets avoids a state space explosion, which makes them an adequate intermediate representation for code generation. Moreover, these Petri nets can be viewed as an inherently parallel generalization of constraint automata, which makes it possible to generate concurrent code. This approach is a refinement of the so called synchronous region decomposition of Reo connectors [JCP16].

Moreover, we also present an abstraction operator for multilabeled nets that eliminates internal transitions, which optimizes the execution of multilabeled nets.

1.4.3 Scheduling

The Reo compiler developed in Part II produces code that relies on the operating system (OS) scheduler to assign computational resources to its tasks. The OS scheduler aims at fair distribution of processing time amongst its given tasks. As mentioned earlier (Section 1.2.2), this objective usually conflicts with the scheduling objective of any given application. In Part III, we address the main question in Section 1.2.2 by computing an optimal scheduling strategy for a given Reo protocol, and guiding the OS scheduler to follow this optimal strategy.

In Chapter 7, we introduce work automata to express all relevant scheduling information of a given Reo protocol. The work automaton semantics of Reo addresses an important shortcoming of the protocol specifications in Part I by explicitly specifying the workload of each task. The modular structure of the Treo language allows us to add syntax to express primitive work automata. Our generic (semantics-agnostic) Reo compiler composes these primitive work automata to produce the work automaton for the complete application. We potentially[4] benefit from all semantics-independent compiler optimizations, such as the queue-optimization [JHA14] and protocol splitting [JCP16].

Work automata specify behavior and workloads. The specification of behavior is fully controlled by the developer of the application. The workloads, however, are not directly under the control of the application developer, and these workloads can be determined only after compilation. In the current work, we assume that the work automata (including workloads) are given, and we leave the derivation or approximation of workloads as future work.

We provide a formal semantics for work automata, based on which we introduce equivalences such as weak simulation and weak language inclusion. Subsequently, we define operations on work automata that simplify them while preserving these equivalences. Where applicable, these operations simplify a work automaton by merging its different states into a state with a ‘more inclusive’ state-invariant. The resulting state-invariant defines a region in a multidimensional real vector space that potentially contains holes, which in turn expose mutual exclusion among processes. Such exposed dependencies provide additional insight into the behavior of an application, which can enhance scheduling. Our operations, therefore, potentially expose implicit dependencies among processes that otherwise may not be evident to exploit.

In Chapter 8, we use the scheduling information of a Reo protocol (presented as a work automaton) to compute an optimal scheduling strategy for that application. To be precise, we use a generic, game-theoretic scheduling framework to find optimal non-preemptive schedules for an application.

A straightforward approach to implement the resulting schedule is to replace the OS scheduler with a custom scheduling strategy. However, as mentioned earlier, such replacement is highly non-trivial and defeats the purpose of the OS as it turns it into a platform optimized to run only one application.

[4]Our current compiler does not yet implement the queue-optimization or protocol splitting.

Therefore, our answer to the main question in Section 1.2.2 is to alter the application so that its scheduling problem becomes trivial. Our main idea is to use the duality from Section 1.1 to convert the ideal scheduling strategy into a scheduling protocol. Our compositional framework allows us to compose this scheduling protocol with the original application protocol. This composition can be viewed as a source-to-source Reo transpiler that addresses scheduling concerns. The resulting new protocol of the application has only one valid schedule, which is the optimal schedule of the original application. As a result, we can safely outsource the scheduling to the operating system, as the operating system scheduler is forced to(closely) follow this optimal schedule.

We evaluate our work by comparing the throughput of two versions of a cyclo-static dataflow network: one version with the usual protocol, and the other version with a restricted protocol.

In Chapter 9, we present our conclusions and future work.