Symbolic Scheduling

Abu Sayem
Electronic Engineering
Hochschule Hamm-Lippstadt
abu.sayem@stud.hshl.de

***Abstract*—Symbolic scheduling has emerged as a powerful methodology for managing the complexity of task execution in heterogeneous hardware/software (HW/SW) systems. By representing scheduling constraints and system behaviors symbolically, it enables the analysis and synthesis of execution plans without exhaustive enumeration. This seminar paper explores and evaluates various symbolic scheduling techniques, focusing on their application in both hardware and software domains. The paper begins by examining the FunState model introduced by Strehl et al., which combines functional programming and state machines to represent nondeterminism and scheduling explicitly. Subsequently, it delves into the combined scheduling and allocation approach proposed by Cabodi et al., analyzing its methodology and benefits. By comparing these approaches, the paper seeks to identify the most effective symbolic scheduling technique, considering factors such as scalability, efficiency, and applicability to real-world systems.**

I. Introduction

When it comes to embedded systems, combining different types of hardware and software is a big problem when it comes to designing and building systems. For these kinds of systems with different kinds of hardware and software to work well, make the most use of resources, and meet strict timing requirements, scheduling must be done correctly [1]. The difficulty comes from having to coordinate different computing parts, each with its own way of running and needs for resources.Symbolic scheduling has become a very useful way to deal with these problems. By employing symbolic ways to show scheduling constraints and system behaviors, it makes it possible to analyze and create execution plans without having to list all of them. This method makes it easier to create schedules that don't have deadlocks and are limited in time, even when there is nondeterminism and complicated control/data flow specifications [1].The FunState model, which combines functional programming and state machines to show nondeterminism and scheduling, was created by Strehl et al. This model is used internally throughout the design process to facilitate stepwise refinement and hierarchy. It can also be used to show different policies for synchronization, communication, and scheduling [1].Cabodi et al. then came up with an efficient symbolic method that combines scheduling operations and allocating resources at the same time. They use a symbolic scheduling automaton to encode allocation information in order to discover the cheapest way to allocate operation resources that meets a particular timetable while also minimizing the number of registers needed for intermediate outcomes [2].The goal of this seminar paper is to look at and judge different symbolic scheduling methods, with a focus on how they might be used in both hardware and software. The FunState model and how it may be used to show nondeterminism and scheduling will be the first things the paper looks at. After that, it will look at the combined scheduling and allocation technique suggested by Cabodi et al., looking at how it works and what its merits are. The study wants to find the best symbolic scheduling method by comparing different methods and looking at things like how well they work in the real world, how scalable they are, and how efficient they are.

II. Background

Embedded systems are specialized computers that are built into larger mechanical or electrical systems to do specific tasks. They are common in many fields, including as automobile control units, medical devices, industrial automation, and consumer electronics. Task scheduling is an important part of designing embedded systems. It involves figuring out the order and timing of activities such that the system is responsive, efficient, and reliable [1].Rate Monotonic Scheduling (RMS) and Earliest Deadline First (EDF) are examples of static or dynamic priority-based algorithms that are widely used in embedded systems for scheduling. These strategies work well for some situations, but they may not work as well for complex systems with complicated job dependencies, limited resources, and behaviors that are hard to predict. In some cases, the state space of possible task executions might get so big that it becomes impossible to use exhaustive enumeration approaches. [3].To deal with these problems, symbolic methods have been added to the scheduling field. Using mathematical symbols and data structures like Binary Decision Diagrams (BDDs) and Interval Decision Diagrams (IDDs), symbolic scheduling shows tasks, resources, and limitations. This representation makes it possible to express enormous state spaces in a small space and use formal verification methods to make sure that qualities like being deadlock-free and bounded are true. Symbolic scheduling may rapidly assess and create schedules for complex embedded systems without having to list them out clearly. [1].The FunState model that Strehl et al. came up with is a well-known addition to symbolic scheduling. This paradigm uses functional programming and state machines together to clearly show nondeterminism and scheduling in embedded systems. The FunState method makes it easier to create schedules that are free of deadlocks and have limits by using symbolic methods to handle mixed data/control flow specifications and the several types of nondeterminism that are

common in embedded system design [1].Cabodi et al. built on symbolic approaches to come up with a way to schedule operations and allocate resources at the same time. Their method uses a symbolic scheduling automaton to encode information about how to allocate resources. The goal is to identify the cheapest way to allocate resources for operations that meets a certain schedule while also minimizing the number of registers needed for intermediate outcomes. By looking at both scheduling and allocation constraints symbolically, this method makes high-level synthesis more efficient [2].As embedded systems have become more complicated, symbolic scheduling methods have changed over time to keep up. Improvements in how we represent and use symbols have made scheduling algorithms that can handle bigger and more complicated systems more scalable and efficient. These changes have made symbolic scheduling useful for a wider range of embedded system applications, including ones with strict real-time and resource limits. In short, symbolic scheduling is a strong way to handle the problems that come up while scheduling tasks in embedded systems. By using symbolic representations and formal approaches, it makes it possible to quickly create schedules that meet important system needs. As symbolic scheduling approaches continue to improve, they could make embedded systems even more reliable and powerful in application areas that are becoming more demanding.

III. Symbolic Scheduling Approaches

A. The FunState Model: Integrating Functional Programming and State Machines

Strehl et al. introduced the FunState model, which combines functional programming concepts with finite state machines (FSMs) to create a new way to represent and schedule in heterogeneous embedded systems [1]. This integration makes it easier to show both data and control flows clearly, which makes it possible to design and analyze complicated systems that use more than one type of computational model.

In the FunState theory, every part of the system has two parts: one that deals with data and one that deals with control. The data-oriented part uses functional units and FIFO queues to show how data moves, while the control-oriented part uses FSMs to manage the order in which these operations are carried out. This dual structure makes it easy to tell the difference between computation and control. It also provides a modular and structured format that allows for gradual improvement and separation. [1]. One of the most important things about FunState is that it can clearly show nondeterministic behaviors that are common in embedded systems. Nondeterminism, which can happen because of things like varying execution timings, asynchronous events, or inadequate requirements, makes system design very hard. FunState solves this problem by using symbolic methods that model all possible execution routes without having to list them all. This method makes sure that the system's behavior in a wide range of situations is fully recorded and studied, which makes it easier to create schedules that don't cause deadlocks and are limited in time [1]. FunState is an internal representation model that helps schedule and check heterogeneous hardware and software systems in real life. It can be used in systems where parts use different communication and computing models. FunState makes it easier to describe both control and data flows, which lets you make schedules that are free of deadlocks and have limits, making sure that the system works reliably. FunState is strong because it is modular, which makes it easy to scale and reuse, and because it uses symbolic scheduling, which makes it more efficient by not having to list all the paths. But the model does have certain flaws. It might not be able to capture all the details of low-level hardware because it is too abstract, and using symbolic approaches can make calculations take longer, especially in systems with large state fields. FunState is a complete framework for modeling and scheduling different types of embedded systems. It strikes a balance between the necessity for thorough representation and the speed of symbolic analysis. It is a useful tool for designing and analyzing large systems because it combines functional programming and state machines and makes nondeterminism clear.

B. Explicit Scheduler, Symbolic Threads

The Explicit Scheduler and Symbolic Threads (ESST) framework, created by Cimatti et al., is a new way to formally check the correctness of multithreaded software systems, especially those that use cooperative scheduling rules [4]. When threads are scheduled cooperatively, they run without being interrupted until they finish or give control to the scheduler. This approach is used a lot in fields like embedded systems and hardware/software co-design, where it is very important to have exact control over how threads run. When used on cooperative multithreaded systems, traditional software model checking methods sometimes run into problems. Putting these kinds of programs in order can make them less efficient since it hides how threads communicate and how the scheduler works. Also, abstracting the scheduler's actions could make them less accurate, which could lead to false positives during verification. The ESST approach separates the analysis of threads and the scheduler to solve these problems. It uses the unique features of each to make verification faster accurate. The ESST framework turns each thread into its own sequential program and analyzes it using lazy predicate abstraction, which is a method that improves abstractions over time based on counterexamples. This symbolic exploration makes it easy to deal with state spaces that could be unlimited inside each thread. At the same time, the scheduler's actions are modeled directly, directing thread execution in line with the cooperative scheduling policy. By keeping a clear picture of the scheduler, ESST keeps the accuracy of thread interactions and scheduling decisions, which are important for checking properties that depend on certain execution orders [4].ESST uses partial-order reduction (POR) approaches to make the verification process even better. POR finds separate transitions that don't change the overall state of the system when they are run in a different sequence. This cuts down on the amount of interleavings that need to be checked during model checking.

ESST solves the state explosion problem that often comes up when verifying concurrent systems by using POR. This makes the solution more scalable and faster. We used the ESST method on a set of benchmark programs and found that it works well for checking the safety of cooperative multi-threaded systems. ESST works better and more accurately than standard model-checking methods that use sequentialization or abstract scheduler models. This is especially true in systems where scheduling behavior is important for correctness [4].

But ESST does have certain problems. The method is based on a cooperative scheduling model, which could not work with systems that use preemptive scheduling strategies. Also, explicitly modeling the scheduler can help keep execution semantics, but it can make things more complicated when there are complex scheduling behaviors or dynamic thread generation. Even with these problems, ESST is a great step forward in the formal verification of multithreaded software. It is a custom solution for systems where cooperative scheduling is common. The ESST framework does a great job of combining explicit scheduler modeling with symbolic thread analysis to deal with the special problems that come up when trying to verify cooperative multithreaded systems. ESST improves the accuracy and speed of software model verification in this area by keeping the structure of thread-scheduler interactions intact and using advanced abstraction and reduction methods.

C. Symbolic Scheduling in High-Level Synthesis for Packet Processing Pipelines

Adding symbolic scheduling to high-level synthesis (HLS) frameworks has been a key step forward in making hardware accelerators for packet processing pipelines. Soviani's doctoral dissertation presents a complete method that uses symbolic scheduling to improve the allocation of resources and the optimization of pipelines in hardware designs. This is especially important for packet processing applications that have very high performance requirements [5].

Soviani's method adds a high-level synthesis method that uses symbolic scheduling to describe and improve packet processing pipelines. The method allows for the exploration of a large design space by using symbols to express scheduling constraints and resource allocations. This makes it easier to find the best scheduling methods that strike a compromise between throughput, latency, and resource use. This symbolic representation lets us describe data dependencies and control flows very accurately. This is important in packet processing, where timing and resource limitations are very important [5].

This method is very interesting when used in hardware accelerators for processing packets. Processing packets, like classifying, modifying, and forwarding them, needs actions that are fast and have low latency. Soviani's technique makes it possible to automatically create pipelined designs that meet these needs. The method uses symbolic scheduling to make sure that the synthesized pipelines are as fast as possible while still making the best use of resources. [5].

One of the best things about Soviani's technique is that it can handle the complicated scheduling situations that come up in packet processing applications. The symbolic representation of scheduling limitations makes it easier to look at different scheduling options, which leads to the best pipeline configurations. The method also makes it possible to automatically create hardware descriptions from high-level specifications, which speeds up the design process and lowers the chance of human error [5].

But the method does have certain problems. Symbolic scheduling can make the synthesis process take longer since it is so complicated. This is especially true for large designs with limited resources. Also, the method works great for packet processing, but it may need to be changed and tested more before it can be used in other areas with different types of computing [5]. Soviani's use of symbolic scheduling in high-level synthesis makes a strong framework for designing packet processing pipelines that operate best. The method solves the important problems of performance and resource management in hardware accelerator design for packet processing applications by allowing for accurate modeling and efficient exploration of scheduling options [5].

IV. Comparative Analysis of Symbolic Scheduling Techniques: FunState vs. Cabodi et al.'s Approach

Symbolic scheduling is essential in optimizing heterogeneous hardware/software (HW/SW) systems. This section compares two key techniques: the FunState model by Strehl et al. [1] and the symbolic scheduling and allocation approach by Cabodi et al. [2]. The analysis focuses on scalability, efficiency, and real-world applicability.

TABLE I: Comparison of FunState and Cabodi et al.'s Symbolic Scheduling Approaches

Criterion	**FunState Model (Strehl et al.)**	**Cabodi et al.'s Approach**
Scalability	Handles mixed data/control flow and multiple models of computation; suitable for moderate system complexity [1].	Employs BDD-based symbolic representation; scales efficiently with large designs and complex resource constraints [2].
Efficiency	Symbolic techniques avoid path enumeration; enables generation of deadlock-free and bounded schedules [1].	Integrates scheduling and allocation; reduces resource usage and register demand simultaneously [2].
Real-World Applicability	Effective in HW/SW co-design and embedded systems with diverse computation models [1].	Tailored for control-dominated hardware synthesis and industrial-scale circuit design [2].

Each method has its own set of strengths. FunState is better for situations where you need to model system non-determinism and different data/control interactions in a clear way. Its abstraction allows for constrained and deadlock-free

scheduling using symbolic representations, however it doesn't work well for really big designs. Cabodi et al.'s model, on the other hand, is best for high-level synthesis workflows where using resources wisely and combining allocation choices are the most important things. It is not the best tool for modeling how software interacts with people, but it uses Binary Decision Diagrams (BDDs), which makes it very scalable and useful in real-world hardware design flows.

REFERENCES

[1] K. Strehl, L. Thiele, D. Ziegenbein, R. Ernst, and J. Teich, "Scheduling hardware/software systems using symbolic techniques," in *Proceedings of the 7th International Workshop on Hardware/Software Codesign*. ACM, 1999, pp. 173–177.

[2] G. Cabodi, M. Lazarescu, L. Lavagno, S. Nocco, C. Passerone, and S. Quer, "A symbolic approach for the combined solution of scheduling and allocation," in *Proceedings of the 15th International Symposium on System Synthesis*. ACM, 2002, pp. 237–242.

[3] A. Radivojevic and F. Brewer, "Symbolic scheduling techniques," in *Proceedings of the 33rd Annual Design Automation Conference*. ACM, 1996, pp. 529–534.

[4] A. Cimatti, I. Narasamdya, and M. Roveri, "Software model checking with explicit scheduler and symbolic threads," *Logical Methods in Computer Science*, vol. 8, no. 2, pp. 1–42, 2012.

[5] C. Soviani, "High level synthesis for packet processing pipelines," Ph.D. dissertation, Columbia University, 2007. [Online]. Available: https://academiccommons.columbia.edu/doi/10.7916/D8DB88PH