Journal of Systems Architecture 000 (2016) 1–16

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Using design space exploration for finding schedules with guaranteed reaction times of synchronous programs on multi-core architecture

Zhenmin Li[a,*], Heejong Park[a], Avinash Malik[a], Kevin I-Kai Wang[a], Zoran Salcic[a], Boris Kuzmin[b], Michael Glaß[b], Jürgen Teich[b]

[a] University of Auckland, Auckland, New Zealand
[b] University of Erlangen-Nuremberg, Erlangen, Germany

ARTICLE INFO

Article history:
Received 31 March 2016
Revised 13 September 2016
Accepted 15 December 2016
Available online xxx

Keywords:
Synchronous program
SystemJ
Guaranteed reaction time
Design space exploration
Evolutionary algorithm
Scheduling
Multi-core architecture

ABSTRACT

The synchronous model of computation is well suited for real-time systems, because it allows static analysis in order to find and guarantee their reaction times. Today's multi-core systems are becoming the predominant computing platforms. Synchronous programs are typically compiled into single threaded code, which makes them unsuitable for exploiting parallelism of the multi-core platforms. Moreover, static timing analysis becomes highly intractable for multi-core systems. This article proposes a novel methodology that aims at finding the mapping and schedule of synchronous programs that guarantees, statically, reaction times when mapped onto a multi-core system consisting of two types of time-predictable cores. The proposed methodology combines design space exploration based on evolutionary algorithm and scheduling of parts of synchronous programs. It allows minimizing the resource usage in terms of number of cores by finding the mapping and schedule with the guaranteed reaction time for architectures with different number of cores. In particular, we: (a) transform a synchronous program written in synchronous SystemJ to a graph-based model represented with two types of computation nodes suitable for execution on two types of time-predictable cores, (b) perform mapping of computation nodes on a customizable multi-core platform using genetic operations, and (c) generate a resulting static schedule of computation nodes for each mapping as part of the design space exploration. The design flow, from program specification and node mapping to the design space exploration and multi-core scheduling is completely automated.

1. Introduction

The synchronous approach is widely adopted in the design and verification of real-time embedded applications, especially safety-critical systems [1]. Real-time embedded systems are typically implemented as reactive systems that react continuously to input events from the environment by producing appropriate output events in a timely manner. Reactive systems require both functional correctness with respect to input/output specifications and temporal correctness regarding timing constraints. The synchronous approach, embodied in synchronous languages such as Signal [2], Lustre [3] and Esterel [4], supports determinism and concurrency, which are the two essential traits of reactive systems. Both Signal and Lustre are declarative and belong to the family of data-flow languages. The Esterel language is imperative and suitable for modeling control-dominated systems [5]. These languages are based on a formal *Model of Computation* (MoC), which relies on the *synchrony hypothesis* [6]. The synchronous languages based on a formal MoC are amenable not only for functional verification, but also static program analysis for guaranteeing real-time properties of the applications [4,7]. However, they have poor capabilities for data handling. The synchronous subset of the *Globally Asynchronous Locally Synchronous* (GALS) language SystemJ [8] extends Esterel concepts with Java, which significantly enhances data-dominated computations and is used as a design and specification language in our case.

The main feature of the synchronous approach is the introduction of a logical clock, which divides the execution of all synchronous concurrent behaviors (called *reactions* in SystemJ) of a program into discrete instants, called *ticks*. Every synchronous program accepts inputs from the environment at the start of a tick and is assumed to respond in *zero time*, producing outputs as necessary [4]. However, computation always consumes real physical time, which depends on the nature of the computation and the speed

* Corresponding author:
E-mail addresses: zli133@aucklanduni.ac.nz (Z. Li), hpar081@aucklanduni.ac.nz (H. Park), avinash.malik@auckland.ac.nz (A. Malik), kevin.wang@auckland.ac.nz (K.I.-K. Wang), z.salcic@auckland.ac.nz (Z. Salcic), boriskuzmin@gmx.de (B. Kuzmin), glass@cs.fau.de (M. Glaß), teich@cs.fau.de (J. Teich).

http://dx.doi.org/10.1016/j.sysarc.2016.12.003
1383-7621/

of the underlying execution platform. All reactions must complete their computations that correspond to one tick in order to move to the next tick. The time interval between beginnings of two consecutive ticks is typically called the *reaction time*. The longest possible reaction time, which ensures that no input event from the environment is missed, if it can be determined, is called the *Worst Case Reaction Time* (WCRT) of a synchronous program [9]. Statically determining the WCRT of the program is crucial to real-time systems designed using the synchronous programming languages.

Given a synchronous program, calculation of the WCRT always trades off the *analysis time* against the *tightness* of the result. The challenge of finding a *tight* WCRT on a single core itself cannot be underestimated, as it involves both program analysis and analysis of micro-architectural features of the platform it will be executed on [7,10,11]. The latter is becoming more intricate due to the highly unpredictable nature of today's general purpose processors used in real-time applications [12]. This realization has recently prompted the design of time-predictable processor architectures that are amenable to real-time analysis, such as PRET-ARM [13] and *Java Optimized Processor* (JOP) [14].

Due to the growing performance requirements of the real-time embedded applications and the performance limitation of a single-core platform, using multiple *processing elements* (PEs) becomes a logical choice to exploit parallelisms in synchronous programs in order to find as short as possible WCRT. Statically determining the WCRT becomes very challenging for such multi-core systems due to interference of execution caused by thread scheduling, cache organization and replacement policy [15]. The state-of-the-art WCRT analysis techniques for synchronous programs on multi-core architecture isolate the mapping and scheduling problem from WCRT analysis by assuming user-defined or pre-determined thread mapping and scheduling schemes [15,16], which greatly impair the usability of these techniques, as the found WCRT heavily depends on the mapping and schedule of the program.

Because of countless feasible combinations of mapping and schedule of program computation nodes, it is very difficult to determine WCRT statically for a synchronous program on a multi-core platform. Therefore, an alternative and viable method is to find a reaction time which will never be exceeded during execution of the program and at the same time be as close as possible to the WCRT, i.e., *guaranteed reaction time* (GRT). Due to the fact that program mapping and scheduling are both known to be NP-hard problems [17], we perform *design space exploration* (DSE) using an *evolutionary algorithm* (EA), along with a multi-core scheduling algorithm capable of handling conditional branches properly, to find a schedule with the GRT for each instance of a customizable multi-core architecture. EA is inspired from the process of natural evolution [18]. Starting from an initial collection of candidate solutions, the quality of each solution is evaluated, which determines the chance that it will be kept and used as the seed for constructing further candidate solutions. Due to the inefficiency of EAs when solving discrete constrained optimization problems, the SAT-decoding technique utilizing a SAT solver is integrated into our methodology in order to efficiently obtain feasible solutions during DSE [19]. Our target architecture, consisting of two types of time-predictable processor cores, is an efficient execution platform for synchronous programs written in the SystemJ language.

Our contributions can be summarized as follows:

- We propose a DSE-based methodology for mapping the computation nodes of synchronous programs on cores of the target execution architecture and finding the schedule that has guaranteed reaction time. Our methodology starts with a rigorous SAT formulation of feasible mappings of computation nodes onto the multi-core platform and at the end results with a schedule that guarantees correct program execution and reaction time. The resulting schedule is generated using a novel list-scheduling algorithm that schedules the mapped computation nodes in order to find the GRT and is also used by the SystemJ compiler to generate final code for each core.
- We build a fully automated and integrated design flow (framework) that starts with program specification in SystemJ and ends with the scheduled synchronous program with a known GRT.

The remainder of this article is organized as follows. In Section 2, the SystemJ MoC is introduced informally, together with the description of an example of SystemJ program and intermediate graph representation. The multi-core architecture, including the time-predictable interconnect fabric, is also discussed in this section. In Section 3, the examples illustrating different combinations of mapping and schedule are compared, which elicits the idea behind our approach. In Section 4, the design flow based on our proposed methodology for finding the schedules with GRTs using DSE is given, along with the problem formulation. In addition, the mapping constraints represented by a symbolic model encoding, together with the composition of the genetic representation of a solution used by the evolutionary algorithm, are also introduced. Section 5 presents a multi-core scheduling algorithm employed to evaluate the feasible mappings based on found schedules and corresponding reaction times. In Section 6 we present the results of the application of the methodology to different example programs when scheduled on the multi-core system with different numbers of cores. The related work is discussed in Section 7. Finally, this article is concluded in Section 8.

2. Preliminaries

In this section, the synchronous part of SystemJ programming language is introduced, by an example of simple program.

2.1. SystemJ language

For program specification, we employ the synchronous subset of the GALS language SystemJ [8] which can be considered as an extension of Esterel [4]. In addition to reactivity and concurrency statements provided in Esterel, SystemJ allows interleaving of these statements with Java, providing a powerful programming paradigm for the description of complex systems and facilitating both control-driven and data-driven computations. A SystemJ program consists of one or more *clock-domains* (CDs), executing asynchronously with each other. Each CD is a purely synchronous program comprised of one or more concurrent entities called *reactions* executing in lock-step with the logical tick. The reactions within a CD communicate with each other through objects called *signals* using a synchronous broadcast mechanism. Signals have a status and in each logical tick signals are either present or absent. Unlike Esterel, which uses instantaneous semantics, SystemJ uses delayed semantics where emitted signal presence is visible, through the broadcast, in the next logical tick.

The synchronous SystemJ program in Fig. 1(a) consists of three synchronous/parallel reactions: R1, R2 and R3, composed using the synchronous parallel operator (||). The infinite loop in each reaction enforces restart after termination of the current tick, due to the `pause` statement at the end of the loop body. R1 contains a conditional `present` statement that checks for presence of signal IN1 coming from the environment. The presence of signal IN1 leads to the execution of Java method named *method11*, and *method12* is invoked if signal IN1 is absent, in each logical tick. Reaction R2 consists of two sequential method calls, i.e., *method21* and *method22*, separated by an `emit` statement which emits local signal *temp*. Reaction R3 emits output signal OUT1 to the en-

Please cite this article as: Z. Li et al., Using design space exploration for finding schedules with guaranteed reaction times of synchronous programs on multi-core architecture, Journal of Systems Architecture (2016), http://dx.doi.org/10.1016/j.sysarc.2016.12.003