Journal of Systems Architecture 61 (2015) 374–382

Contents lists available at ScienceDirect

Journal of Systems Architecture

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

Message scheduling for real-time interprocessor communication

Stefan Waldherr [a,*], Sigrid Knust [a], Stefan Aust [b]

[a] Institute of Computer Science, University of Osnabrück, Germany
[b] Institute of Computer Science, Clausthal University of Technology, Germany

ARTICLE INFO

Article history:
Received 18 February 2015
Received in revised form 7 May 2015
Accepted 24 June 2015
Available online 2 July 2015

Keywords:
Message scheduling
Real time
Baseline network
Periodic scheduling
Graph coloring

ABSTRACT

In this paper an efficient algorithm is proposed which optimizes periodic message scheduling in a real-time multiprocessor system. The system is based on a many-core single-chip computer architecture and uses a multistage baseline network for inter-core communication. Due to its basic architecture, internal blockings can occur during data transfers, i.e. the baseline network is not real-time capable by itself. Therefore, we propose a scheduling algorithm that may be performed before the execution of an application in order to compute a non-blocking schedule of periodic message transfers. Additionally, we optimize the clock rate of the network subject to the constraint that all data transfers can be performed in a non-blocking way. Our solution algorithm is based on a generalized graph coloring model and a randomized greedy approach. The algorithm was tested on some realistic communication scenarios as they appear in modern electronic car units. Computational results show the effectiveness of the proposed algorithm.

1. Introduction and related work

Computer architectures for real-time applications are widespread but all of them suffer from multiple concurrently executing tasks with real-time constraints on a very small number (usually not more than four) of cores. To guarantee a given time behavior and to solve upcoming conflicts with real time data constraints, an appropriate task scheduling strategy is required. Additionally, data complexity and computational demands of embedded systems are ever increasing. A single-core CPU, however, forces the tasks to be executed inside a given time schedule, and especially in time critical systems with safety requirements an in-depth analysis for the worst-case execution time must be made to guarantee that the chosen sequence of tasks can meet all deadlines. In conventional multitasking computers the task scheduler is responsible for the correct time response for each task, which is more and more difficult to guarantee at a higher number of tasks.

To master the time behavior in complex real-time applications, in [4] the space-sharing concept was introduced that circumvents the competition of tasks for computing resources by using a many-core architecture with an appropriate network on a single chip. This idea takes into account that real-time systems with feed-forward and feedback control, e.g. electronic control units (ECU) in automobiles, operate in a periodic manner. Input data have to be updated cyclically within predefined time periods by sensor polling in the environmental system and processed by the computing system. After a period of time, called cycle time T_{cycl}, the same application code is repeated again which reads sensor data, processes them and outputs actuator data as a computing result. T_{cycl} in turn must be chosen according to the sampling theorem the sensors and actuators must obey. Further, every task i may have its own cycle time $T_{cycl}(i)$.

The space-sharing concept distributes real time software from a powerful processor with complex task scheduling into smaller periodic software components with one dedicated and thus smaller processor for each software component. In this way, task scheduling problems are solved, but on the other hand, the communication network of several hundred processors shows new challenges that cannot be handled by established interprocessor communication systems such as buses. For interprocessor communication on our multiprocessor system on a single silicon chip (MPSoC) we re-use a multistage interconnection network known as baseline network [15], which is able to route a direct connection line from sending to every other receiving node. This network has a simple self-routing mechanism and belongs to the class of $\log_2 N$-networks which are the most hardware-efficient interconnection means. However, the network exhibits internal blockings for some but not all data transfers from inputs to outputs, a fact that usually disqualifies for real-time data transfer. Therefore, the interprocessor communication must be sequenced appropriately

* Corresponding author.
E-mail addresses: stefan.waldherr@uni-osnabrueck.de (S. Waldherr), sigrid.knust@uni-osnabrueck.de (S. Knust), stefan.aust@alumni.tu-clausthal.de (S. Aust).

http://dx.doi.org/10.1016/j.sysarc.2015.06.002
1383-7621/© 2015 Elsevier B.V. All rights reserved.

before a user application can be executed. In this work we present an algorithm that may be used to obtain feasible message schedules. The algorithm is based on modeling the periodic processor communication as a periodic scheduling problem and using graph theoretical concepts for its solution process.

In most cases, messages in the MPSoC have to be sent periodically, where the period is given by the maximum data update rate. Aperiodic events can be handled by the principle of data polling as well, where the update rate is given by the maximum delivery time, i.e., the maximum response time of the system. In fact, based on this update period the message schedule allocates spare time slots that can be used for aperiodic messages, assuming that we are able to abstract the interprocessor communication of the real-time MPSoC to a set of periodic messages. Furthermore, coupled messages, e.g. the speed of certain wheels for stability control, will be scheduled synchronously using identical update periods.

There already exist various approaches to periodic (also called cyclic) scheduling problems. Such problems arise in numerous applications, e.g. in periodic manufacturing or rostering and railroad timetabling. Levner et al. [9] provide a concise survey of applications and solution approaches. Within their survey they discuss several variants: cyclic job shop scheduling deals with the cyclic manufacturing of multiple products which consist of several operations that have to be processed on dedicated machines. The goal is to find a processing order in which the operations are processed, minimizing the time of one production cycle. In cyclic robot scheduling robots are used to periodically transport parts between machines. The robot and machines can only handle one part at a time and after a part is processed by a machine it must be immediately picked up by the robot. Given a robot route and a processing sequence of the parts on machines, the goal is to find completion times of the jobs on the machines such that the time of one cycle is minimized. Another class are cyclic project scheduling problems. Given a set of activities which have to be performed periodically and precedence constraints, the goal once again is to minimize the time of one production cycle. Further variants of this problem consider the minimization of the number of processors on which the projects are processed.

Serafini and Ukovich [14] defined the Periodic Event Scheduling Problem (PESP): for a given set of tasks with a common cycle time T_{cycl} and generalized precedence constraints (defining minimum and maximum delays between the execution of two distinct tasks), find a feasible periodic schedule. It is possible to model situations where each task has its own cycle time $T_{cycl}(i)$ by transforming the problem into a larger problem and using appropriate precedence constraints. However, Serafini and Ukovich deemed this too extensive from a computational point of view and thus defined an extended PESP (EPESP) which is able to handle multiple cycle times. Further, they presented a backtracking algorithm to solve the EPESP.

Within their work, Serafini and Ukovich also performed some experimental computations, testing their PESP algorithm on instances of up to 200 tasks. Experiments using mixed integer programming and constraint programming formulations of the PESP were conducted in [11] solving problems that occur in German railway applications, again only on instances with up to 173 tasks. Further computational studies include heuristic approaches based on local search [11] (173 tasks) and genetic algorithms [12] (100 tasks) among others.

Our problem differs from the cyclic scheduling problems that are found in the literature in an important way. Instead of precedence constraints we have to deal with conflicting processes. Our problem could be well modeled as an EPESP, e.g. by employing generalized precedence constraints for conflicting tasks, stating that there has to be a delay of at least one time unit between them. However, the methods used to solve the general EPESP are computationally intractable for real-world instances since in modern cars 100,000 tasks need to be scheduled. While we will show that our problem is slightly more specific than the general EPESP, our specialized problem is also $\mathcal{NP}$-complete and exact methods are still impractical for real-world instances. Therefore, we propose a heuristic method to obtain results in acceptable time.

The paper is organized as follows: in Section 2, the concept of space-sharing is repeated, additionally, the proposed computer architecture is briefly introduced. This is followed by Section 3, where the used communication system model and the topology of the baseline network are described in detail. In Section 4, we model the problem as a periodic scheduling problem and explain a message scheduling algorithm which is based on an extended graph coloring model. The performance of this algorithm was evaluated in some computational tests which are described in Section 5. Finally, Section 6 draws a conclusion of the performed work.

2. Introduction of space-sharing using a MPSoC

In this section we give a short introduction to the concept of space-sharing and the advantages of the proposed computer architecture for which message scheduling is necessary. A MPSoC may scale up to hundreds of processors or cores as state-of-the-art. In case of up to ten processors, we speak of a multi-core CPU, while the term many-core CPU may be used for higher numbers than that. Multi-core CPUs are commonly coupled via shared memory which is implemented as a 2nd level on-chip cache. However, cache coupling is not scalable with respect to bandwidth and latency. Therefore, if the number of cores is high, in the literature message passing is proposed instead of shared variables. Message passing is often implemented by means of a dynamic multistage interconnection network which is known from supercomputer architectures.

We reuse this approach for a MPSoC with a network-on-chip (NoC) for interprocessor communication. Because of their easier usage we discuss MPSoCs for Field Programmable Gate Arrays (FPGAs) only. During the last few years the capabilities of FPGAs have increased significantly, thus allowing to implement a NoC and a many-core CPU on the same FPGA, albeit with less powerful processors and less memory as in a full-custom design. Additionally, with respect to FPGA technology each CPU core exists as a hardware description only that was synthesized explicitly for the used FPGA type. Therefore, such a core is denoted as soft-core processor.

Space-sharing was firstly introduced in [2]. It can accommodate a large number of tasks that have to obey real-time constraints on one FPGA. Task scheduling as it is known from processor time-sharing is not needed. To avoid competitions for CPU resources between concurrent tasks, each task is statically allocated to an own soft-core processor. The same holds for operating system tasks such as device drivers and IO tasks. As a consequence, the number of processors must match the number of tasks to be executed, and each processor must have enough computing power and memory to meet the timing and memory requirements of its task. Fig. 1 shows the difference between time-sharing and space-sharing. Here, T is the set of tasks and P is the set of processors in the system.

In Fig. 2, a many-core architecture for a MPSoC is shown. It consists of N processor-memory modules (PMMs), which are loosely coupled by an interconnection network for message exchanging between cores. Optionally, a tight coupling is possible as well by a shared memory module which may be connected to the network instead of a core. The network ports can also couple peripheral