Realization of Concurrent Programming in Embedded Systems

Anita Sabo*, Bojan Kuljić**, Tibor Szakáll***, Andor Sagi****

Subotica Tech, Subotica, Serbia

* saboanita@gmail.com ** bojan.kuljic@gmail.com *** szakall.tibor@gmail.com **** peva@vts.su.ac.rs

Abstract — **The task of programming concurrent systems is substantially more difficult than the task of programming sequential systems with respect to both correctness and efficiency. Nowadays multi core processors are common. The tendency in development of embedded hardware and processors are shifting to multi core and multiprocessor setups as well. This means that the problem of easy concurrency is an important problem for embedded systems as well. There are numerous solutions for the problem of concurrency, but not with embedded systems in mind. Due to the constrains of embedded hardware and use cases of embedded systems, specific concurrency solutions are required. In this paper we present a solution which is targeted for embedded systems and builds on existing concurrency algorithms and solutions. The presented method emphasizes on the development and design of concurrent software. In the design of the presented method human factor was taken into consideration as the major influential fact in the successful development of concurrent applications.**

Keywords – concurrent systems, embedded systems, parallel algorithms

I. INTRODUCTION

Concurrent computing is the concurrent (simultaneous) execution of multiple interacting computational tasks. These tasks may be implemented as separate programs, or as a set of processes or threads created by a single program. The tasks may also be executing on a single processor, several processors in close proximity, or distributed across a network. Concurrent computing is related to parallel computing, but focuses more on the interactions between tasks. Correct sequencing of the interactions or communications between different tasks, and the coordination of access to resources that are shared between tasks, are key concerns during the design of concurrent computing systems. In some concurrent computing systems communication between the concurrent components is hidden from the programmer, while in others it must be handled explicitly. Explicit communication can be divided into two classes:

A. Shared memory communication

Concurrent components communicate by altering the contents of shared memory location. This style of concurrent programming usually requires the application of some form of locking (e.g., mutexes (meaning(s) mutual exclusion), semaphores, or monitors) to coordinate between threads. Shared memory communication can be achieved with the use of Software Transactional Memory (STM) [1][2][3]. Software Transactional Memory (STM) is an abstraction for concurrent communication mechanism analogous to database transactions for controlling access to shared memory. The main benefits of STM are composability and modularity. That is, by using STM one can write concurrent abstractions that can be easily composed with any other abstraction built using STM, without exposing the details of how the abstraction ensures safety.

B. Message Passing Communication

Concurrent components communicate by exchanging messages. The exchange of messages may be carried out asynchronously (sometimes referred to as "send and pray"), or one may use a rendezvous style in which the sender blocks until the message is received. Message-passing concurrency tends to be far easier to reason about than shared-memory concurrency, and is typically considered a more robust, although slower, form of concurrent programming. The most basic feature of concurrent programming is illustrated in Figure 1. The numbered nodes present instructions that need to be performed and as seen in the figure certain nodes must be executed simultaneously. Since most of the time intermediate results from the node operations are part of the same calculus this presents great challenge for practical systems. A wide variety of mathematical theories for understanding and analyzing message-passing systems are available, including the Actor model [4]. In computer science, the Actor model is a mathematical model of concurrent computation that treats "actors" as the universal primitives of concurrent digital computation: in response to a message that it receives, an actor can make local decisions, create more actors, send more messages,

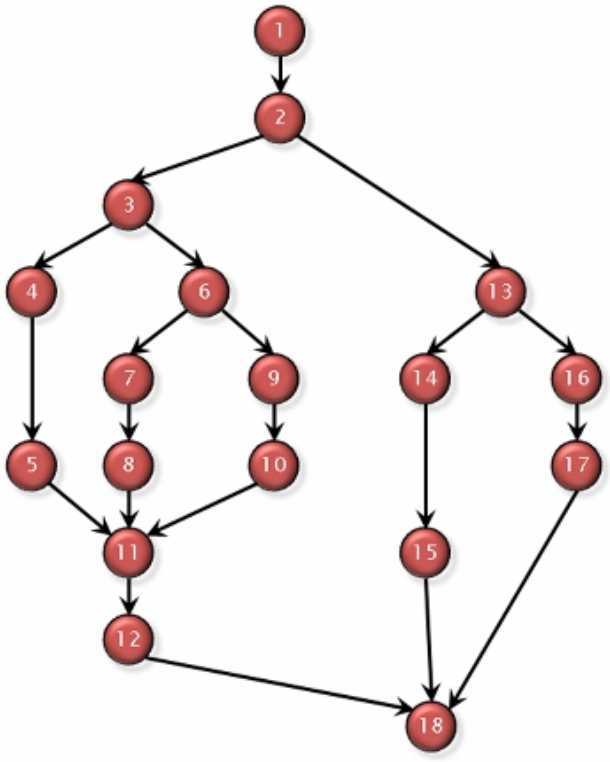

Figure 1. The data flow of a software

and determine how to respond to the next message received. Figure 1 demonstrates the most basic but essential problem in the concurrent programming. Each number represents one process or one operation to be performed. The main goal is not to find the resource for parallel computing but to find the way to pass intermediate results between the numbered nodes.

C. Advantages

Increased application throughput - the number of tasks done in certain time period will increase. High responsiveness for input/output - input/output intensive applications mostly wait for input or output operations to complete. Concurrent programming allows the time that would be spent waiting to be used for another task. It can be stated that there are more appropriate program structures - some problems and problem domains are well-suited to representation as concurrent tasks or processes.

II. COMMUNICATION

In case of distributed systems the performance of parallelization largely depends on the performance of the communication between the peers of the system. Two peers communicate by sending data to each other, therefore the performance of the peers depends on the processing of the data sent and received. The communication data contains the application data as well as the transfer layer data. It is important for the transfer layer to operate with small overhead and provide fast processing. Embedded systems have specific requirements. It is important that the communication meets these requirements.

The design of the presented method is focused around the possibility to support and execute high level optimizations and abstractions on the whole program. The graph-based software layout of the method provides the possibility to execute graph algorithms on the software architecture itself. The graph algorithms operate on the software's logical graph not the execution graph. This provides the possibility for higher level optimizations (super optimization). The architecture is designed to be easily modelable with a domain specific language. This domain specific language eases the development of the software, but its primary purpose is to provide information for higher level optimizations. It can be viewed as the logical description, documentation of the software. Based on the description language it is possible to generate the low level execution of the software, this means that it is not necessary to work at a low level during the development of the software. The development is concentrated around the logic of the application. It focuses on what is to be achieved instead of the small steps that need to be taken in order to get there.

III. REALIZATION IN EMBEDDED SYSTEMS

The architecture of modern embedded systems is based on multi-core or multi-processor setups. This makes concurrent computing an important problem in the case of these systems, as well. The existing algorithms and solutions for concurrency were not designed for embedded systems with resource constraints. In the case of real-time embedded systems it is necessary to meet time and resource constraints. It is important to create algorithms which prioritize these requirements. Also, it is vital to take human factor into consideration and simplify the development of concurrent applications as much as possible and help the transition from the sequential world to the parallel world. It is also important to have the possibility to trace and verify the created concurrent applications. The traditional methods used for parallel programming are not suitable for embedded systems because of the possibility of dead-locks. Dead-locks pose a serious problem for embedded systems [5], because they can cause huge losses. The methods presented in [6] (Actor model and STM), which do not have dead-locks, have increased memory and processing requirements, this also means that achieving real-time execution becomes harder due to the use of garbage collection. Using these methods and taking into account the requirements of embedded systems one can create a method which is easier to use than low-level threading and the resource requirements are negligible. In the development of concurrent software the primary affecting factor is not the method used for parallelization, but the possibility to parallelize the algorithms and the software itself. To create an efficient method for parallel programming, it is important to ease the process of parallelizing software and algorithms. To achieve this, the used method must force the user to a correct, concurrent approach of developing software. This has its drawbacks as well, since the user has to follow the rules set by the method. The presented method has a steep learning curve, due to its requirements toward its usage (software architecture, algorithm implementations, data structures, resource management). On the other hand, these strict rules provide advantages to the users as well, both in correctness of the application and the speed of development. The created applications can be checked by verification algorithms and the integration of parts, created by other users is provided by the method itself. The requirements of the method provide a solid base for the users. In the case of sequential

applications the development, optimization and management is easier than in the case of concurrent applications. Imperative applications when executed have a state. This state can be viewed as the context of the application. The results produced by imperative applications are context-dependent. Imperative applications can produce different results for the same input because of different contexts. Sequential applications execute one action at a given moment with a given context. In the case of concurrent applications, at a given moment, one or more actions are executed with in one or more contexts, where the contexts may affect each other. Concurrent applications can be decomposed into sequential applications which communicate with each other through their input, but their contexts are independent. This is the simplest and cleanest form of concurrent programming.

IV. MAIN PROBLEMS

Embedded systems are designed to execute specific tasks in a specific field. The tasks can range from processing to peripheral control. In the case of peripheral control, concurrent execution is not as important, in most cases the use of event-driven asynchronous execution or collective IO is a better solution [7]. In the case of data- and signal processing systems the parallelization of processing tasks and algorithms is important. It provides a significant advantage in scaling and increasing processing capabilities of the system. The importance of peripheral and resource management is present in data processing systems as well. The processing of the data and peripheral management needs to be synchronized. If we fail to synchronize the data acquisition with data processing the processing will be blocked until the necessary data are acquired, this means that the available resources are not being used effectively. The idea of the presented method is to separate the execution, data management and resource handling parts of the application. The presented method emphasizes on data processing and is made up of separate modules. Every module has a specific task and can only communicate with one other module. These modules are peripheral/resource management module, data management module and the execution module. The execution module is a light weight thread, it does not have its own stack or heap. This is a requirement due to the resource constrains of embedded systems. If required, the stack or heap can be added into the components of the execution thread with to the possibility of extending the components of the execution thread with user-defined data structures. The main advantage of light weight threads is that they have small resource requirements and fast task switching capabilities [8][9]. The execution module interacts with the data manager module which converts raw data to a specific data type and provides input for the execution module. The connection between the data manager and the execution module is based on the Actor model [10] which can be optimally implemented in this case, due to the restrictions put on the execution module which can only read and create new data (types) and cannot modify it. The execution module can be monolithic or modular. The modular composition is required for complex threads were processing is coupled with actions (IO). The execution threads can be built up from two kinds of components, processing and execution/action components. The component used in the execution

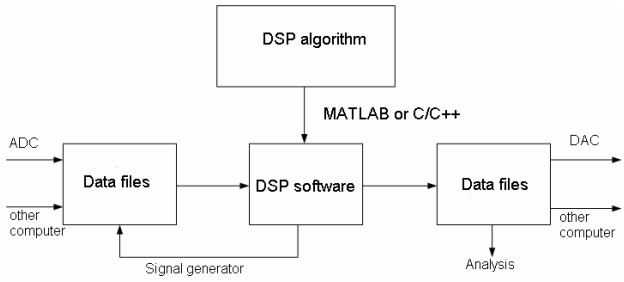

Figure 2. The software development process

module is a type which for a given input type 'a' creates a given type 'b'. This operation will always give the same result for the same input.

The processing component is referentially transparent, meaning it does not support destructive actions [11]. The type variables 'a' and 'b' can have the same types. The action component is similar to the processing component, it is usable in case where one needs to support destructive actions. These components request the execution of specific actions which are received and executed by a transactional unit. The design of the transactional mechanism is based on transactions, just as in software transactional memory. The threads in the execution module are not connected to each other. It is possible to achieve interaction between the threads. One or more execution threads can be joined with the use of the reduce component. The reduce component iterates through the values of the given threads, merging them into one component or value. The merging algorithm is specified by the user, as well as the order of the merging. The joining of the threads follows the MapReduce model, where the map functions correspond to the threads and the reduce function corresponds to the merging algorithm provided by the user [12]. The method introduced in this paper is usable for concurrent programming in real-time embedded systems as well. The complexities of the algorithms used in the method are linear in the worst case. The priority of threads can be specified, this mean that the order of execution can be predetermined. It is possible to calculate the amount of time required to execute a specific action. This way the created systems can be deterministic.

Threads can be separated into two parts. The two parts create a client server architecture, where the server is the data manager and the client is the actions/steps of the thread. The job of the server (producer) is to provide the client (consumer) with data. The server part sends the data to the client part. The server part protects the system form possible collisions due to concurrent access or request to resources. The client part has a simple design it is made up of processing steps and actions.

The job of the asynchronous resource manager is to provide safe access to resources for the server part of the threads. The resource manager does not check the integrity of data, its only job is to provide the execution threads server part with raw data. Parallelization of software is not trivial in most cases. The method presented in the paper takes this fact into consideration. It is an important that the parallelizable and sequential parts of the software can be easily synchronizable. The presented view of software (as seen in Figure 2) is easily implementable into the model of the presented method. Based on the data

flow of the software, it is possible to implement it into the model of the presented method for concurrency.

V. CONCLUSION

Concurrent programming is complex and hard to achieve. In most cases the parallelization of software is not a straightforward and easy task. The realized concurrent programs usually have safety and performance issues. For embedded systems the existing parallelization algorithms and solutions are not optimal due to resource requirements and safety issues. The goal is to realize such a solution for concurrent programming, which is optimal for embedded systems and helps and simplifies the development of concurrent programs. The key to successful development of parallel programs is in the realization of tools which take into consideration the human factors and aspects of parallel development.

The model presented in this paper builds on the advantages of existing parallelization algorithms with human factor as its primary deciding factor. In the development of a concurrent applications, the used parallelization algorithms and solutions are important, but the most important factor is the developer/user itself. To achieve the best possible results, to achieve efficient software, we must concentrate on the most important factor of development, the human (developer).

The presented parallelization model is best applicable if the problem we would like to solve is not trivially parallelizable, which is true for the great number of algorithms and software.

REFERENCES

[1] Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy, "Composable memory transactions," Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 48–60, 2005.

[2] Anthony Discolo, Tim Harris, Simon Marlow, Simon Peyton Jones, Satnam Singh, "Lock -Free Data Structures using STMs in Haskell," Functional and Logic Programming,pp.65–80, 2006.

[3] Tim Harris and Simon Peyton Jones, "Transactional memory with data invariants," ACM SIGPLAN Workshop on Transactional Computing, 2006.

[4] Paul Baran, "On Distributed Communications Networks," IEEE Transactions on Communications Systems, vol. 12, issue 1., pp 1-9, 1964.

[5] César Sanchez, "Deadlock Avoidance for Distributed Real-Time and Embedded", Dissertation, Department of Computer Science of Stanford University, 2007 May.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "MTIO. A multithreaded parallel I/O system," Parallel Processing Symposium,. Proceedings, 11th International, pp. 368-373, 1997.

[7] Girija J. Narlikar, Guy E. Blelloch, "Space-efficient scheduling of nested parallelism," ACM Transactions on Programming Languages and Systems, pp. 138-173, 1999.

[8] Girija J. Narlikar, Guy E. Blelloch, "Space-efficient implementation of nested parallelism," Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1997.

[9] Bondavalli, A.; Simoncini, L., "Functional paradigm for designing dependable large-scale parallel computing systems," Autonomous Decentralized Systems, 1993. Proceedings. ISADS 93, International Symposium on Volume, Issue, 1993, pp. 108 – 114.

[10] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," OSDI'04: Sixth Symposium on Operating System Design and Implementation, 2004.

[11] *Gene Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities," AFIPS Conference Proceedings, (30), pp. 483-485, 1967.*

[12] Rodgers, David P., "Improvements in multiprocessor system design," ACM SIGARCH Computer Architecture News archive Volume 13, Issue 3, pp. 225-231, 1985