Coarse-grain Dataflow Model for Really Distributed Soft-Realtime Systems

Dr. Andrey Sidorov, Lev Shereshovsky
Department of Computer Science and Applied Mathematics
Samara Municipal Nayanova University
196 Molodogvardeyskaya St. Samara 443001, Russia
Phone/Fax: (846 2) 420-182
E-mail: {ac, leo}@nayanova.edu

Abstract

Distributed dataflow architectures present a promising alternative to local control–flow systems. Properly designed dataflow architectures reduce the complexity of a real–time application and facilitate the independent development and validation of the subsystems and support the compositability of the distributed architecture with respect to timeliness, validation, and certification.

Although it is unlikely that pure dataflow architectures as they exist today will be viable commercially in the near future, implementation of dataflow–style processing on existing hardware becomes an increasingly important task. In this paper an abstract dataflow architecture for distributed systems is presented. Unlike most modern dataflow architectures, it does not necessarily require custom–built hardware or specially tuned system software and therefore can be implemented on a wide variety of existing platforms.

The paper is organized as follows. Some background material is presented in Introduction. Section 2 presents the basic details of the developed architectural model. The next section discusses advantages and disadvantages of the suggested model in comparison with major known dataflow architectures. Section 4 describes the software that have been developed to allow the simulation of dataflow behavior in a network of controllers. Some concluding remarks and plans for future work are given in the last section.

Key words: dataflow model, parallel architecture, distributed processing, simulation.

1. Introduction

As the complexity of modern real–time control systems increases the task of specifying, designing and implementing a complete system is becoming more time-consuming and error prone. The tasks to be solved by such systems become ever more complex and place severe limits on the systems' performance and dependability. A common way to increase the system's functionality and achieve fast response times on existing hardware is to employ distributed processing.

One of the biggest challenges in distributed systems is that each of its components, viewed in isolation, may be operating "correctly", but collectively, they exhibit problem behaviors due to
complex interactions among multiple components. While it is possible for a programmer to explicitly choreograph a distributed program by indicating which tasks execute when, how tasks communicate and synchronize, and where data should be placed and how it can be accessed, such an undertaking is extremely complex, especially within real-time systems [5].

At present creating and debugging a distributed real-time system is a difficult and often unsystematic process. Conventional software modules are modified, tuned and integrated by "real-time specialists" during an extensive trial and error period, consuming more than 50% of a project's resources [4]. This deplorable situation may be changed by employing a proper system architecture and a systematic design methodology.

One of the architectures that simplifies concurrency control and improves the system's flexibility is the dataflow model. Dataflow architectures have been promising for years, but little practical results have really emerged. The approach presented here may be of beneficial for practical applications, since it is not a pure dataflow model but rather a programming technique.

The results of recent research indicate that the unification of von Neumann and dataflow models is possible and preferred to treating them as two unrelated, orthogonal computing paradigms [1]. On the one hand, within the dataflow architectures both interprocess synchronization and programming of individual processing elements are relatively simpler with respect to conventional distributed systems [7]. On the other hand, a great number of typical simple tasks of any computational process can be fulfilled very efficiently by computers with conventional architecture, while pure dataflow machines usually perform quite poorly with sequential code. In the next chapters we will introduce a dataflow model and a simulation technique that in some way combine both conventional and dataflow paradigms. This model is expected to be suitable for a wide range of distributed real-time systems.

2. Basic Structure of Macro Dataflow Model

Within the framework of the dataflow architecture a variety of models of computation have emerged. This made 'dataflow' a very vague term, which includes endless variations on the general principle of processing data while it is in motion [10]. All dataflow models can be broadly classified according to several criteria into the following major groups: [9]

- actor granularity (fine-grain vs. coarse-grain (or macro dataflow) systems);
- reconfigurability (static, reconfigurable static and dynamic systems) [12];
- data-driven and demand-driven systems;
- time restrictions (soft real-time vs. hard real-time systems).

This paper will focus on a virtual static data-driven macro dataflow architecture for soft real-time systems. (From now on we'll use the term 'virtual dataflow architecture' or simply 'dataflow architecture' to denote dataflow processing simulation technique.) This model was developed for truly distributed systems but it remains functional within local systems as well. This is one of the coarse-grain dataflow processing techniques, which advocate activating macro dataflow actors by the dataflow principle while executing the represented sequences of instructions by the von Neumann principle.

The proposed dataflow architecture can be modeled by a dataflow graph DFG (sometimes called abstract dataflow machine). A DFG a is a directed graph where the nodes model computation and arcs model communication links. The nodes and arcs of a DFG are usually called actors and channels, respectively. An actor may be considered as an abstract von Neumann machine with a dataflow-oriented interface [9]. The channels are used to carry dynamic data entities between actors. Execution of the graph consists of firing the actors, which then consume input tokens and produce output tokens on the corresponding channels. The tokens allow actors exchange structured data, like arrays, structures, timestamps etc.
An actor is an element of virtual dataflow machine, which independently and in parallel with other elements’ tasks transforms input data $X$ into output data $Y$ according to the specified operator $A$: $Y = A(X)$. Actors are connected to I/O channels by ports through which data flow into and out of the actors. Actors are the principal processing components of the DFGs. An actor automatically fires when all its operands are available on the input channels and the output channels can receive the results of the computation. Communication channels are the only method actors may use to exchange information.

From the point of view of internal implementation, an actor is essentially a virtual sequential von Neumann machine with I/O ports and separate instruction and data space. The virtual processing element is supposed to have mechanisms for activation and suspension of the actor as well as for exchanging tokens with the environment. The state of each I/O port is defined by two Boolean variables — a mask and a flag. The flag indicates the state of the corresponding I/O channel. For input ports the flag is set to true, if and only if the number of tokens in this channel exceeds some predefined threshold. In the case of output ports, the flag is set to true, when the current number of tokens in the channel is less than its volume, i.e. there are ‘empty spaces’ in the channel. A mask in its turn is used to define the state of which ports must be ignored when determining whether the actor can fire or not. If a mask is set to false, the port is considered as masked.

All actors are automatically initialized when the dataflow machine starts its execution. After the initialization phase the way the actor functions depends solely on the internal structure of the its program, and the states of its masks and flags. The dataflow approach requires that execution of an actor’s code is suspended when it attempts to read from or write to an unmasked port whose flag is set to false. A suspended actor fires when all unmasked flags are set to true.

In dataflow models all intercommunications between actors are carried out through communication channels, which model directed arcs of a DFG. Some dataflow architectures for different types of interaction use different communication channels such as informational, temporal, control and regenerative. It can be argued that use of different types of channels is justified unless they are implemented directly in hardware. Our dataflow model has only one unified type of communication channels.

A communication channel can be defined as an element of virtual dataflow machine, which independently and in parallel with other elements’ tasks transfers tokens from an output port of a sending actor to an input port of a receiving actor. The tokens being sent are stored in a channel’s finite queue. A channel works asynchronously, i.e. its operation depends only on the speed of a sender and a receiver. A channel like an actor can exchange tokens only through I/O ports. A channel is not concerned with the formats of data stored in the tokens being sent. Proper interpretation of the data is the programmer’s responsibility.

A communication channel (see Fig. 1) consists of the following main components: i) channel manager which controls data transfer and buffering, ii) a set of state registers and parameters, and iii) local memory used for temporary storage of tokens. The set of channel’s parameters and state registers include:

- channel capacity (CC)
- ready threshold (RT)
- blocking send flag (BS)
- blocking receive flag (BR)
• queue discipline (QD)
• tokens counter (TC)

The first parameter is static, it is set at the model design phase and can not be changed at runtime. The channel capacity sets the maximum number of tokens that the channel can accommodate. The other parameters are dynamic, they reflect the present state of the channel and are stored in its registers. The RT register defines what minimum number of tokens must reside in queue for the channel to be allowed to start the transmission of data. The blocking send flag determines whether actor’s attempts to send a token to a full (TC=CC) channel should be blocked. If BS is set to true, the sender is suspended until there are 'empty places' in the channel’s queue. Similarly, the BR flag defines the behavior of a receiving actor when it tries to read from a channel where TC<RT. Queue discipline sets the order of tokens placement and retrieval such as FIFO, LIFO and priority–based. And finally, the TC register contains the current number of tokens in queue.

The data is sent through channels in the form of tokens. A token is an information entity of the fixed size and structure. There are two main parts of the token — data and meta–data, i.e. data and its attributes. Meta–data in its turn consist of several fields, which describe the essential properties of a token. Among them are token ID, type, priority, creation time, time to live and access mode. Based on this information the system decides how different tokens are handled. The basic operations defined for tokens are: create, delete, read from, write to, duplicate, set priority, set access mode, set type, set time to live, send and receive.

3. Comparison with Existing Models

Due to its elegance and simplicity, the pure dataflow model has been the subject of many research efforts. A number of dataflow computer prototypes have been built and evaluated, and different designs and compiling techniques have been simulated (MIT Static Dataflow Architecture, Manchester Dataflow Computer, MIT Tagged-Token Dataflow Machine, Monsoon,
LAU System, NEC Image Pipelined Processor, and many others) [1], but and the problem of how to map directly dataflow graph programs onto hardware is still an open issue [2].

The next step was to combine the dataflow and control-flow mechanisms. The symbiosis between dataflow and von Neumann architectures is represented by a number of research projects developing von Neumann/dataflow hybrids (MADAME, ETCA Functional Dataflow Architecture, Monsoon, EM-4 and EM-X, MUSE, RAMPS, DTN Dataflow Computer, P-RISC, Stollman Data Flow Machine, ASTOR) [1]. Design of coarse-grain dataflow models was one of the promising directions in first attempts towards hybrid dataflow computing (TAM, *T, ADARC, Pebbles, EARTH). These models were aimed to reach the balance between the parallelism exploitation of dataflow and advantages of conventional architectures. The spectrum of such hybrids was very broad, ranging from conventional multiprocessors with special hardware for message-passing and shared memory access, to associative communication networks of processing units. These machine prototypes had a kind of success but all of them were designed either as compact or tightly coupled architectures. The architecture described here is an attempt to apply principles of coarse-grain dataflow to really (geographically) distributed systems. Some other arguments that justify the viability of the suggested model will be given below.

The rapid improvements in stock microprocessors and the high cost and complexity of developing a modern microprocessor impose practical constraints on what can be built in any experimental project targeted to have competitive performance. For pragmatic reasons the resulting architecture should be as close to conventional architectures as possible to take advantage of the decades of experience in building and programming conventional processors, as well as to make maximum use of existing hardware and software subsystems [5].

Current microprocessors are optimized for sequential code. The synchronization and control overheads in dataflow computing are inherent to fine-grained architectures [3]. Therefore a custom-built hardware will definitely be needed to make dataflow approach a truly viable solution to parallel computing. On the contrary, the coarse-grain dataflow architecture described here can be implemented on stock hardware. This model can be implemented on loosely coupled network architectures, tightly coupled processing systems, such as backplane computers or a combination of the two.

In some sense the macro dataflow model may be viewed as a paradigm of system design and programming rather than actual hardware architecture. The dataflow approach to program construction has proved to be very effective. The great advantage of the dataflow approach is that actors interact in an intuitively clear way, through the channels connecting them. The actors are well insulated from each other in the sense that the internal operation of one cannot affect that of another.

The dataflow approach is asynchronous in nature, because processing nodes may perform their tasks in different speeds [6]. Most coarse-grain dataflow models have no throttling mechanism to prevent data tokens output from a fast actor from piling up on the input to a slow actor [11]. In our model proper use of channel settings (channel volume, blocking modes, queue discipline) and token configuration parameters (time to live, type and priority) can alleviate this problem.

Since most of dataflow models are designed to be directly implemented in hardware, developing a new dataflow architecture making revisions of major parts of the system is a complicated and costly process [8]. The model described here is quite general and does not specify particular implementation details. It is flexible enough to let the designer incorporate additional functionality into his task-specific system, which still retains the inherent properties of dataflow-style processing.

It is universally accepted that macro dataflow architectures are mostly effective for solving computationally-intensive tasks with little data dependency. Typical examples of such
applications are signal processing and virtual reality simulation. Our model is not an exception, it's well-suited for complicated tasks with explicit parallelism and complex pattern of communication between the subtasks.

4. Experimental Implementation

To test the viability of the architecture developed in the framework of this project a data acquisition and control system based on the suggested architecture is being implemented. Currently the system is represented by a loosely-coupled network of 20 Intel and Motorola–based processing units. No custom hardware is required, the entire system is built using only commercially available hardware and system software.

Although it was possible to connect the controllers with a backplane is has been decided to use only a local network in order to investigate the worst-case scenario. The correspondence of logical dataflow architecture and the network topology is not direct. For example several actors may reside in one and the same node and all data channels are physically represented by a common network cable. In this case the efficiency is traded for greater flexibility and reconfigurability of the system.

To implement a macro dataflow–style processing on a network of controllers we developed a set of software modules and C libraries for real–time operating system OS–9. With the help of this software it became possible to program actors in plain C (possibly with OS–9 extensions). As soon as all necessary software modules, compiled actor programs and a valid DFG description are loaded into all network nodes, the system automatically starts its operation.

The first results of experimentation indicate that response times of such macro–dataflow system do not significantly differ from that of systems designed with conventional programming techniques. However the macro–dataflow approach seem to reduce the time required for implementation and debugging of the system by about 10–20 percent.

One of the challenges in dataflow computing is to find a way of executing dataflow style code on a wide variety of platforms. [5] Provided that the software kernel is written in ANSI C with few network extensions, one can easily port this software to any other platform.

5. Concluding Remarks and Future Work

In this paper we have suggested a virtual dataflow architecture for distributed soft real–time systems. It reduces the complexity of a real–time application caused by the system size, the coordination and synchronization of the concurrently executing tasks, and the redundancy management. This architecture facilitates the independent development and validation of the subsystems and supports the compositability of the distributed architecture with respect to timeliness, validation, and certification.

The dataflow model described here can be implemented on a variety of existing hardware using commercially available systems software. We have developed application software to support simulation of the suggested dataflow architecture on a network of industrial controllers. Despite a few shortcomings of the simulator, which are being eliminated, the important impact of the dataflow approach as a paradigm of systems design and programming has been shown.

In future we will investigate the dependability aspects of our dataflow model. Based on the results of this analysis we will probably attempt to develop and simulate a fault–tolerant dataflow architecture. We will also seek a possibility to improve the performance of the simulation software by removing several bottlenecks. Simultaneously, we will try to incorporate the support of interval computations into our application software for dataflow process simulation. The overall goal of our work is to analyze what advantages can give the macro dataflow paradigm when applied to the systems based on existing hardware and systems software.
Bibliography