Parallel Execution of I/O System and Application Functionality

Leif Enblom
Department of Computer Science and Engineering
Mälardalen University, Västerås, Sweden.

Abstract—Many real-time control systems in industry are designed today for single processor architectures. At the same time, more functionality needs to be integrated into the software system. In order to enable correct timely execution of the control and protection applications, designers may need to optimize application code aggressively. Unwanted simplifications of algorithms or low sampling frequencies of the environment may be the result. Functionality in a system, which already has a degree of concurrency, may enable the system to scale onto a multiprocessor environment. This paper discusses and presents results from a study, which separates a substation automation real-time I/O communication system from application level threads in order to exploit existing concurrency. Within the system model described here, as well as in many other system models, it is possible to execute communication mechanisms and applications in parallel. The motivation for this work is let parallel execution of the I/O System and the application enable higher performance for application functionality. The result is more flexibility for the application designers. By describing a model of the real-time substation automation I/O System and extending that model with a mechanism to enable execution in a multiprocessor architecture, we contribute to the understanding of both the composition and the performance issues concerning parallel execution in such industrial systems. Measurements and results originate from execution in an existing system and from the multiprocessor system created.

Index Terms – Real-Time System, I/O System, Multiprocessor.

1. Introduction

Computer systems which operate in an environment in which they are required to respond to external events, not only in a functionally correct way, but also in a correct timely way, are labeled real-time systems. If a system is to be able to respond to and act upon an increasing number of events, or perform an increasing volume of calculations, system performance requirements must be increased. The trend today is to incorporate more functionality into systems both with real-time characteristics and without real-time characteristics. An example of functionality with real-time characteristics is the calculation performed by the real-time parts of the application, while an example of functionality with no or less real-time characteristics is web services. With a real-time software platform built and designed for a single processor hardware architecture, all system components contend for shared resources such as the processor, memory and the interconnects.

Communication and interaction with the environment is an important component in a real-time system. Sensors and actuators respond to and act upon the surrounding environment. The sensors or sampling devices present the collected data to the application, and usually do this by interrupting the processor as data is delivered to an I/O communication system. In the case of a single processor system, the execution of the application is interrupted. Multiple processors can be deployed in order to increase communication performance. Differentiation of the communication system and the application functionality onto separate processor boards has been implemented in different architectures. Examples include the Intel Paragon system [1] and the Spring System [2] in which the purpose was to gain performance and predictability. Dedicated hardware architectures such as the Motorola PowerQUICC architecture [3], have given on-chip support for custom protocol communication, and network processors are becoming commercially available today [9]. Parallel protocol stacks on shared memory multiprocessors have been investigated by, among others, Yates [4] and Björkman [5]. Communication systems can internally exploit parallelism in different forms, such as layer-level parallelism or connection-level parallelism [6], but the work presented in this paper will exclusively investigate the effect of separating I/O communication middleware and application functionality onto different processor boards. In industry, many data collection and sampling I/O boards are developed in-house for special purposes. Protocols and I/O communication systems used in the delivery of data are designed for a specific system. Because of this it can be hard to integrate special purpose hardware accelerators for communication, such as network processors, into the system. The separation of the I/O System and the application onto separate general-purpose processor-based boards can therefore be a way of increasing performance.

The purpose of this work is twofold. Firstly, we are interested in investigating whether a separation of I/O system and application functionality can increase performance for the application functionality. Secondly, we are interested in the real-time aspects of the timeliness of data for systems utilizing such a separation.

In order to investigate the effects of such a separation we first describe in chapter 2, a model of an existing data-driven real-time system. This model describes the I/O system middleware and its interaction with applications. The model is relevant for many industrial control applications and systems. In chapter 3 the model is

extended and we describe how it could be used in distributed multiprocessor system architectures, and in chapter 4 we analyze the execution of the system. Thereafter, in chapter 5, we measure the performance of the different hardware architectures in an existing real-time industrial platform and discuss the benefits and threats of the single processor and multiprocessor configurations of the system. Finally, in chapter 6, we summarize and discuss how further research on the subject could evolve.

2. A Model of a Data-Driven Real-Time System

In this section we describe a model of a data-driven real-time control system. A *data-driven system* is defined as a system in which the execution of the application is dependent on the reception of data from data producers, such as I/O nodes or peripherals. Each time data arrives, the application begins executing on the basis of new data and makes decisions based on the history of the collected data. The core component of the system is the I/O system (from now on abbreviated as the IOSys), which provides access to peripheral boards, actuators and possibly other system components.

2.1. System Architecture

The modules of a processor node in the system are illustrated in Figure 1 below. Components which communicate with peripheral components such as data producers, network peripherals and actuators are illustrated at the bottom of the figure.

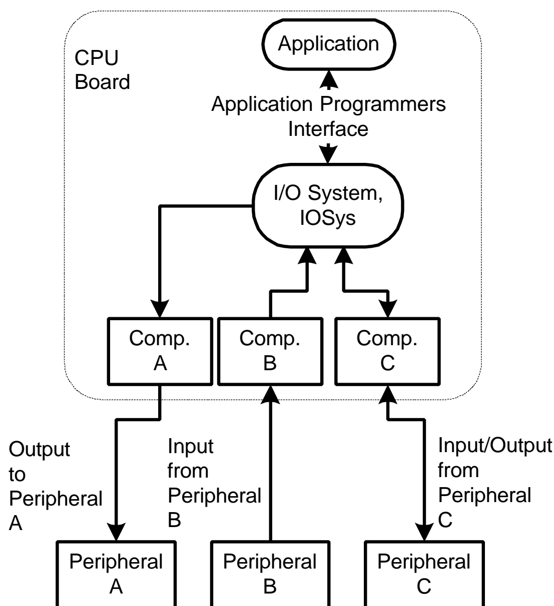

Figure 1. A single processor data-driven node.

Three types of peripherals are represented in the system:

1. An *output peripheral component* which performs actuations toward the environment according to requests from the IOSys via component A.
2. An *input peripheral component* which produces data to corresponding system component B.
3. A combined *input/output component* (for example a network interface) handled by component C.

These three types of peripheral components can be added to the IOSys by the application designer, and the application can define which data is to be received from and/or sent to these. Data is delivered through the Application Programmers Interface (API) to the application, and actuation data is delivered to the IOSys through the same API.

2.2. System Semantics and Functionality

The IOSys provides functionality which can be categorized as *middleware* functionality, serving as a layer of software between the communication facilities, more specifically the transport layer, and the application. Data arriving from I/O producers is delivered to the application thread or threads according to the semantics of the IOSys. The API provided to the application developers enables them to control the run-time functionality of the IOSys. The application can, with the help of the IOSys, be configured towards a certain set of data producers by using this API.

In this data-driven system model, it is possible to combine the delivery of collected and grouped data (see the discussion regarding grouping of data below) from the producers. The application can define data structures (DS) containing data from possibly multiple sources. Thus the application can wait for data items destined for a DS to arrive at the IOSys before the receiving application thread is ready to run. The I/O producers can be said to "publish" data to the IOSys and the application can be said to "subscribe to" data from the I/O producers via the IOSys.

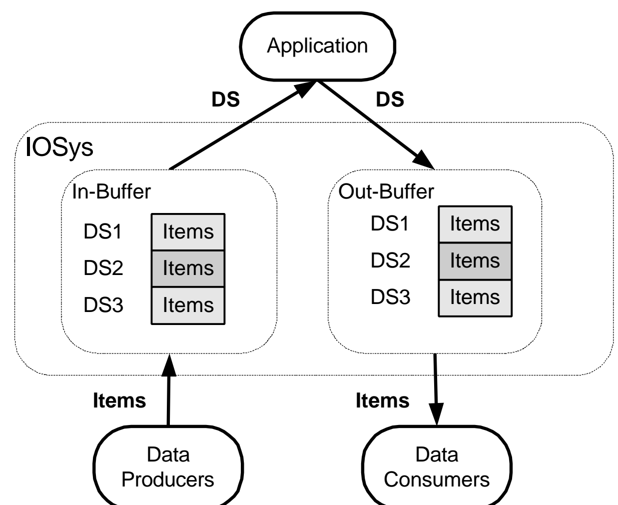

Figure 2, Illustration of data exchange between the data producers/consumers, the I/O communication system and the application.

The IOSys is in effect executing concurrently with the application, buffering and grouping data to be delivered later. Related available mechanisms for real-time systems

are SPLICE [10], NDDS [11] from RTI. A standard for publish/subscribe mechanisms between applications, called Data Distribution Service for Real-Time Systems, is also being defined by OMG [12].

Figure 2 above illustrates a buffer of three data structures, each data set containing a number of data items (possibly from multiple data sources). The communication system will independently from the application group incoming data**.** Data item correlation is based upon timestamps related to each data item in order to achieve a correct snapshot of the environment. Each data producer must therefore be synchronized to a high degree of precision in order to group data into the data structures. Data from remote nodes (which timestamp items produced) to the local node may be delayed, but the IOSys can still group the data into the corresponding DS correctly. The main functionality of the IOSys can be summarized as:

1. Applications can select to receive data items from multiple sources and package them into data structures (DS).
2. Data is delivered whenever a DS is completely filled with items.
3. The correlation, i.e. grouping, of data in the data structures is performed upon the timestamps of each item, i.e. correlation is performed depending on the time at which data items were produced.
4. All data producers must produce data at the same rate; otherwise partly filled data sets would overflow the communication system buffers. The concept of data structure (DS) delivery is dependent on this property.

3. THE DATA-DRIVEN REAL-TIME SYSTEM MODEL APPLIED TO A MULTIPROCESSOR SYSTEM

In a single processor system, as illustrated in Figure 1 above, both software and hardware system components contend for shared resources, such as the processor, the memory hierarchy and the interconnects. Priority-based operating systems therefore provide the assignment of priorities on threads. Threads on a single node are scheduled in an interleaved fashion according to "highest priority first". A thread with a lower priority, ready to execute, may therefore have to wait to run due to the contention for the processor. Whenever such situations develop, the amount of thread level parallelism (TLP) is higher than the underlying computer architecture is able to utilize. Our system model allows for the parallel execution of the IOSys and the application threads. The concurrent execution and buffering that our model provides can therefore be exploited by a parallel system at the interface between the application and the IOSys.

In Figure 3 below, we have introduced a delivery mechanism which enables the application and the IOSys to exchange information. If, for example, the application issues a request to wait for the next data structure, the IOSys will deliver it when it is filled with items through the use of the DS delivery mechanism.

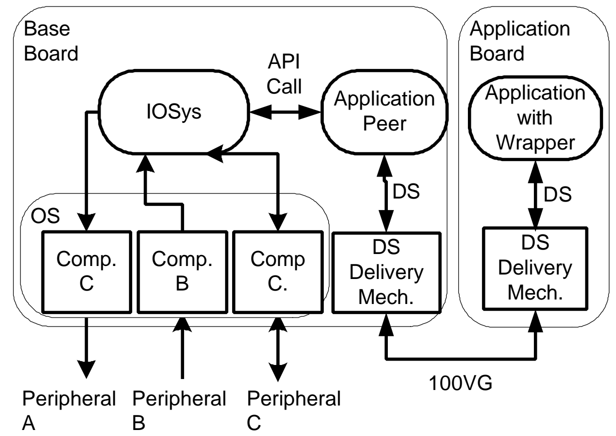

Figure 3, The multiprocessor system model.

In comparison with the single processor system, we have introduced an application peer thread for each application thread which exists on other boards in the system. Calls from the application threads to the IOSys are marshaled by a wrapper class on the application board and demarshaled by the application peer at the base board. This enables the applications to execute Remote Procedure Calls (RPC) across the IOSys API, such as "waiting for data" and "acknowledging data". A problem common to every remote procedure call mechanism is that of opaque references [7]. References to complex data types owned by the IOSys cannot be passed back to the application thread. Therefore such references are substituted with opaque references and complex data structures are flattened. The effect of this mechanism is that the application can be written with the same semantics as are used in the single processor case.

4. EXECUTION ANALYSIS OF THE MODEL

In order to understand the behavior of the system we analyze the execution pattern of the system for a delivery of a data structure to the application for both the single processor case and the multiprocessor case. In Figure 4 we illustrate data delivery over two sample periods (T_{Sample}).

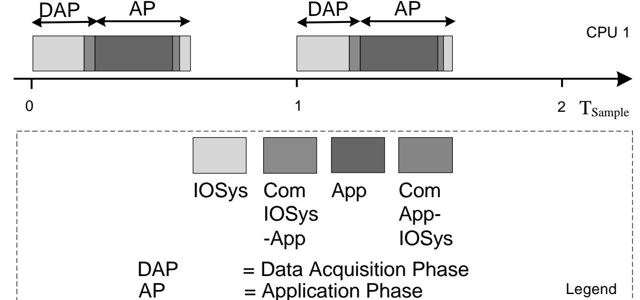

Figure 4. A single processor execution scenario.

We can identify two major phases in the execution, which are common in real-time control systems based on continuous sampling of I/O peripherals. The *Data Acquisition Phase (DAP)* describes the total execution time for all data collection functions and the *Application Phase (AP)* consists of the execution time associated with the application:

$\mathbf{T_{DAP}} = T_{IOSysDAP} + T_{ComIOSysApp}$
$\mathbf{T_{AP}} = T_{App} + T_{ComAppIOSysAP} + T_{IOSysAP}$

Where

- $\mathbf{T_{IOSysDAP}}$ represents the execution time for the IOSys during the Data Acquisition Phase.
- $\mathbf{T_{ComIOSysApp}}$ represents the communication overhead between the IOSys and the application.
- $\mathbf{T_{App}}$ depicts the execution time for the application.
- $\mathbf{T_{ComAppIOSysAP}}$ represents the communication overhead between the application and the IOSys.
- $\mathbf{T_{IOSysAP}}$ represents the execution time in the IOSys during the application phase (acknowledgement of DS).

When the demand on system functionality increases, it may not be possible to execute the application on the single processor as illustrated in Figure 5 below. The first execution of the application thread has not been completed when the data acquisition phase begins. Basically, the rate at which data is produced is higher than the rate at which the application can consume data. This example illustrates only a small timeframe of execution, but is intended to illustrate a transient overload.

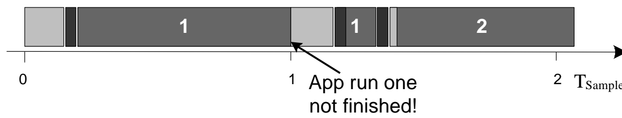

Figure 5. A scenario in which the application has insufficient execution resources.

The execution of the same application in the multiprocessor system would yield an execution diagram as shown in Figure 6 below. A potentially parallel execution of the DAP and the AP which could enable an increase in computing resources for the application is possible. The increased computing resources provided to the application must be compared with how much the communication overhead actually is. As can be seen in Figure 6, there is an overhead in communication which must be weighed against the benefit of having enabled parallel execution.

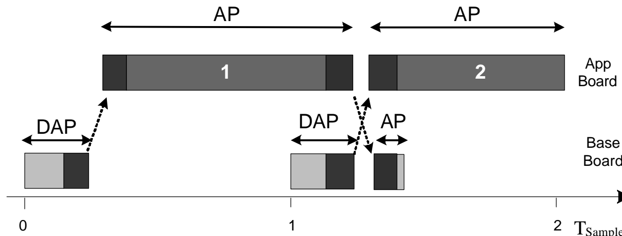

Figure 6. The execution of the application in Figure 5 in the multiprocessor system.

For the application, the communication overhead $T_{ComIOSysApp}$ on the base board plus the communication overhead of $T_{ComAppIOSysAp}$ on the application board is pure overhead. Note that as compared with the single processor system, this overhead is divided into three phases. The first is the execution time of the communication mechanism on the base board. The second is the communication latency of the link (illustrated by the dotted arrows) and the third phase is the execution time of the communication mechanism on the application board. In effect this also means that the "acknowledge" part of the application phase is executing on the base board.

A relevant question is how we view and detect deadline misses in the system. Data structures (DS) are produced periodically by the stream of data items which originate from the data producers. At every instant when a data structure (DS) is ready to be delivered to the application, we can study how many previous DS's have not yet been acknowledged by the application. In short, this property of the system provides the age of buffered data. This view is due to the data-driven structure of the system and deadlines are thus not associated with the execution times of threads, but rather with the delivery and consumption of incoming data.

5. Measurements and Results

The measurement platform that has been used resembles the architecture illustrated in Figure 3. In order to investigate the effects of a separation of the I/O system and the application, we have created a number of system configurations which match the behavior of a data-driven periodic system. The main components in the system which we are interested in investigating are single/multiprocessor configurations with varying I/O data loads and with different application thread characteristics.

5.1. Variation of Hardware and Communication

Different configurations are obtained through the variation of three parameters. These parameters are the hardware configurations, I/O configurations and other system workloads.

The purpose of varying hardware configurations is to permit reasoning about the feasibility of a separation of the IOSys and applications for the respective hardware architectures. The hardware configurations are:

- HW1. Single processor system based on an Intel P3 architecture in a configuration as illustrated Figure 1.
- HW2. Multiprocessor system with two Intel P3 processor boards in a configuration as illustrated in Figure 3.

HW1 represents a computer architecture based on an Intel P3 clocked at a frequency of 266MHz and with a L2 cache. HW2 represents a hardware configuration in which two Intel based processor boards are connected a fiber optical 100VG AnyLAN switched network. The 100VG network technology has been ratified by IEEE as standard 802.12 and achieves a minimum data rate of 100Mb/s.

We also vary the origin of produced data, and four configurations have been set up. As Table 1 states, data originates from remote nodes through communication over a connection-oriented protocol developed in-house. Data is periodically produced at a rate of T_{Sample} and as data items arrive at the node, the IOSys groups them into the data structures (DS).

I/O Configuration	Characteristics
I/O1	I/O originating from *one remote peripheral* producer. Remote peripherals communicate with the processor board through an in-house communication protocol over the 100VG network.
I/O2	I/O originating from *two remote peripheral* producers.
I/O3	I/O originating from *three remote peripheral* producers.
I/O4	I/O originating from *four remote peripheral* producers.

Table 1, I/O Configurations.

5.2. Processor Utilization

A measure of available system performance is the amount of processor utilization over time. The measurements are based on a data collection interval 400 T_{Sample} periods long and in which T_{Sample} is one millisecond in duration. The processor utilization metric gives no actual information regarding for example real-time responsiveness, but indicates the amount of available processing power.

The test includes one application thread which waits for an incoming data structure (DS) and immediately acknowledges this. No other work is performed. All four hardware configurations have been tested together with the four I/O loads, and the results are presented in Figure 7 and Table 2 below.

	I/O1		I/O2		I/O3		I/O4	
HW1	**20,5**		**31,7**		**46,2**		**58,6**	
HW2	Base	App	Base	App	Base	App	Base	App
	50,6	**31,2**	**60,0**	**31,0**	**69,3**	**31,2**	**82,3**	**31,4**

Table 2. Processor utilization over the data collection interval.

We see that the communication mechanism used between the boards does affect performance significantly, but that this overhead is rather constant. For example, HW1 (single Intel P3 board) with the I/O1 configuration leads to a processor utilization of 20,5%. The HW2 multiprocessor configuration indicates that the overhead for the communication between the boards increases the load on each processor by approximately 30%. The actual figures for I/O1 indicate a 30,1% (50,6%-20,5%) and 31,2% overhead for the communication on the base board and the application board respectively. Remember that all I/O from the data producers are handled by the base-board, hence the higher load on that board (50,6% processor utilization with the I/O1 configuration).

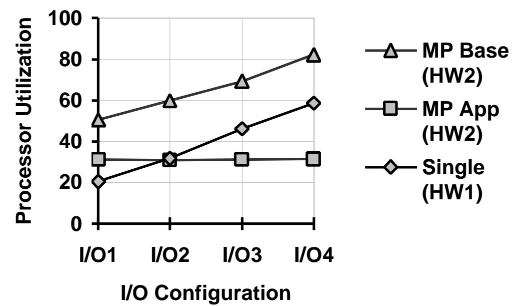

Figure 7. A plot of processor utilization based on the figures in Table 2.

The application thread has, in the multiprocessor case HW2, an almost constant amount of processing power available, regardless of the increased amount of I/O in the I/O1 and I/O2 case. This is due to the effect of only one data structure (DS) delivery across the boards being necessary, irrespective of how many origins the individual items in that DS have. In the I/O1 configuration only one data producer delivers data items, while in configuration I/O2 two data producers deliver data items to the base-board. This form of de-multiplexing of incoming data into data structures (DS) is the foundation of the benefits of such a separation of I/O system and application functionality. The I/O4 multiprocessor configuration showed the largest performance gain for the application functionality configuration. In that case, the gain was 27,2% (58,6%-31,4%) less processor utilization.

Measurements on hardware configurations equipped with PowerPC 603 processors have been conducted as well. The multiprocessor configuration of the PowerPC processor boards does not manage to consume as many data structures (DS) as are produced.

5.3. High Priority System Threads

In order to see how high priority threads affect the execution of application threads we introduce a system thread with various workloads. The priority of the system thread in the single processor configuration was higher than that of the application threads, but lower than that of the communication threads. The thread was to represent functionality such as clock synchronization mechanisms in which synchronization pulses need be handled instantly. Different threads representing different workloads were created, the characteristics of these being the time it took to run them without disturbance on a single board. One-millisecond workloads up to 10-millisecond workloads were created and run on both hardware configurations HW1 and HW2.

In the single processor configuration (HW1) we see that we have a continuously increasing execution time for the system thread compared with the ideal undisturbed execution.

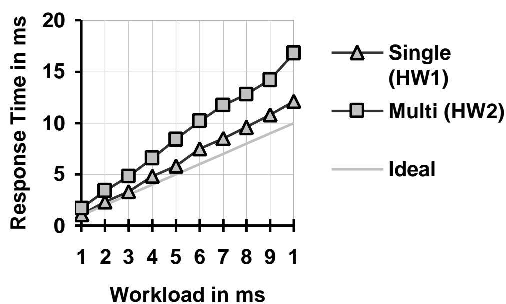

Figure 8. Response time of system thread vs. its undisturbed workload.

The disturbance from the system thread's point of view that leads to this increasing execution time is the data stream arriving continuously from the data producers at a rate of T_{Sample} (in our measurement 1 ms). In the multiprocessor configuration (HW2), the system thread kept the same priority but in this case, the application thread and the system thread did not compete for the same processor. The extra communication overhead between the boards, which is higher prioritized than the system thread, leads however, to an even longer execution time for the system thread (see Figure 8).

For the same measurement, we also kept a log of how many outstanding data structures (DS) not yet acknowledged were queued on the delivery of a new DS, i.e. at each sample period. The result is presented in Figure 9 below.

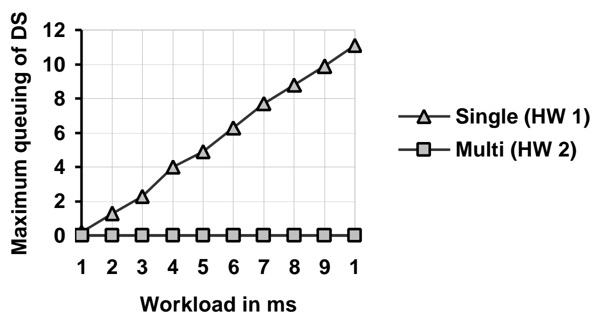

Figure 9. Maximum data structure buffer utilization.

We see that the multiprocessor configuration (HW2) never has the queue to grow**.** This is due to the parallel execution of the application and the system thread. The single processor configuration on the other hand has a continuously growing queue due to the fact that the application thread never has the time to consume data on the incoming queue. A queue with, for example, six queued data structures (DS) will lead to a system that has not reacted on incoming sampled data for at least six T_{Sample} periods. It can be concluded that for such a multiprocessor design (HW2) a tradeoff can be made between the responsiveness of the application vs. how much longer the execution of other system threads will take.

5.4. Synchronous RPC

The multiprocessor design of HW2 assumes a clean interface between the application and the IOSys. In the multiprocessor case, all function calls in the single processor application design must be mapped to a synchronous remote procedure call (RPC). If the function call expects a result of any kind from the base-board, execution of that application thread is stalled. Function calls that do not need a result could be exchanged with an asynchronous RPC call. RPC calls can be very demanding and can have large round-trip times. In our system, in which application threads are executed every millisecond (Tx_{Sample} is 1 millisecond) a high round-trip time can have very degrading effects on performance. We therefore measured the round-trip time of null RPC calls utilizing our inter-board mechanism. The result was a round-trip time of approximately 0.36 milliseconds, which in our system means about a third of a sample period T_{Sample}. A conclusion which must be drawn from this is that RPC calls between the boards must be minimized to the greatest possible extent since even a single RPC call would cause a very high performance degradation of the application. If the semantics of the application permit, all data needed by the application thread should be delivered together with the data structures at the beginning of each sample period.

6. Future Work

Many parameters interact during the execution of a real-time control system. The demand on supporting new functionality is increasing as is the demand for supporting high rates of data from sensors. Multiprocessor solutions need to be considered even in systems which have been designed solely for a single processor environment. We have investigated a separation of application functionality from a communication middleware. Parallelism can be exploited at various other levels of the system and we would like to point out some interesting possible branches of research revolving around this topic for systems with the characteristics similar to the model described in this paper.

The parallel system explored in this paper statically partitions functionality onto different processor boards and into different processing environments. Alternative multiprocessor hardware architectures would for example be Symmetric Multiprocessors (SMPs). The main benefit of such hardware architectures is that it provides a shared view of memory for all the processors and where coherency among processors is achieved by hardware. Since all processors have the same access to hardware components and memory, it should be possible to move a multithreaded application, originally designed for a single processor system, into such an environment. The need for an operating system with SMP support does arise in this context as well as the price/performance ratio. Are SMP systems a valuable alternative and are they feasible in real-time and embedded control environments are questions that need to be answered.

With the introduction of multiple processors, the software developer is faced with more complexity. Much attention needs to be focused on designing multiprocessor software which achieves adequate performance and scales well. As described in this paper, identifying clean interfaces between I/O middleware and applications for existing products can potentially increase performance.

The results have been obtained from a distributed test platform based on network communication, but conceptually, the results should be similar for non cache-coherent non-uniform memory access (NCC-NUMA) hardware architectures based on message passing. Such a solution has been proposed in [8]. An example of NCC-NUMA architecture would be, for example, a Compact PCI (CPCI) back-plane bus-based system equipped with multiple slots, each possibly holding a processor board. The processor boards inserted into the slots are able to access shared memory over the bus-hierarchy, but no memory coherency support is provided by hardware. Issues regarding functional partitioning are much the same as in the distributed system. On the other hand, round-trip latency times for RPC calls would be much smaller due to lower bus latencies and the less processor-demanding message-passing communication mechanism.

The test system presented in this article has been configured working with only one sample frequency. Interesting measures with such systems would be to decrease the sample period time, thus achieving a more frequent data delivery. Questions asked in that area would be how well modern processor architectures behave with this increase in both the amount of arriving data as well as the increased notification overhead in the form of an increased amount of interrupts.

In this paper we have only examined the performance of a system with one application node. Having multiple application nodes with a single I/O node could yield interesting new insights into both the advantages and disadvantages of a separation of the I/O system and the applications.

7. Conclusion

In this paper we have investigated a distributed separation of a real-time I/O communication system (middleware) and application functionality. We have described a model of an existing industrial I/O system and implemented a RPC mechanism between processor boards tailored for the existing I/O system API in order to enable multiprocessor execution.

We have shown that, provided that the interaction between system functionality on the different processors is kept to a minimum, our multiprocessor system can yield up to 27,2% more processor time for the application. Other system functionality, such as high-prioritized system threads, can on the other hand suffer loss of performance. For the substation automation system which has been the target of this work, a distributed multiprocessor system solution may yield more performance for application designers.

8. References

[1] Rudolf Berrendorf, Heribert C. Burg, Ulrich Detert, Ruediger Esser, Michael Gerndt, Renate Knecht, "Intel Paragon XP/S - Architecture, Software Environment, and Performance", *Forschungszentrum Juelich GmbH, Interner Bericht KFA-ZAM-IB-9409*, 1994.

[2] John A. Stankovic, Krithi Ramamritham, Douglas Niehaus, Marty Humphrey, Gary Wallace, "The Spring System: Integrated Support for Complex Real-Time Systems", *The International Journal of Time-Critical Computing Systems*, 16, 223-251, Kluwer Academic Publishers, Boston, 1999.

[3] Motorola Webpage, e-www.Motorola.com.

[4] David J. Yates, "Connection-level parallelism for network protocols on shared-memory multiprocessor servers", *Dissertation at the Department of Computer Science,* University of Massachusetts Amherst, July 1997.

[5] Mats Björkman, "Architectures for High Perfromance Computing", *Dissertation at the Department of Computer Systems*, Uppsala University, ISSN 0283-0574, September 1993.

[6] M. Heddes and E. Rutsche "A survey of parallelism in communication subsystems", *Research Report RZ 2570, IBM Zurich Research Laboratory*, 1994.

[7] George Coulouris, Jean Dollimore, Tim Kindberg, "Distributed Systems, Concepts and Design", Second Edition, *Addison-Wesley*, ISBN 0-201-62433-8, 1994.

[8] Leif Enblom, Lennart Lindh, "Adding Flexibility and Real-Time Performance by Adapting a Single Processor Industrial Application to a Multiprocessor Platform", Proceedings of the ninth Euromicro Workshop on Parallel and Distributed Processing, February 2001.

[9] Stamatis Vassiliadis, Stephan Wong, Sorin Cotofana, "Network Processors: Issues and Prospectives", Proceedings of the 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA2001), Las Vegas, Nevada, USA, June 2001.

[10] Maarten Boasson, "Subscription as a Model for the Architecture of Embedded Systems", Second International Conference on Engineering of Complex Computer Systems, 1996.

[11] Gerardo Pardo-Castellote, Stefaan Sonck Thiebaut, Mark Hamilton, Henry Choi, Real-Time Innovations, inc., http://www.rti.com, September2001.

[12] Data Distribution Service for Real-Time Systems, Request for Proposal, http://www.omg.org, orbos/2001-10-01, October 2001.