Mälardalen University Licentiate Thesis
No. 19

UTILIZING CONCURRENCY TO GAIN PERFORMANCE IN AN INDUSTRIAL AUTOMATION SYSTEM

Leif Enblom

November 2003

Department of Computer Science and Engineering
Mälardalen University
Västerås, Sweden

ISBN 91-88834-21-2
ISSN 1651-9256
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

This work presents and discusses the results from a study, focused on achieving more performance for an industrial real-time control system. The real-time control system is used to protect electrical power stations from being destroyed by strokes of lightning. Sensors in the system continuously collect information on currents and voltages from the electrical power station which the control system protects. The sensors deliver the collected data to a computer system that bases its decisions on the arriving data. When a dangerous situation is detected circuit breakers decouple the hazardous power line.

Today, the computer system is based on a single processor architecture. The problem is that this architecture does not provide enough performance to support demanding system configurations such as more advanced application algorithms and increased amount of data collected from the sensors. In order to obtain correct, timely execution of the protection applications, designers may need to optimize application code aggressively. Unwanted simplifications of algorithms or low sampling frequencies of sensor data may be the result.

The motivation of this work is to study how the real-time control system is affected by being adapted to a multiprocessor or distributed architecture in order to increase the available computing resources. The objective is to improve the performance of system components in general and application components in particular. By identifying components in the existing control system that exhibit a large amount of concurrency and a relatively small amount of data exchange the study found a performance improving solution. The I/O system that is responsible for collecting sensor data and the application functionality both exhibit a large amount of mutual concurrency and may therefore scale on a system with multiple processors. In experimental configurations the I/O system components and an application model were arranged to execute in parallel on two processors. This approach exploits the concurrency available at the interface between the I/O system and application components. Results from measurements show that processing resources (up to 66% when compared with a single processor system configuration) can be freed for application components by utilizing this concurrency in a two processor configuration. The advantage gained is an increase in flexibility for application designers to select a multiprocessor system configuration for demanding applications.

While parallel architectures are used in some industrial systems, not much has been written about the possibilities and threats when legacy systems are adapted to such architectures. By describing a model of an industrial real-time control system and extending that model with a mechanism that enables multiprocessor execution, we contribute to the understanding of both the functional composition and performance issues concerning parallel execution in such industrial systems.

Contents

1 Introduction and Background

The number of applications of electronic systems to the control of industrial processes has increased rapidly during recent decades. Microcontrollers and processors have found their way into processes that were formerly controlled solely by analog or mechanical components. Examples of applications in which mechanical or analog components have been in part or completely replaced by electronic components include vehicles, substation protection equipment and other industrial systems. The benefits of this evolution include increased flexibility for users in configuring functionality, improved possibilities of controlling the process and system under control and reduced costs.

This development has given rise to a growing volume of software executing on microcontrollers/processors embedded in devices located adjacent to the process. Software involved in controlling the system must often satisfy real-time requirements. Depending on the seriousness of the consequences of the system failing to react correctly to the system under control, timing issues become more or less important. The correct and continuous response of a real-time system to stimuli from the environment is of the greatest importance. Figure 1 illustrates the continuous interaction of a real-time control system with the system under control. The real-time control system studied within the scope of this thesis (illustrated in Figure 4) is used to protect electrical power stations from being destroyed by strokes of lightning. Sensors in the system continuously collect information on currents and voltages from the electrical power station (the system under control) which the control system protects. The sensors sample the system under control and deliver the collected data to a computer system that bases its decisions on the arriving data. When a dangerous situation is detected circuit breakers decouple the hazardous power line by issuing an actuation event to a circuit breaker.

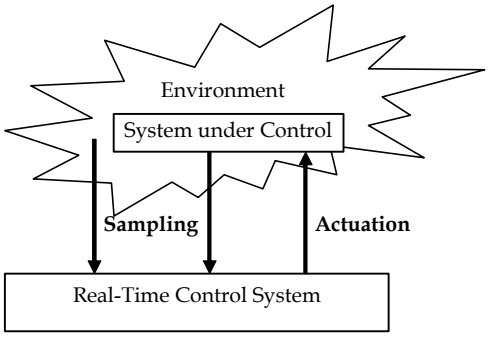


Figure 1, A Real-Time System controlling an industrial process, the System under Control.

Systems in industry often reside in demanding and harsh environments which has called for computer systems with durable mechanical and electrical properties. A

number of bus-architectures and form factors, i.e. the size of the physical boards, have been defined for processor boards and rack-mounts. One architecture designed to reside in such harsh environments is the CompactPCI passive backplane bus-architecture which can be equipped with 6U (Unit) processor boards. In such a system, I/O boards for sampling and actuation purposes reside in the same rack-mount as the processor board executing the application. Figure 2 illustrates such a configuration in which I/O boards produce data for the application to consume.

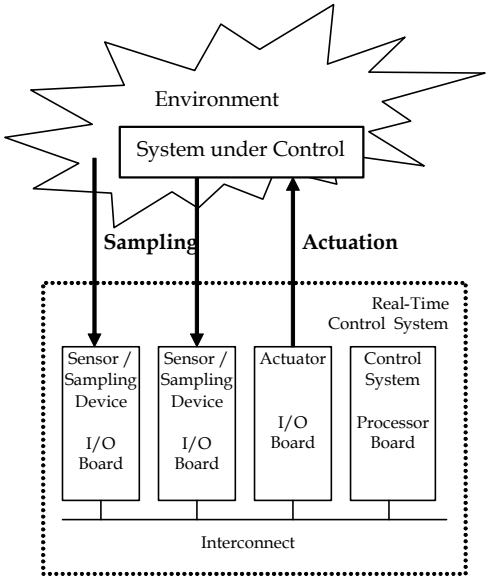


Figure 2, A typical Real-Time Control System and its interaction with an industrial process, the System under Control

This thesis studies a system resembling that illustrated in Figure 2. One or multiple I/O boards provide sampling respective actuation services for the application. The application executes on a single general processor board in the same physical chassis. Figure 2 illustrates a general view of the system that distinguishes between the environment and the system under control. The system under control includes actuators as well as the power grid itself and the environment includes the world surrounding the system under control. The majority of the sampling consists of the sampling of voltages and currents from the system under control (the power grid). We have focused on performance and constraint parameters internal to the real-time control system [Dasdan99 p. 5]. We have therefore not considered the time it takes for data to arrive at the nodes (chassis) but only how the system behaves internally, i.e. the events on the processor boards.

The thesis focuses on a study of parallel execution issues in real-time software system components and how they may scale on a system with multiple processors. The means of achieving scalability is to arrange for the I/O system and the

application components to execute in parallel. This approach allows for exploitation of available concurrency at the interface between the I/O system and application components. Figure 3 illustrates a system configuration where the I/O system components have been partitioned on one processor board and the application component on another processor board.

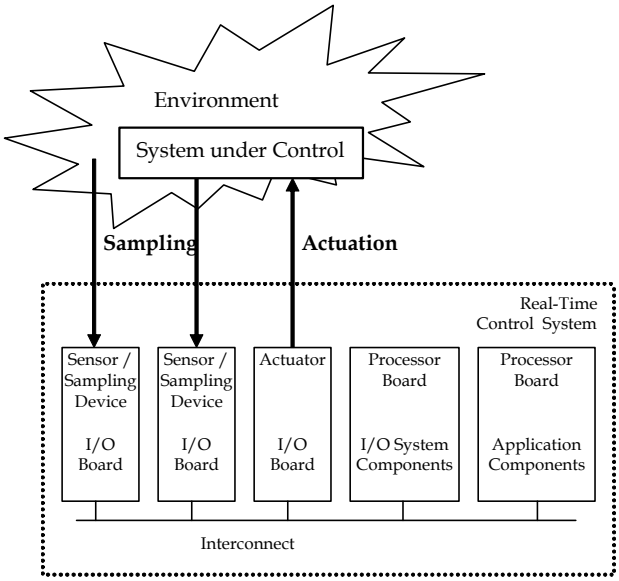


Figure 3, A hardware architecture with multiple processors and the location of the I/O system and application components on two separate processor boards.

The real-time control system is used to protect electrical power systems and especially substations where power distribution lines merge (an example of such systems is illustrated in Figure 4). The protection devices are designated protection relays and are designed to respond to abnormal conditions in the electrical power system [Davies96]. These protection relays operate a circuit-breaker and disconnect a faulty section of the power system when an abnormality occurs. One abnormal condition is a lightning strike on a power line, the excessive current propagating into the substation and threatening to destroy the equipment. In such a situation the circuit breaker must disconnect the power line from the rest of the power grid in as short a time as possible, usually within at the most 20 milliseconds. The system is designed to continuously sample data from the power system grid, delivering this data to interested subscribers. Subscribers to the data are the protection application threads, but other system components, such as event loggers, may also be interested in the data. The protection algorithm needs an amount of historic data (a backlog) to be able to identify the anomaly, which means that the sampling frequency of the system must be high. In our case, the sampling frequency is 1000Hz which means

that the application receives a snapshot of the system every millisecond. This means that the protection application executes once every millisecond, a rather demanding real-time application. If we can find a way of increasing the amount of work performed every millisecond we can increase the volume of sampled data and/or the amount of work which can be performed by the protection application.

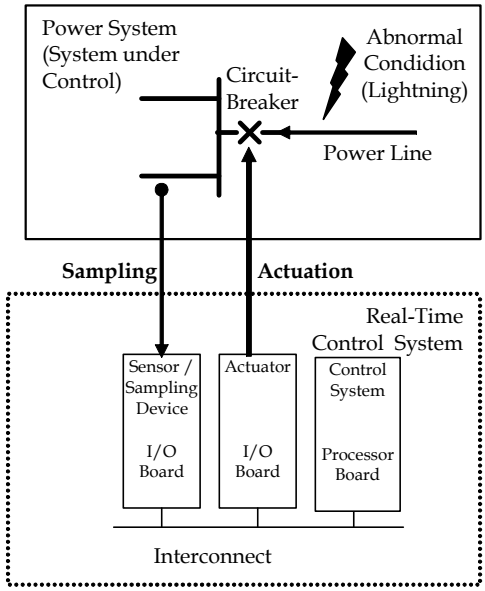


Figure 4, the Real-Time Control System protecting a Power System from abnormal conditions.

1.1 Objective

Large industrial systems are often complex because of the amount of source code and the inherent complexity of the application domain. The history of the design of systems for the specific application domain has led to a complex architecture consisting of different hardware configurations as well as a special software architecture. Application developers base their design on a software platform consisting of components with special functionality tailored for the specific application domain. The objective is to develop a standard way of designing applications without the necessity to "reinvent the wheel" each time. Today application development on the target platform is in part governed by the available performance of the architecture. The accuracy of the application calculations and application components is limited by the available execution capabilities of the architecture, which results in applications having to be optimized and simplified in order to be able to perform their work. This problem can be exemplified by an event

at the beginning of the project. The director of the Platform and Development Department was inspecting the performance of an application prototype running new application code. Performance was poor; the application could not keep up with the amount of data arriving at the node. The director exclaimed: “This is not acceptable. We really need more performance and I hope that the multiprocessor project will be successful soon”. How the problem regarding performance was solved is unknown to the author but illustrates the constant demand for system performance. The obvious way to improve performance is to use higher performance processors but for these, we must wait for the next generation processors to arrive in the market. In the scenario described above, however, such an alternative was not available. Perhaps there is a way to improve performance with the architecture available today? Obviously, the use of multiple processors cooperating in solving the same tasks as before comes to mind. Increased computing resources can help application designers in implementing better systems and thus achieving an advantage over competitors. By introducing multiprocessor or distributed computer architecture we may be able to improve the performance of the system as a whole and especially of application components.

A fundamental issue that arises when introducing a system with multiple processors is the ability for each system component to scale. The operating system must be able to scale on the hardware architecture. Software system components such as application and software platform components must be able to scale on the operating system. The central part of the software platform in this case-study is the I/O system, which enables applications to receive data from I/O boards. The I/O boards can be located in the same chassis as the application, but the I/O system can also establish network connections to I/O boards located remotely (in other chassis). Even though it is possible to connect nodes with the help of a special purpose I/O system, the use of multiple processors to execute the same application has not been discussed widely. These premises were the starting point of the project, leading to the study reported in this thesis.

The overall purpose and motivation of this work is to study how an industrial real-time system designed for a single processor environment is affected by being adapted to a multiprocessor or distributed architecture in order to increase the available computing resources. We therefore need to study how the software system should be designed for execution in such architectures, and we need indications on how performance would be affected by implementing the proposed design. Expected benefits include increased computing resources for the system in general and for application components in particular. Enabling scalability may make increased design space available, giving the designers of applications the freedom to select a more powerful multiprocessor or distributed architecture for demanding applications, and thereby increasing flexibility.

1.2 Research Questions

Given the existing data-driven industrial control system designed for single processor computer architecture, two research questions were asked. These were:

1. "Which time-critical software system components utilize the most processor resources and do they show a large amount of mutual concurrency?"

2. "Given the existing software architecture, can the components requiring the utilization of a considerable proportion of available processor resources scale on a distributed or bus-based multiprocessor architecture and how does the suggested parallel execution of the concurrent system components affect performance?"

When we write "scale on a distributed or bus-based multiprocessor architecture" we mean the ability of the software components to execute on the available multiple processors in the respective architectures. The research questions reflect the iterative process in the search for improved performance and flexibility in the original system. Question 2 is thus a question resulting from question 1. Question 2 targets how performance is affected by parallel execution in the system and we have focused on performance from two points of view (sub-questions of question 2):

2.1. The first point of view relates to *available processing resources* for application and system components in the single and multiprocessor system configurations.

2.2. The second point of view relates to *timing properties* that are important in the real-time system. These timing properties include; latencies on data, response time on important system events as well as inter-processor round-trip latencies.

We conclude the discussion with the answers to these questions, given in Chapter 5, "Conclusion and Future Work".

1.3 Methodology

The starting point of the study was an investigation of the existing system, its components and its behavior. Parallel to that investigation we assembled information on the state-of-the-art as well as the state-of-practice for relevant system components and similar systems. This phase of information collection indicated that a large number of evaluation and design paths were available. At the lowest level, different hardware architectures were available and the choices made at that level affected the possible choices at the higher levels (such as available operating systems). The existing system is designed for a single processor environment which led to different scalability issues depending on the choice of hardware architecture.

The case-study system provided a platform for studying performance and scalability issues for systems with relatively stringent timing requirements. The software and system architecture required us to study different issues related to performance and scalability in parallel systems. Chapter 4 presents the findings of the state-of-the art study. We focused on a bottom-up approach, beginning with parallel hardware architectures and concluding with parallel programming models. Where possible, we have focused on other industrially-related research projects and systems but results from other areas are also included wherever they give relevant insight into the subject.

We have focused on two ways of evaluating the system. The first is how the I/O system can be used in order to scale on a distributed architecture or backplane bus-based multiprocessor architecture. The second is how performance is affected by varying I/O configurations of the system in both the single and multiprocessor architectures.

1.3.1 Limitations

Many parameters, such as price/performance ratio, reliability and fault tolerance are limiting factors that must be considered before a product can be released and sold, but we have limited our study to pure performance and scalability issues. It should be possible, from the results of this study, to obtain information on the available choices and how scalability can be achieved. These performance and scalability results can then be combined with the current prices of processor boards and system components to be able to estimate a price/performance ratio. For example, the cost function of utilizing two chassis (as in the distributed approach) and employing a multiprocessor system in the same chassis can be established. The study has shown how to reduce the processor utilization by approximately 27% at the application node. This represents a 66% increase in computing resources for the application components. It would be possible to obtain a price/performance ratio by calculating the total cost of the hardware used in the distributed architecture configuration as compared with the hardware used in the single processor system configuration. Although we present no price/performance ratio, we are aware of this important factor.

The case-study system is rather limiting because of its design and structure. Periodical sampling of data, the use of a predictable network and fine granular time-synchronization of processor nodes as well as sampling nodes are parameters that contribute to the predictable system behavior. Our results apply to other systems with a similar structure.

1.4 Contribution

This work includes an analysis and model description of a specific real-time industrial system as well as the design and implementation of a mechanism (IOMP) that enables the existing system to scale on a distributed or backplane bus-based multiprocessor architecture. IOMP is an extension of the existing I/O system that

enables applications to utilize I/O system functionality in another execution environment, i.e. address space. IOMP has similarities with a remote procedure call (RPC) mechanism, but it has been tailored and optimized for the existing platform. The new mechanism has been analyzed with respect to performance in a distributed environment and its suitability for use in the existing system. We thereby contribute to the understanding of both the composition and the performance issues of such systems and their use in environments with multiple processors. The work and ideas have been presented in the following published papers and technical reports:

- The paper entitled "Adding Flexibility and Real-Time Performance by Adapting a Single Processor Industrial Application to a Multiprocessor Platform" [Enblom2001], published in the Proceedings of the ninth Euromicro Workshop on Parallel and Distributed Processing, issued in conjunction with the workshop with the same name held in Mantova, Italy in February 2001. It presents a design solution for an industrial system multiprocessor rack-mount CompactPCI NUMA architecture in which the operating system is implemented in hardware. The paper was written mainly by the author under the supervision of Professor Lennart Lindh of Mälardalen University and the approach was a legacy of a master's thesis developed at the university. In addition to the concept of accelerating the operating system with custom hardware, the paper includes the first thoughts on the partitioning issues of existing software.
- The paper entitled "Parallel Execution of I/O System and Application Functionality" [Enblom2003], presented at the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA2003) held in Las Vegas, USA, June 2003. It presents the model of the case-study system and introduces measurements and results from an evaluation in a distributed system environment. The paper was written entirely by the author and the work was performed in an industry environment by the author. Three performance metrics are studied in the comparison between the single processor system and the distributed system; processor utilization, latencies on data and inter-processor latencies. The results of the paper are the basis of the discussion in chapter four.
- The technical report entitled "PCI-Bus Communication Performance" [Enblom2001_2], filed as a technical report at Mälardalen University. It presents the design and performance results from an implementation of a message-based communication mechanism that enables communication between processor boards located on a CompactPCI-bus. Section 4.5 bases its discussion on the results from this technical report.
- The state-of-the-art technical report entitled "Parallel Computer System Components" [Enblom2003_02], presented as a Mälardalen Real-Time Research Center Technical Report at Mälardalen University. It describes multiprocessor hardware architectures, operating systems for multiprocessor architectures, as

well as communication mechanisms relevant to the work performed in this dissertation. It is included as chapter 2 in this thesis.

1.5 Related Work

The problem of scaling the software platform for systems which resemble that described in this thesis involves multiple disciplines. It is related to hardware architectures suitable for such systems, the predictability and bandwidth of the available interconnects as well as operating system support for this architecture. It is also related to offloading I/O system and communication functionality by using dedicated hardware, such as the hardware communication assists described in chapter 3.2.1. The work, however, is conceptually related to the benefits of utilizing proxy servers for offloading application nodes. Conceptually, there are resemblances between the approach of separating the I/O system and the application components on different nodes and the use of proxy nodes in networked environments. We have mentioned a proxy server architecture (CSP) in section 2.4.5 and we identify similarities and differences with our approach with respect to that architecture. Similarities include:

- The offloading and decoupling of communication processing from servers (the application node in our case).
- The data is forwarded to the application nodes with the help of lightweight transport protocol, which applies for our bus-based approach that utilizes a message queue for communication between the I/O system node and the application node.

Differences include:

- The proxy server (CSP) approach is based on the concept of decoupling TCP/IP processing from application nodes. We, on the other hand, decouple a publish/subscribe I/O system from application nodes. The inherent difference is that the CSP approach is connection-oriented from end-point to end-point. The I/O system used in the case-study is not connection-oriented but rather data-oriented.

In summary, mechanisms that dedicate computational resources for the purpose of offloading application components are related to the work performed in the scope of this thesis. The difference and originality of this work lie in the special limitations and properties that were available to us in the case-study system and the semantics of data delivery (correlation and delivery on complete arrival of data) that enabled us to exploit the existing concurrency between I/O system and application components.

1.6 Outline of the Thesis

The thesis consists of four main chapters describing the work performed within the scope of this project. It begins with a state-of-the art report (chapter 2) which gives insight into techniques and concepts that have been of importance for the understanding of the underlying set of problems with systems using multiple processors. In chapter 3 we discuss alternatives for improving performance in the case-study system, and discuss the design alternatives that were available to us, i.e. the design space. Chapter 4 describes the work performed in executing the I/O system and application/client components in a distributed network; creating a model and synthetic execution analysis as well as presenting relevant measurements of important metrics of execution in the distributed system. Section 4.5 presents an implementation and performance measurement of a message-based communication mechanism used for communication over a PCI bus-hierarchy. The objective of that is to determine how communication performance of a more tightly coupled system (as compared with the distributed architecture) would affect parallel execution of I/O system and application/client components. Finally, in chapter 5, we sum up with our conclusions and experiences from studying multiple processor systems available for use in industrial automation systems, and present possible future work on the subject.

2 Parallel Computer System Components

Many issues need to be considered when migrating from a single processor system to an architecture that uses multiple processors. System designers are facing a plethora of design choices and need to understand state-of-the art technologies from many domains including hardware architecture, software architecture and programming techniques. This chapter intends to describe both state-of-the art and state-of-practice of technology domains we have identified and used during the work. We will start by establishing properties and terminology of parallel systems that is needed in the rest of the discussion. Thereafter we start a bottom-up discussion that begins with a chapter on parallel hardware architectures. Next chapter crosses into the domain of software and discusses the lowest level of software, which is the operating system. We focus on operating systems for multiprocessor and distributed system. Important software components include the communication mechanisms, especially for parallel and real-time systems. The chapter labeled "Parallel and Real-Time Communication Mechanisms" discusses such mechanisms. Finally, we bring up and discuss relevant programming models that are used in parallel systems. Focus has been on industrial applications where possible.

2.1 Properties and Terminology of Parallel Systems

Issues regarding parallelism cut through all levels of technology, regardless of whether the technology is targeted at Systems on Chip (SoC) or software executing on commercial off-the-shelf (COTS) architectures. The system designer will have to handle issues such as synchronization, communication and shared resources. This chapter will discuss properties concerning parallel systems as well as introduce terminology that will be used throughout the rest of this text.

2.1.1 Speedup

The term *speedup* is related to parallel computer system performance and is widely used as a metric to quantify the benefit of executing programs on a parallel architecture compared to executing the same program on single processor architecture. Speedup can be defined [Wilkinson99 p. 26] as:

$$Speedup(N) = S(N) = \frac{\text{Time to execute P on 1 processing unit}}{\text{Time to execute P on N processing units}}$$

where P is the program that is being run and N quantifies the number of processing units in the parallel architecture. The definition of speedup is thus the time it takes for a program to execute on 1 processing unit divided by the time it takes to execute on N processing units. A speedup of more than 1, $S(N)>1$, represents a speedup, while a speedup of less than 1, $S(N)<1$, will in effect be a *slowdown*. A

speedup where *S(N)*=N illustrates a situation with a *linear speedup* and a speedup where *S(N)*>N depicts a *super-linear speedup*. This definition of speedup applies to a program that is fixed in problem size (problem-constrained scaling).

2.1.2 Amdahl's Law

In a pioneering article published in 1967 [Amdahl67], Gene Amdahl described the problems of achieving speedup of programs on a parallel architectures. The maximum amount of speedup that a program can achieve is limited to how large proportion of it can execute concurrently and how large proportion must execute sequentially. Amdahl's law is expressed as:

$$S(N) = \frac{1}{f + \frac{1-f}{N}}$$

S(N) defines the speedup, *f* is the fraction of sequential work and *N* quantifies the number of processors. Now, as the number of processors approaches infinity ($N \to \infty$) we get this relation:

$$S(\infty) \to \frac{1}{f}$$

This shows that the maximum amount of speedup possible for the program is dependent on how much of the program that is inherently sequential.

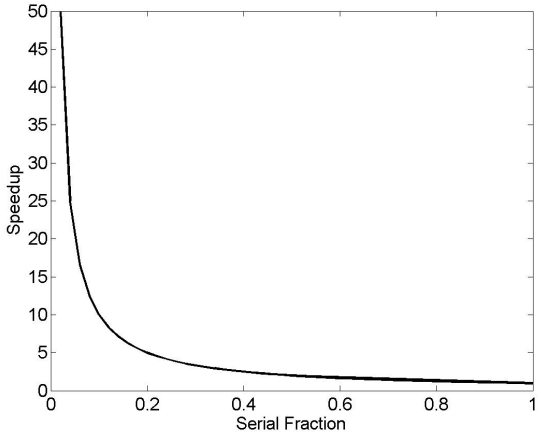


Figure 5, Speedup under Amdahl's law.

Let for example the fraction of sequential work for a certain program be 2%. This will give an upper limit of speedup that approaches 50 (1 / 0.02). The maximum

speedup according to Amdahl's law, as a function of the serial fraction in the program, follows the curve in Figure 5 above.

2.1.3 Gustafson's Law

Amdahl's law portrays a pessimistic picture for parallel computer architects, but despite of this parallel systems are sold and employed successfully. Gustafson [Gustafson88] have tried to explain why the implicated performance problem portrayed by Amdahl does not explain the whole picture of parallel system speedup. Gustafson argues that by changing the problem size, such amount of data used, it is possible to achieve a super-linear speedup. The reason for this is that in practice the problem size is not independent of the number of processors. Amdahl's law will hold true for a system where the problem size does not change. Gustafson argues that this is not the situation for many problems and that it is more realistic to assume a constant run-time in contrast to a constant problem size.

2.1.4 Concurrency and Parallelism

Concurrency is a term that quantifies the amount of operations that can execute independent of each other. If a system component does not interact with or depend on any other component we can say that the components are isolated. The component has its own state and does not change its behavior based on events from other components. A system or program with totally independent and isolated parts is sometimes labeled embarrassingly parallel [Wilkinson99 p.82], but not all problems are this appropriate for parallel execution.

There is a distinction between the definition of parallel execution and concurrent execution [Lawson92 p. 35]. For example, two processes in a uniprocessor system are said to execute concurrently but they never execute in parallel. They execute in what can be called a pseudo-parallel manner (*interleaved*), scheduled by the operating system. Two processes executing on separate processors on the other hand are said to execute in parallel.

Two or more system components, such as processes, can exploit parallelism in a variety of ways. Andrews [Andrews2000 p. 11-26] identifies five concurrent programming paradigms that exploit parallelism differently. These are:

1. *Iterative Parallelism*. An application utilizing iterative parallelism usually consists of a pool of identical processes that cooperate in solving a problem. Each process is parameterized and consists of one ore more loops, each performing a part of the job.
2. *Recursive Parallelism*. In this form of parallelism recursive properties are exploited to utilize multiple processing elements. In each recursive step processes are spawned and execute concurrently. This may lead to a vast amount of processes, which poses a performance threat. Therefore parallel recursive algorithms often prune into sequential recursive calls (without spawning new processes) at a certain level (for example when the number of

processes exceeds a certain number). Sorting algorithms and chess games are problem domains that usually use this kind of concurrency.

3. *Producers and Consumers*. Processes that communicate utilize this form of concurrency, usually in the form of pipelining. Each process performs its calculations and forwards its results to its collaborators and each process is a part of the pipeline.
4. *Client and Servers*. In this is a form of parallelism processes communicate and exchange data, but now in a two-way manner. The client sends requests to the server, which responds with an answer. This form of concurrency has an analogy in procedure calls, invoked either locally or remotely (RPC).
5. *Peers*. This form of parallelism utilizes peer processes that interact and communicate with the help of messages. No global variables are used and usually a coordinator distributes tasks to the waiting processes. This form of parallelism is common in distributed programming.

2.1.5 Shared Resources

There are several resources in a computer that can be shared. The most obvious are the processor, the memory and the interconnects. Sections of a program that act on shared resources are said to execute in a critical section [Dijkstra65]. If two or more processors contend for the same resource they are in a so-called *race condition* [Tanenbaum92 p. 34]. Deadlock occurs when a system cannot progress due to a situation where processes wait indefinitely for a resource. A formal definition of a deadlock situation is [Tanenbaum92 p. 242]:

> *A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.*

If this is the case, all processes will wait forever to acquire the resource. Another form of unwanted lock is the so-called livelock. Livelock [Culler99, p. 379] is a complex form of deadlock, where subsystems interact and transactions are being performed, but where no progress towards the final result is made. The execution is going to continue forever, without the final result ever being presented. When designing system components such as for example message queues for inter-processor communication mechanisms contention and race conditions for shared resources are an important issue.

2.1.6 Three levels of Parallelism

The first computers executed one instruction each cycle, starting the next only after the first had finished completely. These computers utilized *bit-level parallelism*; instructions operated on more than one bit at a time. Over time, the processors incorporated more and more of this parallelism, increasing the width of the data path from 8 bits up to the 64-bit processors that are available today [Culler99 p. 15]. Next level of utilizing parallelism was to exploit *Instruction Level Parallelism*, ILP. The RISC processors developed during the 1990's have improved ILP substantially, but

aggressive machine design studies (with perfect branch prediction and unlimited processing resources) show that the amount of issued instructions per cycle does not increase performance infinitely. A study has shown that ILP has been exploited to what the limit for many applications [Lam92]. Other studies have shown that instruction throughput does not increase significantly, even with a four-way superscalar, as shown in Figure 6.

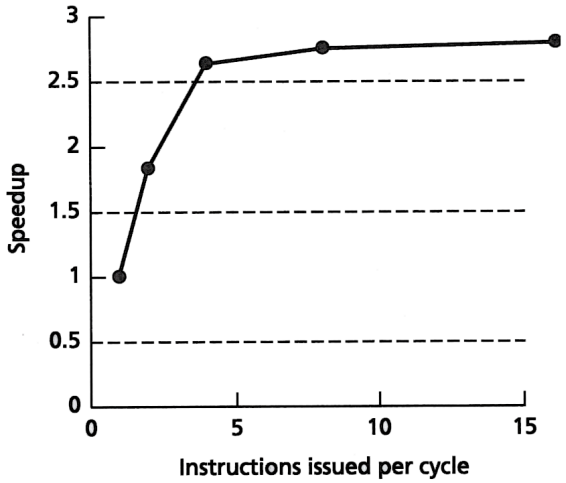


Figure 6, a graph showing the estimated processor speedup with ideal superscalar resources available [Culler99 p. 18].

In many processor architectures where a concurrent flow of instructions is executing, *relaxed memory consistency models* [Adve96] are used in order to promote high ILP. Relaxed memory consistency models let memory operations compete out of order. A processor that completes memory operations in order is said to support serial consistency. In some processor architectures, which utilize relaxed memory consistency models, special memory barriers instructions can be inserted into the sequential instruction flow. This can be a nuisance for the programmer of the operating system, and even for application programmers. For example, the PowerPC architecture [Motorola] requires a program wanting to access a peripheral register to issue a 'sync' or 'eieio' instruction before continuing using values involved in the transaction. As described in [Motorola94]: "The sync instruction ensures that all instructions previously initiated appear to have completed before any subsequent instructions are initiated".

Other ways of increasing processor performance have been studied. Multiprocessors that allow for multiple threads executing simultaneously is an alternative. The implied limit of ILP pushes the evolution of processors and today some processors exploit *Thread Level Parallelism*, TLP. Future processors may include implementations that utilize TLP on chip, such as the Stanford Hydra

[Hammond2000]. This chip supports thread-level speculation, which means that all threads run in parallel until a dependency is detected. We will discuss this issue further in section 2.2.2.

2.1.7 Locality

Data (or spatial) locality is an important factor when designing both hardware architectures as well as software systems. Good spatial locality of a program depicts a situation where memory locations that are addressed close to other addresses have a higher chance of being accessed than other addresses. Caches exploit spatial locality (the cache lines) available in application programs to reduce memory access latencies. The closer the data is to the execution engine of the CPU, the faster the memory accesses can be completed. System architects should strive to keep the data as close to the CPU as possible, ideally the CPU should be able to access all memory within one cycle.

Temporal locality is the property that expresses the inclination of a program to repeatedly access the same memory locations. A program that exhibits good temporal locality has a high probability of accessing the same memory locations multiple times in the near future.

2.1.8 Latency and Bandwidth

The effects of bandwidth and latencies have been studied in multiple papers, among them in a paper by Chong [Chong98]. Latency is associated to communication and is a problem for modern computer system architects. Memory latencies increase every year relative to processor execution engine performance. Three approaches to alleviate this have been identified [Johnson92 p. 2]:

1. *Avoid* long latency operations by introducing a memory hierarchy (caches). Data reuse is enhanced through exploitation of temporal and spatial locality in the program.
2. *Reduce* latencies [Gupta91] by minimizing the physical communication distance and thereby exploit physical locality.
3. *Tolerate* long latencies [Gupta91] by exploiting multithreading, prefetching and relaxed memory consistency techniques. These approaches exploit overlap of long latency operations. Latency tolerance aspects for software distributed shared memory mechanisms has for example been studied by Mowry [Mowry98].

Bandwidth describes the rate at which data is delivered from one component to another. Latency is related to bandwidth in the sense that low bandwidth can incur higher latencies in the system, especially for request-reply protocols.

2.1.9 Synchronization

An important mechanism in parallel and concurrent systems is that of synchronization. The purpose is to achieve program order among processes and/or to achieve mutual exclusion for shared data. These two methods are called synchronization for precedence and synchronization for mutual exclusion [Quinn94 p.91]. Different forms of synchronization include barrier synchronization and point-to-point synchronization. *Barrier synchronization* is used whenever multiple processes agree to wait for every other peer to advance to a certain point. At that point they all synchronize and are thereafter allowed to proceed. *Point-to-point synchronization* depicts two processes and their mutual synchronization. An example of point-to-point synchronization is when a message based communication mechanism utilizes send and receive primitives to synchronize.

Synchronization relies on indivisible atomic read-modify-write actions. Each hardware architecture has its own mechanism to provide for atomic actions. Compare&swap and test&set are two examples of processor instructions that are indivisible, and on which all other forms of synchronization can be built. In a multiprocessor system, atomic actions must be performed by other means. This includes mechanisms that lock the local bus in order to achieve exclusive read-modify-write access to a shared variable. Busy wait/spin-locks can enable *mutual exclusion*, but consume resources since the processor is not doing any useful work at all while spinning. Other threads on the same processor should be able to run while the first thread is waiting for the lock. A common technique of guaranteeing mutual exclusion in a single processor system is by disabling interrupts.

2.1.10 Granularity

Granularity is used to describe the number and size of tasks into which a problem is decomposed [Grama2003 p. 89]. The spectrum stretches from *fine* to *coarsely* grain and the granularity of tasks (fine or coarse grain) depicts the possibility to divide a task into multiple threads and thus exploit concurrency. The granularity of communication depicts for example the amount of data being passed in a memory reference [Culler99 p.186-187]. Granularity as a term can be applied to describe the characteristics of synchronization as well.

2.1.11 Scalability

Scalability is the ability of a computer system component to be partitioned on other computer system components in a flexible and efficient way. To be able to fulfill the requirements of an application to grow in the future, it is important to decide which parts need to scale. Most parts of a computer system are able to scale to a certain degree. Interconnects can support a variable number of devices, and processor boards and the operating system need to be scalable enough in order to support these extra devices. Two approaches concerning scalability have been stated. The first approach is *scalability over machine size* [Culler99 p. 206] which focuses on being able to scale the machine hardware and software while still fulfilling the original

requirements. The second angle of approach is *scalability over problem size*, which focuses on being able to scale the application and the problem, i.e. being able to run new applications and introduce new functionality.

Scalability as a term is used for describing a system component that can be enhanced in some respect. A computer can increase its available amount of memory or increase its number of available processors. Culler [Culler99 p. 456] identifies three domains of interest regarding scaling at the architecture level; bandwidth scaling, latency scaling and cost scaling. These three are the main metrics for determining the scalability of the hardware architecture, but the system user cannot ignore how well the software scales on that particular machine. If the algorithm cannot be partitioned on the underlying hardware architecture in an efficient way, there is a minimal incentive of introducing a scalable multiprocessor architecture.

Culler [Culler99] illustrates the fundamentals of hardware architecture scalability with an abstract view of how the essential components of a system, the processing unit (processor), the memory and interconnects (links and switches), are organized. In a parallel system that scales well, a large number of processing units should be able to interconnect with a sufficient large amount of memory modules. How successful the assembly and design of the system is, depends mainly on the three architecture scaling parameters mentioned earlier; bandwidth, latency and price. Building efficient and high-performance parallel systems leads to many common factors in design, regardless whether the design is for large distributed database systems, industrial automation systems or multiprocessors on chip. Factors such as locality in all levels of design (memory system, location of software modules based on their usage, as well as the physical distance between processors), minimizing the workload on each node by for example creating more efficient algorithms or optimizing code or logic, minimizing overhead in relationship to useful work, as well as partitioning tasks among multiple processing units are important in every multiprocessor system. Loosley and Douglas [Loosley98] discuss these issues in respect to database systems and distributed systems.

An important metric in this context is the computation-to-communication ratio [Culler99 p.132]. This ratio is relevant for scientific applications and industrial systems alike, but the term is mostly used as a metric for large scalable systems. The larger the computation-to-communication ratio is the more important the ability to achieve overlap of communication and computation becomes.

2.2 Parallel Hardware Architecture

This chapter discusses parallel hardware architecture in general, but directs the discussion to systems implemented for industry environments where possible. The discussion strives to present the state-of-the-art designs but also define terms and techniques that are fundamental for the discussion. The text intends to give a walk-through view of the architectural components of multiprocessor architectures, issues that have been relevant for understanding the architectural alternatives that were available in this project.

2.2.1 Single Processor Architecture

In order to describe multiprocessor architectures we will first briefly discuss the basic parts of a common single-processor machine. A computer does essentially consist of five components; input, output, memory, datapath and control [Patterson98 p.16]. Datapath and control are usually combined to what is commonly called the processor. The processor fetches instructions as well as reads and writes data to the first level of memory, the level 1 (L1) cache. If data or instructions are not available to the processor in the first level cache, the memory subsystem will have to fetch the instructions or data from the lower levels of the memory hierarchy. The final instance of random access memory is the primary memory (in this context we overlook facilities such as disks which are seldom used in real-time industrial systems).

Basically, two historical hardware architectures exist; the Von Neumann architecture and the Harvard architecture. The Von Neumann architecture [Heuring97 p. 24] architecture (processor) has a unified view of memory, which is in contrast to the Harvard architecture that has separated instruction and data memory areas. The processor fetches instructions from the memory hierarchy thereby controlling the datapath and manipulating data in memory. Input and output is commonly performed by peripheral devices, which are initialized and controlled by the processor. Modern peripheral devices can usually access memory directly (DMA) and notify the processor by interrupting it. A single processor system has, per definition, only one processor and the system does not need to incorporate facilities that enable multiple processors to work consistently. In a single processor system only one processor accesses memory at every instant even though multitasking operating systems may schedule multiple processes onto one processor in an interleaved fashion.

Processor performance has been increasing according to Moore's law over the past decades, which means that on average performance of a modern processor has doubled every 18 months. The development of memory-technologies has not been able to keep up to this enormous increase in speed. The solution to this problem has been to create a hierarchic memory system, in which smaller but faster memories are high up in the hierarchy (close to the processor) and slow but larger primary memory are at the bottom. Memory hierarchies have been the commonly used mechanism to bridge the gap between processor and memory performance. Memory is also a very important component in multiprocessor systems, and computer architectures can be categorized according to their memory system design, as discussed in section 2.2.3.

2.2.2 Processors from a Parallel Perspective

Modern processors increase performance by employing non-predictive methods or components. Branch prediction, superscalar pipelining, out of order execution and memory hierarchies cause the system to be difficult of predicting in terms of execution time of programs, yet these techniques are a necessary component in

achieving high performance. Even though such processors are less predictable, they are used in many real-time systems since they yield higher performance than simpler architectures.

These modern processors try to exploit parallelism among individual instructions within a sequential flow of instructions. The amount of parallelism available within a block of sequential code is commonly labeled *Instruction Level Parallelism* (ILP) as mentioned in section 2.1.6. Data dependencies limit the ability of the processor to execute and retire more than a few instructions concurrently each cycle. An example of how limiting to performance data dependencies can be, and how important it is to break the data dependency limit of processors, is illustrated by two techniques that exploit value prediction and computation reuse [Wu2001]. *Value prediction* is a speculative method that uses the values of recent executed regions of code in order to predict the results of future calculations of the same region of code. If the result turns out to be incorrect the predicted values must be discarded. *Computation reuse* is a non-speculative technique that caches inputs and outputs of previous computations to be used in later computations. Results from integrating the two techniques shows a speedup of up to 1.4 compared to using one of the techniques alone.

Thread Level Parallelism (TLP) is being exploited in MIMD architectures (see section 2.2.3 for multiprocessor categorization), as for example in symmetric multiprocessors (SMP). The development of processors has been dramatic from the dawn of the microprocessor in the early 1970's. During these years bit-level parallelism and instruction level parallelism, have been exploited extensively. Multithreading architectures and multithreaded processors such as the Stanford Hydra [Hammond2000] or the Tera MTA [Alverson90] have been created but have had limited commercial success.

A multithreading technique that has been studied recently is the so-called Simultaneous Multi Threading (SMT) processors [Tullsen96], where instructions from multiple threads compete for the available execution resources. Unlike other multithreaded architectures, SMT-processors let multiple thread contexts issue instructions simultaneously. A commercial product that employs SMT, i.e. exploits thread level parallelism on chip, is the so-called Hyper-Threading technique [Elektronik] [Intelwww] from Intel. Each logical processor is equipped with two L1 caches and two register sets but only one L2 cache. This allows for two threads to execute simultaneously, which improves latency hiding. With this technique the two threads compete for the same execution resources, such as the ALU. The main purpose of this technique is to "use thread-level parallelism and instruction-level parallelism interchangeably" [Lo97]. The Hyper-Threading technique has been released as an implementation in the Pentium 4 processor.

A way of handing over responsibility of guaranteeing independence between instructions to the compiler is to implement Very Long Instruction Word (*VLIW*) architectures [Patterson98 p. 528]. The compiler analyses the code to find data dependencies and thereafter creates VLIW instructions that enables independent execution at the greatest possible extent. Advantages include simplified instruction

decoding and reduced need for hardware resources. The disadvantage is the need for frequent recompilation of executables, due to the binary incompatibility of different generations of processors. The new 64-bit generation processors from Intel, the IA64 architecture, will incorporate VLIW instructions [Stallings2000]. This technique is named Explicitly Parallel Instruction Computing (EPIC).

Reduced Instruction Set Computers (RISC) [Patterson80] have increased processor performance during the 1990's and almost all processor architectures have introduced pipelined execution of instructions and many high-performance processors are superscalar, i.e. have more than one parallel instruction pipeline. The purpose of the RISC is to achieve effective execution by letting all instructions have the same size, allowing only load and store instructions access memory and execute one instruction per cycle (single pipelined). Example RISC architectures are the MIPS and the PowerPC. Complex Instruction Set Computers (CISC) do not share these properties but can have variable length instruction formats and usually incorporate more functionality per instruction. The common objection to implementing such architectures is that it is more difficult to exploit ILP. Traditional CISC architectures include DEC PDP-11 and the VAX11, and Intel's IA-32 architecture is usually depicted as a CISC as well (even though it is implemented as a RISC architecture internally).

In industrial and embedded systems, *Digital Signal Processors (DSP)* such as the floating point capable DSP SHARC from Analog Devices [Analogwww] provides floating-point and signal analysis capabilities for real-time control applications (especially for frequency analysis). The system we have studied does not incorporate DSPs at the numerical module level (processor boards) but in the sampling peripherals. Since the trend has been to move functionality from DSPs to general purpose processor boards, we have not considered the use of such components in our study.

Some modern processors utilize instructions that act upon multiple data locations at the same time such as the MMX instruction set and the streaming SIMD extensions (SSE) [Intelwww2] for Intel's Pentium processors. These architectures will be discussed further in section 2.2.3.1.

2.2.2.1 Interrupts and External Events

Interrupts is the means by which devices notify the processor of events, as for example when it has completed a memory transfer. Interrupts are in effect the technique for peripherals and the processor to synchronize their actions. In the system we have studied interrupts are an important issue due to the many peripherals that are present.

Modern implementations of interrupt handling have evolved into including handling traps, i.e. internal or program interrupts. Manufacturers use terms such as exceptions, traps, and faults differently but all describe a reaction to an event, collectively known as interrupts. Interrupts occur either synchronously or asynchronously. *Synchronous interrupts* occur due to events that originate from the instruction flow, such as floating-point imprecise exceptions or division by zero

exceptions. *Asynchronous interrupts* on the other hand occur due to external events such as system reset or external interrupts from peripheral devices. Asynchronous interrupts are important from the parallel point of view as well, in that they are the notification mechanism not only for peripheral devices and processors, but between processors and processors as well (as in SMP architectures).

Walker [Walker95] has classified interrupt implementation choices. When an interrupt occurs the state information of the running process/thread has to be saved. This saved information is used to restore the processor to the same state as it was in before the interrupt occurred. This includes at least saving the program counter but usually it also incorporates saving vital registers and switching to another stack. Interrupts can be either *precise* [Smith85] or *imprecise*. Imprecise interrupts allow for instructions executing out-of-order to complete without serial correctness, meaning that the user is responsible for guaranteeing serial correctness. The Alpha processor enables the user to issue special instructions, i.e. memory barriers, to prohibit any instructions from issuing until a following arithmetic instruction has completed. Precise interrupts on the other hand guarantee serial correctness, and the user (usually the compiler) does not need to use memory barrier instructions. Processors utilize precise interrupts when a page fault interrupt occurs. The processor saves the processor state at the time of an interrupt, which enables the system to have another process run while the correct page is collected from the hard drive or another media.

An important issue in real-time and industrial systems is interrupt service time latencies. Modern operating systems such as Linux [Bovet2003] and Windows [Solomon98] have a two layered handling of interrupts, where a minimal interrupt handler acknowledges the interrupt and then schedules a routine to be executed later (Deferred Procedure Call in Windows, and so called tasklets in Linux). Solaris also minimizes long-latency interrupt response by allowing interrupts to be scheduled as threads [Mauro2001]. Real-time operating systems such as VxWorks let all Interrupt Service Routines (ISR) share the same stack and are not allowed to block. This means that ISR are not allowed to take synchronization objects such as semaphores.

2.2.2.2 The Alternative to Interrupts: Polling

The alternative to interrupt handling is polling, in which software or possibly another hardware device periodically examines whether a certain event has occurred. Polling may be useful in systems with fast network interfaces, such as Gigabit Ethernet. Receiving an interrupt for each frame received would not be feasible for a processor to handle, and many fast network interfaces do not even provide the possibility to issue interrupts. In this case polling is an alternative as described by Perkovic [Perkovic99]. The paper discusses both how multithreading and automatic insertion of polls can increase average performance and interrupt latency can be minimized using a watchdog at millisecond intervals. Polling as term can be used in other research areas. Takagi [Takagi88] describe polling models where a single server accesses multiple queues in cyclic order and no asynchronous notification mechanism exists.

In the case-study system an example of a component utilizing polling techniques is the watchdog. It periodically polls devices and their status in order to detect system faults.

2.2.3 Categorization of Parallel Architectures

It is useful to categorize multiprocessor hardware architectures into groups where each category represents hardware architectures with certain common properties. Such attempts exist, where a famous categorization based on the instruction and data streams (control flow) was created by Flynn [Flynn96]. According to the taxonomy created by Flynn four combinations of single or multiple instruction and data streams exist:

- The Single Instruction stream, Single Data stream *(SISD)* computer represents a traditional uniprocessor.
- Single Instruction stream, Multiple Data Stream *(SIMD)* computers work with data parallel constructs on vectors of data. In short, this is an architecture where "a single instruction specifies operations on several data items" [Jordan2003 p.7].
- Multiple Instruction streams, Single Data stream *(MISD)* computers constitute one of the four variations of computers described by Flynn. Pipelined processors that cooperate in performing a part of the computation on the same data stream [Narayanan93] constitute an example MISD architecture.
- Multiple Instruction streams, Multiple Data streams *(MIMD)* computers are the most commonly used architectures today. SMP, CC-NUMA and clusters all belong to this category of computers, and these architectures are described in section 2.2.3.2 below.

2.2.3.1 SIMD Architecture

SIMD computers consist of Processing Elements (PE) that synchronously execute a single instruction from a single program counter [Patterson98 p.749]. Every PE executes synchronously and is controlled by a central control unit (usually a general-purpose processor). Many modern general-purpose processors implement SIMD instruction sets. Example instruction sets include the MultiMedia eXtensions (MMX) from Intel and the AltiVec [AltiVecwww] for the PowerPC architecture from Motorola and IBM. These instruction sets are designed to increase performance for operations that are common in audio and video applications. Algorithms using these instruction sets can issue one instruction and have the processor acting on a vector of data in parallel. The data is contained in special data registers, in the case of MMX they are represented by 64-bit internal registers. The evolution of MMX and AltiVec instructions illustrates the need for specialized instructions that internally utilize parallelism.

Traditionally large SIMD architectures that focus on data parallelism are labeled Massively Parallel Computers. Two such architectures are the MasPar MP-1

[Nickolls90] from MasPar Computer Corporation and The Connection Machine 2 [Tucker88] by Thinking Machines Corporation. SIMD architectures exist for industrial applications as well. The special-purpose Linear Picture Processor (LAPP) [Lawson92 p.431] is a SIMD architecture capable of scanning silicon strings used in the manufacturing process of motor blocks. The massively parallel architecture is capable of performing a visual examination of the silicon strings in real-time.

2.2.3.2 MIMD Architecture

The MIMD architecture is by far the most used computer systems architecture and a plethora of possible MIMD architectures are possible. Hwang [Hwang98] has presented a categorization of MIMD architectures that decomposes into the subcategories shown in Figure 7 below. The first division is between the Uniform Memory Access (UMA, Central Memory) architecture, the Non-Uniform Memory Access (NUMA, Distributed Memory) architecture and the No-Remote Memory Access (NORMA) architecture. Hwang divides the MIMD architecture into sub-categories that describe the access to memory in the parallel system.

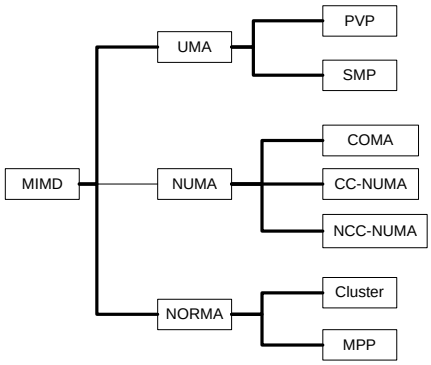


Figure 7, a categorization of multiprocessor architectures [Hwang98 p. 238].

2.2.3.3 Shared-Memory Multiprocessor Systems

Two of the sub-classes in Hwang's categorization allow the processors to share memory (UMA and NUMA). A categorization that divides shared memory architectures according to different memory layouts has been defined by Culler [Culler99 p. 271]:

- The *shared cache architecture* is symmetric, meaning that all processors have equal length to access memory. Since all processors share all cache memory, no cache-coherence problems arise. This approach has only been shown to scale up to eight processors, but may become interesting again for system-on-chip (SoC) architectures. Interconnects are obviously the limiting factor since all processors share the same interconnect to the caches.

- The *bus-based shared memory architecture* is popular in for example server systems due to its good price/performance ratio. Efficient use of shared resources, such as I/O buses and printed circuit boards, contributes to that. The architecture has shown to be scalable up to over 30 processors. The most common bus-based shared memory architecture is the symmetric multiprocessor (SMP). SMP architectures are being used to a greater extent in industrial systems today, and companies such as Concurrent Computer Corporation [Concurrentwww] are designing complete systems for use in real-time industrial applications.
- The *dancehall architecture* exchanges the bus in the bus-based shared memory architecture with a scalable point-to-point interconnect network. All memory modules are still uniformly far away from the processors; therefore the architecture can be described as a symmetric architecture. This memory architecture is not widely used today.
- The *distributed-memory architecture* (DSM) lets all nodes in the system have a share of global memory resulting in that memory accesses are not uniform. This architecture promotes scalability and can be efficient if data locality can be exploited in the local node.

2.2.3.4 Cluster Architectures

Clusters or NOW (Network of Workstations) provide a way of using personal computers and/or workstations to achieve a high price/performance ratio. Personal computers improve their price/performance ratio by approximately 80% per year, while supercomputers only increase the same ratio at a rate of 20-30% per year [Anderson95]. NOW can provide a cheap and highly available (it is possible to compensate for nodes going down) network of computers, using multiple CPUs for parallel computing. The system does usually scale well and every node executes its own operating system (suitable for heterogeneous environments).

Berkeley NOW [Anderson95] and Beowulf [Beowulfwww] are two well known cluster systems that have been developed. Berkeley NOW uses switch-based networks (ATM or the Myrinet) and the original NOW system had a hardware configuration of 105 SUN Ultra 170 workstations. *Beowulf* clusters have become popular mainly due to the fact that they can be implemented using free operating systems as Linux and FreeBSD. Beowulf clusters reside on a dedicated network and nodes can communicate with processes on other nodes through the use of global PID (Process Identification). There is no software package that is called "the Beowulf cluster system" [Beowulfwww2]. Rather, a Beowulf cluster consists of different software packages such as PVM, and a special Linux kernel (supporting global PIDs etc.).

Cluster architecture is being used increasingly in industry, scientific computing and biochemistry. In industry the main use is for simulation purposes, but real-time properties of clusters are being investigated as well [Suzuki2003]. We have seen no obvious use for cluster architecture in the case-study system.

2.2.4 Examples of Multiprocessor and Distributed Real-Time Systems

This section describes three systems that have contributed to the understanding of distributed multiprocessor real-time systems. We have focused on describing aspects of those systems that are relevant for our project, such as for example network nodes and the utilization of functional parallelism.

2.2.4.1 HARTS

HARTS [Shin91], Hexagonal Architecture for Real-Time Systems, is a distributed real-time architecture developed at University of Michigan in the early nineties. The project focused on supporting time-constrained and fault-tolerant communications. The architecture consists of clusters of nodes interconnected by a hexagonal mesh. A HARTS node consists of application processors (AP), a network processor (NP) and an Ethernet Processor interconnected by a VME-bus [Kandlur91] (Figure 8). Every node is synchronized with the help of specialized hardware which provides a global time base which is used by the HARTOS operating system to allocate shared resources [Ghosh94 p.38].

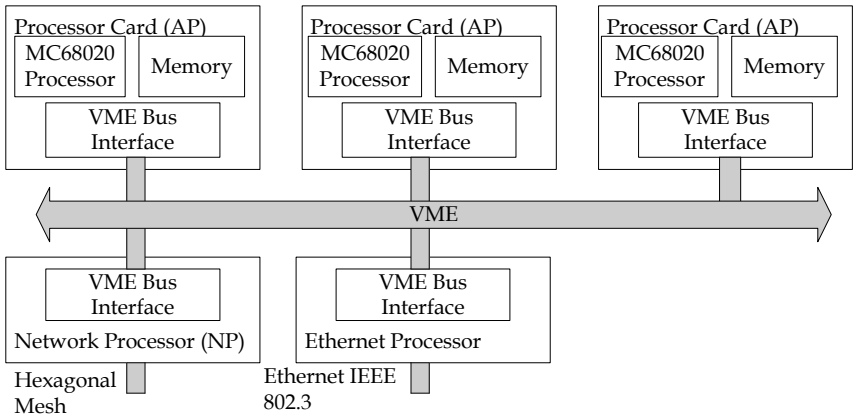


Figure 8, a HARTS Node [Kandlur91].

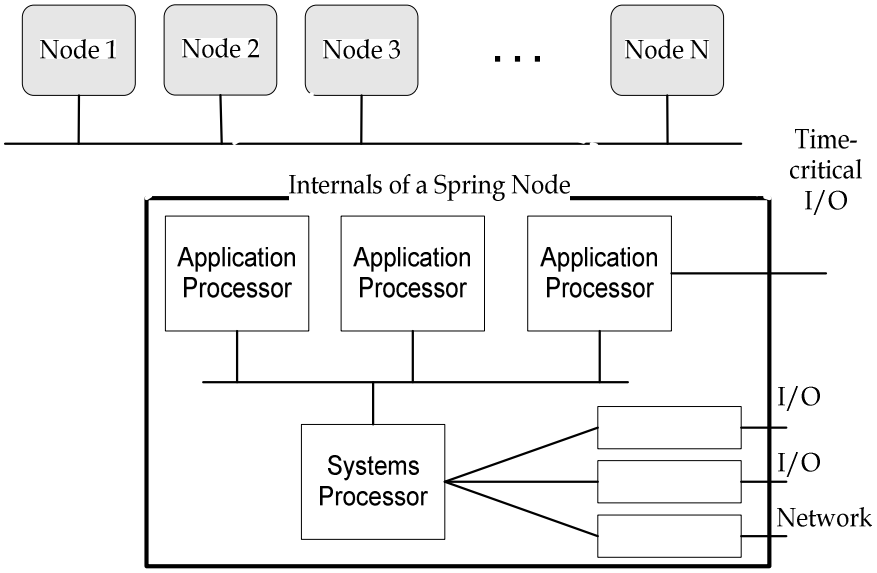
The purpose of incorporating the network processor was to offload the application processors and implement communication functionality from the physical layer up to the transport layer. Communication between the application processors and the network processor is implemented by using a DMA interface on the VME-bus. Achieving low communication latency was one of the goals when designing the network processor and it is also capable of monitoring the load on the network as well marking messages with time-stamps.

The impact of the HARTS project was not as profound as compared with the Spring system, but network and I/O Nodes in real-time systems have parts in common with the approach in our study. The most important similarity is the functional separation of communication and application components.

2.2.4.2 Spring

The Spring system and kernel was an academic project developed by, among others, John A. Stankovic and Krithi Ramamritham [Stankovic91] [Stankovic92] [Stankovic99] [Molesky90]. The motivation for the project was to provide basic support required for large and complex real-time systems, and special focus was on fulfilling timing constraints. Current real-time operating systems, in their view, use the wrong paradigm for enabling predictable execution. These real-time systems are in fact only stripped-down and optimized versions of time-sharing operating systems, stressing fast context switching as well as ability to respond to external interrupt quickly. The main critique brought up against these current real-time systems include the lack of explicit consideration of timing constraints, the difficulty to predict task executions and the lack of explicit handling of tasks with complex characteristics.

With this critique in mind the creators of the Spring kernel tried to create a system that uses the information known beforehand about the tasks in the application. This information is then used during runtime and the result is a system where flexibility and predictability is ensured. The system uses a value system on tasks that values the tasks ability to meet their timing requirements.



include tasks that must meet their requirements (a task deadline miss will result in a negative infinite value to the system). Essential tasks include those that do not cause catastrophic events in the system if their requirements are not met but will degrade performance in the case that the requirements are not met. Unessential tasks on the other hand are usually background tasks that perform maintenance functions and long-range planning. Springnet is the distributed system that utilizes the spring kernel and it consists of spring nodes as can be seen in Figure 9 below. A spring node is a multiprocessor that consists of application processors, system processors and an I/O subsystem.

Three important components of a real-time kernel are highlighted in the Spring kernel, the task management, the scheduling policies and the memory management. A fourth important component, the inter-task communication is omitted in the discussion in [Stankovic91].

The Spring kernel presupposes that the application designer has a priori knowledge about the characteristics of tasks. This includes knowledge about WCET (Worst-Case Execution Time), deadlines, periods, the maximum number and type of resources needed, its type (critical, essential and unessential tasks), and its importance level. Knowledge about precedence graphs and communication graphs is also needed. All this information is maintained by the task descriptor.

The architecture consists of application nodes, system processors and an I/O subsystem and an advantage of the architecture is [Burleson99]:

> *System processors offload the scheduling algorithm and other OS overhead from the application tasks both for speed and so that external interrupts and OS overhead does not cause uncertainty in the execution of guaranteed tasks.*

The Spring Kernel has been used in different environments, such as for example manufacturing industry [Stankovic94] and a co-processor [Burleson99] has been developed to accelerate scheduling in the Spring kernel. The paper [Burleson99] claims a threefold speedup compared to software scheduling. The coprocessor is designed as a memory module and processors communicates with the accelerator by means of memory accesses.

2.2.4.3 UltraSmart

UltraSMART [Boxer94] is a parallel real-time architecture that was developed by a company called Concurrent Computer Corporation [Concurrentwww]. The UltraSMART architecture was at that time (1994) developed for the widely used industry standard VME bus but today the company supplies a more modern symmetric multiprocessor architecture for industry and military under the name of PowerMAXION. UltraSMART provides a directory based cache-coherent mechanism that lets the code executing on the processing modules (XPU) access a coherent memory space. An ASIC, the so called Crosspoint Processing Module (XPU), handles

memory accesses to local and remote memory. The UltraSMART architecture is accompanied by a UNIX operating system with a kernel that is preemptive and that is specially developed to handle disk system I/O. Features, aside from common features such as an optimized interrupt system and priority real-time scheduling policies, is its capability to dedicate a processor to specific tasks. The company's motivation for this mechanism is that "reserving processors guarantees optimum response for the dedicated tasks". As with the systems described above, this allows for functional partitioning. It would be possible to statically pin communication tasks to a dedicated processor, thereby offloading the other functionality (applications).

2.2.5 Interconnects

Together with processor performance and memory access latencies, interconnects have historically been a major bottleneck in computer systems. Insufficient bandwidth and high latencies of interconnects have rendered distributed and parallel computing difficult. The maximum possible length of interconnects has affected the physical topology of systems. Interconnects enable communication between nodes and components, and a broad range of commercial products are available, spanning from on-chip buses to local area networks and wireless data links. Parameters that affect which interconnect to use include; physical extent, fault tolerance requirements, how easily links are affected by electro-magnetic interference (EMI) and price. The goal of this section is to introduce interconnects that are used in industry environments and will include some new and emerging interconnects (RapidIO and PCI-Express).

2.2.5.1 Bus Functionality

A bus provides paths between various parts of a computer system and implies the possibility of communication between more than two devices [Gustavson84]. Devices on a bus request access grant to the bus by communicating with the arbitrator. The arbitrator is usually a centralized component that resolves the contention between transaction initiators [Dawson99]. The time that it takes for a bus master, i.e. a device that initiates a transfer of information, to gain bus mastership, is labeled access latency. Each current bus-master, only one at a time, is labeled the commander. The time that the bus is held by a commander is called bus tenure. Multiple masters on the bus may contend for initiating a transfer, which leads to contention for the bus. Masters and slaves exchange information by using different address ranges. Masters can therefore select the slaves which it wants to communicate with. When connected, the master and the slave can exchange information. The devices that participate in the transfers are called the listener and the talker. After data has been exchanged, the commander can chose to break the connection, i.e. disconnect. This sequence of actions – setting up a connection, transferring data and breaking the connection - is defined as a transaction. The set of rules that governs the use of the bus by masters and slaves is called the bus protocol. A bus can be either parallel or serial. A completely parallel bus dedicates each bit of the bus protocol to a dedicated signal

line. In a serial bus all information is multiplexed over a dedicated signal line. Multiplexing is a time sharing technique that reduces the number of bus-lines by enabling both address and data phases on the same set of signal lines.

DMA (*Direct Memory Access*) provides a way for peripheral devices to directly access memory. The main advantage of using DMA is that the processor does not spend cycles [Heuring97] by moving data from peripheral devices to primary memory. An example of an integrated circuit that provides DMA functionality (channels) in the PC architecture is the 8237A [Intel8237A]. It is a remnant from the PC-AT architecture and is used mostly to transfer data from slow devices such as floppy disk drives [Bovet2003 p. 436] to main memory. The setup time for the DMA channels is usually high which makes it more efficient to let the CPU perform small data transactions.

The next sections will describe I/O buses and relevant network based interconnects. We have decided to separate interconnects into two categories; I/O buses and local area interconnects. The categorization divides according to the physical extent of the interconnects, where I/O Buses have the least physical extent and local area network interconnects have the largest physical extent. More categories of interconnects, such as processor buses and field buses (for example ProfiBus [Profibuswww]), could have been added to the discussion, but we have limited our overview to interconnects that are most relevant for the case-study system.

2.2.5.2 I/O Buses

This section describes some well known I/O buses used in industrial systems today and that are used for connecting both peripherals and processor boards.

The *PCI-bus* is together with the VME-bus the most common bus for industrial automation applications. A large amount of devices and appliances have been constructed for the PCI-bus, and a wide range of tools are available to the users of the bus. The first version of the PCI-bus appeared 1992 (v1.0) while version 2 was released in 1995 [Shanley99]. The latest version is v2.2, which dates to back to 1999. The bus is processor independent and supports up to ten electrical loads. Logically the PCI-bus consists of buses (up to 256 buses), devices and functions within devices. A function is a form of logical device in a physical device and each function contains its own individually addressable configuration space. The bus hierarchy can be scanned by software in order to find available devices and functions. Each device has a unique vendor and device identifier and each PCI-device is assigned those unique identifiers when manufactured [PCIwww2]. The purpose is to have a configurable bus hierarchy where peripherals may physically occupy different slots. Physically the PCI-bus is a reflected-wave switched bus [Shanley99], meaning that a wavefront reflection of the signal issued needs to be propagated back across the bus to be able to drive the signal lines to the desired logic state. The PCI specification states that the devices must only sample their inputs on the rising edge of the PCI-clock signal.

Important performance related techniques of the PCI-bus includes the possibility of performing burst read- and write- transfers, i.e. a transfer consisting of one

address phase and multiple data phases. Burst functionality eliminates the need of constantly regaining mastership of the bus. Theoretically, optimal use of bursts gives rise to a maximum transfer rate of 132MB/s for the 33MHz 32-bit specification (v1.0). In practice, the sustained bandwidth may be much lower.

The PCI specification allows for arbitration requests to be issued while another bus-master is performing a transfer. Overlapping arbitration and data transfers are thereby enabled, thus promoting performance. The arbitration scheme is required to be fair by the PCI-bus specification, meaning that all devices must be allowed to eventually perform a transaction on the bus. The specification defines the arbitration implementation very loosely, but most implementations are reported to adhere to round-robin arbitration [Wolf2001 p.455].

PCI-to-PCI bridges glue together multiple PCI-buses in a hierarchy, and the bridge forwards data from one side to the other (upstream and downstream). A PCI-to-PCI bridge acts as a target on one side and master on the other when forwarding data. Latencies through PCI-to-PCI bridges have been reported to be in the order of tens of nanoseconds up to milliseconds [Chamé98].

PCI-Express [PCIwww] is a serial backward compatible extension of the PCI Bus. Backward compatibility is assured on the software level in that devices are enumerated as on a conventional PCI-bus and that devices are accessed in the same manner. Therefore device drivers need not to be altered when migrating to PCI-Express. Instead of defining a parallel bus, PCI-Express provides a serial point-to-point I/O interconnect that can scale better than the conventional PCI-bus. The number of physical pins as well as the frequency used is configurable, which promotes bandwidth scalability. The serial interconnect does not have as stringent synchronization restrictions as the conventional PCI-bus. Data throughput performance has a maximum of 250MB/s per lane (PCI-SIG terminology for a link) resulting in a total available bandwidth of 16 GB/s for a 32 lane configuration.

RapidIO [RapidIOwww] is a packet-switched, point-to-point interconnect that is designed for a memory mapped programming model. Peer-to-peer communication is possible and the interconnect provides for a globally shared distributed memory [EETimeswww]. Furthermore, the hierarchy is not strictly fixed, and up to 64,000 devices can be addressed on the bus. Each device can be targeted through memory accesses depending on an offset. The I/O system is packet oriented; each node is either rejecting or forwarding packets from one side to the other, assuming the packet is not bound for the node itself. Up to 256 outstanding transactions between each sender and receiver pair can be active.

While not designed for process control or for real-time industrial use, we mention the *Universal Serial Bus* (USB) [Anderson97] here due to its increased popularity. The maximum bandwidth of the first USB version is 12 Mb/s, while USB v2.0 enables a data rate of up to 480 Mb/s. USB devices connect to the host system via a USB host controller and the host controller driver software defines the transactions that are to be scheduled during the next timeframe. This timeframe is usually 1 ms in which all pending transactions are broadcasted on the serial bus. While many devices can be

attached to the USB-bus, each device does not occupy system resources such as IO or memory address space, IRQ lines or DMA channels, which is beneficial compared to for example conventional serial devices (UART).

2.2.5.3 Local Area Network Interconnects

Networks are usually packet-based, where a *packet* is the segment of information that is transported over the channels and that is unpacked at the receiver [Culler99]. A *switch* is a device that acts as a "man in the middle" on the physical link. It can interpret the information contained in the packets and forward them to the correct destination. The wires or fibers that carry the analog signal constitute the *link* to which the *transmitter*, a physical device that converts digital information from the system to analog signals on the link, is connected. The *receiver*, the physical device that converts analog signals on the link to digital signals is also connected to the link. The *physical protocol* converts the stream of digital symbols into an analog signal. The amount of data transmitted across a link in one cycle is called a physical unit, or a *phit*. Now, a *channel* for digital information consists of the transmitter, the link and the receiver. The channel has a *width* w, which denotes the number of signal lines, and a signaling rate of $f=1/\tau$, where τ denotes the cycle time. Therefore the channel bandwidth can be defined as $b=wf$ [Culler99]. The *link-level protocol* segments the stream of symbols crossing a channel into larger logical units, such as the packets or messages.

Ethernet (IEEE 802.3) [Molle96] [Tanenbaum96] is the most commonly used network technology used today, and it is even used in real-time systems. The IEEE 802.3 protocol is a Carrier Sense Multiple Access / Collision Detect (CSMA/CD) protocol where transmissions are scheduled by each node independently. The nodes transmit as soon as the link is quiet and if a collision occurs, each node backs off a random amount of time until it tries again. The most prominent performance effect is that a highly loaded network will show poor channel utilization. Furthermore, delay guarantees cannot mathematically be proven on the Ethernet datalink layer [Banerjea96 p.5], which is of high importance in real-time systems.

The *100VG-AnyLAN* [HP95] [Molle96] [100VGwww] network is a shared-medium LAN protocol that has been ratified by IEEE as the 802.12 standard. It is a protocol that was released in 1995 and originates from the IEEE 802 project. The minimum data rate is 100Mbits per second and the standard is capable of transmitting both IEEE 802.3 (Ethernet) and IEEE 802.5 (Token Ring) frames on the link, and it is available both for twisted-pair cable and fiber-optic cable. It differs from the IEEE 802.3 standard in that it does not define a CSMA/CD medium access control (MAC) algorithm. Instead it defines a so called Demand Priority MAC protocol that uses a switch repeater as a node in the network topology. The repeater continuously polls the connected nodes according to a round-robin policy to determine which node is allowed to send. Two priorities are allowed, normal and high, and high priority frame requests will immediately have priority over normal frames. The protocol was created so that the LAN would provide deterministic access and priorities [Martini95]. Therefore the protocol is able to provide guaranteed-performance

services [Banerjea96 p.5]. The technology was used by industry after the introduction in 1995, and components for 100VG AnyLAN were sold a couple of years after the protocol introduction. Despite the deterministic advantages over Ethernet, the market for 100VG-AnyLAN has vanished, and virtually no manufacturers produce 100VG-AnyLAN related devices today.

2.3 Multiprocessor Operating Systems

The purpose of an operating system is to control the computer resources and provide the base upon which application programs can be written [Tanenbaum92 p. 1]. An operating system has essentially two interfaces, one towards the hardware and one towards the application. This chapter focuses on operating systems that are able to operate in multiprocessor environments. Even though main focus is on embedded and real-time operating systems we also describe some operating systems that are not usually described as "real-time". There is a difference between how academia and industry defines a real-time system. Our discussion is more inclined to follow the industry "definition". Industry tends to define a real-time system as a stripped-down and optimized version of a general time-sharing (often pre-emptive) operating system.

2.3.1 Fundamentals of Operating Systems

Before discussing the peculiarities of each selected operating system some fundamental issues of operating systems are discussed. The first concept is that of processes and threads. *Processes* and *threads* consist of a code segment (that may be shared among other threads and/or processes) as well as memory regions (a stack, static data and memory allocated from the heap) and a process/thread descriptor in the kernel. The process descriptor is commonly denoted Process Control Block (PCB) and the thread descriptor is commonly denoted Thread Control Block (TCB). The layout of the contents of a thread or process is illustrated in Figure 10.

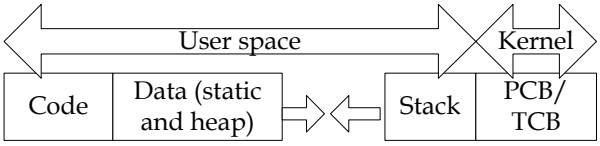


Figure 10, Address space layout of a process or thread [Hwang98 p. 65].

A thread is more "lightweight", sharing memory together with other threads of the same process. Threads are the contexts of a process, acting upon the shared resources held by the process. An opened file can for example be written to by multiple threads in the same process. Threads must therefore use operating system mutual exclusion mechanisms to be able to correctly access these shared resources. Due to the more lightweight characteristics of the threads, they are usually not as time consuming to context switch as processes.

Most modern operating systems introduce at least two processor execution modes, *privileged* and *non-privileged* modes. These modes are sometimes called *kernel mode* and *user mode*. The use of these different modes depends on the processor's ability to automatically switch mode whenever an event such as a system calls, traps and interrupts occur. Code running in user mode can thereby be prevented from manipulating data structures that are internal to the kernel, such as the page-tables.

The scheduler is the operating system entity responsible for controlling context switching of processes and threads. Employing correct scheduling algorithms is important in real-time systems. Some systems have stringent requirements on timing, as for example operating systems used in airplanes and cars. These systems are used in products where human lives are in danger, and are labeled *safety-critical systems* [Storey96]. In order to verify these systems, all parts of the system have to be predictable, from the hardware architecture to the software application. This does also include the scheduling policies, and many different scheduling algorithms and methods have been proposed over the last decades.

The seriousness of improper execution of the system leads to a distinction of real-time systems. *Hard real-time* systems are located in safety critical environments where a need of temporal verification is of importance. Timing constraints are set on threads in order to guarantee temporal correctness. A system designer associates a number of parameters to a thread or process that must be fulfilled during run-time. The thread may not start executing before a specified start time, called the *release time*. After the start the thread or process must complete within the specified *deadline interval*. To be able to predict the deadline it may be necessary to try to determine the *worst-case execution time* (WCET, or sometimes called Maximum Execution Time, MAXT [Puschner89]) of the code that represents the threads. In a periodic system it is also meaningful to talk about the *period* of a thread, the known interval between release-time. Events occur in the system, and their arrival times may be either *aperiodic* (events occur irregularly), *sporadic* (the maximum frequency of the arrivals of the events is known) or they may be *periodic* (The events occur at known regular intervals). The designer has the possibility of either allowing scheduling to occur before run-time (pre run-time / static scheduling) or dynamically at run-time. Dynamic run-time scheduling algorithms include *rate-monotonic* [Liu73], *Earliest Deadline First* (EDF) or the *Priority Inheritance Protocol* [Sha90]. *Soft real-time* systems have less safety critical requirements. A definition of a soft real-time system is where the "consequences of failures are of similar magnitude as the system benefit".

Every thread in the system is at any instant in a special state. These states decide when threads are to be scheduled. Only one thread per processor can at any instant be in a *ready* state. Whenever a thread is not able to gain access to a resource the thread enters a blocked state. As the resource is freed, the thread leaves the *blocked* state and is linked to the *ready* queue. When the priority of the thread is high enough it is made running again. Another thread may sometimes suspend another thread by putting into a *suspended* state.

Schedulers may use different methods to schedule threads in the system, basically using one of three methods [Hwang98 p. 67]. The first is the use of *batch-queues* where jobs are executed one after another until they are finished. The second is a method where threads execute in short interleaved time-slices (quantum). These schedulers are labeled *time-sharing*. The third a *pre-emptive event-driven* scheduler where threads can be interrupted by a higher prioritized thread or interrupt.

In multiprocessor and distributed systems, a non-uniform distribution of work on individual processor can yield less than optimal system performance. The general technique of distributing work evenly on processors is called *load balancing* [Kakulavarapu99] [Culler99 p. 123-131] [Wilkinson99 p. 198-207]. In large multiprocessor systems, fine-grained tasks are easier to load balance due to the large amount of processes that can be spread out on multiple processors [Culler99 p.129] (a higher degree of concurrency is utilized). Counteracting parameters to that are the implied increase in task management overhead and the increase in contention. In distributed systems, *migrating processes* [Elson98] from one node to another can help in achieving efficient load balancing. It is expensive and difficult to move the state of a process to another node. Pending requests to services, open files and bindings to other services must be modified to reply and point to the new node. In the case-study system, dynamic process or thread movement has not been of immediate interest.

Inter-Process Communication (IPC) defines data exchange and synchronization events among processes in different execution environments. It is therefore, per definition, erroneous to use the term IPC to describe data exchange between threads in the same execution environment. IPC mechanisms used in modern operating system include sockets, POSIX signals, semaphores, message queues, monitors and shared memory between processes. In the scope of this work we have implemented an inter-processor FIFO message queue for a bus-based multiprocessor system.

2.3.2 A Small Survey of Multiprocessor Operating Systems

This section describes operating systems that were interesting for the work performed within the scope of this project. One of the first questions encountered at the start of the project was of which importance the operating system had for enhanced system scalability. Therefore the functionality of selected well known operating systems was studied. Focus has been on the operating system's suitability in multiprocessor systems, SMP-support, scheduling, as well as fundamental kernel characteristics.

2.3.2.1 Operating Systems for Bus-Based Cache-Coherent Multiprocessors

Symmetric Multiprocessing is a hardware architecture where all processors have symmetric access to system components. Operating systems for SMP architecture reside in a memory area shared by all processors and where threads are scheduled on all available processors. True SMP is often referred to as "shared everything", since threads can be scheduled on any processor at any time. In a SMP multiprocessor system every processor can potentially access every region in the

entire memory space. This means that two or more processors can execute kernel code at the same instance, and may thereby act on shared resources in the kernel simultaneously. Reading data from a resource may be allowed, but writing to a shared resource must always be protected from parallel intervention. In an SMP operating system every shared data structure must be identified and protected. Kernel synchronization becomes an issue since both other processors and external interrupts can intervene when a processor is executing a critical section. Single processor operating systems usually protects the resources by disabling interrupts. Symmetric multiprocessor operating system kernels rely on spin locks [Solomon98 p. 125] to protect the same resources. Kernel components that need to protect critical resources include the scheduler, the memory management facilities and device drivers.

SMP operating systems performance is dependent on how fine-granular the critical sections are. Spin locks deny all other processors access to the shared resource and minimizing the time a processor execute while holding a spin lock is important. Generally, operating system developers try to make the kernel critical sections as fine-granular as possible. The developers of QNX [QNXwww] argue that there are less shared resources that have to be protected in a micro kernel compared to a monolithic kernel such as Linux. QNX is a micro-kernel with very little kernel code compared to for example Linux. The Linux kernel has on the other hand been equipped with more fine-granular kernel locks with each release. First versions used a single lock on the whole kernel, while the in the latest releases employ fine-granular kernel locks.

2.3.2.2 Amoeba

The Amoeba project [Mullender90] was started at Vrije University in Amsterdam, as early as 1981. We include this operating system in this survey since it is a historical example of a microkernel. The main goal was to create a transparent distributed operating system where a user is not aware of exactly where the program he or she started is executing. It may execute on any node in the system depending on where the load is low at the moment.

Amoeba is not limited to certain hardware architecture but can run on heterogeneous systems. The memory system architecture supports both shared memory programming and message-passing programming. Figure 11 illustrates the architecture consisting of simple workstations (running for example a shell or X-Windows), a processor pool (abstracted into one box, but they can be geographically widespread) as well as dedicated servers.

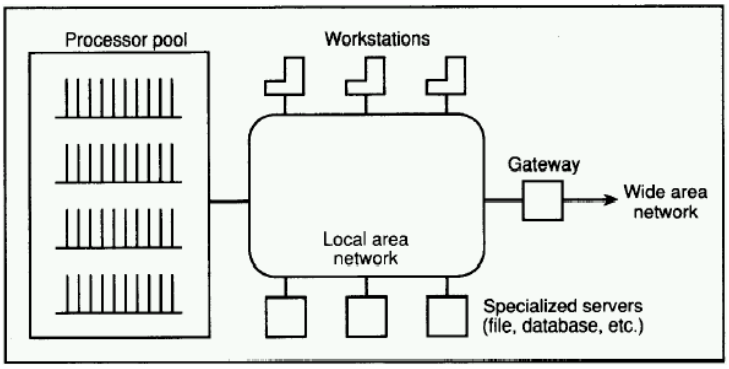


Figure 11, The Amoeba architecture

The kernel is a microkernel, i.e. a small kernel where only the basic functionality of processes and threads, memory management, IPC and low level I/O is provided. All other functionality is handled in the servers that execute as processes. The servers are abstracted into so called objects, which is an encapsulated piece of data upon which operations may be performed. The objects return a so-called capability, a long binary number, to the client that is used in the future correspondence. All communication is performed with the help of point-to-point RPC (Remote Procedure Calls) or group communication over RPC. The RPC layer uses a protocol called FLIP (Fast Local Internet Protocol) that provides network layer functionality.

The pool of servers is available to the users (workstations). The user can utilize a file system based on a file storage server (bullet server) a directory server (handles naming and directory management) and a replication server. A boot server manages fault tolerance aspects, such as continuous polling of servers in order to determine if they are still alive. Other servers include a TCP/IP stack, a disk server, an I/O server, a time-of-day server, a random number server as well as a mail server. The Amoeba kernel is not an SMP operating system in that sense that it runs in the same address space on a SMP system. There is no discussion about real-time properties in [Mullender90] or [Tanenbaum92 p. 588-636] and the operating system is categorized as a distributed time-sharing microkernel.

2.3.2.3 QNX

QNX [QNXwww] is a microkernel (Neutrino) based real-time operating system implementing minimal functionality in the microkernel (synchronization, IPC, signals, mutual exclusion and POSIX message queues). Every other service execute as optional processes. Neutrino supports POSIX processes, threads as well as virtual memory. A Neutrino thread can be in 14 different states, depending on what action or event it is currently waiting on. Neutrino supports FIFO and round robin scheduling, as well as priority inheritance based scheduling (64 priorities). QNX can execute on SMP architectures, which makes it unique among the large real-time operating systems (for example VxWorks). SMP scheduling support was simple to

implement according to the company since Neutrino is a microkernel. Only a small amount of code had to be adapted in order to protect critical sections. Today, SMP support is only provided for the Intel platform and up to eight processors are supported. It is possible for the user to control processor affinity of individual threads, i.e. it is possible to tie (or pin) a thread to a specific processor (by manipulating a bit-field). By default scheduling of threads is allowed on all available processors. Only one thread can access the kernel at any instant, but the developers of QNX argue that time spent in the kernel is only a small fraction of the other work performed. No published material has been found that confirms that assertion.

2.3.2.4 RTU

The RTU [Lindh98] is a hardware accelerator for operating systems. The concept is applicable to many operating systems but research has mainly been performed in a real-time architecture. The main goals have been to increase performance and predictability in the system with the help of the accelerator. The benefits are achieved mainly due to the use of parallelism in the hardware. The parallel nature of the RTU can increase scheduling performance substantially. The RTU has not been tested on SMP hardware, but has successfully scheduled tasks in a CompactPCI backplane multiprocessor architecture. In section 3.2.2.2 we discuss the use of such architecture in the case-study system.

2.3.2.5 UNIXes

FreeBSD is a UNIX-like operating system [FreeBSDwww] that is free for use and modification. The operating system is ported to many hardware platforms including Alpha processors, PowerPC, Intel x86 and Sparc64. The FreeBSD project has a SMP branch [FreeBSDSMPwww] and many open-source programs and applications can run under FreeBSD.

The *Linux* kernel [Linuxhqwww] (from version 2.0 and upwards) supports SMP [LinuxSMPwww] for the Intel x86, UltraSparc, SparcServer, Alpha and PowerPC architectures. As mentioned earlier the kernel lock granularity is becoming finer for each new kernel release. Version 2.0 of the kernel locked the whole kernel during the execution of a service call. The next version (2.2) improved kernel locks on signal handling, interrupts and I/O handling. The most recent version (2.4) has added fine-grained locks in the VFS (Virtual File System), the VM (Virtual Memory) component as well as the scheduling and I/O kernel components.

The most recent stable kernel (x86), version 2.4, conforms to Multiprocessor Specification from Intel [Intelwww4] [Maxwell99], and therefore supports up to 16 processors in a symmetric multiprocessor system. The official kernel release has three scheduling policies; real-time non-preemptive scheduling, real-time preemptive scheduling and time-sharing scheduling. Preemptive scheduling in Linux refers to user processes; the kernel is still not preemptive.

Solaris is a mature UNIX implementation from Sun [SUNwww]. The operating system [Mauro2001 p.10] has real-time capabilities in the sense that it implements a pre-emptive kernel and supports kernel threads (which means that kernel

functionality can block). High-resolution timers and fixed priority scheduling along with fine-grained processor control are other real-time implemented properties. The kernel supports dynamic loading of modules (at run time), such as schedulers, file systems, system calls and device drivers. This is a useful addition to an otherwise monolithic kernel. The dispatcher (scheduler) selects which kernel threads are to be run, and there are scheduling classes involving 170 priorities. There are four scheduling classes for kernel threads as well as a special scheduling class for interrupt threads as can be seen in Figure 12 below. TS (Time Sharing, default thread class) and IA (Interactive, for faster windowing) share the same priority range. Threads assigned with system class priorities (SYS) run until they are blocked. The highest priority level of thread priorities is the RT (Real-time) scheduling class where threads are assigned a fixed priority.

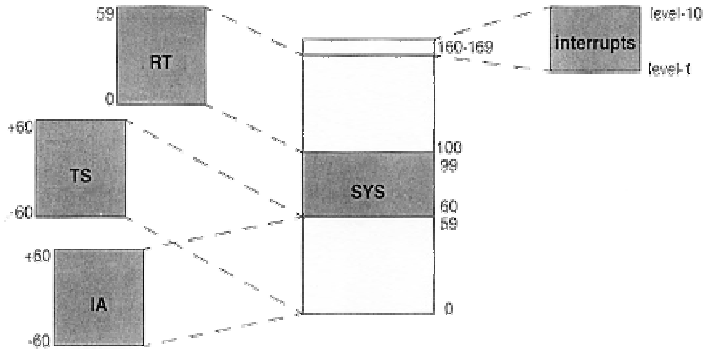


Figure 12, Solaris thread priorities [Mauro2001].

As in most UNIXes there exists a distinction between user level threads and kernel level threads. Figure 13 illustrates the relationship among threads, processes and LWPs (Light Weight Processes). The black boxes at the bottom depict the available processors and the dotted lines illustrate the mapping of threads onto those. In the multithreaded process (with its own virtual memory range) the user is aware of at least one LWP or kernel thread at a time. Additional libraries are available that allow for the use of user threads that the kernel is not aware of. Multiple user level threads are thus able to be executed in the context of one LWP. In Figure 13 this is illustrated as multiple threads above to the leftmost LWP in the process.

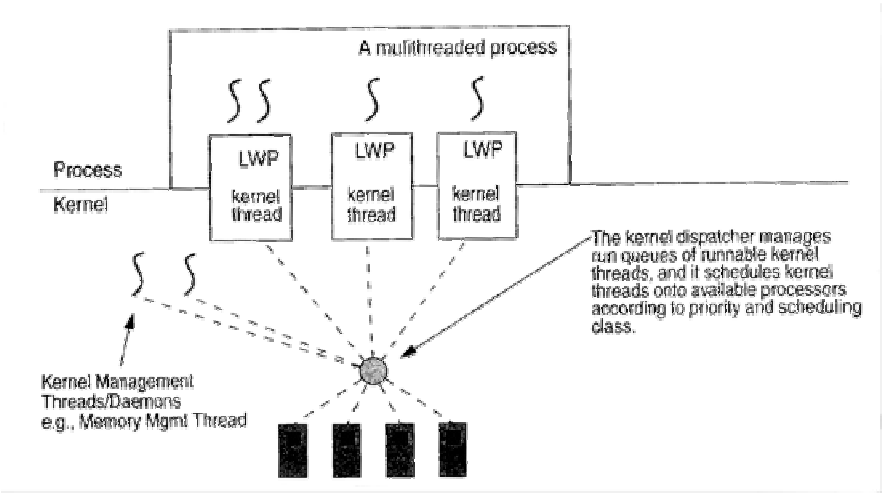


Figure 13, Solaris Threads, Processes and Lightweight Processes [Mauro2001].

Interrupt threads have the highest priority in the whole system. At the time of an asynchronous interrupt (external interrupt), all lower priority interrupts are masked and the interrupt handler executes in the context of a partially initialized interrupt thread. Only if this interrupt thread is being blocked at synchronization object is the thread completely initialized. This decreases the necessary time needed for handling the interrupt. The thread that was interrupted by the interrupt is pinned, meaning that the interrupt thread borrows the LWP (Light Weight Process) from it. This eliminates the need of a complete context-switch. Lower level interrupts are disabled while handling the interrupt, even when the interrupt thread is being blocked.

Solaris is a capable of executing on SMP architectures and is employed in many server systems. It supports up to 64-way symmetric multiprocessors and the SMP scheduling module supports processor affinity (binding), i.e. supports a process to be fixed at a certain processor. Solaris has for example been employed in a billing gateway system, which collects billing information about calls from mobile phones [Häggander2001]. Originally the system ran on in single processor architecture but tests were made on running the application on an 8-way Sun Enterprise 4000. The application suffered from a slowdown when using multiple processors. One of the reasons for the performance loss was that multiple threads needed dynamic memory allocation. The C-library functions used were not re-entrant and the interface to the malloc() and free() functions was protected by a global mutex [Häggander2001 p. 37]. Exchanging the lock with parallel heaps, enabling threads to allocate memory in parallel, gave a near linear speedup. This illustrates the need of creating fine-granular locks in SMP operating systems.

2.3.2.6 VxWorks

VxWorks [Windriverwww] is one of the mostly employed real-time and embedded operating systems in the world. Distinguishing characteristics include a memory footprint that scales down to less than 100 KB and support for a large variety of hardware architectures. The kernel (Wind) is a priority based preemptive microkernel that provides scheduling with the priority inheritance algorithm. The kernel implements a subset of the POSIX standard including POSIX semaphores and message queues. Wind is a kernel that does not provide memory protection or virtual memory support by default (can be added through the VxVMI module) and the kernel is constructed to handle resources in single processor environments. A user that wants to communicate between two or more processors in an architecture that provides for shared memory can add the *VxMP* [Windriverwww] module. This is an add-on to the Wind kernel that provides access to shared memory objects through the use of global memory areas (to be used in for example CompactPCI based architectures). Additional libraries have been added to support this: smObjLib, smObjShow, semSmLib, msgQSmLib, smMemLib, smNameLib. These libraries provide the support for shared semaphores, shared message queues and shared message partitions. Every participant (as for example tasks on different processor boards) has to know where to access each shared object, and this is implemented with the help of a shared name database. The id and the name of the object can be stored together and resolved by the participants. VxMP provides a transparent interface for the use of the above features, only the create routines are different. The shared memory objects can be used in a single processor system as well and it is possible to run VxMP in heterogeneous environments, executing on both the Intel architecture and the PowerPC architecture.

While VxMP enables programs to communicate and synchronize their activities, no support for executing VxWorks as a single kernel image on a SMP multiprocessor system does exist. Board Support Packages (BSP) from SMP processor board manufacturers enables users to start multiple kernel images on SMP architectures. But with such a solution each processor executes one kernel image. The kernel does not contain kernel locks that are necessary for true SMP execution. A dual processor board from Synergy [Synergywww] (SBC-KGM5) is an example where two kernel images are booted. The implementation relies on the VxMP module for synchronizing and communicating between processors. The implication is that two threads cannot invoke the same kernel functionality, and the kernel cannot load balance between the processors.

An alternative approach to the above is called Real-Time Asymmetrical Multi-Processing (R.A.M.P.) from General Micro Systems [GMSwww]. R.A.M.P. does not implement a pure SMP kernel but lets slave processors (application processors) load a microkernel (RAMP/MK) that is able to communicate with the VxWorks kernel image executing on the master (bootstrap processor) with the help of mailboxes and semaphores. This communication is provided by a module contained in the VxWorks BSP. Consequently, only one VxWorks kernel image is needed (one license).

2.3.3 Monitoring and Measurement Techniques

Monitoring and measuring techniques are used to estimate system performance and run-time behavior. Information that can be observed includes *data flow* information, e.g. input and output, or *control flow* observations that include task switches, interrupts, kernel execution overhead as well as clock tick rate and *resources* in the system, such as memory, CPU utilization and network utilization and contention.

The *probe-effect* in concurrent systems has been described in a paper by Gait [Gait86]. The definition of the probe-effect is "an alternation in the frequency of run-time computational errors observed when delays are introduced into concurrent programs". The delays that a software debugger introduces in a system can either mask errors in such a way that a non-functional concurrent program works or make a functional concurrent program stop working.

Software monitor tools install a software component that enables the debugger or monitor device to continuously extract information. This includes collecting information about the average amount of IPC-messages or the number of cache misses during execution. The main disadvantage of software monitors and other software debug facilities is the probe-effect that is the result of the debug mechanism using the CPU, memory and other resources in the system. The main advantage is flexibility and the ease of configuration. Most commercial tools available are intrusive, usually buffering information on a fast medium (i.e. memory) or continuously streaming the data to a host system. One such tool is WindView from Windriver [Windriverwww] that is bundled with development tool Tornado.

Collecting information from a running system will almost always be intrusive. Some attempts have been made to minimize the probe effect, and a project called MAMon [Shobaki2002] has successfully shown that it is possible to monitor system level events without interference. This approach is dependent on special customized hardware that includes an operating system hardware accelerator.

Probing the system and saving the data in memory can be one way to extract data from the system, but it is also intrusive (memory overhead and memory-bus congestion). It might be more attractive to use an external peripheral device attached to the target that continuously snoops and saves data on an external medium. This is an example of a *hybrid monitoring tool* consisting of a minimal software probe that sends collected data to the external device.

2.4 Parallel and Real-Time Communication Mechanisms

The purpose of this section is to define general communication terminology and properties as well as providing a survey of modern communication mechanisms. We will start with a general monologue on the anatomy of communication in computer systems since many of the techniques discussed have been encountered in the scope of this work.

2.4.1 The Anatomy of Communication in Computer Systems

Communication involves at least two participants that exchanges information with each other. The *information* needs to be delivered through some *medium* (does not apply to light) and participants (peers) involved in communication need to agree on a common *protocol* for data exchange. The protocol defines both the sequence of the messages between the peers as well as the format of the data [Coulouris94 p.69]. *Peer-to-peer* communication occurs between two participants and information can flow in either one direction (*simplex*) or in both directions (*duplex*). It is also possible for one participant to communicate with multiple peers in so-called multicast or broadcast communication. *Multicast* communication is targeted to a selected group of receivers, while *broadcast* communication is targeted to everyone who is listening. Broadcasts can be targeted at all possible receivers in the network, while multicasts are targeted at a selected group of receivers within the network, as for example a set of nodes. Both multicast and broadcast communication is categorized as *one-to-many* communication.

Communication can be initiated either by the sender or by the receiver, called *sender-initiated* or *receiver-initiated* communication. In the first case, the sender is the initiator and in the second case, the receiver is the initiator. *Message passing* is categorized as sender-initiated communication, while *shared memory* communication between processing elements is categorized as receiver initiated communication (every read to a shared memory location is initiated by the receiver). Senders are labeled *producers* or *suppliers* of information. Receivers are labeled *consumers* of information. Message passing can combine synchronization with data exchange while shared memory communication requires the use of some explicit synchronization mechanism, as for example semaphores, in order to combine data exchange with synchronization.

Communication can be asynchronous or synchronous. *Asynchronous* (non-blocking) communication allows the sender to continue execution directly after the sending phase. *Synchronous* (blocking) communication implies that the sender blocks until the receiver has entirely received the message. *Rendezvous* is a special case of synchronous communication where both the sender and the receiver agree to synchronize before continuing.

The *medium* over which the communicating peers are sending information can be either *reliable* or *unreliable*. If the protocol that utilizes an unreliable medium requires reliable delivery of data, the protocol implements a feedback technique such as *acknowledgement* of data exchange. Acknowledgements give information to the sender that the data has arrived at the receiver, the *end-point*. The time for data to arrive at the receiver from the time it was sent is called the *latency*. Multiple parameters are the cause of latencies or delays. The medium or communication channel can experience traffic *congestion* which can cause delays in the communication and the *bandwidth* of the medium also limits the amount of data that can be exchanged per time unit.

During the communication phase, both explicit and implicit data can be exchanged. *Explicit communication* is for example performed when data is sent over a reliable communication channel. *Implicit communication* is for example performed when a reliable communication protocol transmits acknowledgement packets. The protocol does also introduce *artifactual* communication, i.e. data that is not explicitly attached by the user.

Multiplexing occurs when a data exchange is split up over multiple channels (logical or physical). A data stream may for example be *fragmented* into multiple packets and sent over a network to a receiving node. At the end-point, the receiving node, the stream of fragmented packets can be *de-multiplexed* into the original order. Both the sending and the receiving node can *buffer* data. Output buffering (sender buffering) includes temporal storage of data that is to be dispatched onto the communication channel. Input buffering (receiver buffering) includes the buffering of incoming data in a node before delivering it to the receiving processing element or process.

In order for peers to *notify* each other of events, two basic approaches can be employed. The first one is for the peer to *interrupt* the other and the other is for the receiver to continuously *poll* a shared synchronization object (such as for example a shared memory location). Some communication protocols allow receivers to *subscribe* to events. In distributed computer environments it is common for peers to interact as *clients* and *servers*. Clients send *requests* to servers that execute the requested services. Servers do *reply* with the result to the clients.

2.4.2 Parallel Communication Protocols and Architecture

Heddes [Heddes94] has completed survey of parallelism in communication subsystems that describes the levels of parallelism that can be targeted in a parallel protocol implementation. These levels include stack level, layer level, entity level, function level and operation level parallelism. Parallel protocol stacks on shared memory multiprocessors have been investigated by, among others, Yates [Yates97] and Björkman [Björkman93]. Communication systems can internally exploit parallelism in different forms, such as layer-level parallelism or connection-level parallelism. Erich M. Nahum [Nahum97] as well as Björkman [Björkman93] has presented a taxonomy of protocol stack parallelism:

- *Layer parallelism* is a coarse grained level of parallelism according to Nahum. Each distinct layer of the protocol stack is the unit of concurrency and the approach exploits clean interfaces between layers. Drawbacks include increased amount of context switching and synchronization as well as concurrency being limited to the number of layers in the protocol.
- *Connection-level parallelism* exploits concurrency among processing elements that are associated with different connections. Multiple connections can be processed independently and this level of parallelism exploits the natural

concurrency among connections and keeps locking to a minimum. A drawback is that it is difficult to achieve speedup within individual connections.

- *Packet-level parallelism* dedicates a processing element to the processing of individual packets. With this approach it is possible to achieve parallel execution both with multiple and single connections. A drawback is that each layer usually needs locking of the shared states (as for example sequence numbers in the TCP/IP stack).
- *Functional parallelism* exploits parallelism in a single protocol layer, such as for example checksum and acknowledgement generation. Parallelism within the layer can be exploited, but synchronization among the functional entities can become a problem.
- *Data-level parallelism* lets separate pieces of data (from the same message) be processed by multiple processing elements.

The appropriateness of deciding whether to use one or multiple levels of parallelism of those presented above depends on the available concurrency within a protocol stack. Kaiserswerth [Kaiserswerth93] identified three forms of parallelism that can be exploited within a protocol stack:

- Pipelining among protocol layers represents an approach where parallelism is exploited between layers. An example is for example network interface cards that implement the medium access control layer and the host processor executes the higher layer functionality of the OSI reference model.
- Parallelism and Pipelining within a protocol layer represents a parallelism that for example exploits independency among connections to handle data from separate connections independently. Another example is the parallel execution of checksum and routing decisions within a layer.
- Parallelism among shared components represents a form of parallelism that exploits parallelism of components that are used by multiple layers. An example is buffer managers or timers.

2.4.3 Hardware Communication Assists

One way of increasing performance for communication mechanisms is to dedicate special purpose hardware, or a dedicated processor, for the task of assisting the processors with communication. *Network processors* are used to relieve the processor/s in the system from handling communication. Most commercial network processors are suitable for use together with Internet protocols, e.g. IP and TCP. Network processors are configurable, but usually a special development environment targeted at a special operating system will be needed. For example, the Intel IXP series of network processors [Intelwww3] are programmed with the help of a software developer kit (SDK) tailored for certain operating systems such as Linux and VxWorks. The IXP 1200 is equipped with a StrongARM processor and six

programmable micro-engines that operate on the incoming IP packets [Vassiliadis2001].

Another architecture providing communication assistance is the QUICC [Motorolawww] architecture developed by Motorola. It has an integrated network processor with dedicated memory that is able to operate in parallel with the main processor. The architecture is described as an "Integrated Multiprotocol Processor (IMP)" where both the general purpose processor and the communication processor reside in the same die. The main MC68000 processor is offloaded by a RISC processor that can be programmed operate on incoming data from the I/O ports. Compared to the network processors described above, the QUICC architecture is more simple, yet useful for many embedded applications. Other products and techniques for accelerating communication exist and we discuss some of them in section 3.2.1.

2.4.4 Parallel and Distributed Communication Mechanisms and Frameworks

Data originating from data producers (e.g. I/O boards) needs to be transmitted to consumers (e.g. applications) efficiently. In order for an application to connect and receive data from the data producers, an I/O systems service is needed. In its simplest form, an I/O board may be located on an I/O bus in the system, notifying the processor of a data delivery by issuing interrupts. In such a system, the application designer is forced to explicitly program Interrupt Service Routines (ISR) and to read and handle incoming data, as well as synchronize with the receiving application thread or threads. Therefore, it is desirable to let a system component provide an abstraction toward the underlying hardware, operating system and communication mechanisms. Such a system component is commonly termed a *middleware* software layer [Andrews2000 p. 32]. The benefits of middleware are:

- To provide a standardized interface for users of the middleware. This promotes execution in heterogeneous architectures by hiding the underlying hardware architecture, communication mechanism and operating system. The middleware hides the distributed system components so that local and remote functions can be accessed uniformly and that their location does not have to be known. The middleware is said to be *transparent*.
- It provides a programming model for the user that unifies the components using the middleware. For example, middleware relying on remote method invocation steer the users of the middleware to design with request-reply semantics.
- The middleware should promote scalability in the sense that multiple nodes can access the available distributed resources efficiently.

2.4.4.1 RPC and RMI

In heterogeneous systems the need of creating an abstract data representation becomes important. Nodes in such systems must be able to exchange data structures regardless of hardware architecture. The process of mapping data structures into

forms suitable for transmission over the communication medium is called *marshalling* [Coulouris94 p. 103]. The reverse process is called *unmarshalling*. RPC and RMI mechanisms provide for a standardized way of data representation. XDR is an example of such a data representation language and is used in the Remote Procedure Call (RPC) component developed by SUN. Other remote invocation techniques include JavaRMI and XML-RPC [Allman2003].

Remote Procedure Call (RPC) and Remote Method Invocation (RMI) are related in terms of the interaction model. Both allow clients to remotely invoke procedures or methods at the server side. The major difference between RPC and RMI is that RMI adheres to an object oriented design where methods of remote objects are invoked while RPC mechanisms are useful in non-object oriented mechanisms. Remote Procedure Calling (RPC) [Bloomer92] is an important concept in a client/server environment. By utilizing RPC the process can access and receive references to procedures and data that are not accessible in its local memory space. ONC RPC by SUN and NCS by HP [Bloomer92] are examples of two RPC mechanisms that are commercially available. Many UNIX services are built on RPC mechanism, but remote invocation is much slower than invocations in the same address space. Null RPC-calls, i.e. remote invocations carrying no parameters and no data in response to the requester, have shown to take hundreds of microseconds to complete in a 100Mbit LAN environment. Bandwidth is not the limiting factor in this case but rather delays that originates from operating system operations such as network driver execution and context switches as well as RPC-mechanism code [Coulouris01 p. 234].

Many RPC calls actually occur between processes on the same local machine. In these cases it is inefficient (due to overheads originating from marshalling and data copying) to execute a complete RPC-mechanism designed for remote invocations between nodes. Therefore, RPC optimizations such as Lightweight RPC (LRPC) [Bershad90] have been proposed. Since local RPC-calls can use local memory to pass parameters, they do not need to copy data. A node-to-node RPC call can have up to four copying phases.

2.4.4.2 CORBA

CORBA, the Common Object Request Broker Architecture [OMG2002], is a specification of a middleware platform that enables objects to be distributed on multiple nodes in the system. Its main use is in client/server environments, where method invocation can be performed on remote objects as well as local objects. The purpose is to decouple communication between objects, thus increasing scalability and reusability of the distributed applications. Since the CORBA middleware platform is capable of running on various operating systems and architectures, heterogeneous computing is made possible.

The reference model consists of clients making requests to objects via the ORB core. While remote invocations of methods promote scalability and reusability, the mechanism relies on a request/response communication paradigm (server method

invocation). Request/Reply design uses a two-way communication pattern which may be ineffective in real-time systems.

OMG has defined a real-time extension to CORBA. The extension targets real-time related issues such as priority queuing, priority inheritance and ORB guarantees on execution [Wolfe97]. A CORBA implementation that targets real-time system design is the *TAO* (The Ace ORB) architecture [Harrison97] [Schmidt98] [Kuhns99]. It is operating on many real-time operating systems including VxWorks. TAO enables the user to specify Quality of Service requirements, requirements that then affect the way TAO schedules messages and resources. TAO has been used in many real-life applications, spanning from avionics systems to telecom systems [Schmidt98].

The CORBA model is object oriented while the publish/subscribe mechanisms described in the next section are data centric. Some, among them the creators of NDDS (described below), therefore claim that publish-subscribe mechanisms are more suitable for distributed real-time systems.

2.4.4.3 Publish/Subscribe Mechanisms

RPC mechanisms are widely used today in distributed applications. Internet services use RPC, but embedded and real-time systems [Windriverwww] can also utilize RPC. However, there are limitations and difficulties with this communication paradigm. The most notable is the inherently synchronous exchange where the requester needs to wait for the reply, which leads to static applications and possible performance degradation due to busy wait. Publish/Subscribe mechanisms [Rajkumar95] on the other hand decouple the producer of events and data from the consumer of events and data. Subscribers inform the publish/subscribe mechanism, i.e. the middleware, of its interest in certain events or data. Publishers then publish data to the middleware, which asynchronously sends the data to the subscribers. Some interesting properties apply for such mechanisms [Eugster2001]. First, subscribers need not need to know of the producers, not their exact numbers nor their locality. The same applies for publishers; the number or locality of the subscribers need not be known. Secondly, the interaction between the subscribers and the publishers does not necessarily need to occur at the same time, publishers may even unregister before all events have arrived at the subscribers. Finally, subscribers can be notified asynchronously thus enabling the subscriber to perform concurrent work. A positive aspect of publish/subscribe mechanisms is that scalability is promoted. This is mainly because publishers can dynamically connect to subscribers without explicitly stating dependencies [Hill2002]. This promotes flexibility; the whole system does not have to be reconstructed because of adding a new application or new functionality in an existing application. In recent years both academic and commercial publish/subscribe mechanisms and real-time object brokers have been released. As we shall see in the following sections, the need for publish/subscribe mechanisms is growing and a new standard has been released from OMG.

The publish/subscribe paradigm is implemented in distributed event and notification servers where objects of interest change their states and deliver the

notification of that event to interested subscribers [Coulouris2001 p.190]. Objects of interest, i.e. the event publishers, connect to an event server that forwards the event notification to the subscriber. In such an approach the event service has the opportunity to filter and correlate the events. This can for example include publishing a notification whenever a defined number of events have arrived. This is similar to the semantics of the case-study I/O system. Another mechanism that enables subscribers to receive notifications from publishers is the CORBA Event Service [OMG97] and its real-time enhancement. The real-time event service enables the user to filter events and select to correlate events either by conjunction (AND) or disjunction (OR). Thereby the application can for example request the event service to wait for events from supplier objects A and B to arrive before pushing the notification. The CORBA Event Service provides three main features that enhance the original CORBA specification [Harrison97]:

1. Asynchronous message delivery.
2. Allows one or many suppliers to send messages to one or more consumers.
3. Suppliers and consumers are decoupled in the sense that they do not know about each other explicitly.

2.4.4.4 SPLICE

SPLICE (Subscription Paradigm for the Logical Interconnection of Concurrent Engines) is an early work implementing the subscription paradigm for embedded systems [Boasson93] [Boasson96]. The work was performed for the military and the experiences from the project are now used by the company THALES. THALES has been a contributor to the OMG Data Distribution Service Specification described below. The basic functionality of SPLICE is that it provides a shared data model based on the two primitives read and write. A receiver performing a read on a set of data will block until it arrives. The architecture consists of applications, agents and the network. Agents handle all communication needs on behalf of the application, and provide the necessary functions for maintaining the shared data space.

The designers of SPLICE criticize the CSP (Communicating Sequential Processes) model as well as the client/server paradigm for connecting the senders and receivers tightly, thus leading to complex designs. They argue that the subscribe paradigm, by providing a loosely coupled connection between data producers and data consumers, will provide a more flexible system design.

2.4.4.5 NDDS

NDDS (Network Data Delivery Service) [Pardo-Castellote1997] [Pardo-Castellote2001] is a publish/subscribe mechanism available as a middleware for many platforms and operating systems. The company that has developed this product is Real-Time Innovations (RTI), which is one of the contributors of the Data Distribution Service Specification described in the next section.

NDDS consists of a run-time library, a database and tasks (threads) that perform all the necessary marshalling, addressing and transporting services. The creators of

NDDS claim to provide symmetric design and real-time performance [NDDS]. By symmetric design they mean that all nodes in the distributed system are equal, no specific node is for example responsible for address lookup, thus avoiding a single point of failure. NDDS is built on top of the unreliable connectionless transport protocol UDP. All semantics provided by NDDS, such as reliability, quality of service (QoS), and data representation (CDR) is implemented on top of UDP. NDDS has been used in multiple commercial and real-time systems and the mechanism has its heritage from the robotics industry.

2.4.4.6 The Data Distribution Service for Real-Time Systems Specification

A consortium of companies including THALES, RTI and Objective Interface Systems have submitted a specification for a Data Distribution Service (DDS) for distributed real-time systems [DDSRTS2003] [EETimeswww2] to the OMG group. The model is called as a data-centric publish-subscribe mechanism (DCPS) and the application domain is expected to be high-performance and predictable real-time systems, found in industrial automation, distributed control, telecom equipment and sensor networks. The previous work of SPLICE and NDDS has been the driving technique for this specification and the goal is that the mechanisms should scale to hundreds and even thousands of publishers and subscribers. The data model consists of unrelated data-structures that are identified by topic and type. The topic uniquely identifies data items in the global data space, while the type gives information to the middleware about how to handle items (resembles marshalling functionality of RPC mechanisms). The Data Distribution Service for Real-Time Systems Specification has been issued as a mechanism that can be used together with the CORBA standard. Even though object models and the design can be mapped to CORBA platforms, the standard is largely platform independent, i.e. it can be used on other platforms as well.

2.4.4.7 Functional Parallelism, Parallel I/O and Data Concentrators

Depending on area of subject, I/O denotes different aspects of architecture. In industrial systems, I/O usually depicts data transfers related to sensors and actuators, or the signals connecting an Integrated Circuit (IC) to other IC. In the high performance computing (HPC) community, the term *parallel I/O* is commonly used to describe the system exchanging data with the file system [May2001]. I/O system can therefore come to mean different mechanisms depending on context. We mention parallel I/O in order to clarify the difference between the use of the term in industry and in the high-performance community.

Parallel I/O can be defined as providing a form of *functional parallelism.* Functional parallelism was exploited in early parallel systems that utilized special-purpose file processors and dedicated programmable peripheral processors [Lawson92 p.252]. The approach of separating I/O system and application functionality on separate nodes can be described as exploiting functional parallelism.

The term *Data Concentrator* is used in different contexts ranging from devices that provide monitoring and protocol conversion [BVMwww] (for example transmitting

serial data over Ethernet) to devices providing data collection functionality in SCADA systems [IPCASwww]. The IPCAS system is called ipDaco and one of its purposes is to increase performance by limiting communication with the data concentrator. It lets a communication server handle communication with the SCADA system. In that sense the ipDaco acts as a proxy server for the data acquisition devices.

2.4.5 Scalable Network Architectures and Parallel Communication Services

In applications that are larger than targeted in the scope of this work, scalability for nodes in a network and especially the Internet has been examined. This section gives an introduction to those issues since some of the concepts are related to the parallel execution of I/O System and application components examined within the scope of this work.

Shah [Shah2001] has presented a scalable system architecture called *Comm Services Platform (CSP)* that offloads and decouples the TCP/IP processing from servers. The purpose is to improve the performance of services built on top of TCP/IP and the architecture is illustrated in Figure 14 below. Network nodes accelerated by network processors perform the first level of packet forwarding to the proxy nodes, mainly for load balancing purposes. The proxy nodes decouple network transport protocol processing cycles from application node compute cycles. The proxy node is the terminating end-point for TCP connections, and the data contained in the connection is forwarded to the application nodes with the help of a lightweight transport protocol.

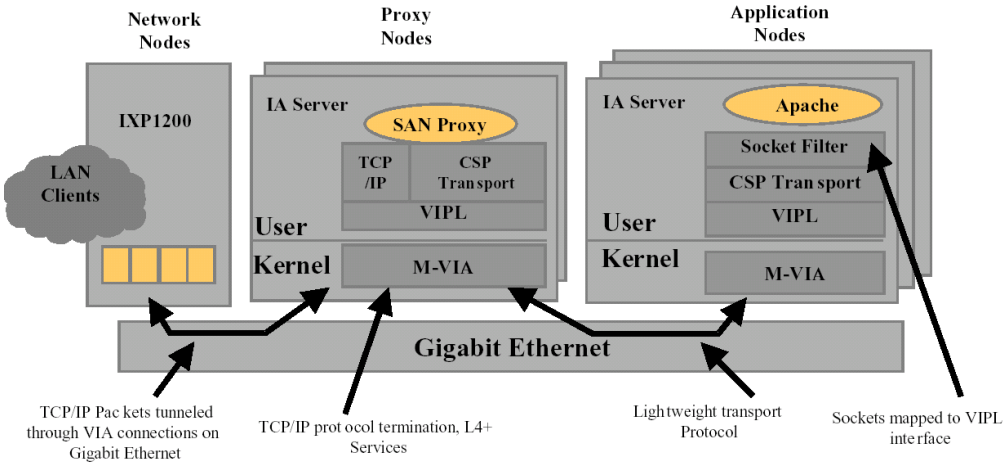


Figure 14, the CSP architecture [Shah2001].

The creators of the CSP architecture identified these main benefits of using their system platform:

- The decomposition of the system into a functional pipeline allows scaling of each pipeline stage independently. Therefore proxy nodes can be scaled independently from application nodes.
- Proxy nodes can also execute higher level functionality (above the transport layer) and thereby offload protocol processing in the application nodes.

The platform is based on the idea of functional pipelining and independent scaling of application and communication. The use of proxies is a technique related to building secure and scalable networks. A proxy is a communication mediator that allows for two nodes to communicate indirectly. A proxy usually maintains two data connections, one to each node, and the proxy ("in the middle") can thereby govern the data flow as well as the content of the data. This technique is discussed in a paper written by Spatscheck [Spatscheck98] et. al., who have defined TCP Forwarding as: "…communication relayed over two TCP connections via a proxy".

2.5 Programming Models

A parallel programming model specifies what data can be named by the threads, what operations can be performed on the data, and what ordering exists among these operations [Culler99 p. 53]. A parallel programming model is the data parallel processing [Culler99 p. 44] model. The data parallel processing model enables operations to be performed on each individual element in a data structure (SIMD architecture according to Flynn [Flynn96]). This model has evolved into the SPMD (Single Program Multiple Data) model thus converging with message passing and shared memory.

The two most commonly used parallel programming models are the message passing model and the shared memory model. The message passing programming model has become popular and widely used with the advent of MPI (Message Passing Interface) [MPIwww]. MPI is based on the use of send and receive primitives. While MPI has been used in clusters and in the HPC (High Performance Computing) world, the deployment of MPI in the embedded and real-time system domain has not been common. In real-time and embedded systems message passing is commonly implemented as an IPC mechanism.

The shared memory programming model needs an underlying hardware or software mechanism that provides access to shared memory. When such mechanisms are available, the application designer may use multithreading to support parallel execution. PThreads [Nichols96], i.e. POSIX Threads, and OpenMP [Chandra2001] are widely used mechanisms for parallel programming on a shared address space.

In industrial automation and control systems the factor governing the programming model are standards. IEC 1131-3 [Lewis98] is for example a standard that governs the programming of programmable logic controllers (PLC), devices common in industrial control systems. The most important feature in the IEC 1131-3 standard is the function block. A function block defines input and output data parameters that interconnects with other function blocks, and also defines an

algorithm that executes every time the function block is invoked. In today's modern industrial systems it is getting more common to execute logic and algorithms on general purpose processors, but the programming models and techniques remain. This means that functional blocks resembling IEC 1131-3 functional blocks are executed as a part of the application on the general purpose processor.

2.5.1 Message Passing versus Shared Memory

There has been much debate regarding which communication model is preferred by parallel application designers. In our case-study we had the choice of adhering to message passing or shared memory (the bus-based architecture provides for non-cache coherent shared memory) and we therefore studied the pros and cons of the two communication models. An important remark is that this summary concerns cache coherent shared memory architectures. Not all attributes of the cache coherent shared memory column are therefore applicable to a non-cache coherent shared memory architecture. For example, a bus-based backplane architecture utilizing a shared memory area does not reuse remote data; every access is to its primary storage.

The advantages and disadvantages of message passing mechanisms and shared memory mechanisms are summarized below in Figure 15. The information originates from multiple papers, which are referenced below.

	Message Passing	Cache Coherent Shared Memory
+	Offers efficient bulk-transfer of data [Culler99 p. 187] if DMA transfers cover for the gathering overhead [Chong98 p. 5].	Fine-grained data transfers (cache-lined) are efficient.
+	Offers good performance with known communication patterns [Chong98 p. 1]. Data can be communicated when produced which enables overlapping of the transfer with computation [Woo94 p. 219].	Global naming and coherent replication of data eases programming [Shan2001 p. 1], especially for irregular, dynamically changing communication patterns [Woo94 p. 219].
+	Combines synchronization with data transfer [Chong98 p. 1].	Reuse of remote data.
+	Robust to relative changes in processor to network latencies and bandwidth [Chong98 p. 2].	The user accesses the hardware directly with loads and stores, which eliminates a need for an extra software layer [Culler p. 269].
+	Messages are usually unacknowledged [Chong98 p. 2].	
+	Only a single pass through the network.	
+	Data is replicated in local main memory [Woo94].	

+	Easier to scale to large numbers of processors compared to shared memory architecture [Byrd98 p. 1].	
+	No cache or memory consistency problems [Andrews2000 p. 9]	
-	Higher overhead for fine-grained data transfers [Chong98 p. 1].	Overhead increases when shared data is frequently modified on different processors [Chong98 p. 1].
-	Extra copying to and from buffers is needed when data is not consecutive [Chong98 p. 1].	Adds round-trips latencies in the network. This can be facilitated by implementing prefetching [Kranz93] or using a more relaxed memory consistency model.
-	Higher communication overhead since messages (headers) must be constructed.	Generally requires more network bandwidth.
-	Message Passing can be extremely difficult to program, especially for irregular structured computations [Shan2001 p. 1].	
-	Low computation to communication ratio yields many messages and thus much overhead [Chong98 p. 6]. Interrupt synchronization may be a major overhead.	
	Messages are sender-initiated and thus asynchronous for the receiver.	Shared memory communication is receiver-initiated [Byrd94].
	Messages are *value oriented;* all data must be contained in the message to make computation progress at the receiver [Byrd98 p. 1].	Shared memory communication is reference oriented. References to memory locations can be passed to the receiver for further computation [Byrd98 p. 2].
	Pre-communication may avoid round-trips needed for a read-on-request model [Chong98 p. 5]	Shared memory programmers should be aware of locality.
		Shared Memory Prefetching depends heavily on a low computation to communication ratio [Chong98 p. 4]

Figure 15, Advantages and disadvantages of message passing mechanisms versus cache coherent shared memory mechanisms.

Frederic Chong [Chong98] has addressed the subject by performing a series of experiments on the MIT Alewife system. The goal was to "gain insight into the relative performance of communication mechanisms as bisection bandwidth and

network latency wary" and the study compares message-passing mechanisms to cache-coherent shared memory. Chong used the bisection bandwidth as a measure on the bandwidth of the system as a whole. *The bisection bandwidth* is the "sum of the bandwidth of the minimum set of channels that, if removed, partition the network into two equal unconnected sets of nodes" [Culler99 p. 761]. This metric is rather useful if the communication pattern is completely uniform in the whole system. If this is not the case, the metric is pessimistic. The access latencies to memory are variable in the MIT Alewife (NUMA), in converse to a SMP system (UMA) in which all processing elements have equal latencies to primary memory. Figure 16 shows the expected performance scaling of the two communication mechanisms when the bisection bandwidth varies.

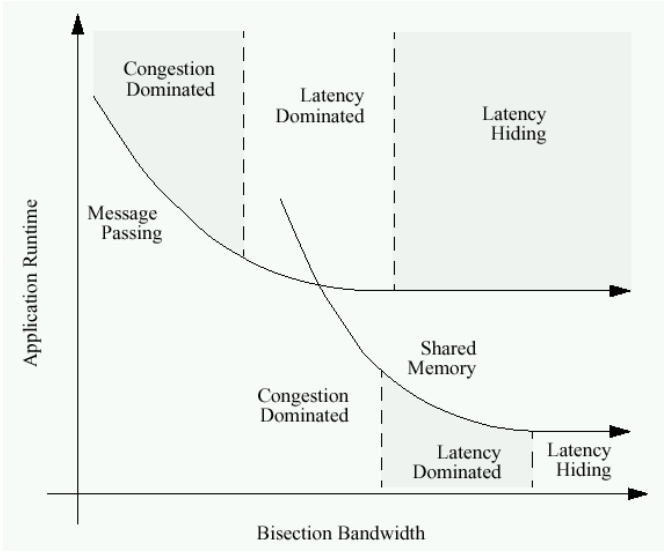


Figure 16, Regions of performance in processor cycles as bisection bandwidth varies [Chong 98].

Three regions are identified; the congestion dominated region, the latency dominated region and the latency-hiding region. In the *congestion-dominated* region the effects of congestion are bigger than the decrease in bisection bandwidth. Shared memory mechanisms require much communication and suffer from congestion problems earlier than the message passing mechanisms. In the *latency dominated* region message-passing communication suffers because of lack of parallel work compared to latency, while in the shared memory (sequential consistency) case the processors are stalled when data is not in the cache. In the *latency hiding* region latencies in the network are hidden by low communication volume compared to the amount of work being performed. Network latencies can also be varied as illustrated in Figure 17. Here it is worth mentioning that message passing tolerates network latencies better due to one-way nature of the mechanism. Noteworthy is also that

shared memory architectures need to use prefetching techniques in order to hide network latencies. Other studies such as the one performed by Chandra [Chandra94] et al. suggest that the differences between message passing mechanisms and shared memory mechanisms are not big (no prefetching or DMA transfers were used). In one case where computation was dominant, the shared memory's higher cost was offset by the higher latency of handling messages in software. The higher overhead from handling buffers for message passing was visible. In another test case the shared memory mechanism required a noticeable amount of explicit synchronization.

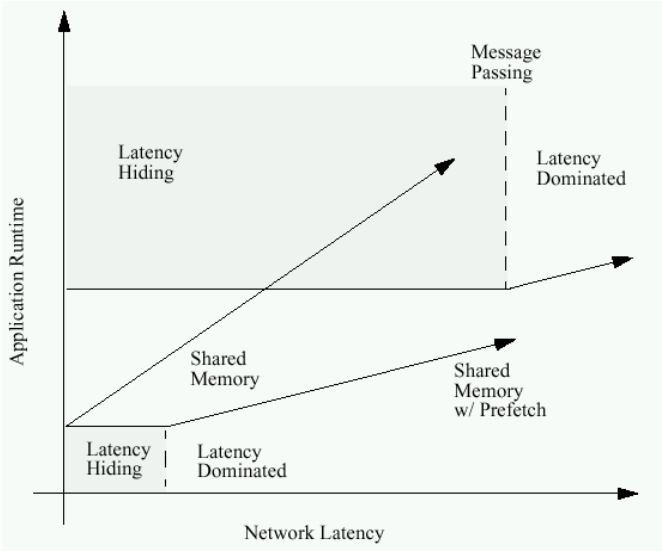


Figure 17, Regions of performance in processor cycles as network latency varies [Chong 98].

2.6 Definitions

We adhere to the definitions below.

- *Concurrency*. The degree of concurrency is defined as by [Grama2003 p. 89]: "The … number of tasks that can be executed simultaneously in a parallel program".
- *Correlation*. Actions that relate individual items into a group according to a specification. Simple correlation specifications include AND and OR logic operators on item characteristics. Item characteristics include topic, time and type. Correlation actions are common in event services and notification services [Liu97].
- *Distributed System*. Coulouris [Coulouris01 p. 2] defines a distributed system to be a system where: "Hardware and software components located at networked computers communicate and coordinate their actions only by passing messages".

- *Execution Environment*. An execution environment consists of an address space, thread resources such as semaphores and higher level resources such as open files [Coulouris01].
- *Independence of Parallel processes*. Let the read set of a part of a program be the variables it reads but do not alter. Let the write set of a part be the variables it writes into (and possibly also reads). Two parts of a program are independent if the intersection of their write sets is empty. This definition is taken from [Andrews2000].
- *Multiprocessor system*. We adhere to the definition of a multiprocessor by Stone [Stone90 p.452]: "A parallel computer composed of multiple independent processors and facilities for controlling their interaction and cooperation". According to the definition of a multiprocessor by Enslow [Enslow77] only a system that can be controlled by a single operating system can be depicted as a multiprocessor.
- *Parallel System*. Almasi and Gottlieb, cited from [Culler99 p.1] define a parallel computer to be: "A collection of processing elements that communicate and cooperate to solve large problems fast".
- *Process*. A process consists of an execution environment and one or many threads of control.
- *Task*. There is much confusion concerning where to use the term thread and task. Usually the terms represent the same entity, a context of a program, where memory may be shared among other tasks. Stankovic [Stankovic91] defines a task to be a schedulable entity that consists of reentrant code, local and global data, a stack, a task descriptor and a task-control block. See also *Thread*.
- *Thread*. A context of a program, which may run concurrently with other threads or processes, and may share memory with other threads.

3 Performance Improving Alternatives for an Industrial System

In the search for more computing performance, system designers face a plethora of possibilities. Choices regarding the platform and components need to be made in at least four main areas of the system, each level asserting certain requirements on the others. A layered model, as illustrated in Figure 18 below, illustrates how the system components depend on each other (the grey area depicts the definition of the whole platform). Requirements flow from the hardware and upwards, as well as from the application and downwards. Each layer enhances and hides the functionality of the layer immediately below it [Kaiserswerth93].

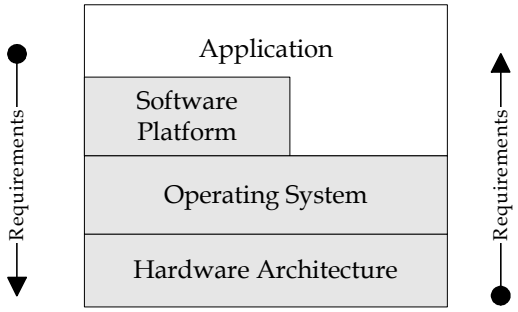


Figure 18, a layered view of the system.

The design decisions are governed by the needs and requirements of the application, and in a broader sense the requirements and needs of the system under control. The application asserts requirements on the platform and operating system layers. Operating systems expose interfaces to its users and where its main purpose is to abstract away the hardware architecture from the design of the application and the platform. The application should for example not be responsible for managing a file system. Conversely, the operating system interface asserts requirements on how the higher levels of the system are designed, including the application. In order to promote easy porting to different hardware architectures, operating interface standards such as for example POSIX [Quarterman93] have been defined. The POSIX standard is an example of the need for standardized interfaces among system components as well as the need for platforms and applications to be easily ported to multiple hardware architectures.

This chapter will discuss architectural design decisions encountered within the scope of this work and during the study of the case-study system. The discussion is

based on the precondition that we have an existing system platform that has been designed for single processor architecture and where instructions of any thread are allowed to access any memory position.

3.1 The Case-Study System

The target of this study is an industrial system that is used to control and protect power system equipment such as for example transformers. Papers that describe the research with this system include [Enblom2001][Enblom2003][Enblom2001_02]. Applications are of a large variety and therefore the company has created a platform consisting of useful and necessary services to be used in application design in order to promote reusability. The platform has been tailored for the need of applications and has four major characteristics:

- Hardware architecture dependencies have been hidden in the sense that devices, such as I/O boards, are handled and configured by the platform.
- I/O boards and the flow of data to and from these is governed by a publish/subscribe middleware; the I/O System. The I/O System is capable of subscribing to data from remote nodes as well as devices located in the local chassis.
- The platform is modular in terms of file-structure and interfaces to components. Yet, the platform has been designed for a single execution environment in the respect that every platform and application compiles to a single executable image presently executable only in a single processor environment.
- The platform has been designed for a multi-threading environment, where services are executing in the context of threads with various priorities. These threads are at present scheduled by the priority-based real-time operating system VxWorks [Windriverwww].

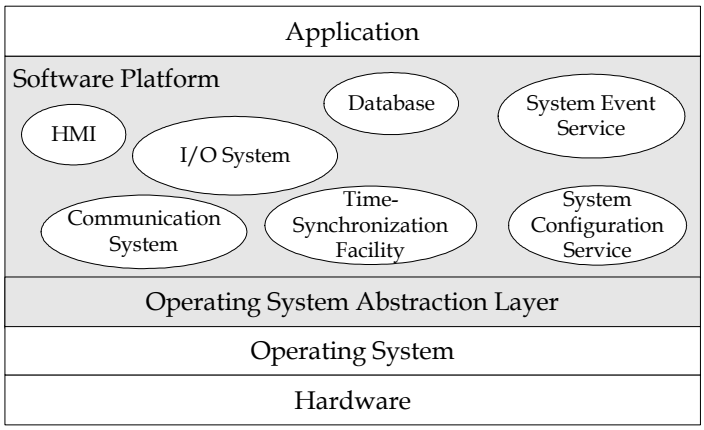


Figure 19, System Overview.

Figure 19 above presents an overview of the system and its most important components. The software platform components are abstracted away from the operating system by means of an operating system abstraction layer that is to be used by the components of the software platform and the application alike. Choosing to use such an abstraction promotes portability, enabling developers to move existing applications and framework from one operating system platform to another without having to explicitly adopt the code to a new operating system. Operating systems providing the services required by the abstraction layer can be used in the system.

The real-time requirements of the system-under-control are challenging, delays in the removal of the anomaly may increase the damage to the faulty equipment or damage healthy equipment [Davies96]. Therefore the maximum end-to-end response time for the system to respond to a critical event in the environment is less than 20 milliseconds. Within this timeframe parameters of the environment and the system under control must be sampled and the data must be transmitted to the receiving application node. The operating system and I/O System on the node must receive the data in a timely fashion and the application must have enough history of the data in order to take the correct action. The decisions taken by the application must thereafter propagate to the actuator via the I/O System and the operating system, and finally over the interconnect to reach the actuator (circuit breaker). This implies a maximum allowed response time of the computer system of less than 10 milliseconds. Our case-study system relies on a periodic sampling of the environment and system under control that currently has a period of 1ms (1000Hz). This means that 20 sets of sampled data are produced before a fault in the environment (the power grid) must have been isolated. Ideally the application completes executing its protection algorithm each period, a pattern applications are designed according to. This periodic execution of the application is defined to be

data-driven since the application execution is dependent on the arrival of data from data producers.

The hardware architecture is based on a rack-mount Compact PCI system consisting of I/O boards and a processor board. The existing software system is divided into different modules where the time-critical components are the I/O system and the execution of the application protection and control functionality. The time-critical part of the application is written in such a way that it blocks for data to be delivered from the I/O System. The most common approach is to wait for a complete set of sensor data has arrived in a data structure (DS). The common structure this part of the application is illustrated with C-like semantics in Figure 20 below. The sequence of events is such that the application waits for data to arrive, executes the application semantics and thereafter acknowledges the data structure so that it can be reused by the I/O System. The application performs the same task every cycle of execution (sample period).

```
Identifier dataStructureId;
DataStructure *dataPointer;
while( running == true ) {
    dataPointer = waitForDataStructure( dataStructureId );
    executeApplicationCode( dataPointer );
    acknowledgeDataStructure( dataPointer );
}
```

Figure 20. Application structure example.

The system is based on the periodic collection of data from the environment and the last functional component executing before delivery of data structures to the receiving application is the I/O System thread. Function calls to components that return references to data structures that have been allocated internally by the I/O System may be invoked. It is expected that the caller should read and write to those data structures with ordinary processor loads and stores (using RISC architecture terminology).

Figure 21 below illustrates a function call to a software module that returns a reference to a data structure allocated within it. First, as illustrated by arrow 1, the receiver invokes a wait function that blocks at a synchronization point, awaiting notification from the producer (I/O System thread). Thereafter, illustrated by arrow 2, data is collected, correlated (defined later) and prepared for delivery. Arrow 3 represents the notification of new data to the receiving thread, in turn enabling the application thread to progress (arrow 4). A reference to the delivered data is passed back (arrow 5) along with other parameters. At some point later in time, the application thread may act upon the data as illustrated by arrow 6.

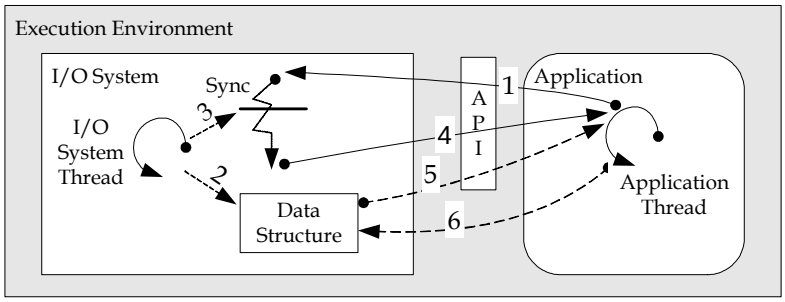


Figure 21, Component interaction in the system.

When separating the application and the I/O System into separate execution environments (such as in a distributed system), reading the data in the data structure may not be possible. In such as system the data structure must be copied to the execution environment of the application which can be achieved with the help of IOMP described in section 4.3.

3.2 Hardware Architecture Alternatives

For a system designer that wants to increase performance by migrating a product and platform that is designed for a single processor environment to a distributed or multiprocessor architecture, a number of issues arise. The main purpose for considering such a migration is usually to increase performance, e.g. increasing the number of operations that the application and the system as a whole can handle. Other reasons may be to provide redundancy and fault-tolerance [Storey96], but we have not analyzed any explicit fault-tolerance aspects, even though that would be of interest for the application domain. Three approaches that are potential ways of increasing performance in the system have been identified during the analysis of the case-study system:

1. Accelerate a system component with the help of dedicated hardware.
2. Introduce a multiprocessor architecture.
3. Upgrade the existing system with faster single processor architecture.

Figure 22 below illustrates these approaches and we have also identified three hardware architectures with multiple processors.

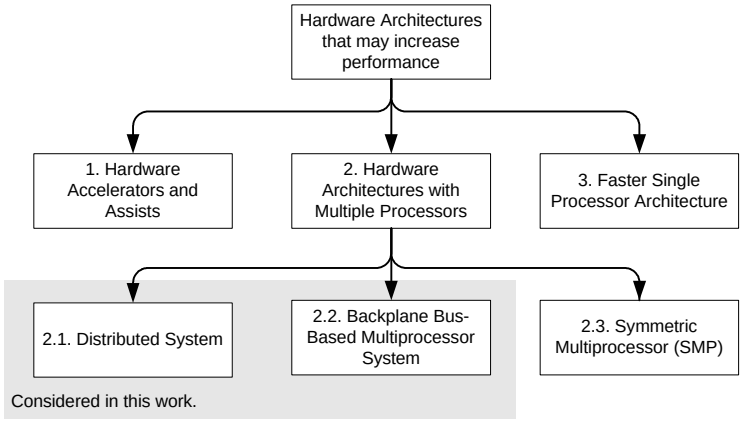


Figure 22, Hardware Design Alternatives.

The symmetric multiprocessor (SMP) is a Uniform Memory Access (UMA) architecture according to the classification of Hwang (see section 2.2.3.3). The backplane bus-based multiprocessor can be classified as a Non Cache-Coherent Non-Uniform Memory Access (NCC-NUMA) architecture. The distributed system alternative can, when only considering the ability to access memory, be classified as a NORMA architecture.

3.2.1 Hardware Accelerators and Assists

Increasing performance with the help of hardware assists and accelerators is possible for many systems. An obvious component that may be accelerated is the communication system. But, the communication system of industrial automation systems is usually developed by the manufacturing company itself, and is therefore unique. The system may execute on off-the-shelf processor boards while peripheral sampling and actuation devices are special purpose custom designs. This is the case with the case-study system. Therefore it can be hard to find third party components that can easily increase performance in the system by offloading the communication system. For systems relying on standardized communication protocols, such as for example TCP/IP, there exist products such as protocol processors, network processors [Vassiliadis2001] [Vitessewww] and TCP/IP Offload Engines (TOE) [Tensilicawww] [10GEAwww] that can offload the system in order to gain performance. These solutions are usually deployed for use in gigabit networks where the network speed is more demanding compared to the data rate at which the host can process, i.e. the host processing capability is the bottleneck. Three factors that are processing intensive for such protocol stacks are reassembling packets, copying of memory and interrupts [10GEAwww].

Another approach is to use an operating system hardware accelerator in order to increase performance. Research on hardware accelerated operating systems has been

performed at Mälardalen University [Lindh98] and a similar approach has been proposed for the Spring [Burleson99] system. A hardware accelerator for the operating system can offload time-critical components in the operating system, as for example interrupt handling, scheduling and synchronization. An approach with a hardware accelerator for the case-study target system has been proposed in [Enblom2001], and this approach is discussed in more detail in section 3.2.2.2. One possible hardware architectures for accelerating functionality in the case-study architecture includes reconfigurable boards such as Wildcard [Annapmicrowww] from Annapolis Micro Systems, but these products are still quite expensive.

We have analytically identified three main subsystems that could benefit from being hardware accelerated by hardware:

1. Application functionality such as algorithms, filter (FFT) and/or logic are candidates for hardware acceleration. In our case-study system application functionality is mainly generated from graphical tools, that automatically generate software. After a brief review and analysis of the design flow using those tools we (together with an application designer) concluded that the amount of concurrent tasks in the application is to low to be considered within the scope of this study.
2. A second approach would be to accelerate subsystems of the I/O system. This could include accelerating the in-house real-time protocol stack or the dispatch engine of the I/O System. After a brief review and analysis of the I/O System together with a designer we found no obvious ways for parallelizing and/or accelerating the I/O System.
3. Platform functionality, such as for example the database engine and the time synchronization facilities are also candidates for hardware acceleration. No comprehensive review or analysis has been performed on the possibilities of accelerating these components, so the possible impact is unknown at present.

3.2.2 Hardware Architectures with Multiple Processors

The second approach is to introduce a multiprocessor architecture for parallel execution of system components. We have identified three parallel architectures that are possible to utilize with the current case-study system; a distributed architecture utilizing the existing network topology, a bus-based (CompactPCI) multiprocessor architecture and a symmetric multiprocessor architecture. The next sections will discuss these architectures as well as the approach of hardware accelerating the operating system on a bus-based architecture.

3.2.2.1 Distributed Architecture

Today nodes in the system can communicate over a network (AnyLAN-100VG) in order to exchange data. It is possible to connect processor boards using this network, and in chapter 4 we analyze the approach of executing I/O system and application

functionality separately in such architecture. The following applies for the distributed architecture approach:

- The nodes do not share the same execution environment.
- Communication is limited to the passing of messages between the nodes.
- Bandwidth is limited to approximately 12,5 MB/s.
- The processor clock of each node must be synchronized independently.
- Each node consists of a chassis containing processor boards as well as peripheral boards. Therefore the distributed approach will require more hardware and will thus be more costly than the backplane bus-based architecture approach.

3.2.2.2 Backplane Bus-Based Multiprocessor Architecture

A passive backplane is a printed circuit board with connectors placed at regular intervals to allow connection between printed circuit boards. Processor and peripheral boards are inserted into slots that enable boards to connect to the backplane bus, and an example is the CompactPCI-bus. It is possible to connect multiple processor boards in the same chassis (rack-mount). An approach of utilizing a centralized operating system accelerator in hardware for such a system has been discussed in [Enblom2001] and the proposed hardware architecture is illustrated in Figure 23 below. In such an architecture context switches are triggered by the hardware operating system (RTU), and the remaining software RTOS on each board saves current context and makes next task running. The RTOS does also provide an interface for accessing the features of the hardware operating system such as changing priority of the tasks, setting periodic timers as well as synchronization and inter-process communication functionality.

While an operating system accelerator certainly can increase performance for functionality such as IPC in a system [Furunäs2000], the main part determining whether the system will scale on a multiprocessor is how well the software system scales on the particular hardware architecture. An efficient synchronization mechanism does for example not provide that enough concurrent tasks are found in the system in order to enable parallel execution. Figure 23 below illustrates an approach where a hardware operating system (from now on abbreviated RTU, i.e. Real-Time Unit) controls operating system related functionality. The RTU implements the following operating system functionality in hardware; task/thread handling, timers, interrupt handling, shared resource handling (semaphores) as well as message passing functionality [Furunäs2001]. The remaining software of the operating system (RTU Software OS) provides for functionality that cannot be performed by the hardware accelerator, such as for example saving and restoring the context of a thread. An inherent benefit of utilizing an external operating system is a performance increase due to the lack of timer-interrupts on processor boards. The tasks do not require rescheduling or corresponding actions until the RTU issues a taskswitch command. This gain can be as large as 32% [Furunäs2001], compared to a system with a conventional scheduler in software having to administer clock ticks.

Yet, in the system we have studied, the processor administrative load for each clock tick was 16μs (the Intel Pentium 3 hardware platform). The period time for the clock tick was 1ms which means that the clock tick administration overhead for our case-study system is 1,6%. Clock tick administration is not judged to be a serious overhead in the case-study system.

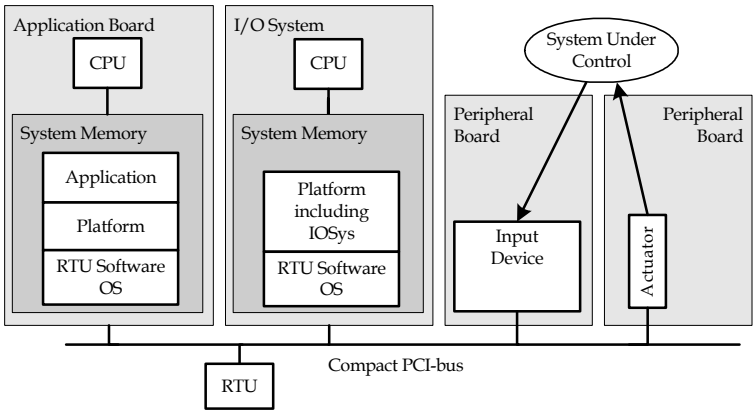


Figure 23, Proposed Hardware Architecture with an operating system accelerator [Enblom2001].

As described in section 2.3.2.5, VxMP enhances VxWorks with the ability to share semaphores, provide mutual exclusion to shared data structures, shared message queues and shared memory management. A possible system configuration using these components is illustrated in Figure 24 below. A comparison between the RTU and the functionality provided by VxMP showed that the same operating system functionality necessary for functional separation of the existing platform is provided by both approaches. Functional differences between a VxWorks+VxMP system and the RTU includes that the RTU provides a centralized system clock, provides scheduling of external interrupts and provides a global scheduling policy for tasks on multiple boards. The conclusion is that it is not necessary to utilize a centralized hardware operating system accelerator in order to achieve functional scalability for a backplane bus-based multiprocessor system. Therefore, in order to understand how scalability could be achieved in the target system we decided to study how system components could be partitioned on multiple processors in the system in order to achieve both functional scalability and increase performance. This approach led to the study of the separation of I/O System and application level functionality in a distributed system, as described in chapter 4.

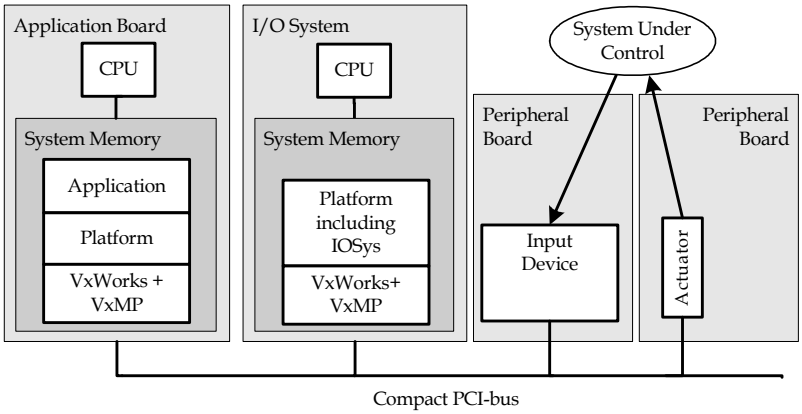


Figure 24, a possible alternative approach using VxWorks + VxMP.

3.2.2.3 Symmetric Multiprocessor (SMP) Architecture

A third architecture alternative is Symmetric multiprocessors (SMP) that represent a hardware architecture where all system components such as primary memory, peripherals and buses are equally distant to each processor. The most important difference between this architecture and the two described above is that it can execute the same system image, meaning that both processors have access to the same primary memory for fetching instructions and data. This in turn means that two processors can issue the same operating system call simultaneously thus executing operating system code in parallel. Possible dual processor SMP system architecture is illustrated in Figure 25 below and the SMP System board could for example be the cPCI-DT64 equipped with dual Pentium III processors [Kontronwww].

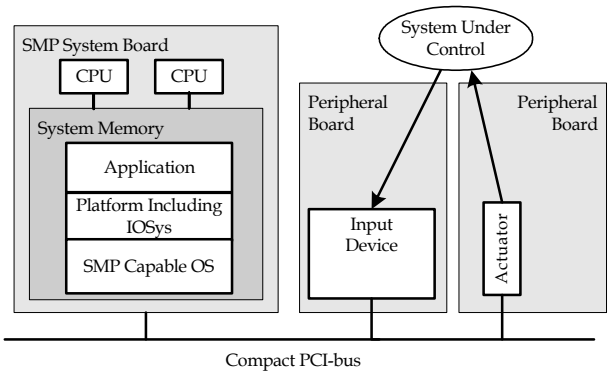


Figure 25, possible dual processor SMP system architecture.

The following applies for the SMP system architecture approach:

- The nodes share the same execution environment.
- Communication between processors is performed by accessing shared memory, equally distant from each processor and where cache coherence is enforced by hardware.
- Bandwidth to memory is high, over 1GB/s for most contemporary architectures.
- All processors share the same clock.
- The SMP system board can fit into one chassis. Therefore such an approach will require less hardware than the distributed approach and will thus be less costly.
- It is necessary to utilize a SMP-capable operating system.

SMP architecture may be interesting for future consideration for the following reasons:

- No functional partitioning is necessary in order to scale on the multiple processors of the SMP.
- Individual dynamic load balancing of threads and processes governed by the operating system is possible.
- Minimal or no software differences are needed for execution on a single processor and the multiprocessor.

Obstacles to introducing SMP architecture include:

- The power dissipation of each processor board is larger than for a single processor board. Since the systems studied in this work cannot have any moving parts such as for example fans, power dissipation for each board is limited.
- The cost for SMP architecture boards have generally been high, but lately the prices have dropped compared to single processor boards.

3.2.3 Faster Single Processor Architecture

Waiting for faster single processor architecture is presently the way of increasing performance in the target system. Today the target system has supports two generations of hardware architectures, a 100MHz PowerPC based architecture and a Pentium III 266 MHz based architecture. While the next generation system, as for example a faster clocked Pentium 3 architecture, can increase performance dramatically, an architecture with multiple processors can potentially enable execution of demanding applications and configurations that cannot execute in the most powerful single processor architecture available. This is what is defined as *flexibility*; the possibility of introducing a multiprocessor system and thereby having the ability to scale beyond the performance of the most powerful single processor

architecture. It is not necessary to run the new system on a multiprocessor architecture, so for the cases where extra performance is not needed a single processor solution is sufficient [Enblom2001]. At the same time as we have introduced flexibility with the multiprocessor solution we have also introduced complexity [Pancake96] in terms of increased difficulty debugging the application and more parameters to remember when writing applications.

3.3 Software Architecture Considerations

When functionally partitioning existing system components onto multiple processors, load balancing issues become important. The challenging part is the adaptation of the application and software platform, which has been written for a single-processor system environment and a single processor RTOS. Partitioning of threads on separate processor boards therefore becomes an important question. There is a need to differentiate between the real-time parts of the system (I/O System and real-time tasks having stringent timing constraints) from less timing constrained tasks such as logging functions and HMI (Human Machine Interface) functionality.

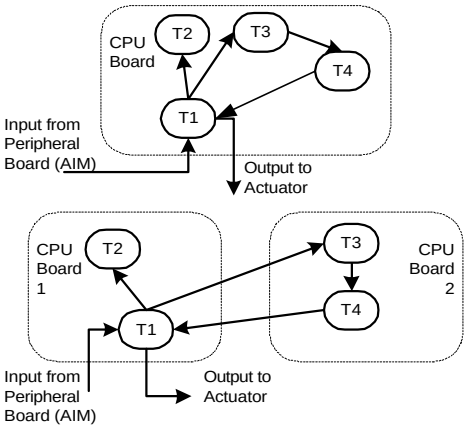


Figure 26, example of tasks in the original system as well as the new multiprocessor system and a possible configuration [Enblom2001].

In the beginning of the project we reasoned about how functionality should be partitioned among the processors in order to achieve the goals of increased performance. The initial ideas are illustrated in Figure 26 above where the original application and platform has been created and partitioned into many tasks as (the tasks merely illustrate how the application can be configured). One initial conclusion was that communication between the boards has to be minimized to gain performance in the multiprocessor system. Different groups of tasks that share common assignment and cooperate intensely can be identified in the original system. Tasks that communicate intensely are from the communication point of view appropriate to let execute on the same processor. A counteracting factor is the

amount of concurrency available between two or more tasks in the system. Concurrency can be exploited by partitioning tasks on separate processors in order to set free more computing resources. The I/O System and the application were identified of showing a high degree of mutual concurrency and were therefore candidates for parallel execution. The counteracting factor of communication between those components must thereafter be taken into consideration. Fortunately, the communication between I/O System and application has some favorable properties:

- The delivery of sensor data is periodic, meaning that the application will not be temporarily be overloaded by communication data.
- When configured, the same amount of data is delivered from the I/O system to the application each data delivery period. Between applications configurations, the amount of data varies from a few tens of bytes up to hundreds of bytes (the data delivered during the DDP as described in section 4.1.).
- Protection applications deliver a small amount of out-data, actuation actions are only necessary when an anomaly occurs. For those applications it is not necessary to continuously deliver actuation events during the AADP phase (actuation events delivered during the AADP is described in section 4.1.). Control applications on the other hand may yield a higher amount of out-data, where continuous delivery of data during AADP is necessary. We have focused mainly on protection applications within the scope of this work.

At the beginning of the project we considered separating application code from the complete software platform (called the base system) onto multiple processors. This would mean that all functionality available in the software platform would execute in one execution environment and the application code in another execution environment leading to that all interfaces between application and base-system components would be mapped to RPC-calls. The round-trip latencies of RPC makes such an approach unfeasible. We therefore turned from the approach of separating the application from the software platform (base-system) to the approach of separating the I/O system from the rest of the system (including the application and its use of the software platform).

3.4 Identifying Concurrency

In the system that has been studied within the extent of this project an architecture with multiple processors requiring different execution environments was chosen for evaluation. Partitioning of existing components is an important task in order for the system to be efficient. Since we are investigating an existing system, the functionality and semantics of the system must be preserved. The main task will therefore be to identify existing concurrency in the system. If components in the existing product have a large amount of concurrency, the potential of speeding up the total execution of the product is also large. The software system executes as a number of tasks (i.e.

threads) in the VxWorks [WindRiverwww] priority based real-time operating system. These threads express an amount of thread level parallelism, TLP, which can be utilized in a parallel system. We have identified parallelism among threads in order to find the best option for parallel execution. Parallel execution of components within each existing thread is certainly possible, as for example hardware acceleration of mathematical functions. But that would require new analysis of existing functionality and how acceleration would be implemented. The designers of the system have already identified parallelism in the form of the existing threads. An easy and natural way of scaling on multiple processors is to rely on this existing concurrency.

The system that has been studied in the case-study consists of a platform which is used for application designers to connect to I/O boards, synchronize the system boards to a high precision to an external clock, data base components as well as logging and monitoring of system events. Concurrency can be exploited either within existing multithreaded components or between components. The two approaches have different characteristics that may enable more or less parallel execution, and some of these aspects will be discussed below.

3.4.1 Concurrency within Components

- **Concurrency within applications**. Existing concurrency within or between application threads may enable parallel execution and thus an performance increase. Overlapping independent activities within or between threads is usually depicted as *slackness* in literature. Finding slackness between application threads can indicate that they can scale on multiple processors, i.e. be configured to execute in parallel on multiple processors. An analysis of the execution of a an existing application and an analysis of the code structure of applications have showed little opportunity of parallel execution among application threads. Existing applications must complete their job within a fraction of a millisecond, typically less than 0,5 milliseconds and the real-time critical parts are contained within one thread of execution. Partitioning application threads onto processors in a network will yield a very high communication overhead, latencies for nullRPC calls have been measured to 0,36 milliseconds [Enblom2003]. Even with a PCI communication mechanism, round-trip latencies for messages consisting of 100 bytes are in the domain of 50 microseconds [Enblom2002], even for contemporary processor architectures. It will be very hard to find enough concurrency within application threads so that a feasible communication /computation ratio can be achieved. This is true for both the distributed and board based architecture.
- **Parallelism within the I/O System**. A system component that consumes a large amount of computing resources is the I/O System. Depending on how many input and output devices are connected to the application via the I/O system, a processor utilization of approximately 10% up to 100% derives from I/O system activity. This includes interrupt handling, interrupt service routine (ISR)

execution, as well as data correlation and dispatching of data to receivers (usually applications). Parallelism within the I/O system component for data originating from local peripherals (on the PCI-bus) is hard to achieve mainly due to the intricate software structure. Memory references between modules and functions are common and only one thread context exists in the I/O System. Remote data reception (via the network) utilizes a real-time transport mechanism which could be separated from the dispatcher functionality. No obvious way of exploiting parallelism within the I/O System was found.

- **Parallelism within other components**. Other components such as time synchronization and data-base components are other candidates for exploiting concurrency. Many of those components are designed for existing in the same execution environment as the callers, which makes redesign necessary. The components would need to rely on copying of data instead of returning references to memory locations in the same address space. We have not seen any straightforward way to achieve speedup of those components.

3.4.2 Parallelism between Components

- **Parallelism between I/O System and Applications**. Parallel execution of I/O system and application components is the approach that has the best chances of increasing performance for system configurations subscribing to data from many remote nodes. Both components, the I/O system and the applications, execute in a way that enables parallel execution. The I/O System receives data and dispatches it to the receiving application threads. Parallel execution of this approach in a distributed configuration shows up to 27% less total processor utilization on the application node [Enblom2003] (66% more computing resources available for application functionality). With a PCI board based solution, even more computing resources are expected to be available for applications, see section 4.5.
- **Parallelism between other system components**. Other components may also show the possibility of parallel execution. No measurements have been performed to prove the possibility of parallel execution between other components.

4 Parallel Execution of I/O System and Application Components

This chapter consists of five sections describing and analyzing issues regarding parallel execution of the existing I/O System and application/client components. In section 4.1 we introduce a model of the existing industrial I/O System that has been studied, and we then extend the model to enable parallel execution on multiple processors. Thereafter, in section 4.2, we present a synthetic execution analysis of the system in both a single and multiprocessor configuration consisting of two processor nodes. In section 4.3 we describe how we designed and implemented a software component called **I**nput **O**utput for a **M**ulti **P**rocessor (IOMP) in order to exploit the concurrency between the I/O system and application/client components. Section 4.4 then presents the results from execution in both single and multiprocessor configurations and we discuss possibilities, threats and suggest improvements to the approach. Finally, in section 4.5, we present results from inter-processor communication between two processor boards in a CompactPCI bus-based architecture.

4.1 A Model of a Data-Driven Real-Time System

Before describing the system and introducing the system model we discuss the concept of modeling. A model consists of a set of assumptions about how a system, i.e. real-world facilities and processes, works [Law2000 p.1]. The purpose is to gain understanding of how the corresponding real system behaves in situations where it is impossible to experiment with it or where the system has not yet been built. Furthermore, a system designer may want to change system parameters that cannot be modified in the real system, such as for example the memory hierarchy, the bandwidth of the interconnect or the clock-speed of the CPU. The assumptions of the model lead to simplifications, i.e. the impact of some parameters is simplified or totally removed in the model compared to the real system. This does in turn lead to that an amount of error is introduced in the model compared to how a real system may behave. The simplifications do by necessity introduce errors in the model compared to the real system. Absolute model validity is impossible to achieve, but it is desirable to achieve as much model credibility as possible compared to the invested effort. It is on the other hand desirable to simplify the model description if we want to achieve an analytical solution [Law2000 p.5], e.g. providing a solution with the help of a mathematical model. A good model should in that case describe the behavior of the system as accurately as possible as well as be simple enough to solve [Hu97]. Law et. al. have categorized ways to study the behavior and parameters of a system, a categorization that is illustrated in Figure 27 below.

In this work we have emphasized on experimenting with an actual system, but we have also described and presented an execution model of the system. Generally, if it

is possible and cost-effective, experimenting with the actual system increases validity of the results. In our case we had an existing system available, enabling us to alter and add functionality in order to examine the impact of parallel execution.

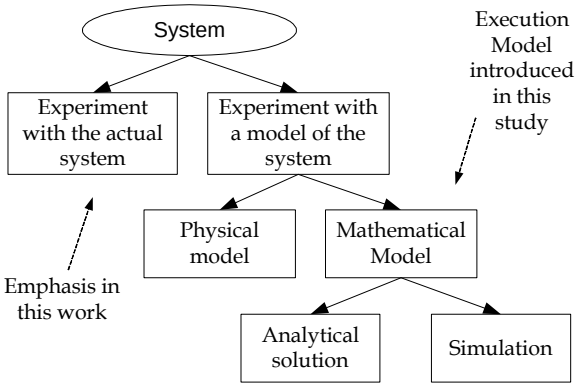


Figure 27, possible ways to study a system [Law2000] and the emphasis in this work.

The execution model is not based on assumptions about how the system is behaving, but rather on the actual system. Execution traces from the system as well as a functional study of program code has been the source of information for the system model. The model, including its simplifications, has been the foundation for identifying concurrency in the system.

The alternative to experimenting with an actual system is to experiment with a model of the system. One of the ways of experimenting is by *physical modeling*, such as for example models of cars in wind tunnel experiments. This method can be excluded for our purposes. The alternative to physical modeling is *mathematical modeling* which, if it is simple enough can be solved analytically or analyzed by means of simulation. *Simulation* may be performed in various forms and at various levels of abstraction. Simulation is the task of imitating a system together with a model of the surrounding system under control, in order to evaluate, understand and identify characteristics of the system. A definition stated by Shannon [Paul93 p.1] is: "Simulation is the process of designing a model of a real system and performing experiments with this model for the purpose of either understanding the behavior of the system or of evaluating various strategies for the operation of the system".

The purpose may be to increase performance by understanding the execution behavior of the system as a whole. The model of the system and the environment may be simple or complex. For example, in order to understand how the queuing of sensor data arriving to a computer system affects the response time of tasks it may be interesting to model the inter-arrival time of sensor data and the associated notification (i.e. interrupt) together with a software model. Simulation of a system can also be a way of decreasing the volume of faults of a design in the early phases of

construction. This can save time and money in a project as well as helping in the verification of the design.

Different levels of abstraction can be chosen in order to study a computer system [Banks2001 p. 529]:

1. Gate Level (Lowest abstraction level).
2. CPU Level, including models of microcode, ALUs, memory interfaces etc.
3. Processor Level, including models of caches, memory and disks.
4. Computer System Level (Highest abstraction level).

Choosing abstraction level affects parameters such as the speed of the simulation, the complexity of the hardware and software model, the accuracy of the simulation results as well as the ability to model in-house peripherals. Generally, low abstraction levels of simulation result in high accuracy, slow simulation and high complexity of the models. The converse applies for high abstraction levels.

A number of simulation techniques are available for the system evaluator. A subset of potential simulation techniques available are listed below (beginning with the lowest abstraction level):

- Co-simulation [Adams96], or co-design, is a technique using the same methodology when designing digital systems including both hardware and software systems. Since our case-study system consists of standard hardware components co-simulation is not very useful for our purposes.
- Complete system simulation is a technique where the complete target architecture is modeled. Peripherals and system components such as network interface controllers, interrupt controllers and the processor instruction set as well as address space are accurately modeled at the functional level. Unmodified software may run on such simulators. Available simulators are SimOS [Rosenblum97] and Simics [Albertsson2000] [Virtutecwww]. We have in our work chosen not to create such a simulation environment due to factors such as complexity of the tools, complexity of modeling in-house peripherals (the data producers) and the initial lack of interest in studying factors such as cache hit ratio and memory footprint. If factors such as cache behavior, memory footprint and interaction with peripheral devices is assumed to play a significant role in performance in future systems, complete system simulators may become interesting tools for system analysis.
- Discrete-event system simulation [Banks2001] [Arshamwww] is a technique where a model of a system is run and where state variables change at discrete points in time. It would be possible to implement a discrete-event system simulator for the system model and vary parameters such as task execution times, network latencies and input data distributions. But when examining the existing system, very little variation in for example input data arrival was detected due to the predictable dispatching of messages on the AnyLAN

network and the system-wide fine-granular time synchronization. In order for us to perform sensible experiments we had to invent input data distributions that were not in accordance with the real system. In the future, if it is interesting to study the effects of varying input data frequency, discrete-event simulation may be an alternative. Discrete-event simulation packages have been criticized of showing poor scalability and that "the complexity of the simulation model approaches the complexity of the system under development" [Chatterjee97].

As described in the next couple of sections, we have created a model of the system. We had three main reasons for creating and describing this model:

1. To gain a comprehensive understanding of how the system is designed and enable us to disseminate results without exposing proprietary information.
2. To enable us to reason about system properties at a high abstraction level and identify relations between system parameters and system performance.
3. To enable us to describe and present metrics from the performance experiments, e.g. processor utilization, response time, latencies and communication/computation ratio.

Having discussed modeling in general, we now progress by describing a model of a data-driven real-time control system, as used in the case study and in part described in [Enblom2003].

4.1.1 System Architecture

The modules of a processor node in the system are illustrated in Figure 28 below. Components which communicate with peripheral devices such as data producers, network peripherals and actuators are illustrated at the bottom of the figure. Three types of peripheral devices are represented in the system:

1. An *output peripheral device* which performs actuations toward the environment according to requests from the IOSys via software component A.
2. An *input peripheral device* which produces data to corresponding system software component B.
3. A combined *input/output device* (for example the AnyLAN network interface) handled by software 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 Program Interface (API) to the application, and actuation data is delivered to the IOSys through the same API.

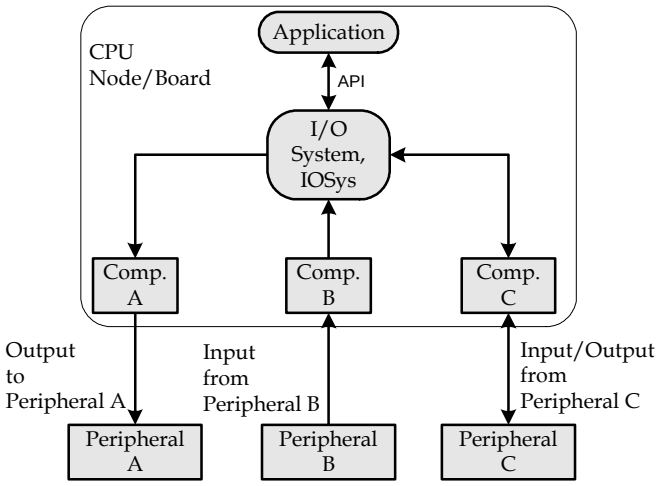


Figure 28, a single processor data-driven node.

A *data-driven system* is in this context 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 peripheral devices. Each time data arrives at the node, 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, remote nodes and possibly other system components.

4.1.2 System Semantics and Functionality

The IOSys provides functionality which can be categorized as being a *middleware*, serving as a layer of software between the I/O devices and the application. The purpose is to hide heterogeneity of the I/O devices and enable portability to many hardware architectures. 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 for example 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 correlated and correlated data (see the discussion regarding correlation 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.

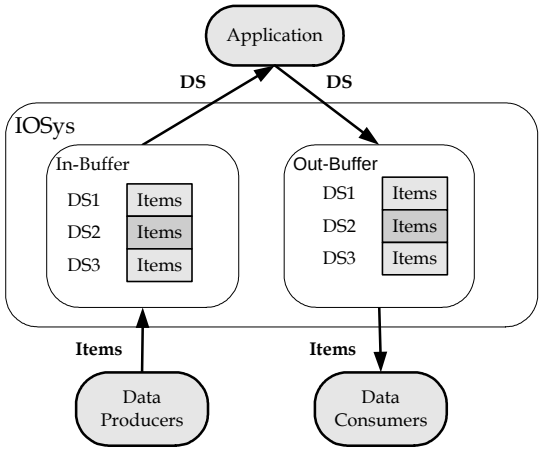


Figure 29, Illustration of data exchange between the data producers/consumers, the I/O system and the application.

The IOSys is in effect executing concurrently with the application, buffering and correlating data to be delivered later. Figure 29 above illustrates a buffer of three data structures, each data set containing a number of data items (possibly from multiple data sources). The IOSys will independently from the application correlate/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 correlate data into the data structures. Data being delivered from remote nodes (items are time stamped) to the local node may be delayed, but the IOSys can still correlate 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 have IOSys correlate 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 produce data at the same rate in the case-study.

4.1.3 The Data-Driven Real-Time System Model Applied to a System with Multiple Processors

In a single processor system, as illustrated in Figure 28 above, both software and hardware system components contend for shared resources, such as the processor,

memory and the interconnects. Priority-based operating systems therefore provide the assignment of priorities to threads depending on task importance. 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 occur, the amount of thread level parallelism (TLP) of the software is higher than the underlying computer architecture is able to utilize. An execution analysis (section 4.2) of the system model allows for the parallel execution of the IOSys and the application threads. The concurrent execution and buffering that the model provides can therefore be exploited by a parallel system at the interface between the application and the IOSys.

In Figure 30 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.

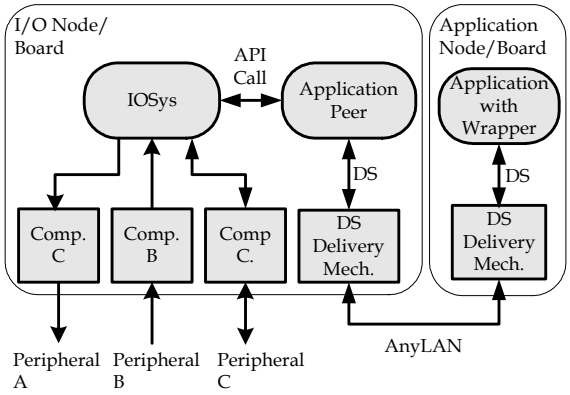


Figure 30, The multiprocessor system model.

In comparison with the single processor system, we have introduced an application peer thread for each application thread that 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 I/O 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 references to data in different execution environments. 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 [Coulouris94 p.129] and complex data structures are flattened. The effect of this mechanism is that the application can be written with the same semantics as in the single processor case.

4.2 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 31 we illustrate data delivery over two sample periods (T_{Sample}).

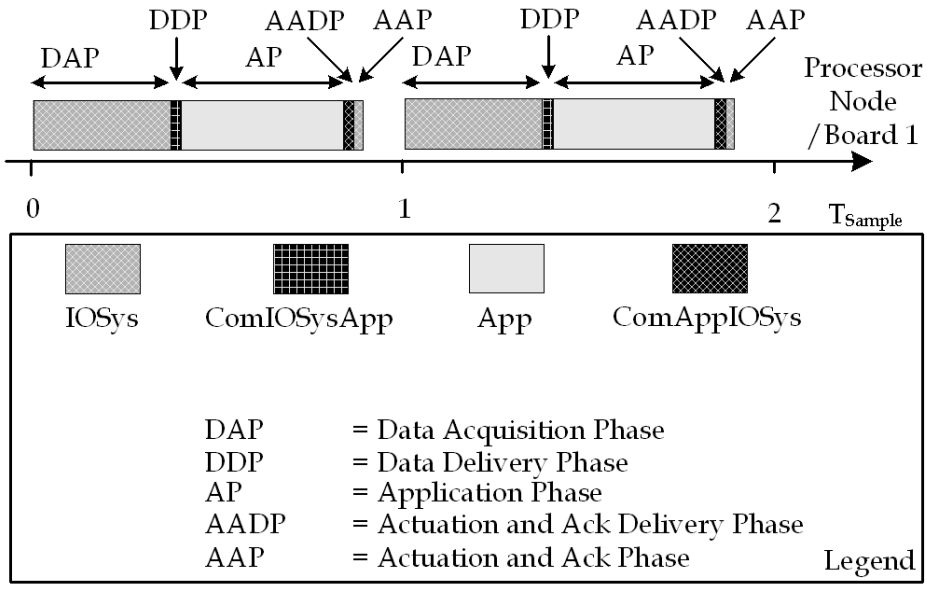


Figure 31. A single processor execution scenario.

We can identify five major phases in the execution, which also are common in real-time control systems based on continuous sampling and actuation of I/O peripherals. The *Data Acquisition Phase (DAP)* represents the total execution time for all data collection functions. The *Data Delivery Phase (DDP)* represents the time for delivery of the collected data during the DAP phase to the receiver. The receiver is usually an application thread, and the execution of the application is represented by the *Application Phase (AP)* in the execution model. At the end of the application phase the buffer used in delivering the data structure is acknowledged and actuation actions that may be due are delivered to the IOSys as well. These actions are represented by the *Actuation and Ack Delivery Phase (AADP).* Finally, during the *Actuation and Ack Phase (AAP)*, the buffer of the data structure is released in order to be used again by the system and data is sent to respective actuation peripheral board.

We introduce five timing parameters that are representing the executing time of each component during the five phases:

- $\mathbf{T_{DAP}}$ represents the execution time of IOSys during DAP.

- T_{DDP} represents the communication overhead between the IOSys and the application during DDP.
- T_{AP} represents the execution time for the application during AP.
- T_{AADP} represents the communication overhead between the application and the IOSys during AADP.
- T_{AAP} represents the execution time in the IOSys during AAP (acknowledgement of DS).

When the load on the system increases, i.e. application that subscribes to more I/O and, it may not be possible to execute the application on the single processor as illustrated in Figure 31 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.

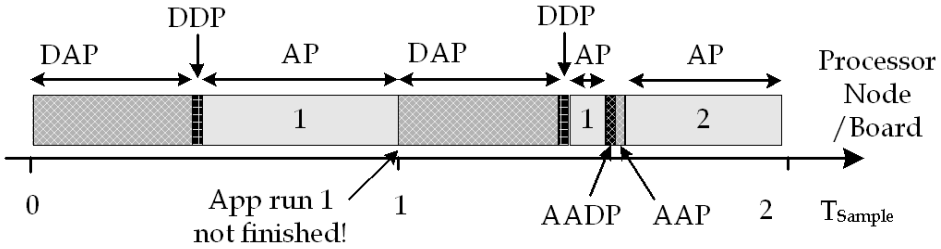


Figure 32, A scenario in which the application has insufficient execution resources.

The execution of the same scenario in a multiprocessor system could yield an execution diagram as shown in Figure 33 below. Parallel execution of DAP and AP could enable an increase in computing resources for the application. But, 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 33, there is an overhead in communication which must be weighed against the benefit of having enabled parallel execution. For the application, the communication overhead T_{DDP} plus the communication overhead of T_{AADP} on the application board is pure overhead. Note that compared to the single processor system, this overhead is divided into three phases from a system-wide perspective. The first is the execution time of the communication mechanism on the I/O board. The second is the communication latency of the channel (illustrated by the dotted arrows) and the third phase is the execution time of the communication mechanism on the application 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.

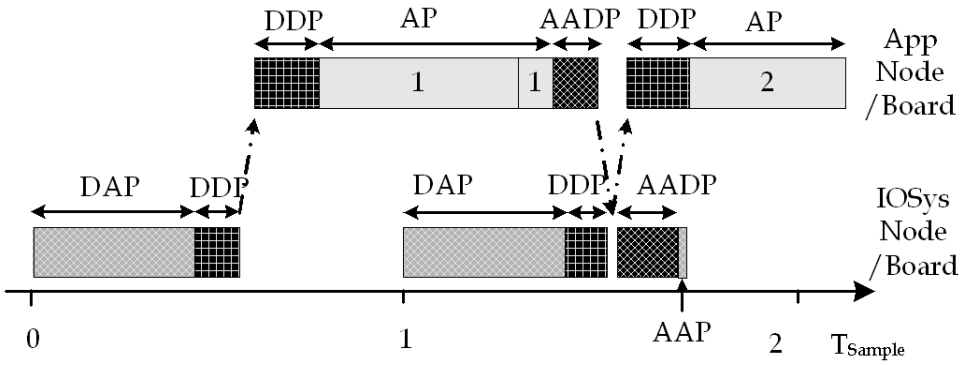


Figure 33, example execution of the application in Figure 32 in the multiprocessor system.

4.2.1 Performance Metrics

Performance metrics [Lilja2000], i.e. quantitative values that are used to describe performance of the system, are introduced in this section. We discuss the relevance of each of the chosen parameters and how they are calculated.

- Processor Utilization is a metric that represents the amount of work (in percent) performed each measured time interval compared to the amount of processor idle time. The granularity of the metric can vary, ranging from fine-grained processor utilization measured each sample period (millisecond granularity) up to coarsely grained processor utilization measured over the whole measurement period (second granularity). In our measurements we have measured processor utilization over the whole measurement interval ($400T_{Sample}$). This is due to that processor utilization has traditionally been measured at a coarse granularity by system developers. We have concluded that processor utilization is an important metric to use for two main reasons; firstly it is the number one metric used by system developers to measure performance, and secondly it is a metric describing performance of the most central resource of the system; the processor.
- In a system where the execution of main application functionality is dependent on delivery of sampled data, the age of buffered data is important. A commonly used term for describing unimplemented work or unhandled data is the backlog. Its importance is due to its ability to indicate a temporary overload of the system, as well as indicate response times on for example actuation events that are causally related to input data.

- Communication-to-computation ratio is a metric that is commonly used to indicate the impact of communication in parallel systems. In the high performance computing subject area the ratio is usually calculated as the ratio between the amount of bytes sent and the computation time, i.e. amount of bytes over time [Culler99 p.132]. In our case we are more interested in a metric that indicates the performance as communication time over the time available for useful work. The definition of communication-to-computation ratio (CCR) is thus:

$$CCR = \frac{T_{Communication}}{T_{Computation}} \tag{1.1}$$

  This definition of communication-to-computation ratio resembles the performance definition of parallel architectures introduced by Tokhi et. al. [Tokhi2003 p.15]. The difference is that they introduce a ratio that is the inverse of our definition, i.e. time of computation over communication. Since we compare an architecture consisting of multiple processors with a single processor architecture with no communication overhead we would get an undefined ratio (division by zero) for the single processor system with this definition. Therefore we chose to use our definition.
- *Speedup* is a metric that describes the ratio between execution time for one system configuration over another system configuration. Usually it is measured as the execution time of the task on one processor over the execution time of the same task on multiple processors. In our approach of parallel execution of I/O system and application components speedup has not been a useful metric. We have not intended to speed up the execution of neither the I/O system, nor the application, but have had the goal to free more computing resources for the system as a whole. In that sense, we adhere more to the *scalability over problem size* definition, see section 2.1.11 and 2.1.3. In a system where each execution of application functionality needs to finish each sample period, i.e. application with real-time properties, it is more interesting to assume a constant run-time in contrast to a constant problem size. Therefore measuring speedup as defined by Amdahl was not relevant for our measurements.
- The *round-trip time of nullRPC calls* is a metric that measures the time it takes to execute a procedure call in a remote execution environment, and where the procedure does not perform any actual work. The metric is good as it quantifies the minimum communication latency for request-reply protocols, and therefore latencies that are a consequence of such requests can be given a minimum value.

4.2.2 Extensions to the Model

It is possible to extend the model to be more accurate with respect to the actual system. These extensions include:

- The Data Acquisition Phase (DAP) can be extended to include timing for tasks executing within the phase. This would include:
    - Interrupt handling routines during data reception, including operating system interrupt dispatchers (ISR) and interrupt service routines for the network interface card (NIC).
    - Timing of the protocol stack during reception of data from remote data producers.

  Figure 34 below illustrates a scenario where the DAP has been extended to show the interrupts from the devices delivering data and the execution of IOSys for each interrupt.

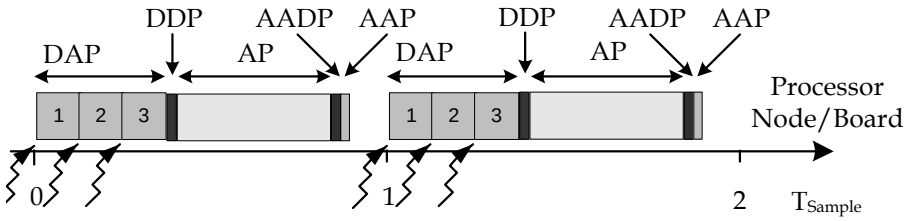


Figure 34, Extended model of DAP including interrupts and execution time per data reception.

- Execution time induced by correlation dependent on number of items, i.e. the work performed in context of the I/O System thread, in order to group (correlate) incoming data into respective data structure (DS).
- Amount of data in every delivery from remote producers. We performed a test to see the impact of this in the actual system and found that a variation from the minimal amount of data to maximum amount of data yielded a processor utilization increase of 3% (20% increase compared to the minimum). Most of this overhead is due to correlation activity in the I/O System.

4.3 Implementation and Design of IOMP

This section describes the design of a software component called **I**nput **O**utput for a **M**ulti **P**rocessor (IOMP) designed in order to exploit the concurrency between I/O system and application/client components.

4.3.1 Design

We introduce a mechanism that allows for an application remotely use the I/O System and describe the implementation of it and how it can be used together with the existing control system. The system has been called IOMP, Input Output for a MultiProcessor, and consists of the following components:

- An application side wrapper (IOMP Wrapper) for the I/O System API.
- Application Peers (IOMPPeer) on the I/O System side, one thread servicing each application thread.
- A server (IOMPServer) for servicing application thread in the creation of peers.
- A protocol (IOMPProtocol) utilizing a reliable communication protocol (an existing mechanism), enabling communication between IOMPWrapper and IOMPPeer.

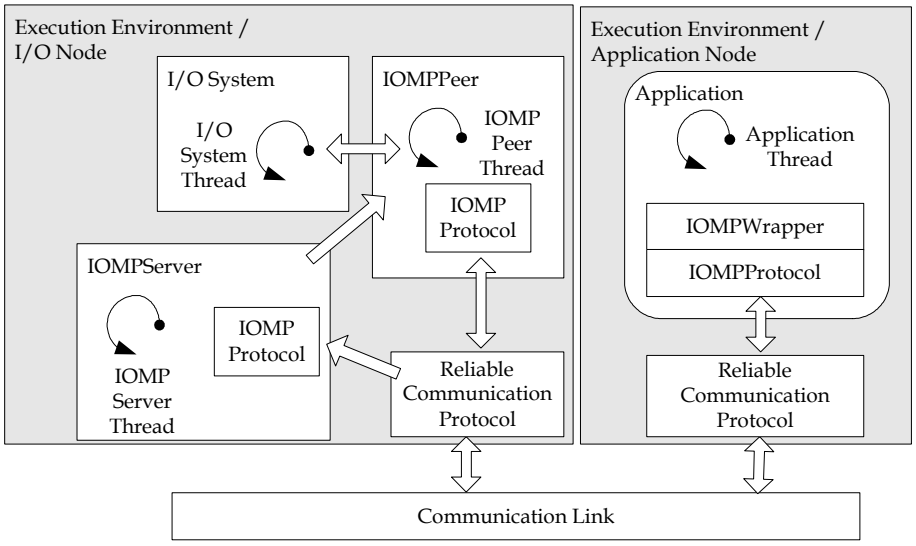


Figure 35, Overview of IOMP.

Figure 35 above illustrates the IOMP mechanism and its interaction with the I/O System. The following sections will describe the internals of each component.

4.3.2 IOMPWrapper

IOMPWrapper was designed in order for I/O System API calls to be executed in other execution environments. This includes the mapping of the interface for creating, reception and acknowledgement of data structures. The purpose is to enable easy porting of the original application code towards the IOMP system. As an example we illustrate in Figure 36 below how the application code presented in Figure 20 should be changed to enable execution in the IOMP framework. A goal with the design is to minimize the changes of the application when used in a multiprocessor environment. The user (application designer) only has to instantiate a

IOMPWrapper object in order for a peer-thread to be created on the I/O System side. The wrapper object does, in the context of the application thread, send a request to the IOMPServer on the I/O System side, which in turn creates the specified IOMPPeer. The design could be described as a tailored RPC mechanism for the specific I/O System API. We will see in Section 4.4.3 below that different optimizations need to be implemented in order to achieve optimal or near optimal performance.

```
IOMPWrapper wrapperObj;
Identifier dataStructureId;
DataStructure *dataPointer;
while( running == true ) {
    dataPointer = wrapperObj.waitForDataStructure( dataStructureId );
    executeApplicationCode( dataPointer );
    wrapperObj.acknowledgeDataStructure( dataPointer );
}
```

Figure 36, Application code for execution in IOMP.

4.3.3 IOMPPeer

The IOMPPeer thread is created when an IOMPWrapper object is instantiated and a connection between those two (the context the IOMPWrapper is executing in and the IOMPPeer) is established. If a reliable interconnect is used, it would not be necessary to establish a connection, only a uniquely addressable endpoint would be needed. The basic design of the IOMPPeer thread is that of a non-terminating loop waiting in a blocked state for requests from the IOMPWrapper. Say for example that the application invokes the waitForDataStructure() method in the wrapperObj object as illustrated in Figure 36. This call will trigger a request to be sent from the wrapper object to the peer, which in turn will invoke the waitForDataStructure() function in the I/O System.

4.3.4 IOMPProtocol

IOMPProtocol is a component that defines how requests and replies between wrappers and peers are constituted. The protocol is used by the wrapper objects and the peers in order to transparently, from the applications point of view, execute I/O System services in the other execution environment.

The protocol includes the data structures that are being sent between peers and wrapper objects, including their sizes and content. Memory references cannot be passed back to the caller (in this case the wrapper and application) by the I/O System. Such references must be converted to *remote object references* [Coulouris2001 p.144] (opaque references). An object does not exclusively represent objects in an object-oriented sense. It could also include identifiers and references to data structures located in another execution environment. For example, the I/O System

does return identifiers and memory references embedded into data structures. These references must not be accessed in the other execution environment, but they will be used by the peer when a method is invoked.

4.3.5 IOMPServer

The IOMPServer is not a necessary part for the execution of IOMP, but provides a dynamic way of binding application threads and peers together. An application can request for a peer to be created by IOMPServer via the instantiated IOMPWrapper. In the case that the user does not want to use IOMPServer, it is possible to explicitly define the number of peers and their interaction with each application thread. The communication between the application threads and the server is initiated through a predefined port.

4.4 Measurements and Results

The measurement platform that has been used resembles the model architecture illustrated in Figure 30. In order to investigate the effects of a separation of the I/O system and application components, we have created a number of system configurations which match the behavior of a data-driven periodic system. The configurations we are interested in investigating are single/multiprocessor configurations with varying I/O data loads.

4.4.1 Experimental Setup

We have experimented with actual hardware in the industrial environment, building on existing components. Two processor architectures were available and we have experimented with both, but focus has been on the more powerful Intel P3 based architecture. We wanted to vary hardware configuration where possible, compare single and multiprocessor configurations as well as vary the input stream of data into the system. The single processor setup is illustrated in Figure 37 below. Every node, including the remote data producers, is synchronized to a station clock. The station clock is connected to the nodes by optical fiber, periodically transmitting a synchronization pulse. In the extension, the station clock is usually synchronized to some other external clock, e.g. the GPS satellite system. The effect is that every node is synchronized at a high precision, a necessary condition for the publish/subscribe I/O system to operate correctly (in the cases we need to correlate or group data that have been sampled at the same instant).

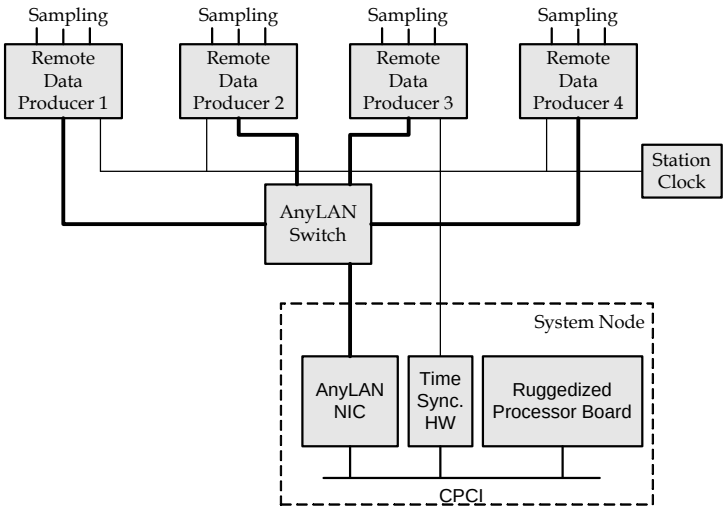


Figure 37, Single Processor Experimental Setup

Figure 38 below illustrates the two processor based experimental setup. We have named the two nodes differently based on their intended operation. The I/O node operates the I/O system including the peer part of IOMP. The application node operates the application thread.

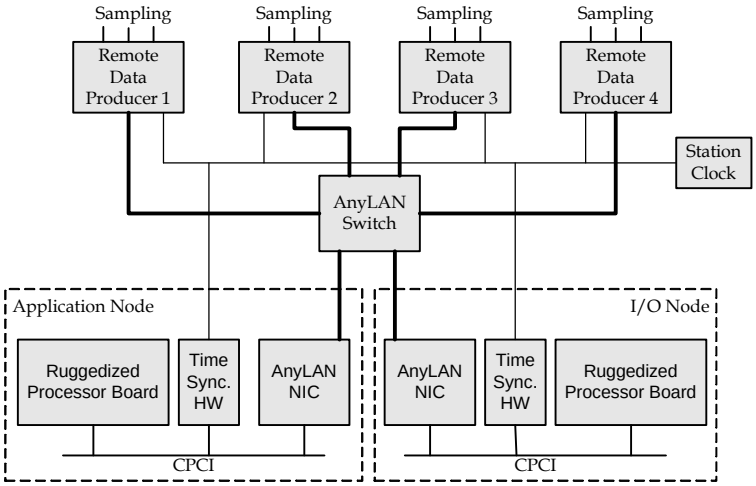


Figure 38, Two Processor Experimental Setup

All measurements were performed using the VxWorks [Windriverwww] operating system. In order to collect data and monitor the system under runtime we used

Windview [Windriverwww], a software monitoring tool provided in the Tornado IDE environment. Windview is integrated with the kernel, and we used the tool exclusively in post-mortem mode. In this mode the monitoring tool continuously collects data in a buffer without transmitting any data to the monitoring host. By using this mode we minimized the interference and the probe effect compared to using a mode where data is periodically sent to the host during runtime. Every measurement was performed similarly in order to minimize measurement errors. We used a cold start period of 400 T_{Sample}, thus allowing for the system to stabilize in terms of data deliveries and threads executing during startup. All measurement periods were thereafter 400 T_{Sample} long during which performance parameters such as processor utilization were collected. Since we were experimenting with a complete system platform we inspected every trace to ensure that no other system services, e.g. the flash control thread, were executing. The flash control thread executes periodically on low priority, but since our processor utilization measurement figures are based on the fraction of background thread execution, we had to ensure that no such threads executed during our measurement interval.

In order to more easily understand how the system is constituted we have included a picture showing the experimental setup of the P3 architecture. The I/O node is located to the left and the application node to the right. Optical fibers originating from the AnyLAN switch are connected to the NIC located on a CPCI carrier board, i.e. the time synchronization hardware board. The nodes used for experimenting with the PowerPC based architecture appears in the background.

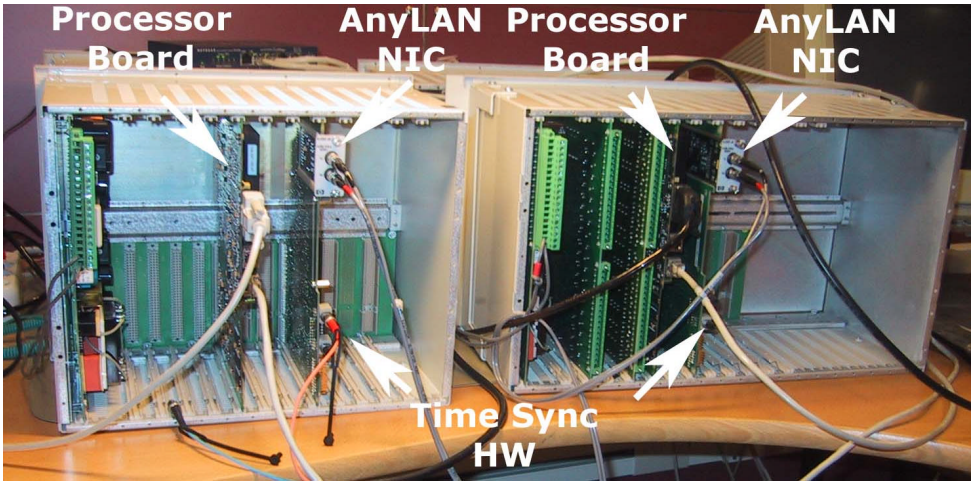


Figure 39, Picture of the system setup, I/O node to the left and application node to the right.

Different configurations are obtained through the variation of three components. These components 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:

- Single_HW1. Single processor system based on an Intel P3 architecture in a configuration as illustrated in Figure 37.
- Multi_HW2. Multiprocessor system with two Intel P3 processor nodes/boards in a configuration as illustrated in Figure 38.

Single_HW1 represents a computer architecture based on an Intel P3 clocked at a frequency of 266MHz and with a L2 cache. Multi_HW2 represents a hardware configuration in which two Intel P3 based processor boards (same as in Single_HW1) are connected a fiber optical 100VG AnyLAN switched network. The 100VG network technology [Molle96] 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, please refer to Figure 37 and Table 1. As Table 1 lists, 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 (correlates) them into data structures (DS).

I/O Configuration	Characteristics
I/O1	I/O originating from *one remote peripheral* data 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* data producers.
I/O3	I/O originating from *three remote peripheral* data producers.
I/O4	I/O originating from *four remote peripheral* data producers.

Table 1, I/O Configurations.

4.4.2 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 client thread which waits for an incoming data structure (DS) and

immediately acknowledges it. 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 Table 2 and Figure 40 below.

	I/O1		I/O2		I/O3		I/O4	
Single_HW1	**20,5**		**31,7**		**46,2**		**58,6**	
Multi_HW2	**I/O**	**App**	**I/O**	**App**	**I/O**	**App**	**I/O**	**App**
	50,6	**31,2**	**60,0**	**31,0**	**69,3**	**31,2**	**82,3**	**31,4**

Table 2, Processor utilization over the measurement interval.

We see that the communication mechanism used between the nodes does affect performance significantly, i.e. the overhead of DDP and AADP, but that this overhead is rather constant. For example, running Single_HW1 (single Intel P3 board/node) with the I/O1 configuration leads to a processor utilization of 20,5%. The Multi_HW2 multiprocessor configuration indicates that the overhead for the communication between the nodes 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 I/O node and the application node respectively. All I/O from the data producers are handled by the I/O node, hence the higher load on that board (50,6% processor utilization with the I/O1 configuration).

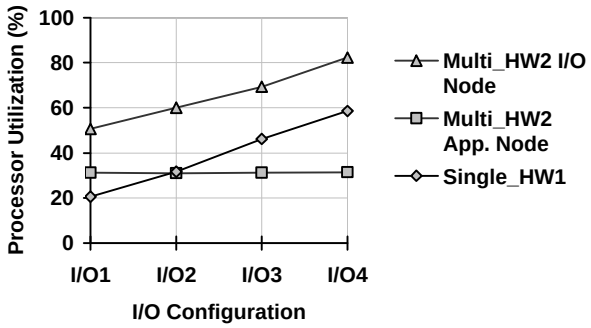


Figure 40, A plot of mean processor utilization based on the figures in Table 2.

The application thread has, in the multiprocessor case Multi_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 I/O 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 components. 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 (in absolute processor utilization terms). The relative improvement for application components is thus 66% more available processing resources.

We observe that the crossover point, where the parallel execution of IOSys and application components is beneficial, occurs when loading IOSys with configuration I/O2. Loading the system with configuration I/O1 gives a system where available processing resources for the application is less when utilizing the multiprocessor system compared to the utilizing the single processor system. At the crossover point the processing resources available in the single and multiprocessor systems is approximately equal. Not until we load the system with the I/O3 and I/O4 configurations we increase the computing resources for application functionality when utilizing the multiprocessor system.

Measurements on hardware configurations equipped with PowerPC 603 processors have been performed as well. The multiprocessor configuration of the PowerPC processor boards does not manage to consume as many data structures (DS) as are produced, even when loading the system with configuration I/O1.

4.4.3 Optimizing for Performance

We have identified three interaction approaches that between I/O System and application components. They are called the RPC approach, the PreWait Approach and the Subscribe Approach respectively.

4.4.3.1 The RPC Approach

A RPC mapping of each function call of the I/O System interface is functionally feasible for a multiprocessor system, but the approach does not perform well. As an example, please refer to Figure 36. Both the waitForData and acknowledgeData function calls are request-reply oriented. This implies that the application thread is busy waiting for the reply for each of the function calls, thus decreasing performance drastically.

```
IOMPWrapper wrapperObj;
Identifier dataStructureId;
DataStructure *dataPointer;
while( running == true ) {
    dataPointer = wrapperObj.waitForDataStructureAndAck( dataStructureId );
    executeApplicationCode( dataPointer );
}
```

Figure 41, the RPC approach with merged dataWait and ack.

We (refer to section 4.4.5) have measured nullRPC calls to have a latency of 0.36 ms in the system, which implies that approximately one third of the processing resources are used busy waiting (with a T_{Sample} period of 1ms) when this approach is used. The RPC approach illustrated in Figure 36 leads to two request-reply round-trip communication cycles. This would lead to a approximate total of 0,36 * 2 = 0,72 ms (based on nullRPC measurements in section 4.4.5) of busy wait at the application board. This would in effect nullify the benefits of parallel execution of application and IOSys components. As a first step in alleviating the negative effects of two RPC calls we merged the two waitForDataStructure and acknowledgeDataStructure function calls into the single waitForDataStructureAndAck function call. The effect of one RPC call still persists, and the disadvantage of this request-response based approach is illustrated in Figure 42 below. Data delivery is triggered by the waitForData call issued during the AADP phase.

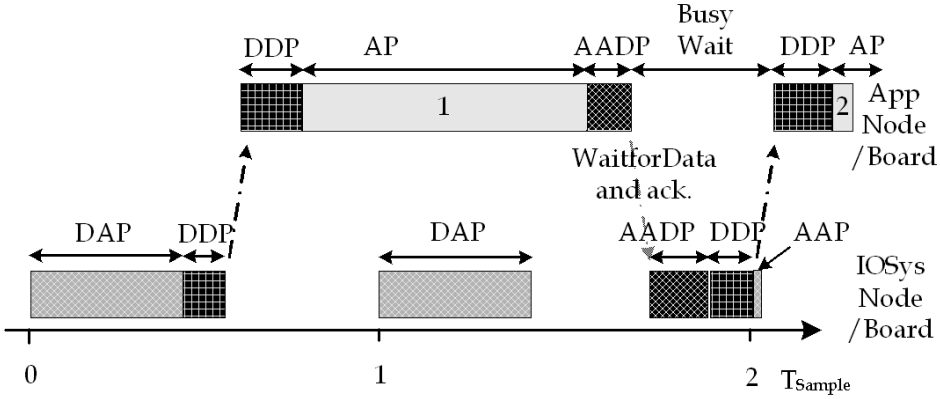


Figure 42, the disadvantage of a pure request-response based approach.

Of importance is the high negative impact of inter-processor latencies when using the consumer-initiated (receiver-initiated) communication of RPC. The goal is to utilize the application board with useful work. Therefore we needed to change the interaction between the application and the I/O System.

4.4.3.2 The PreWait Approach

An improvement over the RPC approach is to create a new interface where waitForData is issued before the AP is finished calculating on previous data structures. In that way the I/O System can operate concurrently with the application, and sending the next data structure as soon as all items are received. The buffering of data structures is handled by the communication mechanism, i.e. the receive buffers of the connection endpoint. This optimization is possible if the application is aware of that it will need next data structure at some point of time in the future and where the communication mechanism between the peers and the application or communication mechanism does support buffering of at least one message. The structure of an application we have studied does not prevent this optimization and it would have to be rewritten accordingly in order to function without the performance decrease incurred by the RPC approach:

```
IOMPWrapper wrapperObj;
Identifier dataStructureId;
DataStructure *dataPointer;

while( running == true ) {
   dataPointer = wrapperObj.acknowledgeDataAndPreWait( dataStructureId );
   executeApplicationCode( dataPointer );
   dataPointer = wrapperObj.waitForDataStructure( dataStructureId );
 }
```

Figure 43, the PreWait approach.

The acknowledgeDataAndPreWait is called at once at data reception at the application board. Next function call, waitForDataStructure, is local and progresses whenever the next data structure has arrived. The PreWait approach can be compared to what is called *precommunication* for shared memory architectures [Culler99 p.838]. With precommunication the receiver/consumer issues a request for data before it is needed, in hope that whenever it is needed data has arrived at the receiver/consumer. The purpose is to allow the sender, in our case the IOSys board, to deliver the data structure when it becomes ready. The effect is that we hide the latency of the request-reply approach. The approach was used during measurements and the execution diagram for two T_{Sample} periods would be as shown in Figure 44 below.

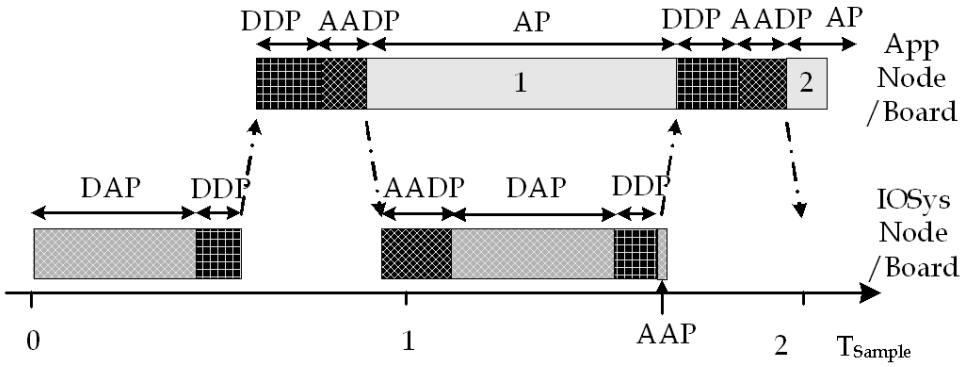


Figure 44, Execution diagram with the acknowledgeDataAndPreWait approach.

When we compare the PreWait approach to the RPC approach we see that the amount of communication overhead is the same with both approaches. Still the communication overhead is approximately 30% at each board and can be expressed as (on each board):

$$T_{CommunicationOverhead} = T_{DDP} + T_{AADP} \quad (1.2)$$

The relation determining whether a separation of I/O System and application components is profitable compared to the single processor system can be expressed as (execution time on the I/O Node):

$$T_{DAP} > T_{CommunicationOverhead} \quad (1.3)$$

and thus:

$$T_{DAP} > T_{DDP} + T_{AADP} \quad (1.4)$$

In short; this relationship states that the amount of work performed within the DAP phase must be greater than the overhead of communicating the data structures to the other node. The Communication-to-Computation (CCR) ratio as defined in section 4.2.1 is an indicator of the same relationship, i.e. the relationship between inter-processor communication overhead and the amount of useful work. A CCR of 1 indicates an equal amount of communication overhead and useful work performed during the measurement period (indicated by the crossover point). A smaller ratio indicates a system where a separation of the I/O system and application components is beneficial. Figure 45, presents the CCR ratio for the measurements performed. The single processor configuration is defined of having a CCR of zero; the time of parameter passing of the API function call is neglected. The CCR of the I/O node crosses 1 (from 1,47 to 0,89) when introducing a second I/O producer (I/O2), thus indicating that we have crossed the crossover point. We have also introduced the system wide CCR for the multiprocessor configuration indicating how much total communication overhead there exist in the system.

	I/O1			I/O2			I/O3			I/O4		
Single_HW1	**0**			**0**			**0**			**0**		
Multi_HW2	**I/O**	**App**	**Sys**	**I/O**	**App**	**Sys**	**I/O**	**App**	**Sys**	**I/O**	**App**	**Sys**
	1,47	**0,31**	**0,31**	**0,89**	**0,31**	**0,30**	**0,5**	**0,31**	**0,27**	**0,40**	**0,31**	**0,28**

Figure 45, the calculated Communication-to-Computation Ratio (CCR) on each node.

The relationship that indicates at which point the IOSys board becomes overloaded is (as measured during one T_{Sample}):

$$T_{DAP} + T_{DDP} + T_{AADP} > T_{Sample} \tag{1.5}$$

In the Intel Pentium system configuration a system overload would occur if another I/O producer is added, i.e. when subscribing to data from five data producing nodes. The conclusion is that the PreWait approach is not suitable for system configurations subscribing to data from one or five (or more) remote data producers. The PreWait approach gives approximately the same available processing resources for the application when subscribing to data from two remote data producers and shows an increase in processing resources for configurations with three and four remote data producers.

4.4.3.3 The Subscribe Approach

The overhead of the RPC and PreWait approach leads us to continue looking for more efficient approaches. The IOSys is developed with a publish/subscribe methodology in mind. Therefore it would be natural to adopt a system executing on a multiprocessor towards this methodology as well. The acknowledgement semantics of AADP of the model presented above could be removed since its purpose is to free a buffer in IOSys (in a separate execution environment). The data structure is already copied into the receiving execution environment. If the continuous actuation of peripherals is not needed, which is true for protection applications, and calculated values of the application is not remotely exported via the IOSys, we may remove the AADP phase during execution (until an event that needs actuation occurs). The execution diagram for this approach would then look like illustrated in Figure 46 below.

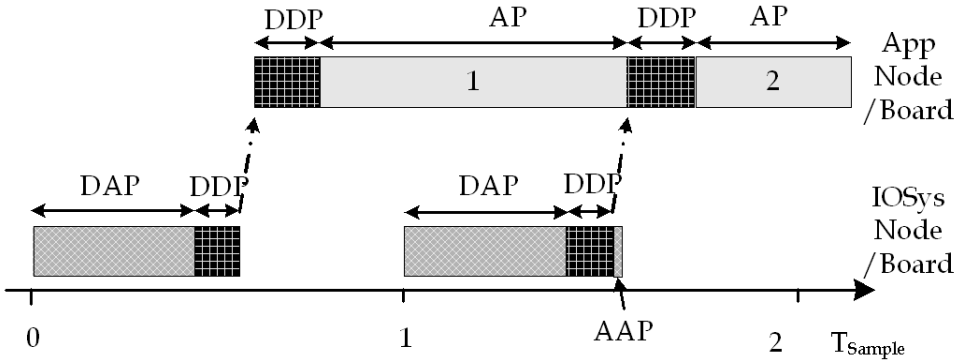


Figure 46, Execution diagram without the AADP phase.

In order to be able to estimate the effects on processor utilization for such an approach we measured the difference of receiving and sending data, i.e. the processor utilization of DDP and AADP at the IOSys and application board respectively. The result was that receiving data is approximately 50% more processor demanding compared to sending data. According to this line of reasoning we have reduced the communication overhead with 60% on the IOSys board by removing the receiving part of the AADP. We also reduced the communication overhead with 40% on the application board by removing the sending part of AADP. Figure 47 below illustrates the estimated processor utilization when removing AADP, based on the figures of Figure 40. Now, the crossover point we observed in Figure 40 has been removed, meaning that we release more processing resources for application functionality in all I/O configurations. Please observe that Figure 47 does not contain measured values, only estimated values when removing the communication overhead of AADP.

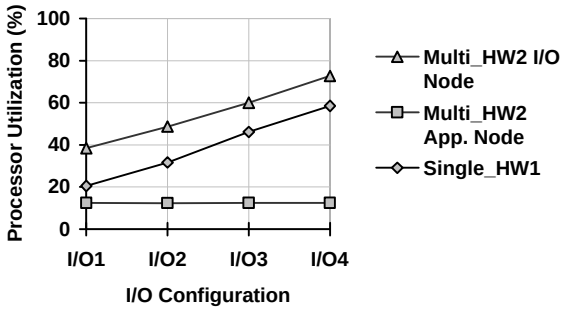


Figure 47, Estimated processor utilization with the subscribe approach.

According to the model we achieve an increase in available processing resources for application functionality when:

$$T_{DAP} < T_{CommunicationOverhead} \text{ where } T_{CommunicationOverhead} = T_{DDP} \quad (1.6)$$

which results in:

$$T_{DAP} < T_{DDP} \quad (1.7)$$

The estimated maximum performance improvement (with I/O4) in terms of less total processor utilization is approximately 47% (58,6%-12%) and the relative improvement for application components is thus 111% more available processing resources.

4.4.4 Latencies

We have already been reasoning about latencies that deprecate performance for the RPC approach in section 4.4.3, Optimizing for Performance. Now, we extend our study to include what effects on responsiveness various configurations of the system give rise to. Data responsiveness is defined as the response time of the system on sampled data. From a system-wide perspective it is defined to be the time from sampling the environment to the actuating the system under control. In our model and measurements responsiveness on data it is defined to be the time from the arrival of data to the processor board until the application is finished using the data. It is measured as the number of buffered data structures to be delivered to the application.

4.4.4.1 High Priority System Threads

In order to estimate 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 that need to be handled instantly.

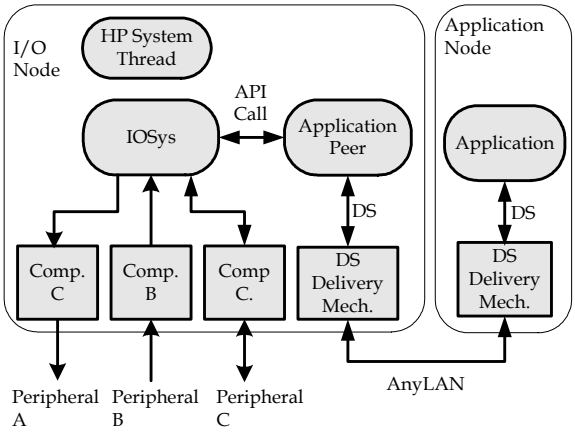


Figure 48, Introduction of a high priority system thread.

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.

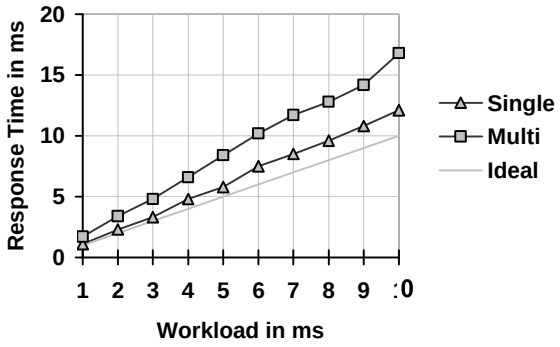


Figure 49, Response time of system thread vs. its undisturbed workload.

One-millisecond workloads up to 10-millisecond workloads were created and run on both hardware configurations Single_HW1 and Multi_HW2. In the single processor configuration (Single_HW1) we see that we have a continuously increasing execution time for the system thread compared with the ideal undisturbed execution. The disturbance, from the system thread's point of view that leads to its execution time increase, is the data stream arriving continuously from the data producers at a rate of T_{Sample}. In the multiprocessor configuration (Multi_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 nodes, which is higher prioritized than the system thread, leads however, to an even longer execution time for the system thread (see Figure 49).

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 50 below.

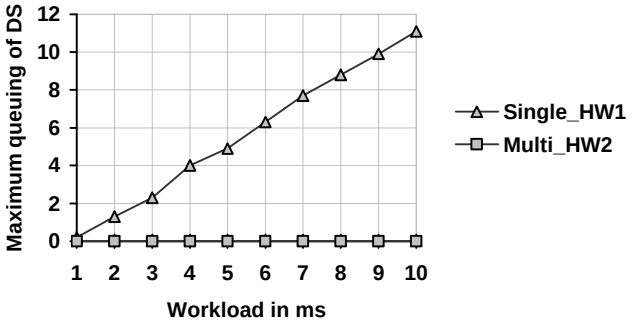


Figure 50, Maximum data structure buffer utilization.

We see that the multiprocessor configuration (Multi_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. While this synthetic test is somewhat unfair in since it introduces starvation, it can be concluded that for a multiprocessor design (such as Multi_HW2) a tradeoff can be made between the responsiveness of the application vs. the responsiveness of other system threads. The test also shows that unexpected delays not accounted for during design and erroneous behavior of system threads in the I/O node does not affect the execution of the application. Thus the robustness of the system has been improved.

4.4.4.2 Network Related Latencies and Contention

Of importance is the timely delivery of data over the interconnect. The case-study system uses an AnyLAN network and while the AnyLAN network is predictable regarding traffic scheduling the links are not secure. This meaning that packets transmitted over the network can be lost. It is up to the transport protocol to secure the delivery of data, detecting a situation where packets are lost and retransmitting them. During a measurement period we detected an incurred latency of 1293µs from the time the data structure was delivered until the response from the application arrived.

During measurements we found another source of latency originating from contention between the AADP and the DAP. Figure 44 does actually illustrate such a situation where AADP (on the IOSys board) delays the DAP and ultimately the AP. This situation occurs whenever the application board wants to execute AADP concurrently with the DAP on the IOSys board. For most applications this incurred latency (in the range of 150-200µs) is not a serious problem, but an effect the application designer should be aware of.

Actuation events should in most cases be higher prioritized than DAP transmissions on the link. In the current implementation only normal prioritized AnyLAN [HP95] frame transmissions are used. Therefore AADP could use frame transmission with high priority in situations where critical actuations need to have precedence over data acquisition.

4.4.5 Synchronous RPC

The multiprocessor design of Multi_HW2 assumes a clean interface between the application and the IOSys. In the multiprocessor case, all function calls in the single processor architecture design must be mapped to inter-processor synchronous remote procedure calls (RPC). If the function call expects a result of any kind from the I/O board/node, execution of that application thread is stalled. Function calls that do not need a result could be exchanged with asynchronous RPC calls. RPC calls can be resource demanding and can have large round-trip times. In our system, in which application threads are executed every millisecond (T_{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 is that RPC calls between the nodes 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.

4.5 PCI-Bus Communication Performance

In order to estimate performance when utilizing a PCI-bus based multiprocessor system, we implemented a communication mechanism that enables tasks to exchange messages over the Compact PCI bus. The motivation for this work was to analyze and identify threats to efficient communication in backplane bus-based computer architecture that is suitable as the multiprocessor platform in a separated I/O System and application approach. This section and Appendix A describe the implementation of the communication mechanism, called PCICom, and presents measurement results as well as a discussion regarding the potential use of it together with the IOMP approach.

4.5.1 Overview

The software implementation of PCICom consists of a C++ class that utilizes a shared memory area. It is possible to use PCICom in shared memory architectures where it is possible to issue processor loads and stores to memory located externally to the processors own execution environment. A passive CompactPCI-backplane bus architecture, with a system and a non-system board connected to the bus is an example of such an architecture.

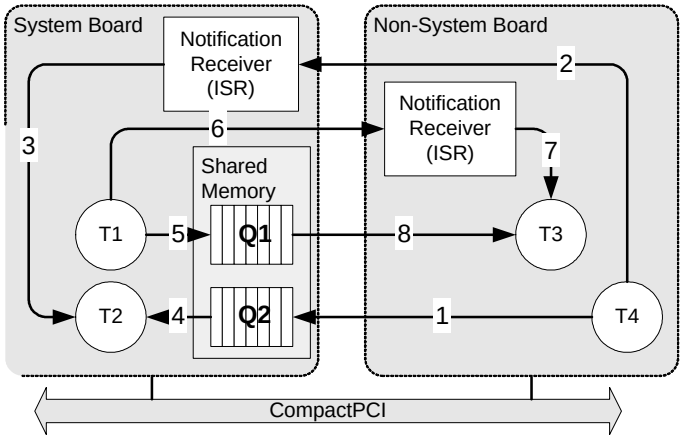


Figure 51, Overview of PCICom.

The difference between a system board and a non-system board is that the system board has a transparent PCI-PCI bridge and the arbiter for the CompactPCI bus (PCI-PCI bridges not shown for simplicity in Figure 51). A non-system board does have a PCI-PCI bridge with configurable address translating windows, enabling filtering of CompactPCI memory transactions not intended for the non-system board. In our tests, both the System and the Non-System boards were Pentium III boards running at 850MHz (CT7 single processor boards produced by SBS Technologies [SBSwww]).

Each processor board has a notification receiver, responding to interrupt events from other boards and the notification receiver is represented by an Interrupt Service Routine (ISR). The ISR will be run by an interrupt triggered by transactions to the doorbell register in the Intel 21554 PCI-PCI bridge [21554www] residing on the non-system board. Figure 51 also illustrates how two pairs of threads exchange messages over the PCI-bus. A receiving thread, for example T2, will take a semaphore and block forever until the release of the semaphore. The sending thread, T4, writes (1) a message into next available message entry in queue number two, Q2. Thereafter a write to the doorbell register (2) on the Intel 21554 bridge invokes the notification receiver ISR on the system board. The ISR will in turn release the semaphore (3) that has been taken by T2 and T2 can thereafter read (4) the message in Q2. The same

procedure is repeated whenever T1 wants to send messages to T3, but in this case the receiver (T3) reads (8) data from shared memory located on the system board.

4.5.2 Memory Layout

The first test with PCICom were using a shared memory area on the CompactPCI system slot board. VxWorks by default takes control all of the available primary memory, which cannot be allowed when defining a shared memory area such as the one needed by PCICom.

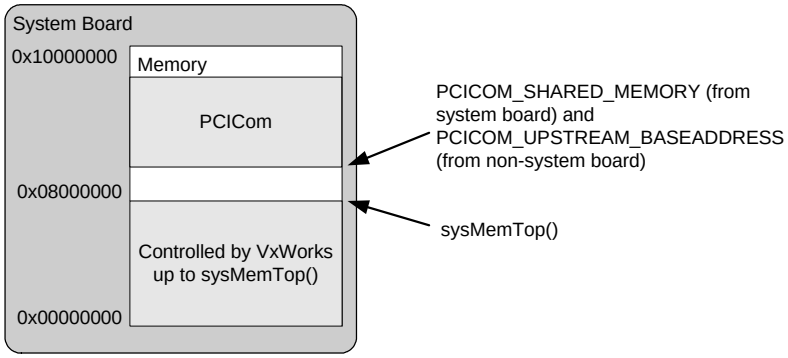


Figure 52, Memory Layout on the System Board

A definition (USER_RESERVED_MEM) in the Board Support Package (BSP) for VxWorks was altered so that VxWorks would not use memory allocated by PCICom (this limit can be read with the sysMemTop() function). This definition changes the amount of memory that will be controlled by VxWorks from the top of the address space and downward. The only information that has to be provided by the user to find the shared memory area on either processor board (system and non-system) are the respective base addresses. On the system board the base address is provided with the PCICOM_SHARED_MEMORY definition and on the non-system board it is defined by PCICOM_UPSTREAM_BASEADDRESS. Figure 52 illustrates how the memory of the message queues are placed in memory with respect to the operating system. From PCICOM_SHARED_MEMORY and towards higher addresses, message queues with index 0, 1, 2 up to N are located.

4.5.3 Performance Measurements

A number of measurements have been performed to study the behavior of communication using the PCI-bus. Each test was made on two SBS CT7 [SBSwww] CompactPCI boards running at 850MHz. The first test performed was a uni-directional communication from the system to non-system board. With the help of the spy tool available in VxWorks, the average execution percentage of the background task could be observed. A clear tendency could be seen at once; the non-

system board was being heavily loaded and the processor utilization figures are presented in Figure 53 below.

Message Size	Frequency	System Board Processor Utilization	Non-System Board Processor Utilization
10 byte	1000Hz	1%	3%
100 byte	1000Hz	1%	7%
1000 byte	1000Hz	2%	59%

Figure 53, Processor Utilization with uni-directional communication from system to non-system board.

It is clear that performance is suffering from reads from the non-system board over the PCI-hierarchy. Even at moderate 1000 bytes sent at a frequency of 1000Hz (~1MB/s) the non-system board was suffering under heavy load.

In a second test two message queues were created and we stressed the system with a ping-pong test. One thread on each board exchanges messages in an interleaved fashion, without any delay. Complete copying from the memory area of each thread to the other is performed in the process. This test yielded results that are presented in Figure 54 below. The SendReceive threads represent user threads performing ping-pong message passing. Bkgnd is a thread running at a low priority, i.e. the idle thread. Kernel is the fraction of time spent in the kernel, e.g. scheduling, and Int represents the fraction of time spent at interrupt level.

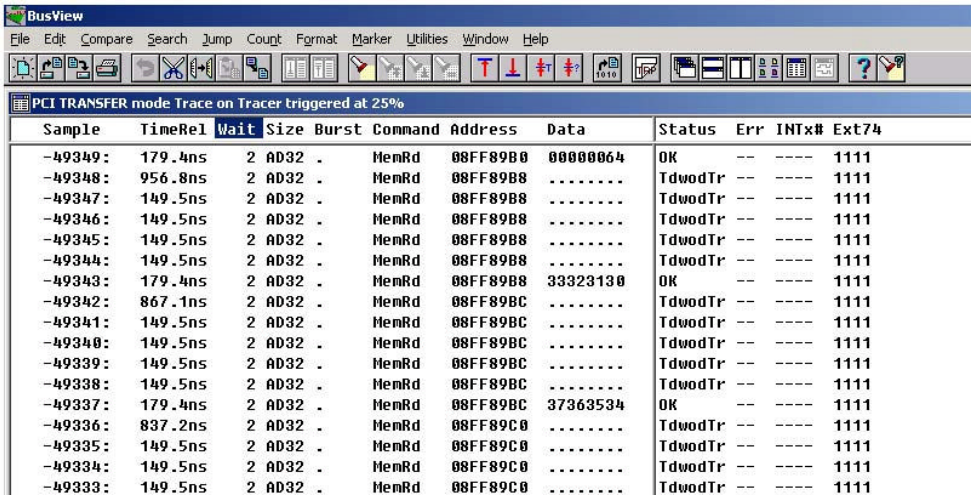
Message Size	Messages per second and direction	Load on individual threads or modules on System Board		Load on individual threads or modules on Non-System Boards	
1000 byte	2200	SendReceive	1%	SendReceive	95%
		Bkgnd	96%	Bkgnd	2%
		Kernel	0%	Kernel	0%
		Int	1%	Int	1%
100 byte	15200	SendReceive	4%	SendReceive	75%
		Bkgnd	87%	Bkgnd	13%
		Kernel	2%	Kernel	2%
		Int	5%	Int	7%
10 byte	34000	SendReceive	9%	SendReceive	51%
		Bkgnd	70%	Bkgnd	28%
		Kernel	6%	Kernel	5%
		Int	14%	Int	14%
0 byte	45400	SendReceive	11%	SendReceive	38%
		Bkgnd	60%	Bkgnd	35%
		Kernel	6%	Kernel	6%
		Int	20%	Int	20%

Figure 54, Performance of Ping-Pong test between system and non-system boards.

Two conclusions can be drawn from these figures. Firstly, as has been noted earlier, the reads from the non-system slot are disastrous for performance and is reflected in the load on the SendReceive thread on the non-system board. The SendReceive thread is responsible for reading the message, and in the case of large messages the execution of this thread is occupying the whole processor. As the messages become smaller and smaller, the significance of the reads become lesser, while the significance of context switches and interrupt handling becomes more significant. The Kernel module and interrupt module figures in Figure 54 are good indicators on context switch overhead and interrupt overhead. The performance of pure notification, where the message size is zero, is quite impressive. The boards are able to handle 45400 interrupts, including one semaphore release and one context switch to the receiving thread per second.

Another interesting figure to note is the load on SendReceive on the system board in the case of large messages of 1000 bytes where the CPU load is as low as 1%. This is due good memory locality, i.e. PCICom copies to a memory with primary memory on the board itself.

The significance and latency of bus transactions on the PCI-bus was analyzed with a bus-analyzer from VMETRO [VMETROwww], and a timing sample is presented in Figure 55 below. The figures are taken from a test with a message size of 100 bytes and the analyzer was placed on the CompactPCI-bus. Bus access latencies, the amount of time that expires from the moment a bus master requests the use of the PCI-bus until it completes the first data transfer of the transaction, consist of three components; arbitration latency, bus acquisition latency and target latency [Shanley99]. In a bus-hierarchy as in the case of a local PCI-bus and a CompactPCI-bus, these latencies will accumulate.



Sample	TimeRel	Wait	Size	Burst	Command	Address	Data	Status	Err	INTx#	Ext74
-49349:	179.4ns	2	AD32	.	MemRd	08FF89B0	00000064	OK	--	----	1111
-49348:	956.8ns	2	AD32	.	MemRd	08FF89B8	....	TdwodTr	--	----	1111
-49347:	149.5ns	2	AD32	.	MemRd	08FF89B8	....	TdwodTr	--	----	1111
-49346:	149.5ns	2	AD32	.	MemRd	08FF89B8	....	TdwodTr	--	----	1111
-49345:	149.5ns	2	AD32	.	MemRd	08FF89B8	....	TdwodTr	--	----	1111
-49344:	149.5ns	2	AD32	.	MemRd	08FF89B8	....	TdwodTr	--	----	1111
-49343:	179.4ns	2	AD32	.	MemRd	08FF89B8	33323130	OK	--	----	1111
-49342:	867.1ns	2	AD32	.	MemRd	08FF89BC	....	TdwodTr	--	----	1111
-49341:	149.5ns	2	AD32	.	MemRd	08FF89BC	....	TdwodTr	--	----	1111
-49340:	149.5ns	2	AD32	.	MemRd	08FF89BC	....	TdwodTr	--	----	1111
-49339:	149.5ns	2	AD32	.	MemRd	08FF89BC	....	TdwodTr	--	----	1111
-49338:	149.5ns	2	AD32	.	MemRd	08FF89BC	....	TdwodTr	--	----	1111
-49337:	179.4ns	2	AD32	.	MemRd	08FF89BC	37363534	OK	--	----	1111
-49336:	837.2ns	2	AD32	.	MemRd	08FF89C0	....	TdwodTr	--	----	1111
-49335:	149.5ns	2	AD32	.	MemRd	08FF89C0	....	TdwodTr	--	----	1111
-49334:	149.5ns	2	AD32	.	MemRd	08FF89C0	....	TdwodTr	--	----	1111
-49333:	149.5ns	2	AD32	.	MemRd	08FF89C0	....	TdwodTr	--	----	1111

Figure 55, Timing on reads from the non-system board to the system board.

Each address that will result in a 4 byte transfer from system memory on the system board to the non-system boards, will also take a long time to complete. The first transfer shown in Figure 55 takes as much as 956,8ns + 149,5ns + 149,5ns + 149,5ns + 149,5ns 179,4 = **1734,2**ns. This represents 1743,2ns / 29,9ns = **58** PCI clock cycles. During this time the processor on the non-system board is busy waiting for the result of the transaction. A large amount of processing power is thus wasted.

Figure 56 below shows the figures from our test where the non-system board issues writes over the CompactPCI-bus to the shared memory area on the system board. Timing is much better compared to issuing reads from the non-system board, but not optimal. The first write marked in the figure takes 149,5ns (five PCI-clock cycles) and the next, due to burst effects, allocates only one PCI-bus cycle (29,9ns).

The positive effects of using write-buffers is a well-known fact [Patterson98 p.554], but the tests also show the necessity of know the limitations of PCI-bus transactions in a bus-hierarchy. Memory references issued transparently transferred over the PCI-bus hierarchy to memory locations with bad locality can sacrifice many clock cycles

for the processor. The correct utilization of bursts must be exploited as well as correct use of the 256 bytes of posted write and 256 bytes read buffers in each direction in the Intel 21554 PCI-PCI bridge. It is interesting to note that latencies through a PCI-to-PCI bridge has been reported to be in the order of tens of nanoseconds to milliseconds [Chamé98].

-49183:	119.6ns	1 AD32 .	MemWri	08FF0CC0	39383736	OK	--	----	1111
-49182:	149.5ns	1 AD32 Start	MemWri	08FF0CC4	33323130	OK	--	----	1111
-49181:	29.9ns	. AD32 B	MemWri	08FF0CC8	37363534	OK	--	----	1111
-49180:	29.9ns	. AD32 B	MemWri	08FF0CCC	31303938	OK	--	----	1111
-49179:	119.6ns	1 AD32 .	MemWri	08FF0CD0	35343332	OK	--	----	1111
-49178:	119.6ns	1 AD32 Start	MemWri	08FF0CD4	39383736	OK	--	----	1111
-49177:	29.9ns	. AD32 B	MemWri	08FF0CD8	33323130	OK	--	----	1111
-49176:	29.9ns	. AD32 B	MemWri	08FF0CDC	37363534	OK	--	----	1111
-49175:	119.6ns	1 AD32 .	MemWri	08FF0CE0	31303938	OK	--	----	1111
-49174:	149.5ns	1 AD32 Start	MemWri	08FF0CE4	35343332	OK	--	----	1111
-49173:	29.9ns	. AD32 B	MemWri	08FF0CE8	39383736	OK	--	----	1111
-49172:	29.9ns	. AD32 B	MemWri	08FF0CEC	33323130	OK	--	----	1111
-49171:	119.6ns	1 AD32 .	MemWri	08FF0CF0	37363534	OK	--	----	1111

Figure 56, Impact of writes over the CompactPCI-bus and the bus hierarchy.

4.5.4 Message Passing Utilizing Writes over the PCI-bus

PCICom was modified to support multiple shared memory areas, enabling message passing with only writes over the bus hierarchy as illustrated in Figure 57 below. The timing and performance to message-passing was greatly improved as can be seen in the figures shown in Figure 58.

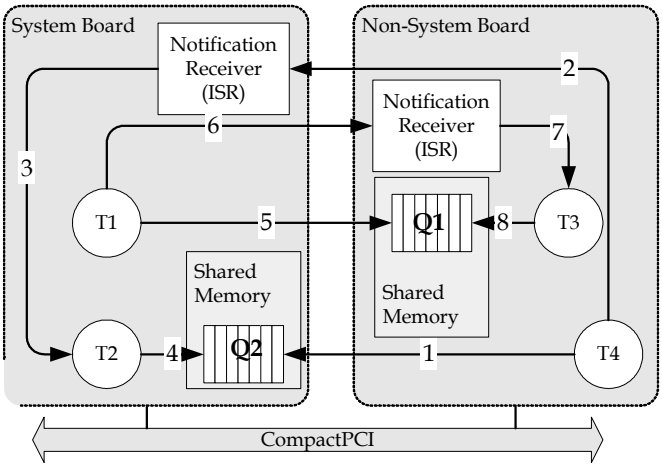


Figure 57, PCICom modified to support multiple shared memory areas.

The throughput reaches its maximum when 1000 byte large messages are sent. In that case 14450 messages * 1000 byte ≈ 14,5 MB/s is transmitted in each direction. The theoretical maximum throughput of the PCI-bus is 32bit * 33MHz ≈ 132MB/s

which is much better that the figure presented by PCICom (14,5MB/s * 2 = 29 MB/s). PCICom is thus approximately 132/29 ≈ 4,5 times slower than theoretically achievable. Theoretical throughput is only achievable when achieving long bursts on the bus. Sustained throughput of PCI-based communication has been reported to be in the range of 70-80 MB/s [PLXwww] and lower figures are common for most practical situations, i.e. a system with multiple competing devices. DMA performance for devices located on the local PCI bus have shown to be able to achieve 80-90 MB/s or even 130MB/s in some cases and as low as 68 MB/s in other cases [Moll97].

Message Size	Messages per second and direction	Processor Utilization of individual threads or modules on System Board		Processor Utilization of individual threads or modules on Non-System Boards	
1000 byte	14450	SendReceive	43%	SendReceive	45%
		Bkgnd	48%	Bkgnd	46%
		Kernel	2%	Kernel	2%
		Int	6%	Int	6%
100 byte	36750	SendReceive	31%	SendReceive	26%
		Bkgnd	46%	Bkgnd	50%
		Kernel	5%	Kernel	5%
		Int	16%	Int	16%
10 byte	43150	SendReceive	27%	SendReceive	23%
		Bkgnd	47%	Bkgnd	53%
		Kernel	6%	Kernel	6%
		Int	18%	Int	17%
0 byte	45400	SendReceive	25%	SendReceive	23%
		Bkgnd	48%	Bkgnd	50%
		Kernel	7%	Kernel	7%
		Int	19%	Int	19%

Figure 58, PCICom issuing only writes over the PCI-bus.

Based on the measurements described above the conclusion when implementing a communication mechanism where the processor is responsible for the data exchange is:

> *In order to avoid costly request-reply data exchange in a PCI-bus hierarchy, issue processor writes from the sender processor instead of reads issued from the receiving processor.*

4.5.5 PCI-bus as the Communication Mechanism in the Case-Study System

The figures presented in Figure 58 above can give us indications on latencies for the round-trip times for request-reply protocols, e.g. RPC calls. Based on the amount of messages sent in each direction each second (from column 2 in Figure 58) we can calculate approximate timing for round-trip times (the uni-directional latency times two):

Message size (bytes)	Estimated Round-Trip Latency (µs)
0 (nullRPC)	44
10	46
100	54
1000	138

Figure 59, Round-Trip latency.

In section 4.4.5, synchronous RPC, we measured round-trip latency for function calls over the 100VG network. It is interesting to compare the difference between the round-trip latencies for the distributed approach and the CompactPCI bus-based approach. They are both executing on Pentium 3 architectures, but the processor boards were clocked approximately 3 times faster (850MHz vs. 266 MHz). In the distributed system using IOMP, the round-trip latency was 360µs. Compare this to 44µs for the estimated round-trip latencies for nullRPC over the CompactPCI hierarchy we get a 360µs/44µs ~ 8 times ratio. Assuming a performance ratio of 3 between the two architectures we may estimate a lower limit of the ratio to 8/3 ~ 2,7. These figures are approximate but can indicate if synchronous RPC is feasible for a PCI bus based approach. 44µs corresponds to 4,4% of the sample period, T_{Sample}, which is a substantial amount of time. The conclusion is that RPC is still inefficient as a programming model when using a Compact PCI bus-based architecture and where system periodicity is high.

In our case-study system the transmission of data structures (DS) during the DDP and AADP phases for the distributed system accounts for approximately 18% communication overhead on the receiving side (as discussed in section 4.4.3.3) and 12% communication overhead on the sending side. With a properly tuned PCI communication mechanism, the load of transmitting 1000 bytes at 1000Hz will yield a 2% system load (according to Figure 53) and transmitting 100bytes in 1000Hz will yield a system load of 1%. The overhead of delivering Data Structures (DS) between processor boards in such a system is minimal.

5 Conclusion and Future Work

Industrial systems designed for execution in a single processor environment are not necessarily able to scale on multiple processors effortlessly. Scalability is dependent on parameters such as the available hardware architecture alternatives, the available operating systems for those architectures, the available communication system as well as the ability of middleware and application level functionality to scale. Given an industrial system with an intelligent I/O system, it is possible to improve performance by executing the I/O system and application components in parallel.

5.1 Research Questions Revisited

This section summarizes our conclusions and the answers found to the research questions stated in section 1.2. The first question was formulated as:

1. "Which time-critical software system components utilize the most processor resources and do they show a large amount of mutual concurrency?"

The answer to this question was found by studying source code as well as executing systems, monitoring the operating system, platform and applications. The system consists of the major components illustrated in Figure 19, each with its own software structure and run-time properties. Each component exhibits a certain execution pattern depending on how it is triggered by the arrival of data (the I/O system and protection system of applications) and other external events (such as the synchronization mechanism that is triggered by synchronization pulses). Other system components are triggered by internal events such as periodic triggers (the watchdog task) or data delivered by other system components (such as system event functionality and logging activity). The results of system analysis showed a system execution pattern that periodically begins with data acquisition followed by application protection functionality. In order to describe and reason about system performance we developed the model described in section 4.1 and performed the execution analysis in section 4.2 (also described in [Enblom2003]).

Why did we choose to study the separation of I/O system and application components? Firstly, because the two components showed a large amount of mutual concurrency (in a pipelined producer-consumer fashion) and secondly, because of the combination of the time-critical and resource-demanding parameters. Data arrives periodically and must be handled by the system immediately. Applications must be adapted to finish before the next data arrival, i.e. be tuned to fit in the time-slot available. Other system functionality such as flash disk recovery tasks or logging tasks execute in the remaining slack available in the system (the system is tuned to show an average processor utilization rate not exceeding a certain amount less than 100%). Leaving the uniprocessor system intact and executing logging functionality

on a different processor is an alternative to the parallel execution approach. This functionality is however subordinated to (given lower priority than) the time-critical data acquisition and application functionality and this approach cannot help in freeing more processing resources, for neither the I/O system nor the application. The approach could on the other hand free more processing resources for the logging task, but this was not of great interest in our system. Thus, the choice of separating I/O system and application components was a combination of the time-critical and resource- demanding nature of the system architecture. Components that exhibit a large amount of mutual concurrency and a relatively small amount of data exchange are suitable candidates for parallel execution. The I/O system and application have both of these properties.

In addition to the mutual concurrency that exists among I/O system and application components, we utilized the correlation functionality of the existing I/O system to offload the application node. The correlation functionality can be used with the following effects on a separation of I/O system and application components:

- The I/O node running the I/O system shields the disturbances of interrupts, communication protocol processing, as well as correlation functionality from the execution of application functionality on the application node.
- The concept of data structure delivery upon completion gives rise to only one notification event, we shield the application node from a varying load of data producers.
- Correlation is necessary in shielding application nodes from the varying amount of I/O deliveries. If the correlation functionality was not present a separation of I/O system and application components would yield as many notifications to the application node as to the I/O system node. Much of the benefit of separation would then be lost.

2. "Given the existing software architecture, can the components requiring the utilization of a considerable proportion of available processor resources scale on a distributed or bus-based multiprocessor architecture and how does the suggested parallel execution of the concurrent system components affect performance?"

Our studies have shown that, given a sampling frequency of 1000Hz and off-the-shelf hardware components, the system can scale on a two-processor distributed-system architecture by introducing an I/O system node. The inter-node communication overhead however is quite processor demanding. Yet, the approach can free computing resources for system configurations that need data from multiple remote data producers. The case study showed approximately 30% processor utilization on each processor for the communication between the processor boards. In cases in which the I/O system utilizes the processor less than the processor utilization overhead of inter-processor communication with the distributed solution,

no processor resources are freed for the application. If, on the other hand, the I/O system processor utilization becomes greater than the distributed overhead, the available processing resources have increased for the application (as discussed in 4.4.3.2). Performance measurements with a message-based communication mechanism on a CompactPCI system indicate that overhead for the communication between the processors can decrease significantly for such architectures. When the interaction pattern between I/O system node/board and the application node/board is retained, CompactPCI architecture can lower the communication overhead to the range of a few percent (see section 4.5).

The results have been obtained from a distributed test platform based on communication over an AnyLAN network, 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 [Enblom2001]. An example of NCC-NUMA architecture would be, for example, a Compact PCI (CPCI) backplane 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 timing for RPC calls would be much lower due to lower bus latencies and the less processor-demanding message-passing communication mechanism. In this context it is relevant to identify the similarities between the distributed and a backplane bus-based architecture. They include:

- All resources in both architectures, such as interrupt controllers, local memory and peripheral buses are local to each processor node/board. The only means of communicating between processors is over the interconnect (backplane bus or network).
- Moving tasks/threads from one processor environment to another is performance-demanding and employed seldom in real-time systems. In the backplane bus-based architecture however, the system can access data in a global memory area. Therefore the same task, if the code is located on multiple boards, can potentially execute on multiple processor boards. Yet, we have seen (in section 4.5.5) that accessing memory on another processor board can be a threat to performance due to latencies. A task/thread usually obtains resources such as file-pointers, memory allocated from the heap as well as the stack, from its local processor. Migrating from one processor to another in a backplane bus-based architecture includes moving the complete state of the task/thread from one processor to another. An approach in which the stack and the heap are placed completely in a global shared memory will suffer from the same performance limitations as the study in 4.5.5 suggests. Thus, efficient partitioning of system components, such as locating I/O system and application

components on separate boards/nodes will be the most fundamental factor in both the distributed architecture and backplane bus-based architecture.

Differences do however exist between the distributed architecture and the backplane bus-based architecture:

- The inter-node bandwidth differs by a factor of approximately 10 (~132MS/s compared to ~12,5MB/s)
- In the distributed architecture, inter-processor communication is limited to the sending of packets over the network. In the backplane bus-based architecture processor load/stores can access remote memory. Sending and receiving data packets over the network involves a respectable amount of code to execute on both the sender and receiver side. The startup time of sending and receiving packets is high compared with the amount of data sent. In the backplane bus-based architecture, the ratio between startup time and the time per sent packet is smaller.

Conclusions and results regarding processor utilization include:

- Given the interconnects available in this study (100VG-AnyLAN and a backplane PCI-bus) the fine-granular execution of data acquisition and application execution give rise to relatively high communication/computation ratios. The communication mechanism used between the boards should require the least possible resources in terms of processor utilization. The communication mechanism used for inter-processor communication should ideally be less demanding of processor resources than the communication mechanism used to receive data from remote I/O boards during DAP.
- Parallel execution of I/O system and application components has a number of positive and negative effects on the system. The main positive effect is that more processing resources are released for application components for systems that subscribe to data from many remote nodes.
- Due to less contention with the I/O system activities, application functionality gives a stable amount of processing resources with the approach using multiple processors.
- Powerful hardware architectures are more suitable for use in the distributed architecture approach because they demand less processor resources for communicating between the nodes (we compared the PowerPC 603 and the Intel Pentium3 architectures). Even more powerful future architectures will improve this further.
- The amount of data delivered between the I/O node and application node has little effect on processor utilization. The application node shows the same processor utilization figures (between 31,0% and 31,4% when using configuration I/O1 to I/O4).

- Instruction level parallelism (ILP) is exploited by the processor architecture. We have exploited existing thread level parallelism (TLP) in the form of pipelining (producer/consumer parallelism).
- We have shown that, provided that the interaction between system components on the different processors is kept to a minimum, the distributed system can reach 27,2% less total processor utilization for the application node (with the RPC approach). This represents 66% more processor resources available for application components. The estimated increase in performance (with respect to reduced total processor utilization) reaches 46% when using configuration I/O4 (adhering to the subscribe approach in section 4.4.3.3). This represents approximately 111% more processor resources available for application components. For the substation automation system which has been the target of this work, a distributed multiprocessor system solution can therefore increase the performance capability of the application components.
- Other system functionality, such as system threads with high priority suffer loss of performance in terms of increased response-times.
- The semantics and communication pattern at the interface between the I/O system and application components can affect performance significantly. We identified three interaction patterns; the RPC, the PreWait and the Subscribe patterns (called approaches in chapter 4.4.3). The only practically feasible patterns are the PreWait and the Subscribe patterns where the Subscribe pattern will introduce the least communication overhead.
- The latency-hiding technique of precommunication used in the PreWait approach enables the system to tolerate the high round-trip latencies of a request-reply protocol. The RPC approach cannot exploit any latency-hiding techniques unless the system is able to find concurrent work when waiting for the reply. If multiple threads are available for execution, the application node could exploit this available thread level parallelism (TLP), but unfortunately the critical path within applications is contained within the context of one thread.
- With both the RPC approach and the PreWait approach, the communication is receiver-initiated, with the inherent round-trip latency as a result. The subscribe approach on the other hand is sender-initiated, inherently eliminating the need for a two-way protocol.
- In both the RPC and the Prewait approach, issuing a complete two-way request-reply cycle for each function call is a waste of processing resources. Merging multiple independent function calls into a single call can therefore be advantageous. We did this with the waitForDataStructureAndAck() function call, thereby eliminating one RPC round-trip.

Conclusions and results regarding data latencies, response time to important system events as well as inter-processor round-trip latencies include:

- Inter-processor synchronous RPC, i.e. client/server semantics between processor boards, is performance-costly semantics and should be avoided.
- By exploiting the inherent correlating functionality of the I/O system we are able to shield the numerous interrupts and data receptions occurring on the I/O node from the components of the application. For example, configuring the system with four remote data producers (I/O4) leads to four interrupts from the AnyLAN NIC each T_{Sample} period. Correspondingly, only one interrupt originating from the AnyLAN NIC occurs each T_{Sample} period on the application node.
- By communicating between two nodes in a network we have introduced another source of latency in comparison with the single processor system. In section 4.4.4.2 we describe the issues of lost packets in the network and contention between I/O system communication traffic and inter-node communication traffic. For most systems, lost packets introduced an acceptable delay while contention was the origin of a minimal delay.

5.2 A Condensed Summary

The most important conclusions from this work are:

- By introducing an I/O Node as shown in Figure 60 below can shield the application from the disturbing communication activities, thus enabling a more predictable execution of application components.
- Utilizing two nodes has been measured to give up to 66% more processing resources for the application components.
- The interaction pattern between the I/O node and the application node is of significance. An analysis showed that relying on a subscribe approach between the nodes is appropriate.

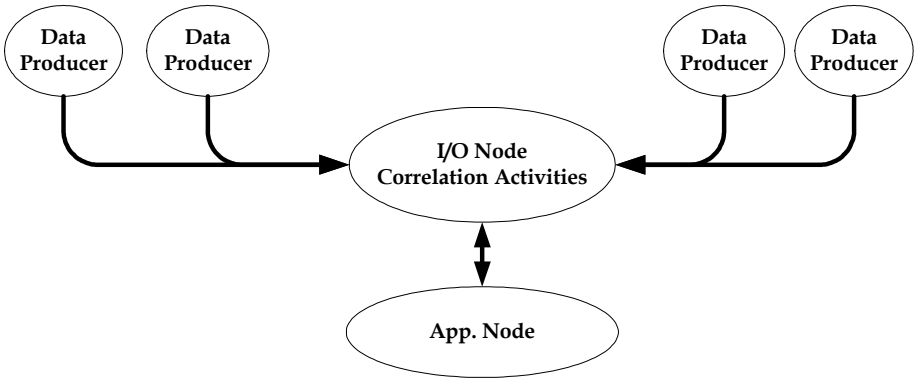


Figure 60, Conceptual view of the I/O Node approach.

5.3 General Applicability

The focal point of this work was the case-study system and the problems of parallel execution for that particular system. It is therefore hazardous to draw any conclusions general applicable to all industrial automation systems. However, systems with the same general design and structure as the case study in this thesis are potential candidates for the improvement of performance by parallel execution. System properties such as periodic data arrival, data acquisition from remote nodes via a network, concurrent data acquisition and application phases, data driven application execution and limitations on power dissipation from processor boards are important system characteristics. In brief, we have exploited pipelined concurrency already present in a real-time industrial automation system by introducing an I/O system proxy node in order to improve the system performance. Performance was possible through the use of the correlation functionality of the existing I/O system. This enables applications to specify the data that is it interested in to the I/O system. Systems that have correlation functionality (on for example timestamps) and have the same execution patterns as the model in this thesis describes, can potentially improve performance for application functionality by parallel execution of I/O system and application components. There are many issues of interest for further study in this field and a number of such issues will be outlined in the next section.

5.4 Future Work

Many parameters are involved during the execution of a real-time control system, and the system designer needs to have a broad holistic system view and to understand how parameters and properties cooperate. The demand for more performance for new functionality is continuously increasing as is the demand for supporting high rates of I/O sensor sampling. Multiprocessor solutions need to be considered even in systems which have been designed solely for a single processor environment. We outline below two areas and some other general directions for future study we consider to be of interest:

1. **Hardware architectures and hardware accelerators**. We have in chapter 3 already discussed possible hardware architecture alternatives intended to improve the performance of the system. We now briefly discuss investigations in other directions which could be undertaken.

    1.1. The parallel system explored in this thesis statically partitions components onto different processor boards and into different processing environments. Alternative multiprocessor hardware architectures would be, for example, Symmetric Multiprocessors (SMP). The main benefit of such hardware architectures is that they provide a shared view of memory for all the processors and achieve coherency among processors by means of hardware.

Since all the 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 arises in this context as do the price/performance ratio and power dissipation issues. Are SMP systems a credible alternative and are they feasible in real-time and embedded control environments? These are questions that need to be answered.

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

1.3. Mixed configurations of cheaper/less powerful processor boards and more expensive/more powerful processor boards can lead to a more balanced workload for different I/O configurations. This is one parameter in a future price/performance study that could also include the study of different processor architectures, performance issues regarding cache sizes and different system memory footprints.

1.4. Operating system accelerators. Research in this field needs to advance into accelerating more functionality, traditionally implemented in operating systems. If the operating system functionality such as clock tick administration, scheduling and IPC is a major bottleneck in the specific system, an existing operating system accelerator might improve performance. Areas of study which would be of interest include memory allocation accelerators and hardware support for accessing performance-demanding devices such as network adapters. But, we are certain that the sequence of system design and development should focus on system components scaling on multiple processing elements before identifying the magnitude of the operating system overhead.

1.5. Hardware acceleration of specific application algorithms and logic may benefit from hardware acceleration. We determined that concurrency of application protection functionality was inadequate for exploitation within the architectures we studied, but the use of a dedicated hardware accelerator for FFT algorithms could still be a performance-improving approach.

2. **Communication and I/O system middleware for industrial automation systems**.

2.1. Data flow patterns in industrial automation systems and distributed real-time systems are a challenging topic. We have investigated the separation of application components from a communication middleware with specific publish/subscribe characteristics. Such characteristics of the flow of data are

common in sampling systems, but companies usually develop their own methodology for handling such systems. The Data Distribution Service (DDS) for distributed real-time systems [DDSRTS2003] has been issued recently by the OMG group (see section 2.4.4.6). This shows how immature the interfaces for such systems have been and that the specific needs of various target systems should be compared with what such a standard offers.

2.2. The need for an increased sample rate is not acute but could give more precision in applications and thus an advantage over competitors. If we should increase the sample rate to 2000Hz, how does this affect the system? Will application components be able to execute once every sample period or must other approaches be exploited? Which I/O system characteristics are then needed to fulfill system requirements?

2.3. Further study of the need to utilize correlation as a means to offload the clients/applications from I/O system activity would give valuable insight into predictability, efficiency and performance characteristics.

3. **Other directions**.

3.1. With the introduction of multiple processors, the software developer is facing more system complexity. Attention needs to be focused on the design of multiprocessor software which achieves adequate performance and scales well. Identifying and designing clean interfaces between I/O middleware and applications for existing products can pave the way for parallel execution and improved performance. Component based design that avoids request-reply semantics would be an interesting subject for study for maintainability, reuse and performance reasons.

3.2. In this work 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.

3.3. Future work could also include a study of the parallelization of other system components such as parallel databases. Even though our impression is that the granularity of this functionality is too fine to scale well in a bus-based and distributed processing environment.

The future work topics above have not been listed by priority. We think that the topics regarding middleware for industrial systems are the most important to study, especially the use of correlation functionality to enable the execution of an I/O node. The TAO [Schmidt98] project team has incorporated correlation functionality in their framework but do not seem to evaluate the possibilities of correlation to introduce an I/O node or proxy.

6 References

We have chosen to split references into three categories; published, unpublished and other material. Published material includes papers, journal papers as well as books. Unpublished material includes references to material available on the web. Other references include research material not published at a conference, e.g. technical reports, as well as articles published in newspapers or journals without peer reviewing.

6.1 Published References

[Adams96] Jay K. Adams, Donald E. Thomas, "The Design of Mixed Hardware/Software Systems", 33rd Design Automation Conference, p 515-520, 1996.

[Adve96] Sarita V. Adve, Kourosh Gharachorloo, "Shared Memory Consistency Models: A Tutorial", IEEE Computer, Vol. 29, No. 12, pp. 66-76, 1996.

[Albertsson2000] Lars Albertsson, Peter S. Magnusson, " Using Complete System Simulation for Temporal Debugging of General Purpose Operating Systems and Workloads", International workshop on modeling, analysis and simulation of computer and telecommunication systems, 2000.

[Alverson90] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, Burton Smith, "The Tera Computer System", Proceedings of the1990 International Conference on Supercomputing, pp 1-6, 1990.

[Amdahl67] Gene M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", *AFIPS conference proceedings / American federation of information processing societies*, pp. 483-485, 1967.

[Anderson95] T.E Anderson, D.E Culler, D. Patterson, "A case for NOW (Networks of Workstations)", *IEEE Micro, Vol 15 Issue 1*, pp 54 – 64, February 1995.

[Anderson97] Don Anderson, "USB system architecture", *MindShare Inc*., ISBN 0-201-46137-4, 1997.

[Andrews2000] Gregory R. Andrews, "Foundations of Multithreaded, Parallell and Distributed Programming", Addison Wesley Longman Inc., ISBN 0-201-35752-6, 2000.

[Banerjea96] Anindo Banerjea, Domenico Ferrari, Bruce A. Mah, Mark Moran, Dinesh C. Verma, Hui Zhang, "The Tenet Real-Time Protocol Suite: Design Implementation, and Experiences", ACM Transactions on Networking, 1996.

[Banks2001] Jerry Banks, John S. Carson II, Barry L. Nelson, David M. Nicol, "Discrete-Event System Simulation", Prentice-Hall Inc., ISBN 0-13-088702-1, 2001.

[Bershad90] Brian N. Bershad, Thomas E. Andersson, Edward D. Lazowska, Henry M. Levy, "Lightweight Remote Procedure Call", ACM Transactions on Computer Systems (TOCS), Vol. 8, Issue 1, pp 37-55, February 1990.

[Björkman93] Mats Björkman, "Architectures for High Perfromance Computing", *Dissertation at the Department of Computer Systems*, Uppsala University, ISSN 0283-0574, September 1993.

[Bloomer92] John Bloomer, "Power Programming with RPC", O'Reilly & Associates Inc., ISBN 0-937175-77-3, 1992.

[Boasson93] Maarten Boasson, "Control Systems Software", IEEE Transactions on Automatic Control, Vol. 38, No. 7, July 1993.

[Boasson96] Maarten Boasson, “Subscription as a Model for the Architecture of Embedded Systems”, Proceedings of Engineering of Complex Computer Systems, 1996.
[Bovet2003] Daniel P. Bovet, Marco Cesati, “Understanding the Linux Kernel”, Second Edition, O´Reilly & Associates, Inc., ISBN 0-596-00213-0, 2003.
[Boxer94] Aaron Boxer, Dan Malek, “UltraSMART: A Scalable Multiprocessor Architecture for Real-Time”, Proceedings of the Sixth Euromicro Workshop on Real-Time Systems, p. 118 –123, 1994.
[Byrd98] Gregory Thurman Byrd, “Communication Mechanisms in Shared Memory Multiprocessors”, A Dissertation of Stanford University, August 1998.
[Chamé98] Al Chamé, “PCI Bus In High Speed I/O Systems Applications”, Proceedings of the IEEE Aerospace Conference, Vol. 4, p. 505-514, 1998.
[Chatterjee97] Saurav Chatterjee, Kevin Bradley, Jose Madriz, James A. Colquist, Jay Strosnider, “SEW: A Toolset For Design and Analysis of Distributed Real-Time Systems”, Proceedings of the third IEEE Real-Time Technology and Applications Symposium, pp 72-77, 1997.
[Chong98] Frederic T. Chong, Rajeev Barua, Fredrik Dahlgren, Johan D. Kubiatowicz, Anant Agarval, “The Sensitivity of Communication Mechanisms to Bandwidth and Latency”, *Proceedings of the 4th Internationsl Symposium on High Performance Computer Architecture*, 1998.
[Coulouris94] George Coulouris, Jean Dollimore, Tim Kindberg, “Distributed Systems Concepts and Design”, *Addison-Wesley Longman Inc.*, ISBN 0-201-62433-8, 1994.
[Coulouris01] George Coulouris, Jean Dollimore, Tim Kindberg, “Distributed Systems Concepts and Design”, *Addison-Wesley Longman Inc*., ISBN 0-201-61918-0, 2001.
[Culler99] David E. Culler and Jaswinder Pal Singh, “Parallel Computer Architecture, a hardware/software approach”, Morgan Kaufmann Publishers Inc, ISBN 1-55860-343-3, 1999.
[Dasdan99] Ali Dasdan, “Timing Analysis of Embedded Real-Time Systems”, Dissertation at University of Illinois at Urbana-Champaign, 1999.
[Davies96] T. Davies, “Protection of industrial power systems”, 2nd edition, Newnes, Reed Educational and professional publishing Ltd., ISBN 0-7506-2662-3, 1996.
[Dawson99] W.K. Dawson and R.W. Dobinson, “*Buses and bus standards*”, Computer Standards & Interfaces 20, 201-224, Elsevier Science B.V., 1999.
[Dijkstra65] E.W. Dijkstra, “Solution of a Problem in Concurrent Programming Control”, Communications of the ACM, Vol.8, No.9, 1965.
[Enblom2001] 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.
[Enblom2003] Leif Enblom, “Parallel Execution of I/O System and Application Functionality”, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA2003)*, CSREA Press, Las Vegas, USA, June 2003.
[Flynn96] Michael J. Flynn, “Parallel Architectures”, ACM Computing Surveys, No.1 March 1996.
[Furunäs2000] Johan Furunäs, "Benchmarking of a Real-Time System that utilizes a Booster", *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA2000)*, Las Vegas, USA, June 2000.

[Furunäs2001] Johan Furunäs, “Interprocess Communication Utilising Special Purpose Hardware”, *Licentiate thesis 01/42, Mälardalen University Press and Department of Information Technology*, Uppsala University, December 2001.

[Gait86] Jason Gait, “A Probe Effect in Concurrent Programs”, Software-Practice and Experience, Vol. 16(3), pp 225-233, March 1986.

[Grama2003] Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, “Introduction to Parallel Computing”, Pearson Education Limited, ISBN 0-201-64865-2, 2003.

[Gupta91] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, Wolf-Dietrich Weber, “Comparative Evaluation of Latency Reducing and Tolerating Techniques”, ACM Computer Architecture News, SIGARCH, vol. 19, nr. 3, p. 254-265, 1991.

[Gustafson88] John L. Gustafson , "Reevaluating Amdahl's law”, Communications of the ACM, Vol. 31, Issue 5, p 532-533, May 1988.

[Hammond2000] Hammond, L., Hubbert, B.A., Siu, M., Prabhu, M.K., Chen, M., Olukolun, K., “The Stanford Hydra CMP” *IEEE Micro*, Vol. 20 Issue 2, pp 71 – 84, March-April 2000.

[Heuring97] Vincent P. Heuring, Harry F. Jordan, “Computer Systems Design And Architecture”, Addison Wesley Longman Inc, ISBN 0-8053-4330-X, 1997.

[Hwang98] Kai Hwang, Xu Zhiwei, „Scalable Parallel Computing“, WCB/McGraw-Hill, ISBN 0-07-031798-4, 1998.

[Häggander2001] Daniel Häggander, “Software Design Conflicts, Maintainability versus Performance and Availability”, Doctorial Dissertation, Blekinge Institute of Technology, ISBN 91-7295-004-8, 2001.

[Johnson92] Kirk L. Johnson, “The Impact of Communication Locality on Large-Scale Multiprocessor Performance”, *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[Jordan2003] Harry F. Jordan, Gita Alaghband, “Fundamentals of Parallel Processing”, Pearson Education, Inc., Prentice Hall, ISBN 0-13-901158-7, 2003.

[Kaiserswerth93] Matthias Kaiserswerth, "The Parallel Protocol Engine", IEEE/ACM Transactions on Networking, Vol. 1, Nr. 6, p. 650-663, 1993.

[Kakulavarapu99] P. Kakulavarapu, J. Amaral, ”A survey of load balancers in modern multi-threading systems", In Proc. of the 11th Symp. on Computer Architecture and High Performance Computing, pp 10 - 16, 1999.

[Lam92] M. S. Lam, R. P. Wilson, "Limits of control flow on parallelism", Nineteenth International Symposium on Computer Architecture", ACM and IEEE Computer Society, p. 46-57, 1992.

[Law2000] Averill M. Law, David W. Kelton, “Simulation modeling and analysis”, 3rd edition, McGraw-Hill Book co., ISBN 0-07-116537-1, 2000.

[Lawson92] Harold W. Lawson, “Parallel Processing in Industrial Real-Time Applications”, *Prentice-Hall, Inc.*, ISBN 0-13-654518-1, 1992.

[Lewis98] R.W. Lewis, “Programming industrial control systems using IEC 1131-3”, Revised edition, The Institution of Electrical Engineers, ISBN 0-85296-950-3, 1998.

[Lilja2000] David J. Lilja, “Measuring Computer Performance, A practitioners guide”, *Cambridge University Press*, ISBN 0-521-64105-5, 2000.

[Lindh98] Lennart Lindh, Johan Stärner, Johan Furunäs, Joakim Adomat, Mohammed El Shobaki, “Hardware Accelerator for Single and Multiprocessor Real-Time Operating

Systems", Seventh Swedish Workshop on Computer Systems Architecture, Göteborg, Sweden, June 1998.

[Liu97] Liu Guangtian, Aloysius K. Mok, "An Event Service Framework for Distributed Real-Time Systems", IEEE Workshop on Middleware for Distributed Real-Time Systems and Services, December 1997.

[Lo97] Jack L. Lo, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, Dean M. Tullsen, "Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading", ACM Transactions on Computer Systems, Vol. 15, No.3, p. 322-354, August 1997.

[Loosley98] Chris Loosley, Frank Douglas, " High-Performance Client/Server, A Guide to Building and Managing Robust Distributed Systems", *Wiley Computer Publishing, John Wiley & Sons*, inc., ISBN 0-471-16269-8, 1998.

[Mauro2001] Jim Mauro, Richard McDougall, "Solaris Internals, Core Kernel Components", Sun Microsystems Press, Prentice Hall, ISBN 0-13-022496-0, 2001.

[Maxwell99] Scott Maxwell, "Linux Core Kernel Commentary", *The Coriolis Group*, ISBN 1-57610-469-9, 1999.

[May2001] John M. May, "Parallel I/O for High-Performance Computing", Morgan Kaufmann Publishers, ISBN 1-55860-664-5, 2001.

[Molesky90] L. Molesky, K. Ramamritham, C. Shen, J. Stankovic, G. Zlokapa, "Implementing a predictable real-time multiprocessor kernel - the Spring kernel", Proceedings of the 7th IEEE Workshop on Real-Time Operating Systems and Software, 1990.

[Moll97] Laurent Moll, Mark Shand, "Systems Performance Measurement on PCI Pamette", IEEE Symposium on FPGAs for Custom Computing Machines, 1997.

[Molle96] Molle M, Watson G, "100Base-T/IEEE 802.12/packet switching", IEEE Communications Magazine, p 64-73, Volume 34, Issue 8, Aug 1996.

[Mowry98] Todd C. Mowry, Charles Q. Chan, Adley K. W. Lo, "Comparative Evaluation of Latency Tolerance Techniques for Software Distributed Shared Memory", Proceedings of the 4th IEEE symposium on high-perfromance computer architectures, p. 300-311, 1998.

[Mullender90] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, "Amoeba, A Distributed Operating System for the 1990s", *Computer Volume: 23 Issue: 5*, May 1990.

[Narayanan93] P. J. Narayanan, "Processor autonomy on SIMD architectures", *Proceedings of the 1993 international conference on Supercomputing, ACM Press*, ISBN 0-89791-600-X, p 127-136, 1993.

[Nichols96] Bradford Nichols, Dick Buttlar and Jacqueline Proulx Farrel, "Pthreads Programming", O'Reilly & Associates, Inc., 1996.

[Pancake96] Cherri M. Pancake, "Is parallelism for You?", *IEEE Computational Science & Engineering*, pp 18-37, Summer 1996.

[Patterson80] David. A. Patterson, David R. Ditzel, "The case for the reduced instruction set computer", Computer Architecture News, Vol. 8, No. 6, p. 25-33, October 1980.

[Patterson98] David A. Patterson, John L. Hennessy, "Computer Organization & Design The Hardware/Software Interface", Second Edition, Morgan Kaufmann Publishers Inc., ISBN 1-55860-491-X, 1998.

[Perkovic99] Dejan Perkovic, Peter J. Keleher, "Responsiveness without Interrupts", *Proceedings of the 1999 international conference on Supercomputing*, pp 101 – 108, 1999.

[Quarterman93] John S. Quarterman, Susanne Wilhelm, “UNIX, POSIX, and Open Systems. The Open Standards Puzzle”, Addison-Wesley Publishing Company Inc., ISBN 0-201-52772-3, 1993.

[Quinn94] Michael J. Quinn, “Parallel Computing Theory and Practice”, *McGraw-Hill Inc*, ISBN 0-07-113800-5, 1994.

[Rajkumar95] Ragunathan Rajkumar, Mike Gagliardi, Lui Sha, “The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation”, IEEE, Proceedings of the Real-Time Technology and Applications Symposium, 1995.

[Rosenblum97] Mendel Rosenblum, Edouard Bugnion, Scott Devine, Sephen A. Herrod, “Using the SimOS Machine Simulator to Study Complex Computer Systems”, ACM Transactions on Modeling and Computer Simulation, Vol. 7, No. 1. p 78-103, January 1997.

[Sha90] Lui Sha, Ragunathan Rajkumar, John P. Lehoczky, “Priority Inheritance Protocols: An approach to Real-Time Synchronisation”, *IEEE Transactions on Computers*, Volume 39 Issue 9, pp 1175 –1185, 1990.

[Shah2001] Hemal V. Shah, Dave B. Minturn, Annie Foong, Gary L. McAlpine, Rajesh S. Madukkarumukumana, Greg J. Regnier, “CSP: A Novel System Architecture for Scalable Internet and Communication Services", 3rd USENIX Symposium on Internet Technologies and Systems (USITS’01), p.61 - 72, 2001.

[Shanley99] Tom Shanley, Don Andersson, “PCI System Architecture”, Third Edition, Mindshare, Addison Wesley Longman Inc., ISBN 0-201-30974-2, 1999.

[Shin91] K.G. Shin, “HARTS: a distributed real-time architecture”, *IEEE Computer*, Vol. 24, Issue 5, May 1991.

[Shobaki2002] Mohammed El Shobaki, “On-Chip Monitoring of Single- and Multiprocessor Hardware Real-Time Operating Systems”, In Proceedings of the 8th International Conference on Real-Time Computing Systems and Applications (RTCSA), March 2002.

[Smith85] James E. Smith, Andrew R. Pleszkun, “Implementation of precise Interrupts in Pipelined Processors”, *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pp 291-299, 1985.

[Solomon98] David A. Solomon, “Inside Windows NT Second Edition”, *Microsoft Press*, ISBN 1-57231-677-2, 1998.

[Spatscheck98] Oliver Spatscheck, Jorgen S. Hansen, Johan H. Harman, Larry L. Peterson, “Optimizing TCP Forwarder Performance”, IEEE/ACM Transactions on Networking”, Vol. 8, Nr. 2, p.146-157, 1998.

[Stallings2000] William Stallings, “Computer Organization and Architecture”, *Prentice Hall Inc*, ISBN 0-13-081294-3, 2000.

[Stone90] Harold S. Stone, “High-Performance Computer Architecture, Second Edition”, Addison Wesley, ISBN 0-201-51377-3, 1990.

[Storey96] Neil Storey, “Safety Critical Computer Systems”, 2nd Ed, *Prentice Hall*, ISBN 0-201-42787-7, 1996.

[Suzuki2003] M. Suzuki, H. Kobayashi, N. Yamasaki, Y. Anzai, “A Task Migration Scheme for High Performance Real-Time Cluster System”, Proceedings of 18th International Conference on Computers and Their Applications, pp. 228-231, March 2003.

[Takagi88] Hideaki Takagi, “Queuing Analysis of Polling Models”, ACM Computing Surveys 20, pp 5-281, March 1988.

[Tanenbaum92] Andrew S. Tanenbaum, “Modern Operating Systems”, Prentice Hall International Inc., ISBN 0-13-595752-4, 1992.
[Tanenbaum96] Andrew S. Tanenbaum, “Computer Networks”, Third Edition, Prentice Hall International Inc., ISBN 0-13-394248-1, 1992.
[Tokhi2003] M.O. Tokhi, M.A. Hossain, M.H. Shaheed, “Parallel Computing for Real-Time Signal Processing and Control”, Springer-Verlag London, ISBN 1-85233-599-8, 2003.
[Tullsen96] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm, “Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor”, *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp 191-202, May 22-24, 1996.
[Vassiliadis2001] 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 (PDPTA’01), Las Vegas, 2001.
[Walker95] Wade Walker, Harvey G. Cragon, “Interrupt Processing in Concurrent Processors”, *IEEE Computer*, pp 36-46, June 1995.
[Wilkinson99] Barry Wilkinson, Michael Allen, “Parallel Programming, Techniques and Applications Using Networked Workstations and Parallel Computers”, *Prentice Hall Inc*., ISBN 0-13-671710-1, 1999.
[Wolf2001] Wayne Wolf, “Computers as components: principles of embedded computing system design”, *Academic Press*, ISBN 1-55860-693-9, 2001.
[Wu2001] Youfeng Wu, Dong-Yuan Chen, Jesse Fang, “Better Exploration of Region-Level Locality with Integrated Computation Reuse and Value Prediction”, International Symposium on Computer Architecture (ISCA-28), 2001.
[Yates97] 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.

6.2 Unpublished References, Mainly WWW Material

[100VGwww] Unofficial 100VG AnyLAN FAQ, http://www.io.com/~richardr/vg/vgfaq.htm.
[10GEAwww] Introduction to TCP/IP Offload Engine (TOE), http://www.10gea.org/SP0502IntroToTOE_F.pdf.
[21554www] Intel(R) 21554 Non-transparent PCI-to-PCI Bridge Documentation, http://www.intel.com/design/bridge/docs/21554_documentation.htm
[Altivecwww] AltiVec.org, http://www.altivec.org
[Analogwww] Analog Devices, http://www.analog.com/
[Annapmicrowww] Wildcard™ http://www.annapmicro.com/wildcard2.html
[Arshamwww] Systems Simulation, http://ubmail.ubalt.edu/~harsham/simulation/sim.htm
[Beowulfwww] Beowulf Project Homepage, http://www.beowulf.org
[Beowulfwww2] Beowulf mailing list FAQ, http://www.canonical.org/~kragen/beowulf-faq.txt
[BVMwww] Programmable Data Concentrator/Protocol Converter, BVM Limited, http://www.bvmltd.co.uk/system3500pr.html
[Concurrentwww] Concurrent Computer Corporation, http://www.ccur.com/realtime/
[EETimeswww] “Throughput fuels data revolution”, eetimes.com 2001-04-20, http://www.eetimes.com/story/OEG20010420S0061

[EETimeswww2] Emerging Standards enable distributed real-time systems", http://www.eetimes.com/in_focus/embedded_systems/OEG20020307S0074
[FreeBSDwww] FreeBSD, http://www.freebsd.org.
[FreeBSDSMPwww] The FreeBSD SMP Project, http://people.freebsd.org/~fsmp/SMP/SMP.html
[GMSwww] "Embedded Applications Benefit from Mainstream SMP Computing Architectures", http://www.gms4vme.com/gmshome/products/vxworks.html
[IPCASwww] IPCAS ipDaco Datat Concentrator, http://www.ipcas.de/english/download/ipdaco.pdf
[Intelwww] Intel Outlines Future Technology Directions Press Release, http://www.intel.com/pressroom/archive/releases/20010828corp_a.htm
[Intelwww2] IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide. Can be found at http://www.intel.com
[Intelwww3] Intel Network Processors, http://developer.intel.com/design/network/products/npfamily/
[Intelwww4] MultiProcessor Specification Version 1.4, May 1997, http://developer.intel.com
[Kontronwww] Kontron Webpage, http://www.kontron.se/
[Linuxhqwww] The Official Linux Kernel Homepage, www.linuxhq.com
[LinuxSMPwww] Linux SMP HOWTO, http://www.linuxdoc.org/HOWTO/SMP-HOWTO.html
[Motorolawww] Motorola Website, www.motorola.com
[MPIwww] http://www.mpi-forum.org
[QNXwww] QNX Whitepaper, http://www.qnx.com
[Pardo-Castellote2001], Gerardo Pardo-Castellote, Stefaan Sonck Thiebaut, Mark Hamilton, Henry Choi, " Real-Time Publish-Subscribe Protocol for Distributed Real-Time Applications", Instrument Society of America, http://www.rti.com, 2001.
[PCIwww] PCI-SIG Homepage, http://www.pcisig.com/home
[PCIwww2] http://www.yourvote.com/pci/
[PLXwww] „PCI to Local Bridge Performance Study", http://openarm.linuxforum.net/doc/nanoPCI/docs/bus21bridge.pdf
[Profibuswww] Profibus International, http://www.profibus.com/
[RapidIOwww] RapidIO Trade Association, http://www.rapidio.org/home
[SBSwww] SBS Technologies Inc., http://www.sbs.com/
[SUNwww] SUN Documentation, http://docs.sun.com
[Synergywww] Synergy Microsystems, http://www.synergymicro.com/
[Tensilicawww] NEC's TCP/IP Offload Engine, http://www.tensilica.com/html/pr_2003_05_12.html
[Virtutecwww] Virtutec Simics, http://www.simics.com/
[Vitessewww] Samuel J. Barnett, Mark R. Fauber, "Network Processors, Uncovering Architectureal Approaches for High-Speed Packet Processing", www.vitesse.com
[VMETROwww] VMETRO Webpage, http://www.vmetro.com
[Windriverwww] Windriver Webpage, http://www.windriver.com

6.3 Other References

[Elektronik] "Listig arkitektur ger falska processorer", in Elektroniktidningen nr 13, page 22, 14 September 2001.

[Elson98] Jeremy Elson, “A Survey of Process Migration Schemes”, http://citeseer.nj.nec.com/elson98survey.html, 1998.
[Enblom2001_02] Leif Enblom, “Performance of a Communication Mechanism over the PCI-bus”, Mälardalen University Technical Report Id 0346, November 2001.
[Enblom2003_02] Leif Enblom, “Parallel Computer System Components”, MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-89/2003-1-SE, November 2003.
[Eugster2001] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, ”The Many Faces Publish/Subscribe", Technical Report DSC ID: citeseer.nj.nec.com/442483.html, 2001.
[Ghosh94] Kaushik Ghosh, Bodhisattwa Mukherjee, Karsten Schwan, "A Survey of Real-Time Operating Systems -- Draft", Technical Report, http://citeseer.nj.nec.com/35747.html
[Harrison97] Timothy H. Harrison, Carlos O´Ryan, David L. Levine, Douglas C. Schmidt, “The Design and Performance of a Real-Time CORBA Event Service”, IEEE Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems, 1997.
[Hill2002] Jonathan C. Hill, John C. Knight, Aaron M. Crickenberger, Richard Honhart, “Publish and Subscribe with Reply”, Technical Report CS-2002-32, Department of Computer Science, University of Virginia, 2002.
[HP95] “100VG-AnyLAN Technology Guide”, Hewlett-Packard Company, Publication Number 5963-6588, August 1995.
[Hu97] Lei Hu, Ian Gorton, “Performance Evaluation for Parallel Systems: A Survey”, Technical Report UNSW-CSE-TR-9707, Department of Computer Systems, University of NSW, Sydney, Australia, October 1997.
[Intel8237A] “8237A High Performance Programmable DMA Controller (8237A-5)”, Order Number: 231466-005, September 1993.
[Kandlur91] D. Kandlur, K. G. Shin, ”Design of a communication subsystem for HARTS”, Technical Report, CSE-TR-109-91, U. of Michigan, 1991.
[Motorola94] Motorola, IBM Microelectronics, “PowerPC Microprocessor Family: The Programming Environments”, MPRPPCFPE-01, 1994.
[OMG97] Object Management Group. Event Service Specification, Framingham, Mass. 1997.

7 Appendix A, PCICom

This section encapsulates implementation specific parts of PCICom.

7.1 Message Queues

Each queue is represented by a data structure and the memory layout of PCICom is illustrated in Figure 61 below.

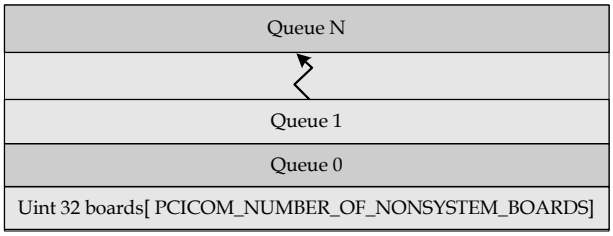


Figure 61, PCICom memory layout.

Each message queue consists of the data illustrated in Figure 62. A top and bottom pointer will govern the message queue in order to make it function as a ring-buffer. Since only one writer and only one receiver is active at every instant, there will be no requirement on mutual exclusion. A flow control field has been added for future use. Each message entry is limited to PCICOM_ENTRY_SIZE and is accompanied by information of the size of the message and the id of the sender.

top	bottom	flow Cont.	index	not allocated
size	sender	message 0 [PCICOM_ENTRY_SIZE]		
size	sender	message 1 [PCICOM_ENTRY_SIZE]		
size	sender	message 2 [PCICOM_ENTRY_SIZE]		
size	sender	message PCICOM_NUMBER_OF_ENTRIES [PCICOM_ENTRY_SIZE]		

Figure 62, Data structure of each queue.

This version supports only one reader and one writer per message queue, in order to avoid synchronization among multiple writers/senders. If there is a need for multiple writers this functionality can be added in the future by introducing a mutual exclusion mechanism among writers or by letting every sender have a unique message queue. It is important to understand that if the senders are located on

multiple processors, mutual exclusion has to be achieved between them. It is possible to use the LOCK# functional signal to achieve atomic read/modify/write transactions on the PCI-bus and thus being able to create shared semaphores among nodes. At present PCICom is capable of these features and limitations listed below:

- One reader per message queue.
- The system board is responsible for initialize the message queues at startup.
- Senders cannot synchronize. If multiple senders/writers want to send a message to the same queue in the future, they must synchronize. With the current implementation it is possible that an interrupt is issued from multiple boards to the receiving boards before a previous interrupt has been acknowledged. If used together with the IOMP mechanism, where only one application thread interacts with exactly one peer, this limitation is acceptable.

The PCICom class acts directly upon the peripheral registers and data structures, which makes PCICom feasible to use in operating system environments where user threads are allowed to act upon hardware directly. In this work VxWorks has been used as the operating system. When using PCICom in an operating system environment supporting kernel and user space, some redesign is necessary (introducing system calls, copying between user and kernel space as well as adapting the code to a specific device driver methodology).

7.2 Interface

Every sender and receiver has to instantiate the PCICom class, and has to provide parameters that will point out a specific index, setting the type of the queue and determine if it is a sender or a receiver .

PCICom(Uint32 in_index, Uint32 in_senderReceiver, Uint32 in_queueType)

The constructor that will enable the user to access the queue through the instantiated object. If the user instantiated the object as a receiver, a semaphore will be created that afterwards will be used by the receive method to wait for messages.

Uint32 PCICom::**send**(const void *data, Uint32 length, Uint32 timeout, Uint32 priority)

Sends a message of size length to the receiver (receiving queue was declared at instantiation.

Uint32 PCICom::**receive**(void *data, Uint32 *length, Uint32 timeToWait)

Through this method a thread can wait a period of time defined by timeToWait for a message to arrive to the queue. The user is responsible for having allocated at least the size of an entry at the memory area where the data pointer points to.

void PCICom::**notify**(void)

This method is used by the send method and notifies the receiver about a sent message. It is not intended to be used by the user and is therefore declared to be private to the class.

Uint32 PCICom::**init**(Uint32 in_boardNumber, Uint32 in_numberOfBoards)

This method should be used at initialization of the application before any objects are created. It configures the board according to if it is a system or a non-system board. The 21554 bridge and the queues are configured according to the definitions in PCIComDefines.h.

void PCICom::**boardInitialized**(Uint32 boardNumber)

This method should be used by the non-system boards to notify the master that they are configured and executing.

void PCICom::**synchronize**(Uint32 nice)

A method that enables the master to wait for all other boards to come up before it continues. The user can through the nice parameter calibrate the polling period of the status of the non-system boards.

void PCICom::**ISRNonSystemBoard**(int parameter)

void PCICom::**ISRSystemBoard**(int parameter)

The Interrupt Service Routines that are responsible for acknowledge issued interrupts and awakening the receiving thread.