Ken Sakamura (Ed.)

TRON Project 1990

Open-Architecture Computer Systems

Proceedings of the Seventh TRON Project Symposium

With 176 Figures

Springer-Verlag
Tokyo Berlin Heidelberg
New York London Paris
Hong Kong Barcelona

KEN SAKAMURA
Leader, TRON Project
Department of Information Science
Faculty of Science
University of Tokyo
Hongo, Tokyo, 113 Japan

ISBN-13: 978-4-431-68131-1 e-ISBN-13: 978-4-431-68129-8
DOI: 10.1007/978-4-431-68129-8

Foreword

I wish to extend my warm greetings to you all on behalf of the TRON Association, on this occasion of the Seventh International TRON Project Symposium.

The TRON Project was proposed by Dr. Ken Sakamura of the University of Tokyo, with the aim of designing a new, comprehensive computer architecture that is open to worldwide use. Already more than six years have passed since the project was put in motion. The TRON Association is now made up of over 140 companies and organizations, including 25 overseas firms or their affiliates.

A basic goal of TRON Project activities is to offer the world a human-oriented computer culture, that will lead to a richer and more fulfilling life for people throughout the world. It is our desire to bring to reality a new order in the world of computers, based on design concepts that consider the needs of human beings first of all, and to enable people to enjoy the full benefits of these computers in their daily life.

Thanks to the efforts of Association members, in recent months a number of TRON-specification 32-bit microprocessors have been made available. ITRON-specification products are continuing to appear, and we are now seeing commercial implementations of BTRON specifications as well. The CTRON subproject, meanwhile, is promoting standardization through validation testing and a portability experiment, and products are being marketed by several firms. This is truly a year in which the TRON Project has reached the practical implementation stage.

Application projects, such as the Intelligent House, Intelligent Building, Intelligent Automobile and Computer City projects, are also attracting considerable attention. A new addition to these projects this year is the start up of a research group on human/machine interface in home electronics products.

It is my sincere hope that this TRON Project Symposium will be a worthwhile experience for all of you who have favored us with your attendance.

Akio Tanii
Chairman, TRON Association
President, Matsushia Electric Industrial Co., Ltd.

Preface

The last 12 months have seen application-oriented developments in the TRON Project as exemplified by the TRON Intelligent House which was completed on December 1st 1989. Three more application projects began in 1989 and 1990 which means the application projects now in existence are TRON House, TRON Intelligent Building, TRON Intelligent City Development, TRON Intelligent Automobile, BTRON Multimedia Communication Research, and TRON Electronics HMI.

The TRON Project aims at creating an ideal computer architecture which takes into account the proliferation of computers in today's society. In order to establish the architecture, we need to anticipate the investigations of the future and this is the motivation behind the application projects.

I would venture to say that older computer systems were often created with the developer's interests in mind, perhaps to the disadvantage of the user. We now see many products on the market that cause many problems or inconveniences due to this design approach. The TRON Project aims at establishing computer systems that are truly user-friendly which means we need to proceed with our projects in a way that involves the system developers from the beginning while bearing the user in mind.

We are aiming to incorporate the user's requirements that we learn from the application-oriented projects into the basic subprojects of the TRON Project such as ITRON, BTRON, CTRON, TRON-specification CPU. The feedback from application developments is an indispensable part of the Project.

We regard these application projects as essential in order to prevent us from making the same mistakes that society made in coping with environmental problems; we failed to anticipate environmental problems by neglecting a vigorous assessment of the impact of human activity on the environment. Those involved with the TRON Project do not intend to produce unwelcome surprises when advanced computer systems are in wide circulation. Now is the time to anticipate and resolve the problems before mass installation.

Today, we see real applications being added to the capabilities of ITRON, BTRON, CTRON and TRON-specification CPU and we are receiving active cooperation from application developers. With this support the MTRON subproject will become the basis of a fully computerized society in the future. As the leader of the Project, I hope that this volume conveys our enthusiasm to all readers.

KEN SAKAMURA

Table of Contents

Appendix: Additional Contributions

Key Note Address

Programmable Interface Design in HFDS

Ken Sakamura
Department of Information Science, Faculty of Science, University of Tokyo

ABSTRACT

A programming/communication model on a shared memory base is proposed as a framework for devising standard interface specifications, in a large-scale, loosely coupled distributed environment. Although this model assumes a shared memory base, it is designed to allow different access mechanisms, structures, and a diversity of practical restricting conditions to be introduced at the model level to account for specific applications. The model integrates common data spaces found in a variety of applications, making it possible to realize compact interface specifications optimized to each application.

The aim behind this model is to realize programmable interfaces, as required in implementing the highly functionally distributed systems (HFDS) that are an ultimate objective of the TRON Project. By realizing programmable interfaces on this model, it will be possible to achieve commonality among each of the interfaces in HFDS, and at the same time to allow interface expansion to adapt to technological advances and to application differences.

Keywords: uniformity, extensibility, distributed environment, shared memory, communication model, programming paradigm, programmable interface

1. Introduction

The ultimate aim of the TRON Project is to build highly functionally distributed systems (HFDS). To this end the concept of programmable interfaces has been introduced. A programming model is specified as the basis of programmable interfaces. When programming is performed in a TRON environment, this programming model represents the system as it is seen from the programmer's standpoint, regardless of the language specifications used. In a broad sense, this model is the TRON architecture itself.

In TRON, a shared memory-based programming model has been adopted, since it is semantically close to the sequential programming approach in a von Neumann computer, to a window-based user interface, to file systems, and other basic aspects. The shared memory concept in TRON, however, differs from the conventional notion, in that a framework is provided from which shared memory systems can be derived that are optimized to specific applications. This optimization is done by choosing access methods and memory space configurations suited to each application.

The thinking that led to development of this model is outlined below. The paper describes the resulting programming and communication model based on shared memory, which is proposed as the direction for future design work. The features of this design policy are then discussed as realized in TULS (TRON Universal Language System), an actual TRON system programming environment.

2. Background

In an HFDS, many thousands or even millions of computer systems of all scales, from embedded microcontrollers to huge database servers, will be loosely coupled, forming a network that is vast and diverse, both quantitatively and qualitatively. Moreover, the network will undergo constant reconfiguration as nodes are added or removed, or replaced by different systems. In order to assure that the nodes in this network operate in harmony with each other, a flexible network architecture is required featuring both compatibility and extensibility.[1] This is the purpose of the programmable interface concept introduced in the TRON Project.

2.1 Programmable Interface

Cooperative interaction among elements in an HFDS is possible only if standard interfaces are provided on all communication paths in the network. If, however, such standardization means that all the interfaces in a vast and open network are to have the same fixed specifications, this will stifle the incorporation of future advances in computer technology. For this reason fixed specifications are not desirable. On the other hand, if version updates are allowed, inconsistency in versions throughout the network will soon become a problem. And if strict adherence to upward compatibility is made mandatory, many systems will become overweighted with nonstandard and unneeded aspects.

Similar problems apply not only between networks but also to system calls, which are interfaces between systems and application programs, as well as to data formats, which are interfaces between application programs. In the HFDS system as a whole there are many more interfaces, in the broad sense, that need to be specified, including network protocol, printer control codes, and also data coding formats and human/machine interfaces. Here, too, it is desirable to allow changes in the interface specifications in order to incorporate new technology and adapt to different applications; but at the same time, interface changes can lead to compatibility problems.

The programmable interface concept approaches this dilemma in the following way. A system with which communication is made can be programmed, and in this way interface specifications can be changed dynamically as needed. When communication takes place between systems, first of all the interface specifications on both sides are compared with each other, and if necessary, the side requiring higher-level specifications sends a program to the other party, thus establishing the necessary communication.

Perhaps the simplest example is the problem of user-defined characters. TRON already offers a character code that includes nearly all the characters in use today,[2] but there is no end to the symbols that could be devised which are not supported in the character code. If a user defines such a symbol, there are numerous cases where it will not be recognized, such as when it is transferred between different systems, between application programs, or between an application program and window system. In the programmable interface concept, however, the file containing the symbol can define the symbol in a "program section" at the head of the file, using graphical data, and thereafter this definition can be referred to by a code wherever the symbol appears in the file. The system receiving the file first receives this program section and programs itself accordingly, so that it can interpret the data following.(*Fig. 1*)

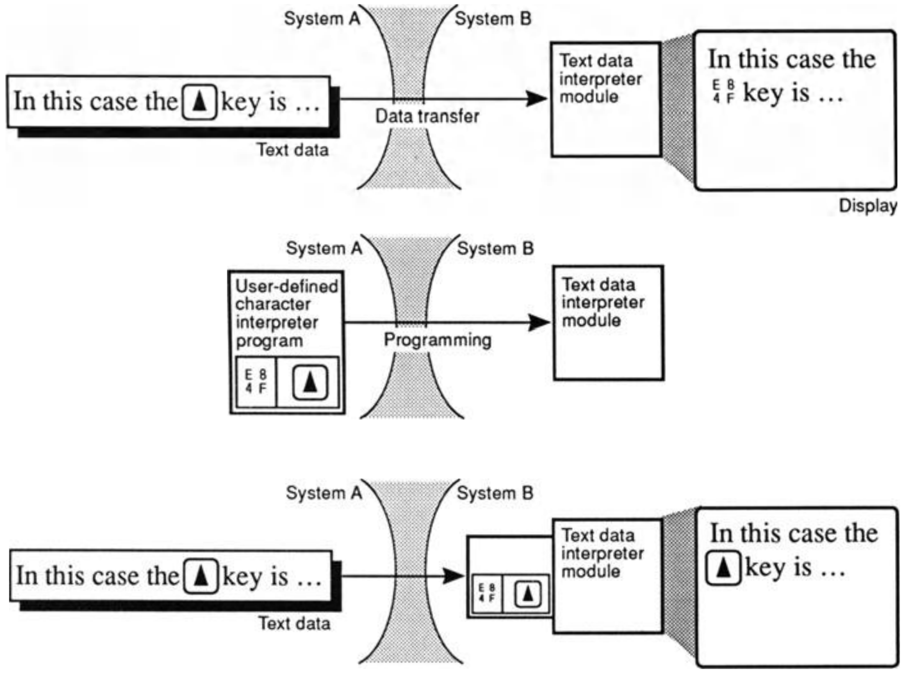
System A
System B
In this case the ▲ key is ...
Text data
Data transfer
Text data interpreter module
In this case the E8 4F key is ...
Display
System A
System B
User-defined character interpreter program
E8 4F ▲
Programming
Text data interpreter module
System A
System B
In this case the ▲ key is ...
Text data
E8 4F ▲
Text data interpreter module
In this case the ▲ key is ...

Fig. 1 Handling of user-defined characters in a programmable interface

Likewise, user expansions such as defining new character formats (outline, for example) can be handled in a similar fashion. This ensures that other systems will reproduce the character format in the same way as intended.[3] In the language processing environment, as well, an extension of this capability is used. For example, if mail is received by a system lacking the environment for interpreting the language in which the contents are expressed, the necessary language processing environment is requested and used to program the receiving system. The mail contents can then be interpreted and displayed. Similarly, where data are in a format specific to an application, the section of the application for interpreting and displaying data in that format can be made non-proprietary (even if other parts of the program are proprietary) and can be sent along with the data.

This concept also means that, besides maintaining interface compatibility, the specifications of interfaces in an HFDS can be changed flexibly at execution, resulting in an ideal load distribution and efficient communication between systems linked via a programmable interface. Moreover, by distinguishing between design guidelines and implementation, both diversity and compatibility can be achieved at the system level.(*Fig. 2*)

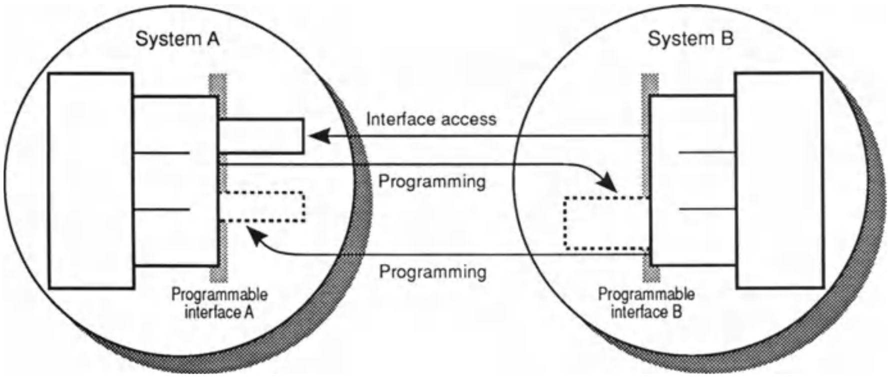

Fig. 2 Interface programming in distributed system

As an example, consider a system in which an ITRON-controlled air conditioner is operated by a BTRON machine in an HFDS. If the air conditioner sends to the BTRON machine a dialog window program for its control, interaction with human beings can be left up to the BTRON machine, while the ITRON side need only receive instructions that have been determined based on a standard control interface.

2.2 Programming Model

The programmable interface concept is applicable to all interface specifications that are conceivable in a TRON system, and in application fields it applies to BTRON as well as to ITRON and CTRON fields. This concept is not simply one in which each interface has in common the ability to be programmed; it also means that this programming takes place in a standard way for all interfaces. In other words, this is not a programmable interface that is valid only between the systems assumed by specific application programs; rather, standard provision is to be made of this capability at the system level, independently of application programs. Only in this way will it be possible for interfaces to interact on communication links between any two systems, and in this way for dynamic standardization of interfaces to be achieved.

Looking once more at the air conditioner example, dialog with the air conditioner does not have to be limited to the BTRON machine's window system, but could also use an ITRON-specification programmable remote controller, or many other combinations of communication. In this sense, as long as a display and pointing device are provided and a system is given

GUI (graphical user interface) capability for dialog with human beings, all kinds of systems can be used for communication.

The problem here comes down once again to whether such a standard programming environment is possible. That is, if a programmable interface is applied to the problem of interface specifications that defy standardization, can standard specifications be devised that fit the object of programming? Further, it is a question of whether the same programming language and environment can be provided for applications as different as window servers and file servers, or to operating systems as diverse as those of a BTRON machine and an ITRON programmable remote controller.

An answer proposed in TRON, based on the von Neumann resource model, is the macro-based programming language/environment design guideline TULS,[4] which has been made specific to programming for BTRON systems in the TACL language and environment.[3] This twofold structure of specifications arises from the judgment that, from the standpoints both of efficiency and programming ease, and for programming of interfaces for the diversity of applications and system scales noted above, relying fully on one programming language and environment would not be feasible. Instead, in this scheme, programming languages and environments are designed in accord with the guidelines (TULS), but can be tailored to specific applications and scales, as in the case of TACL. The programming languages and environments, in other words, do not have to be the same but must have similar specifications.

In subsequent studies it was decided to go beyond mere guidelines and to provide rather a framework for devising very primitive programming languages and environments, like the relation between microprograms and programs. A mechanism could then be provided for putting together programming languages and environments geared to each application and scale, through a process of specializing or fleshing out that framework. This specialization process can be described objectively.

This approach assures that two languages and environments intended for quite different applications or system scales will have sufficient similarity in specifications in the parts where similarity is possible. Even when these parts are viewed from another system, it is only necessary to be familiar with the general coding rules for the specialization process. This will enable programming of the other system without knowing the detailed internal specifications. This abstraction is necessary for a programmable interface in an open network.

This framework for devising programming languages and environments is here called a "programming model."

2.3 Programming Model Requirements, and Direction of Studies

Next a summary will be given of what is required in a programming model of the type described above. These requirements determine the direction to be taken in studies on the programming model.

A programming model for programmable interfaces must be universal enough to be usable in all the systems of different purposes that make up an HFDS. At the same time it must be sufficiently abstract so as to allow programming without detailed knowledge of the interface specifications of the other system.

Ease of programming in the languages and environments based on that model is the next requirement. This is of course important when human beings do the programming directly, but is also desirable in systems with automatic programming capability, which will no doubt be a key feature in future HFDS subsystems that operate autonomously and cooperatively.

Another obvious requirement for this programming model is achievement of execution efficiency and compactness, so as not to impose a burden on the many small-scale hardware resources included in an HFDS.

The need to realize the seemingly irreconcilable objectives of universality and efficiency leads to adoption of a policy of strict adherence to von Neumann architecture, as was the case in TULS studies. The von Neumann model, as it turns out, is simulated in all computer environments, from window systems to file systems. Any model, so long as it properly represents the basic qualities of a von Neumann computer environment, is capable of better efficiency than a purpose-specific interface that ignores those qualities. In this sense, a universal model does not necessarily have to be inefficient.

The model, in other words, expresses the von Neumann architecture at as low a level as possible, and as abstractly as possible. The object of continuing studies is to devise a universal mechanism for specializing this model to fit the needs of various applications. Further, this specialization should achieve a high level of communication between interfaces so that programming can be simplified.

2.4 Communication Model Requirements, and Direction of Studies

In an HFDS, programming must be possible while a system is operating. With a programmable interface, programming takes place without distinguishing between ordinary programming using an interface and dynamic programming. To this end, the programming model must also be a communication model between network-transparent systems, processes, and layers.

The discussion up to now has focussed on the programming model aspect, but the same requirements must be satisfied in terms of use as a communication model. That is, when application programs are programmed in a distributed environment, it must be possible to code programs simply, on a communication model that integrates communication programs that are complex and distributed for each of the diverse objects of communication.

This aim of simplifying programming of application programs by means of an integrated communication model is realized in UNIX by means of a byte stream file model. Integration in this model, however, is performed in a classic environment in which the file is a flat byte stream, the user interface is a teletype, and communication between processes is by message passing only.

Communication model integration that meets the needs of future high-level applications must take into account the emergence of a variety of files, ranging from hash files to hypertext and other types of high-level files. It must also be premised on memory shared among processes, for the sake of real-time performance, and on user-friendly window systems that display application programs independently.

In addition to uniformity, there are of course such general requirements as network transparency, extensibility, security, reliability, maintainability, and real-time performance.

When it comes to guidelines, extensibility can be achieved by means of programmable interfaces, whereas performance needs are met by the policy of "strict adherence to a von Neumann architecture." Network transparency, extensibility, security, reliability, and maintainability can be realized by increasing the degree of abstractness of communication so that when communication takes place, the necessary processing is performed at the system end, independently for each application program.

3. Two Communication Models

Integrated communication models for the sake of simplified programming can be classified as below into message passing and shared memory types.

3.1 Message Passing Model

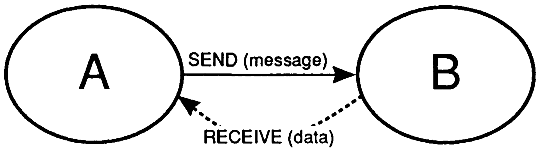

Fig. 3 Message passing model

An advantage of a message passing model is first of all the high degree of abstractness. Thus, even when rather complex communication is realized, there is little need to be concerned about side effects, and in that sense the burden on programmers is lessened. For similar reasons, network transparency, extensibility, security, reliability, and maintainability are high; and the semantic gap with network configuration is small, so that this model is suited to loosely coupled distributed systems.

On the other hand, a disadvantage of message passing is that the message issuer must know the recipient of a message. A further restriction is that the recipient must exist when messages are sent. Moreover, FIFO is the only possible form of access to messages. When messages arrive at random time intervals, they must be tracked from the head of the queue, with their states determined dynamically; and measures must be taken to assure mutual consistency.

When complex communication is attempted, there is the added burden of having to decide data format and protocols, etc. between the sender and receiver; and the problem of consistency noted above must also be taken into account, as a result of which the communication-related aspects of a program can become quite unwieldy.

Even if the programmer's burden is to be reduced by greater abstractness, when it comes to complex communication — especially in applications with severe needs for real-time control — the programming burden will still be considerable. Real-time applications are a basic assumption in TRON, and in that sense the message passing model can be seen as too abstract.

3.2 Shared Memory Model

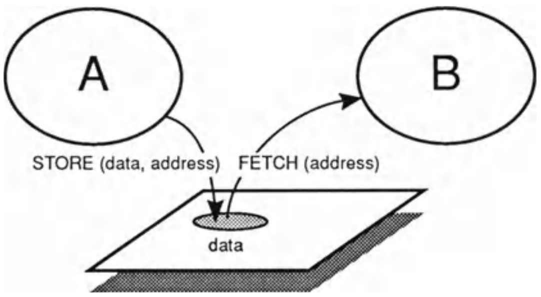

Fig. 4 Shared memory model

The advantages and disadvantages of the shared memory model are the converse of those noted for the message passing model. That is, this model features good execution efficiency due to the narrow semantic gap with sequential programming in the von Neumann model. Moreover, random access is possible, and communication is not restricted by the need for existence of a recipient when sending takes place. In addition, common data are located in shared memory, so that changes in data are inevitably reflected in the data of all those possessing it. This removes the need for concern about data consistency. This model, moreover, is best for exchanging data with complicated structures. For these reasons, programming is generally easy.

Where the shared memory model suffers in comparison with message passing is the low level of abstractness. For this reason, when complex communication is realized, the programmer's burden for those aspects is large. This is because, with programming in a conventional shared memory model, passing of control is treated as a completely separate event from access to shared memory, as a result of which changes in shared memory take place as side operations.

Along with the low degree of abstractness, a fully shared memory model, in which even non-shared data that should be kept separate are all located in the same memory space, would hardly be desirable from the standpoints of network transparency, extensibility, security, reliability, and maintainability.

In addition, when a distributed environment is configured by joining together heterogeneous systems, and especially when systems are linked whose memory structures use different endians, the shared memory model is disadvantageous.[5]

3.3 Shared Memory Object Model

A system that is simply a faithful rendition of one of the conventional models described above would thus be impractical. Instead, loosely coupled distributed systems today adopt what can be called a shared memory object model, in which a general message passing model is put into practice along the lines of a shared memory system.(*Fig. 5*)

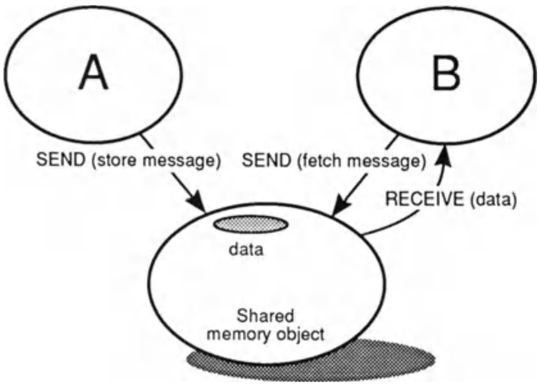

Fig. 5 Shared memory object model

In these systems, data that require sharing are stored in objects, to which those sharing the data (users) request access by means of messages. The users need only be aware of the shared memory objects as message receivers, and the only protocol necessary is that with the shared memory objects. Besides, the existence of message receivers is guaranteed at all times, and random access is possible with this approach. These added qualities also have the effect of reducing the programmer's load.

3.4 Advanced Shared Memory Model

Similarly, by devising a mechanism for triggering high-level processing involved in access to shared memory, it is possible to conceive of a model based on shared memory but which incorporates the advantages of message passing.

If the shared memory access primitives are made more advanced, it is possible to realize functions for abstraction of communication, for objectification of shared memory, for distributed memory and replication, and for distributed atomic transaction.(*Fig. 6*)

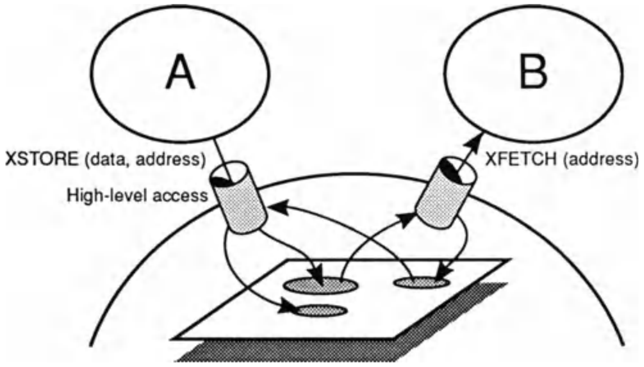

Fig. 6 Advanced shared memory model

Moreover, by means of objectification, and by imposing a restriction requiring FIFO access to shared memory that is used only between certain specific users, the model can be implemented using a message passing approach.

This expansion of the original model makes it possible to overcome the disadvantages of the shared memory model noted earlier, satisfying the need for network transparency, extensibility, security, reliability, and maintainability. The resulting model can be called an advanced shared memory model.

3.5 Consideration as Access Method

The more realistic approaches described above are the result of bringing the two basic models closer together, in the process of which many of the differences between the two are resolved. In deciding whether to adopt a model based on message passing or one based on memory sharing, it is first necessary to consider the essential differences that derive from having started out from one model or the other.

The essence of those differences is in the overhead arising from the method of access to shared data. The shared memory object model, inasmuch as it is a development out of the message passing model, realizes its access method by using messages. For this reason, it is not possible to improve efficiency beyond the message access overhead.

The advanced shared memory model, on the other hand, raises the level of memory access primitives. As for the important question of how the level is to be raised, the model must be tuned sufficiently to the particular application, so that even if overhead becomes greater than that of the pure memory access model, it will still be possible to ensure greater efficiency than message access overhead.