REIHE INFORMATIK
6/94

# Efficient Configuration of Protocol Software for Multiprocessors

– Revised Version –

S. Fischer und W. Effelsberg
Universität Mannheim
Seminargebäude A5
D-68131 Mannheim

# Efficient Configuration of Protocol Software for Multiprocessors

Stefan Fischer and Wolfgang Effelsberg
Praktische Informatik IV, University of Mannheim
D-68131 Mannheim, Germany
email:{stefis, effelsberg}@pi4.informatik.uni-mannheim.de

November 25, 1994

**Abstract**

Efficient implementation of communication software is of crucial importance for high-speed networks. One way to improve the runtime performance of protocol implementations in the network nodes is the use of parallelism. Formal description techniques like Estelle improve the specification process in many respects and allow for semiautomatic code generation. Therefore, they are now widely accepted. We present a code generator for Estelle that compiles and automatically configures protocol software for a multiprocessor. Software modules are distributed over the available processors and executed concurrently. We report performance results on a KSR1 with 28 available processors under the OSF/1 operating system.

## 1 Introduction

Existing protocol suites such as the ISO/OSI or the INTERNET protocols were designed with relatively slow links in mind. The end systems were fast enough to process complex protocols because transmission speed was slow compared to the time needed for protocol execution.

Given the fast transmission media based on fiber optics that are now available, current end systems are too slow for high–performance communication. Communication software has become the major bottleneck in high–speed networks [CT90, Svo89]. Efficient implementation of the protocol stack is of crucial importance for the networking future. One approach to improve protocol performance is the use of parallelism [HEHK92, BZ92, DDK$^+$90, PS92]. Parallel execution of protocol entities leads to a significant speedup in protocol execution [Zit92].

For specification purposes, the formal description technique Estelle (among others) was standardized by ISO [ISO89]. Formal description techniques improve the correctness of specifications by avoiding ambiguities and by enabling formal verification. In addition, they allow semiautomatic code generation. Generating code from specifications has several advantages: The code can be maintained more easily since the system is specified in an abstract, problem–oriented language, and the code is well–structured. It is also much easier to port an implementation to another system. But one of the major problems has been the performance of implementations produced automatically from a formal specification.

Often, today's code generators are primarily designed to easily get rapid prototypes for simulation purposes, e.g. [SS93]. Some of the generators already make use of parallelism [JJ89, Pet91], but mainly for validation and simulation purposes. In contrast we have implemented a code generator for Estelle aiming at good performance of the generated code at runtime [FH93]. It maps an Estelle specification to a multiprocessor system. The runtime system of our compiler allows the concurrent execution of protocol entities.

Though the first results we obtained with our system were good, they were not close to the theoretical limit for the possible speedup for the execution of Estelle specifications derived in [Hof94]. As a consequence, we present in this paper an improved solution for the design of our Estelle code generator. The protocol software is configured for a multiprocessor in an optimized way. Unlike other approaches ([PPVW92] or [HP91]), we do not mean by configuration the selection of certain protocols but the mapping of certain software parts to selected processors. A similar approach can be found in [MT93] where a formal language as well as an implementation environment for the development of parallel systems is described.

The paper is organized as follows: Section two briefly introduces Estelle and concentrates on its features for the description of parallelism. Section three compares the theoretical limit for the speedup for parallel execution of Estelle specifications to results we obtained with our Estelle compiler. In section four, we describe a better design for the compiler, taking into account machine, operating system and Estelle specification characteristics. Section five presents current performance results. Section six concludes the paper.

## 2 Specifying Parallelism with Estelle

Estelle specifications consist of a hierarchically ordered set of finite state machines, so-called *modules*. Modules communicate via bidirectional *channels*. A channel connects two *interaction points* (one per module), both of which have unlimited queues for asynchronous message reception associated with them. Estelle modules can be nested: Within the body of a module, other modules, called *child modules*, can be defined. Thus all modules of a specification form a tree.

When a module has no attribute, it is said to be *inactive*, i.e. it has no transition besides the initialization (often the root module is inactive). All other modules are *active*.

The execution sequence of the modules is controlled in two ways: according to their position within the tree, and by means of an attribute given to each module. The basic tree rule is that a parent module always takes precedence over its children, i.e. a child can only execute if the parent has nothing to do. A parent and a child can never run in parallel.

The *module attributes* control the parallelism between modules at the same level of the hierarchy. There are four attributes: `systemprocess`, `systemactivity`, `process` and `activity`. In the following we use the term *system module* for `systemprocesses` and `systemactivities`. Then the following Estelle rules apply:

- Every active module must have one of the four attributes.
- A system module cannot be contained in another attributed module.
- Each `process` module and each `activity` module must be contained (perhaps indirectly) in a system module.
- A `process` module or a `systemprocess` module can contain other `process` or `activity` modules.
- An `activity` module or a `systemactivity` module can only contain other activity modules.
- Children whose parent module is of type `process` or `systemprocess` may all run in parallel. Children whose parent module is of type `activity` or `systemactivity` are mutually exclusive, i.e. only one of them can run at a time.

As a consequence, a module containing a systemmodule must be inactive; it is typically located at the root of the tree. Also, in each path of the tree, from the root to a leaf, there is exactly one system module, i.e. each active module belongs to exactly one system module.

The dynamics are as follows. At runtime, a module instance can only be dynamically created and destroyed by its parent module. Thus the number of module instances can vary at runtime, but their relative position within the tree is predetermined. When the system is initialized, exactly one instance of each *system module* is created. As opposed to the `activity` modules and the `process` modules, the structure of the `systemactivity` modules and `systemprocess` modules is static at runtime. The system modules themselves are mutually independent and can run asynchronously and in parallel.

The motivation behind these semantics are that a typical communication system has static parts and dynamic parts (to be created at runtime). For example a protocol entity implemented as a `process` can accept a new CONNECT request and then create a new child module to handle the new connection. All child module instances for parallel connections will then be independent of each other, and able to execute in parallel.

Most existing execution environments for Estelle run on sequential machines and use a very simple runtime scheduler to guarantee the above semantics. The execution is organized in cycles. The cycle begins with the system module. If it has a transition to execute ("fire"), it will do so, ending the cycle. If not, it passes the right to fire to its children. There, the same procedure is repeated, all the way down to the leaves of the tree. The number of children allowed to execute their transitions in parallel is determined by the attribute of the parent module: Children whose parent module is of type `process` or `systemprocess` may all run in parallel. An `activity` or `systemactivity` module may only allow one child to execute a transition in each cycle.

It should be noted that asynchronous parallelism in Estelle specifications could be increased considerably by slightly extending syntax and semantics, thereby maintaining the practically relevant logic of the semantics, while relinquishing the unnatural restrictions introduced by the cyclic execution model. This extension provides for much more potential for efficient implementation on multiprocessors. The details can be found in [BG93].

Obviously it is possible to implement an Estelle module tree on a multiprocessor, taking advantage of the two possible forms of parallelism in Estelle specifications:

- Asynchronous parallelism between all system modules
- Synchronous parallelism in subtrees rooted at `process` or `systemprocess` modules (synchronized by the parent module).

In order not to violate the Estelle semantics described above, the parallel implementation still has to guarantee that parent and child modules may never run in parallel, nor any of the modules inside a subtree rooted at an `activity` or `systemactivity` module.

## 3 Theoretical Limit and Practical Results

When implementing Estelle specifications on a multiprocessor, the possible speedup is bounded by two factors: first, by the number of processors, and second, by the Estelle semantics described in section 2.

Obviously, the speedup can never be greater than the number of processors. For the limitations introduced by Estelle semantics, we developed a formula for an upper bound of the speedup [Hof94]:

**Theorem 1** *Let $m$ be the total number of modules in a specification, $b$ the number of leaf modules, $h$ the height of the module tree, $t_{sel}$ the average time for the selection of a transition and $t_{exe}$ the average time for its execution. Then, the following upper bound is valid for the maximum speedup $s_{max}$:*

$$s_{max} \leq \frac{m + b\frac{t_{exe}}{t_{sel}}}{h + \frac{t_{exe}}{t_{sel}}} \tag{1}$$

For $t_{exe} \ll t_{sel}$, this leads to the following bound:

$$s_{max} \leq \frac{m}{h} \tag{2}$$

For $t_{exe} \gg t_{sel}$, the bound computes to:

$$s_{max} \leq b \tag{3}$$

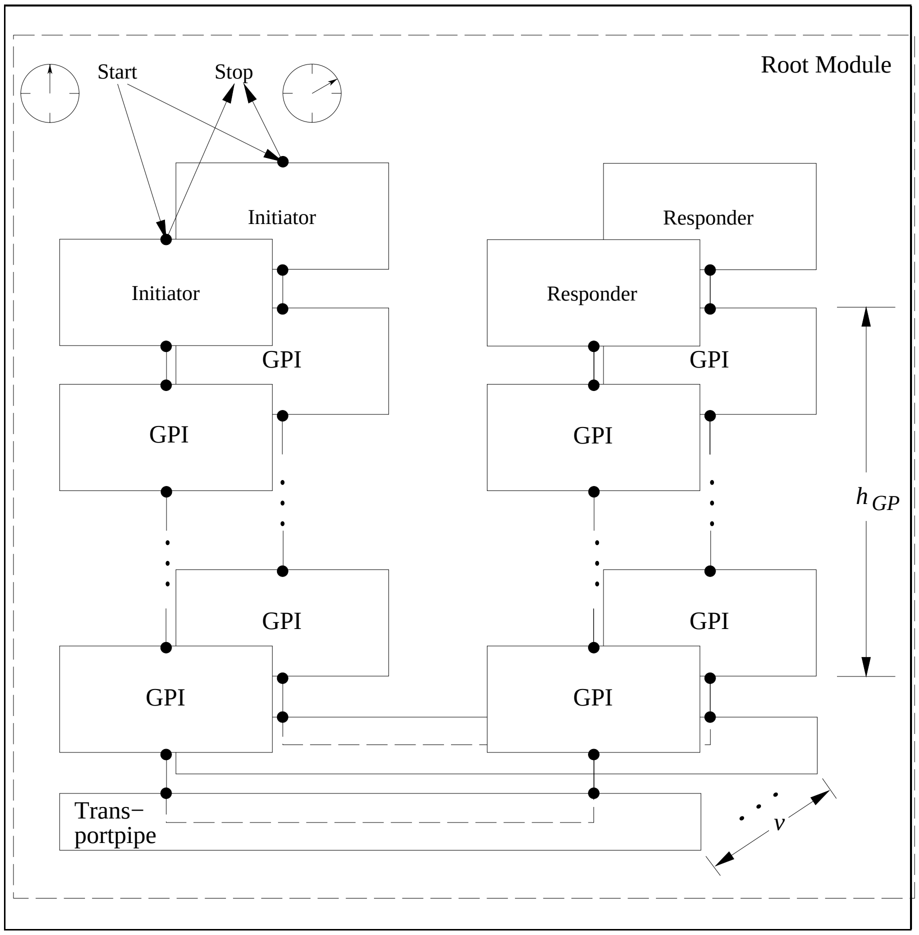

Figure 1: Generalized Protocol Stack for Execution Time Measurements

The derivation and proof of these formulas can be found in [Hof94].

These formulas help us to structure our Estelle specification for optimal parallelism: first, the module hierarchy has to be designed as flat as possible. This keeps the height $h$ small and the number of leaf modules $b$ big thus increasing the upper bound in formula (1). Second, the number of modules should be big, increasing the overall parallelization potential. As a consequence, large modules should be checked if they can be split into more modules. When doing this, the ratio $\frac{t_{exe}}{t_{sel}}$ has to be regarded. The time for transition selection should be smaller than the transition's execution time.

To test our compiler, we used the generalized protocol stack shown in Figure 1. The *Generalized Protocol Instances* (GPI) take messages from their upper resp. lower interaction point, do some processing, and send the message to their lower resp. upper interaction point. The GPI module is specified in a manner that allows us to connect GPI modules to each other. The internal processing is produced by some module–internal Estelle commands. This is a loop with variable upper bound, doing some computations.

This model is very flexible with respect to the number of connections $v$, height of the protocol stack $h_{GP}$ and the time spent on protocol processing. It is possible to vary all these parameters easily. By the third parameter, the time spent on protocol processing, we model the ratio $\frac{t_{exe}}{t_{sel}}$. The higher the number of iterations in the loop, the higher will be the time spent on protocol processing, and the higher will be the ratio.

For measurements, we always set up all connections, start the timer, send 1000 data requests from the initiator to the responder and wait for the last ACK indication. Then we stop the timer. With a varying number of connections, height of the stack and processing overhead, we got the results shown in Table 1. The parameters $m$, $b$ and $h$ can be computed as follows:

$$\begin{aligned} m &= (3+2h_{GP})v+1 \\ b &= m-1 \\ h &= 2 \end{aligned}$$

Thus, we have a very flat specification with a huge number of leaf modules, resulting in a high degree of parallelism.

The results are already good and show a significant speedup when executing protocols in parallel. However, they are not close to the theoretical limit (which is maximally 28, the number of processors). In the next section, we describe the determinants of the time needed for parallel protocol execution and some ways to increase the speedup.

## 4 Improving the Compiler

For our experiments, we have modified the Estelle compiler Pet/Dingo from NIST [SS93]. The most important drawback of this modified compiler is that it has a very simple mapping algorithm: each Estelle module is mapped onto an operating system thread of OSF/1, and all the threads implementing the modules of one system module tree are packed into one OSF/1

| $v$ | $h_{GP}$ | $m$ | Speedup | | | theor. Limit | | Speedup/Module | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | 1 000 | 5 000 | 10 000 | $m/h$ | $b$ | 1 000 | 5 000 | 10 000 |
| 1 | 1 | 6 | 1,8 | 1,9 | 1,9 | 3 | 5 | 0,3 | 0,3 | 0,3 |
| 1 | 2 | 8 | 2,9 | 3,4 | 3,5 | 4 | 7 | 0,4 | 0,4 | 0,4 |
| 1 | 3 | 10 | 3,9 | 4,7 | 4,8 | 5 | 9 | 0,4 | 0,5 | 0,5 |
| 1 | 4 | 12 | 4,6 | 5,8 | 6,0 | 6 | 11 | 0,4 | 0,5 | 0,5 |
| 2 | 1 | 11 | 3,0 | 3,5 | 3,5 | 5,5 | 10 | 0,3 | 0,3 | 0,3 |
| 2 | 2 | 15 | 5,1 | 6,5 | 6,8 | 7,5 | 14 | 0,3 | 0,4 | 0,5 |
| 2 | 3 | 19 | 6,7 | 8,6 | 8,9 | 9,5 | 18 | 0,4 | 0,5 | 0,5 |
| 2 | 4 | 23 | 8,0 | 11,1 | 11,5 | 11,5 | 22 | 0,3 | 0,5 | 0,5 |
| 3 | 1 | 16 | 4,3 | 5,2 | 5,5 | 8 | 15 | 0,3 | 0,3 | 0,3 |
| 3 | 2 | 22 | 7,0 | 9,5 | 9,7 | 11 | 21 | 0,3 | 0,4 | 0,4 |
| 3 | 3 | 28 | 9,2 | 12,1 | 13,3 | 14 | 27 | 0,3 | 0,4 | 0,5 |
| 3 | 4 | 34 | 10,4 | 15,4 | 15,9 | 17 | 33 | 0,3 | 0,5 | 0,5 |
| 4 | 1 | 21 | 5,3 | 6,9 | 7,0 | 10,5 | 20 | 0,3 | 0,3 | 0,3 |
| 4 | 2 | 29 | 8,5 | 11,8 | 12,7 | 14,5 | 28 | 0,3 | 0,4 | 0,4 |
| 4 | 3 | 37 | 10,6 | 15,8 | 17,2 | 18,5 | 36 | 0,3 | 0,4 | 0,5 |
| 4 | 4 | 45 | 9,5 | 13,5 | 15,2 | 22,5 | 44 | 0,2 | 0,3 | 0,3 |

Table 1: Measurement Results

process. This approach does not take the actual execution environment into account. We see three components that influence the execution time of protocols:

- the machine architecture
- the operating system
- the structure of the specification

These issues will be discussed in the remainder of this section.

## 4.1 The Machine

Concerning the machine[1] on which the protocol stack is running, we identify two important performance parameters: the number of available processors, and the communication cost between each pair of processors.

The number of processors is important as it is a bound on the number of threads. If the number of threads exceeds the number of processors, we can show that we lose more time with thread

[1]We do not elaborate on the selection of MIMD as our parallel architecture. More details on this point are given in [FH93]

synchronization and context switching than we gain by parallelism. This implies that it is better to not fully parallelize a protocol implementation when the number of threads exceeds a certain limit. In section 4.3, we show how to adapt the degree of parallelism smoothly.

Another problem on many multiprocessors is the communication cost between processors. This includes thread synchronization and data exchange. Often, threads or processes running on one processor need to access data which is located in the memory of some other processor. The data has first to be located and then to be transferred. Let us look at the KSR1 architecture, our implementation machine. At most 32 processors are connected to a ring which they use to communicate. If more than 32 processors are used, they have to be connected to different rings. Those rings are then interconnected. Data exchange between processors in one ring is executed during one round trip: Receiving a data request, the processor who has the data puts it on the ring. With no processor on the ring having the data, the request will be forwarded to the higher–level ring controller. Obviously, communication cost will then increase.

The solution to this problem is to map communicating threads (i.e. Estelle modules) onto processors in a cost–minimizing way. Threads which are not communicating may be mapped to processor pairs with higher communication costs.

For our new configuration process, the machine configuration will be stored once in a configuration file. It contains the number of processors and, for each pair of processors, the communication costs.

## 4.2 The operating system

For our work, we decided to use a Mach–based operating system which we consider well–suited for MIMD computers. However, there are still some subtle differences between different Mach–based systems. One important aspect is the thread synchronization mechanism. Estelle modules often have to synchronize with their parent or child modules, and as modules are mapped to threads, thread synchronization plays an important role in protocol execution.

Basically there is one mechanism available in any Mach–based operating system: It is synchronization using locks and condition variables. We use it as follows: To synchronize the threads running parent and child modules, we use two shared memory variables and a lock and a condition variable for each of them. One variable is used by the parent thread to specify the action that should be performed by the child (e.g. execute a transition). The other one contains the number of child modules that received the command. Any child that finishes its action will decrement that variable by one. When the variable reaches zero, the parent knows that all children are ready. It will immediately continue with its own work.

On some systems, there is also a so-called **barrier synchronization**. A barrier has a *barrier master thread* and some *barrier slave threads*. They both may *check in* and *check out* from a barrier. When slaves check in, they cannot continue their work until the master checks in. Afterwards, both master and slaves run in parallel. When the master checks out, he has to stop until all the slaves have checked out.

For use in our Estelle synchronization, the parent module is mapped to the master thread while all its children become slave threads. When the parent wants the children to perform an action, he sets the shared variable accordingly and checks in. The slaves begin their execution by first reading the variable and then performing the requested action. The master checks out immediately after his check in, thus waiting for the completion of the children. A ready child simply checks out.

Interestingly, the mechanism of barrier synchronization comes very close to Estelle's parent–child synchronization. In addition, the implementation uses one shared variable less than in lock synchronization, avoiding a considerable amount of communication between the processors. That results in a much more efficient synchronization. However, barriers have a serious drawback: threads waiting at a barrier perform busy waiting, i.e. as soon as the number of threads becomes larger than the number of processors, or other processes run on that machine, the performance will degrade significantly. As a consequence, barriers should only be used when the number of threads is small. For exact measurement results, see section 5.

## 4.3 The specification structure

The solution of mapping modules to threads presented so-far still does not take the structure of the specification into account. There are three important issues which limit the processor usage:

1. The computational complexity of modules running in parallel may be very different.
2. The computational complexity of some modules is so small that synchronization time exceeds protocol processing time.
3. Estelle's prohibition of parent–child parallelism makes at least one processor running idle. When the parent module passes the right to execute to its children, it waits for all responses wasting one processor.

The solution for all three problems is the "intelligent" reduction of Estelle parallelism by mapping more than one module onto one thread. As a result, we get an increase in the overall system utilization, as there are more processors available for other modules.

From the three problems described above, we derive the following **mapping rules**:

1. Threads which are running in parallel but are synchronized due to Estelle semantics (i.e. children of the same parent) should have nearly equal computational complexity. Threads which are ready will not have to wait a long time. For asynchronous modules (i.e. systems), we need no such rule, as threads running in such a way do not have to wait for each other.
2. Modules with higher synchronization than protocol processing time should not run exclusively in their own thread. Running them sequentially together with another module will save synchronization time.

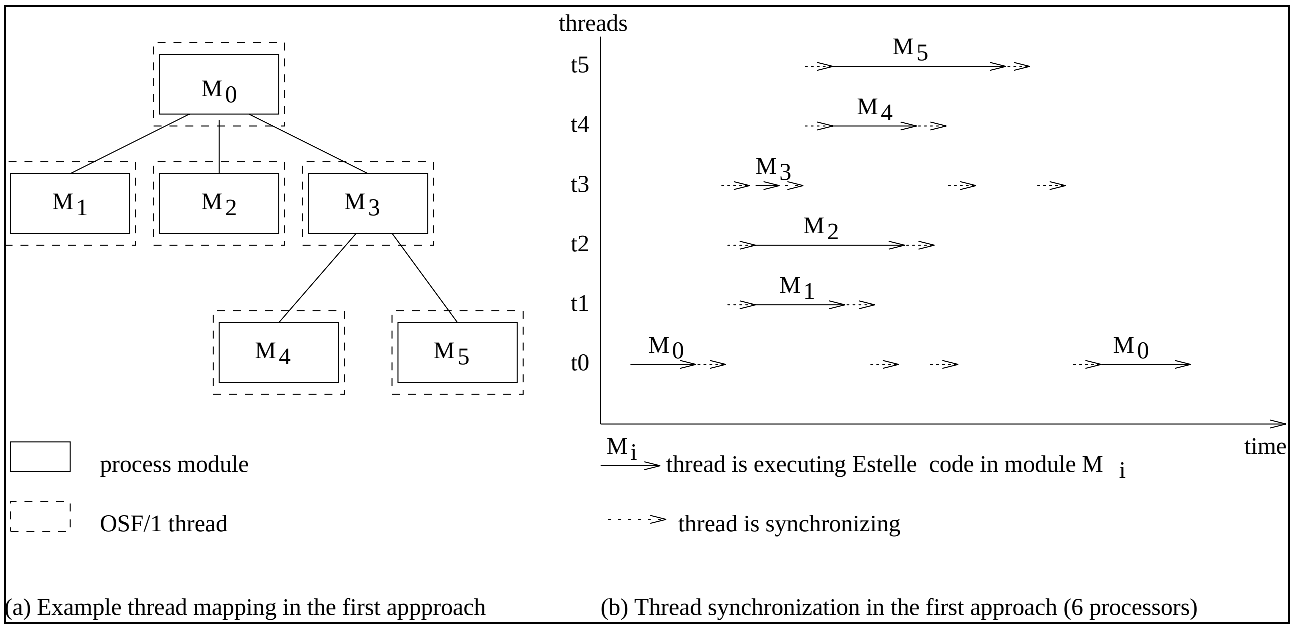
M0
M1
M2
M3
M4
M5
process module
OSF/1 thread
(a) Example thread mapping in the first appproach
threads
t5
t4
t3
t2
t1
t0
time
Mi thread is executing Estelle code in module Mi
thread is synchronizing
(b) Thread synchronization in the first approach (6 processors)

Figure 2: Mapping of modules to threads in the first solution

3. The thread executing a parent module should always include one child module. This thread will then not wait, but execute a child module in parallel to other threads running other child modules of the same level.

A comparison of the traditional and the improved thread mapping scheme is illustrated in Figures 2 and 3. Let us assume that modules $M_0$, $M_1$, $M_2$, $M_3$, $M_4$ and $M_5$ have average execution times of 20, 80, 70, 10, 30 and 30 milliseconds, respectively[2]. So, module $M_1$ and $M_2$ have nearly equal protocol execution times, while module $M_3$ has much less. For that reason, in the new approach, it is grouped together with modules $M_4$ and $M_5$. In the traditional solution, we have a total execution time of 100 ms[3] using six processors; in the new one, we also have 100 ms[4], by only using three processors. The remaining three processors may be assigned to other modules.

To determine the runtime of real protocol modules, we instrument Estelle implementations with time measurement routines allowing us to extract protocol execution times during runtime. So we are able to detect modules which have very small execution times, and we are also able to detect modules which are running in parallel, but have very different execution times.

Currently, our system does not yet allow to compare protocol execution and synchronization time. This comparison is highly implementation– and machine–dependent. Such numbers

[2]We concentrate on average execution times. Experience shows that there is very little data dependency in protocol processing.

[3]20ms for $M_0$ and 80 ms from $M_1$ which is the longest-running module from the three running in parallel.

[4]Again, 20 ms from $M_0$ and 80 ms from $M_1$. The sequential execution of modules $M_3$, $M_4$ and $M_5$ only adds up to 70 ms, less than the execution time of concurrent module $M_1$.

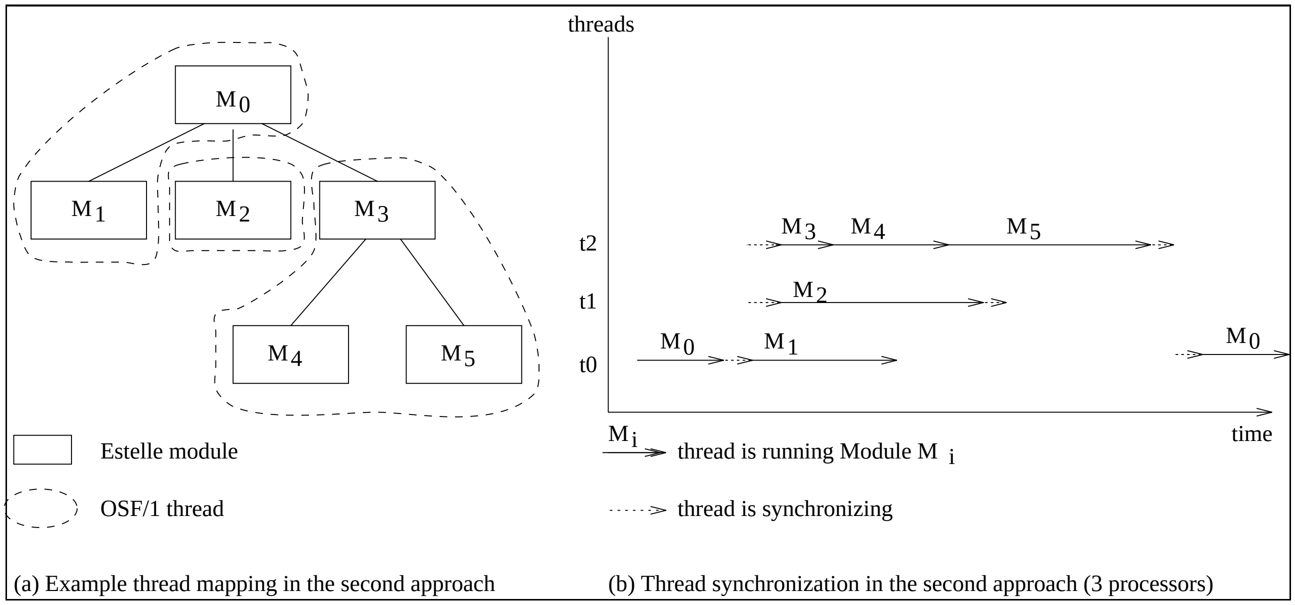
threads
M0
M1
M2
M3
M4
M5
t2
t1
t0
M3 M4 M5
M2
M0 M1 M0
time
Estelle module
OSF/1 thread
Mi thread is running Module M i
thread is synchronizing
(a) Example thread mapping in the second approach
(b) Thread synchronization in the second approach (3 processors)

Figure 3: Mapping of modules to threads in the improved solution

must be added manually to the machine configuration file. For the KSR1, we extracted the synchronization time from [PB93].

## 4.4 The new configuration process

The new protocol software mapping and configuration methodology takes into account all three results described above.

**Configuration Methodology:**

1. Generate the implementation code from the specification, compile it and link it with the modified libraries implementing the Estelle semantics and the module mapping.
2. The protocol software runs sequentially for a certain amount of time specified in the configuration file. Each module measures its runtime.
3. The protocol is configured for parallel execution. The configuration process uses no more threads than processors are available.
4. The protocol software runs in parallel for the amount of time specified in the configuration file. This is the operational phase in the software life cycle. Each module keeps measuring its runtime.
5. Go to step 3.

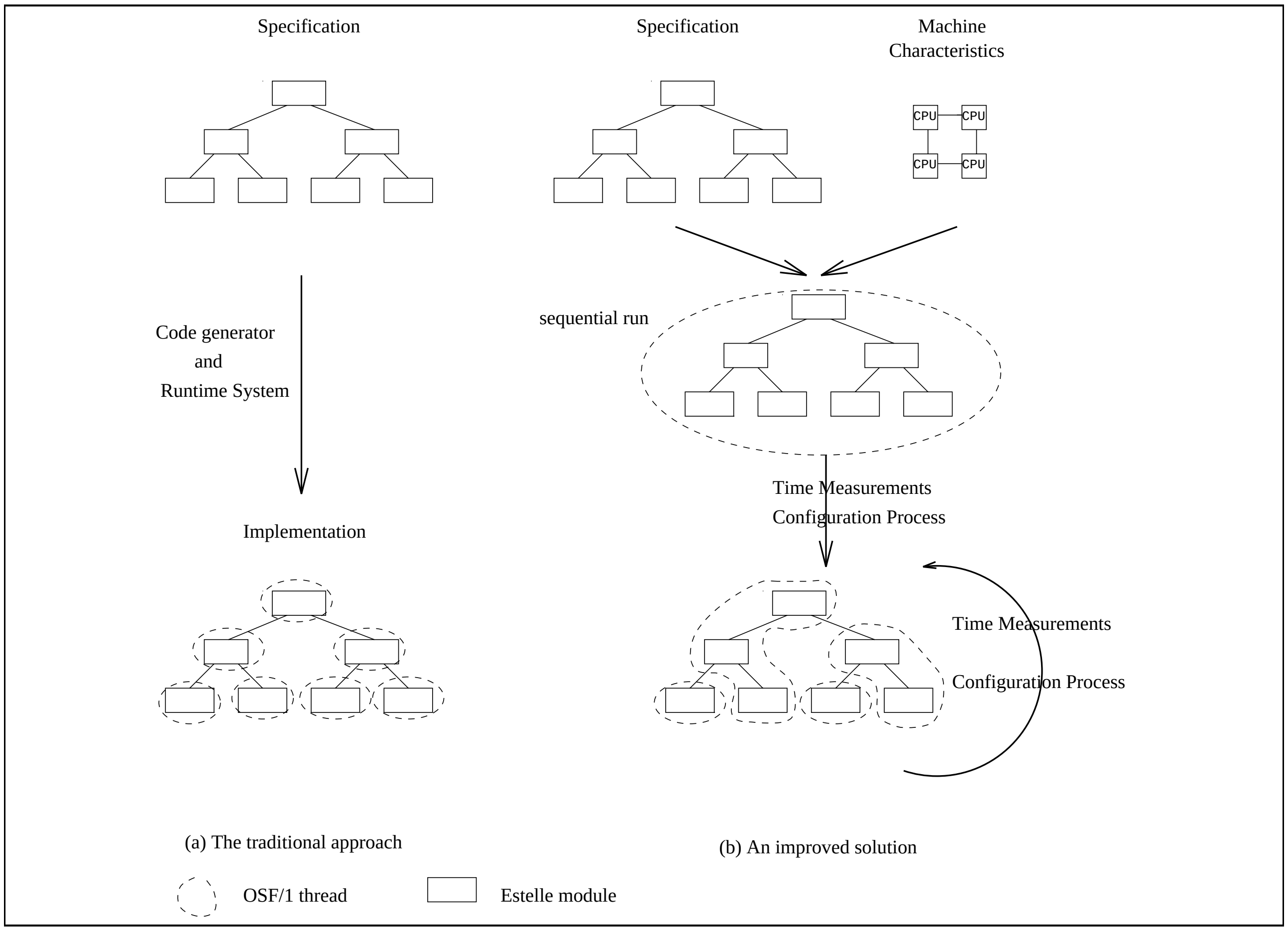

Figure 4: Configuration process

As a result, we get dynamic protocol configuration depending on the number of processors, the Estelle specification structure and the execution time of all the protocol modules. For a comparison between the traditional methodology and ours, see Fig.4.

## 5 Experimental Results

The described process was implemented [Ber94]. As the goal of our code generation was to obtain efficient protocol implementations for high–speed networks, we were very interested in the performance improvements between a standard configuration and the improved version.

**Thread Synchronization**
First, we implemented the barrier synchronization mechanism. Table 2 presents measurement results for Estelle specifications implemented with either 8, 15 or 22 threads. For all three measurements, the maximum number of processors was 16. Obviously, the synchronization times do not depend on the number of data requests exchanged between modules. Barrier

| Data requests | Speedup with 16 processors and | | |
|---|---|---|---|
| | 8 threads | 15 threads | 22 threads |
| 100 | 1.25 | 1.17 | 0.04 |
| 200 | 1.21 | 1.14 | 0.05 |
| 300 | 1.22 | 1.13 | 0.05 |
| 400 | 1.17 | 1.11 | 0.05 |
| 500 | 1.25 | 1.11 | 0.05 |
| 600 | 1.18 | 1.12 | 0.06 |
| 700 | 1.14 | 1.14 | 0.05 |
| 800 | 1.11 | 1.16 | 0.05 |
| 900 | 1.19 | 1.13 | 0.06 |
| 1000 | 1.14 | 1.15 | 0.05 |

Table 2: Barrier vs. Lock Synchronization

synchronization is between 10% and 25% faster than lock synchronization, but only as long as the number of threads is smaller than the number of processors. In that case, barrier synchronization performance decreases heavily due to busy waiting of threads at the barrier.

**Configuration Methodology**
To compare the new configuration methodology to the fully parallel version, we performed measurements with a slightly modified specification. We left out connection establishment and began immediately with the data transfer phase assuming that the connection is already established. Also, vertical dependencies were eliminated. We implemented a configuring and a fully parallel version as well as – for comparison purposes – a sequential version. The configuring version was restricted to four processors. The sequential version used one processor, while, during one experiment, we had the fully parallel version run on four processors and, during another, on as many processors as were needed to get full parallelism. For all experiments, we varied the machine load between one and eight connections. Thus, the fully parallel version used at most 17 processors (2 per connection and one for the root module). The results are shown in Figure 5.

The diagram shows that the new configuration methodology is far better than the fully parallelized implementation. It is not only better on the same number of processors, but also on much fewer processors[5]. We achieve a speedup between 3.3 and 3.5 compared to the sequential execution. This is very near to the optimum of four (the number of processors for the configuring version). It is also obvious that it is unreasonable to use (many) more threads than processors are available. The parallel version on four processors shows a dramatic runtime increase with the number of threads increasing.

In Figures 6 and 7, we see a typical snapshot of the processor usage of the two versions (fully parallel and configured) on the KSR. The usage of the parallel version is very different for single processors. The average load of a processor is relatively small. The configured version shows

[5]This last point is mainly due to the specification structure which introduces a little disadvantage for the parallel version because of synchronization variable accesses

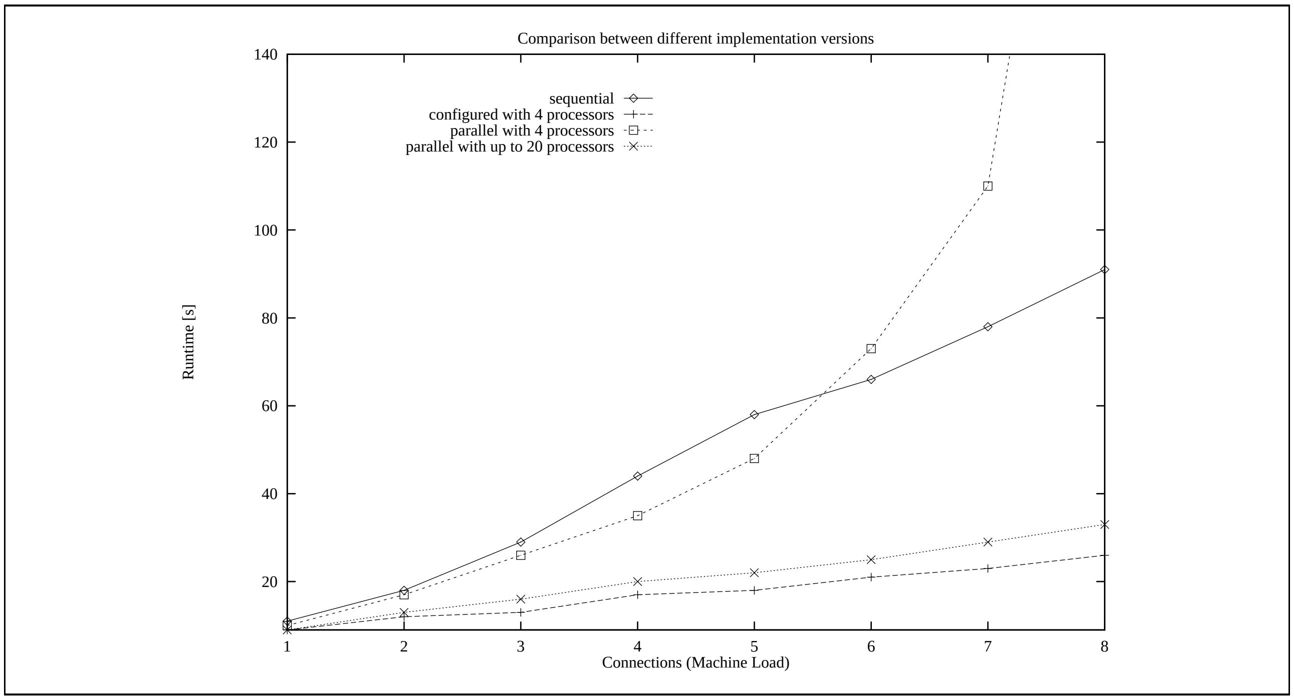

Figure 5: Runtime Comparison between different implementations

an equal load distribution on its four processors. For each processor, the load is relatively high. Thus, we achieve a better usage of the processors involved in system execution, resulting in the same or even better peformance.

These results have two important implications: First, parallel protocol implementations will not only be successful on massively parallel supercomputers such as the KSR or an Intel Paragon etc., but also on much lower priced multiprocessor workstations, equipped with two or four or even more processors, e.g. a SUN SPARC 20. Second, we could easily restrict the protocol excution on fewer processors and achieve the same performance. This may be used for Quality of Service issues. With our compiler, we can e.g. restrict a certain set of connections in a multimedia system (the non-time critical ones) to one or two processors and can then reserve the other processors for connections with very high performance requirements supporting the users' QoS specification.

## 6 Conclusions and Outlook

In previous work, we developed an Estelle code generator that automatically derives parallel code from an Estelle specification and configures parallel protocol software for a multiprocessor. As we were not fully satisfied with the performance results in comparison with theoretical speedup bounds, we developed an improved code generator and runtime system which uses information about the specification structure and the runtime environment to increase the speedup.

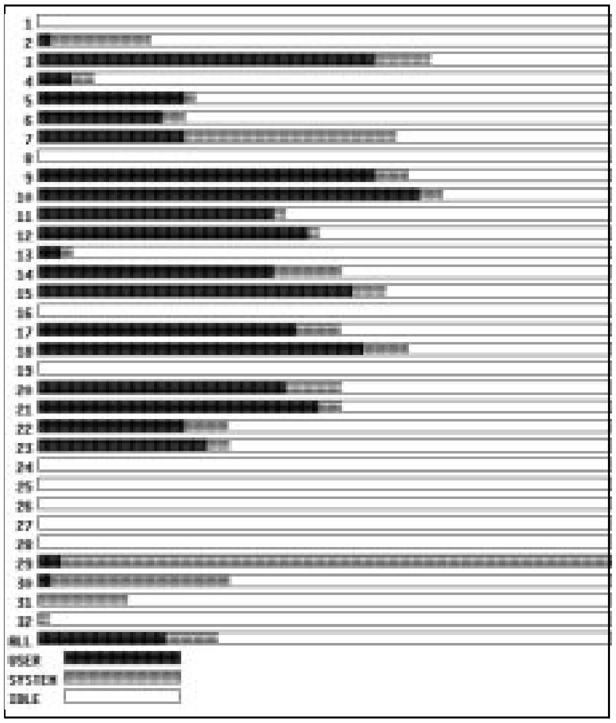

Figure 6: Typical Processor usage for the parallel version

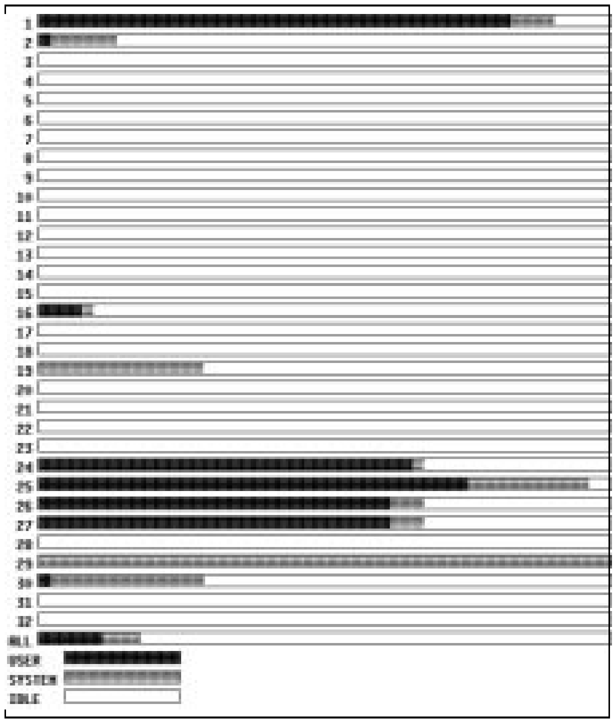

Figure 7: Typical processor usage for the configured version

We performed several measurements with the new systems and showed its usefulness especially for machines with few processors.

Our current measurements are based on the generalized protocol stack described in Fig.1 and on some protocol subset specifications. But we are also interested in results for "real" systems. We already have results for the ISO Session and Presentation Layers [Hof94], and we are currently implementing an application layer protocol for multimedia systems using our technology [KFE94].

We are also thinking about a Quality of Service-based mapping of Estelle modules to processors. Thus, the configuration would become more requirement-driven and could be controlled by the user. Currently, it is driven by actual module load and does not take into account any QoS specifications.

## References

[Ber94] Nicole Beriér. Ein Werkzeug zur effizienten Implementierung von Kommunikationsprotokollen auf Multiprozessorsystemen. Master's thesis, University of Mannheim, Praktische Informatik IV (in German), to appear in December 1994.

[BG93] Jan Bredereke and Reinhard Gotzhein. Increasing the concurrency in Estelle. In Richard L. Tenney, Paul D. Amer, and Ümit Uyar, editors, *Formal Description Techniques VI – Forte'93, Boston, USA*. Participants' proceedings, October 1993.

[BZ92] T. Braun and M. Zitterbart. Parallel Transport System Design. In Danthine and Spaniol [DS92], pages H3:1–H3:16.

[CT90] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *SIGCOMM '90 Symposium Communication Architectures & Protocols*, pages 200–208, Philadelphia, September 1990.

[DDK+90] W. Doeringer, D. Dykeman, M. Kaiserswerth, B. Meister, H. Rudin, and R. Williamson. A Survey of Light–Weight Transport Protocols for High–Speed Networks. *IEEE Transactions on Communications*, 38(11):2025–2039, November 1990.

[DS92] A. Danthine and O. Spaniol, editors. *4th IFIP conference on high performance networking*, Liège, 1992.

[FH93] Stefan Fischer and Bernd Hofmann. An Estelle Compiler for Multiprocessor Platforms. In Richard L. Tenney, Paul D. Amer, and Ümit Uyar, editors, *Formal Description Techniques VI – Forte'93, Boston, USA*. Participants' proceedings, October 1993.

[HEHK92] B. Hofmann, W. Effelsberg, T. Held, and H. König. On the Parallel Implementation of OSI Protocols. In *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, Tucson, Arizona, February 1992.

[Hof94] B. Hofmann. *Deriving Efficient Protocol Implementations from Estelle Specifications*. PhD thesis, Universität Mannheim, Praktische Informatik IV, 1994. (in German).

[HP91] N.C. Hutchinson and L.L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, January 1991.

[ISO89] Information processing systems – Open Systems Interconnection – Estelle: A formal description technique based on an extended state transition model. International Standard ISO 9074, 1989.

[JJ89] C. Jard and J. M. Jézéquel. A Multi–Processor Estelle to C–Compiler to Prototype Distributed Algorithms on Parallel Machines. In E. Brinksma, G. Scollo, and C. A. Vissers, editors, *Protocol Specification, Testing, and Verification, IX*, pages 161–174. IFIP WG 6.1, Elsevier Science Publishers B.V. (North–Holland), Amsterdam, 1989.

[KFE94] Ralf Keller, Stefan Fischer, and Wolfgang Effelsberg. Implementing Movie Control, Access and Management - from a Formal Description to Working Multimedia System. In Liba Svobodova, editor, *Int.Conference on Distributed Computing Systems – ICDCS14, Poznan, Poland. Participants' Proceedings.* IEEE, June 1994.

[MT93] A. Mitschele-Thiel. The DSPL Programming Environment. In *Proc. Conf. on Programming Models for Massively Parallel Computers, Berlin.* IEEE Computer Society Press, September 1993.

[PB93] Jean-Daniel Pouget and Helmar Burkhart. Performance Studies on the KSR1. In Robert Schumacher, editor, *One Year KSR1 at the University of Mannheim – Results and Experiences, Technical Report No.35/93.* Computing Center, University of Mannheim, 1993.

[Pet91] D. Peter. Entwurf, Realisierung und Integration eines Protokolls zur verteilten Ausführung von Estelle–Spezifikationen. Master's thesis, Universität Hamburg, Februar 1991.

[PPVW92] T. Plagemann, B. Plattner, M. Vogt, and T. Walter. A Model for Dynamic Configuration of Light–Weight Protocols. In *IEEE Third Workshop on Future Trends of Distributed Computing Systems*, 1992.

[PS92] T.F. La Porta and M. Schwartz. A High–Speed Protocol Parallel Implementation: Design and Analysis. In Danthine and Spaniol [DS92], pages C3:1–C3:16.

[SS93] Rachid Sijelmassi and Brett Strausser. The PET and DINGO tools for deriving distributed implementations from Estelle. *Computer Networks and ISDN Systems*, 25(7):841–851, 1993.

[Svo89] L. Svobodova. Measured Performance of Transport Service in LANs. *Computer Networks and ISDN Systems*, 18(1):31–45, 1989.

[Zit92] M. Zitterbart. Parallel Protocol Implementations on Transputers — Experiences with OSI TP4, OSI CLNP, and XTP. In *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, Tucson, Arizona, February 1992.