A Kernel to Support Computer-Aided Verification of Embedded Software

Leon D. Grobler

THESIS PRESENTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE AT THE UNIVERSITY OF STELLENBOSCH.

Supervised by: Prof. P. J. A. de Villiers

April, 2006
Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and has not previously in its entirety or in part been submitted at any university for a degree.

Signature: ........................ Date: ........................
Abstract

Formal methods, such as model checking, have the potential to improve the reliability of software. Abstract models of systems are subjected to formal analysis, often showing subtle defects not discovered by traditional testing. Model checking is not widely accepted in industry, partly because a verified model does not guarantee the corresponding implementation will be correct. A special-purpose CSP-based programming language, LF, has been designed which is meant to support model checking directly. LF is designed to support the development of small to medium scale embedded software.

This thesis is concerned with the development of a kernel to support the LF language. The design of the kernel needs to support model checking directly while it is efficient enough to support real embedded applications. The main technique employed to allow model checking directly on compiled code, is to execute processes in indivisible steps, known as actions. Actions lead to fewer states, but reduce responsiveness to interrupts.

Performance measurements of the efficiency as well as interrupt response are presented and evaluated. The suitability to model checking at the code level was also examined, with promising results.
Opsomming

Verifikasietegnieke, soos modeltoetsing, het die potensiaal om die betroubaarheid van sagteware te verbeter. Tydens modeltoetsing word ’n abstrakte model van ’n stelsel aan ’n formele analyse onderwerp. Die analyse kan dikwels subtiele defecte uitwys wat nie maklik deur tradisionele toetsing gevind word nie. Modeltoetsing het nog nie wye aanvaarding in die industrie ontvang nie, gedeeltelik omdat ’n geverifieerde model nie noodwendig waarborg dat ’n implementasie daarvan afgelei word ook korrek sal wees nie. ’n Taal, LF, is ontwikkel om verifikasie deur modeltoetsing direk te ondersteun. Die taal is gemik op die ontwikkeling van klein- en mediumgrootte ingebedde stelsels.

Die ontwikkeling van ’n ondersteuningsomgewing vir LF is die fokus van die tesis. Die ontwerp van die ondersteuningsstelsel moet modeltoetsing direk op implementasies toelaat terwyl dit steeds effektief genoeg is om werlike ingebedde stelsels te ondersteun. Die belangrikste tegniek wat toegepas word om modeltoetsing van vertaalde kode moontlik te maak, is om prosesse uit te voer in ononderbreekbare stappe, wat aksies genoem word. Aksies verminder die aantal state wat deur die programtoetsers ondersoek moet word, maar maak die stelsel minder reaktief tot onderbreking.

Die resultate van metings van die effektiwiteit van die ondersteuningsstelsels, en ook metings van die reaksietyd in die hantering van onderbrekingen word gebied en bespreek. Die toepaslikheid daarvan om direk op vertaalde kode modeltoetsing toe te pas is ook ondersoek, met belowe resultate.
Acknowledgements

I am grateful to all the people who have assisted and supported me throughout my studies, especially:

- Prof. Pieter de Villiers;
- Frank van Riet, Riaan Swart and Erick Gerber for their work on the LF project;
- My parents, family and friends;
- The Department of Computer Science;
- The National Research Foundation (NRF) and the University of Stellenbosch for financial assistance.
Contents

Abstract iii

Opsomming iv

Acknowledgements v

1 Introduction 1

1.1 Goals of the Project 2

1.2 Thesis Outline 2

2 Background 4

2.1 The LF Language 4

2.2 Supporting CSP-based Languages 7

2.3 Microkernel Design 9

2.4 Processes 10

2.4.1 Memory Management 11

2.4.2 Protection 14

2.4.3 Scheduling 14

2.5 Interprocess Communication 16

2.5.1 Message Passing 16
2.5.2  Guarded Communication Commands ......................... 17
2.5.3  Conditional Receive ........................................ 17
2.5.4  Mobile Communication ...................................... 18
2.6   Device Drivers .......................................................... 18
2.7   The LF Model Checker ............................................... 20

3  The LF Kernel ............................................................. 24
   3.1  Supporting the LF Language ...................................... 25
   3.2  The LF Kernel .......................................................... 26
       3.2.1  Memory Management .......................................... 26
       3.2.2  Support for Processes ........................................ 28
       3.2.3  Calling Processes .............................................. 34
       3.2.4  Inter-process Communication ............................... 35
       3.2.5  Complex Messages ............................................ 38
       3.2.6  Conditional Receive ........................................... 40
       3.2.7  Device Drivers .................................................... 41
       3.2.8  Scheduling .......................................................... 43
   3.3  Supporting Model Checking ........................................ 44

4  Evaluation ................................................................. 46
   4.1  Benchmark Methodology .......................................... 47
   4.2  Overhead of Actions ................................................. 48
   4.3  System Calls ........................................................... 49
   4.4  Implementing System Calls ....................................... 52
   4.5  Memory Management ............................................... 53
   4.6  Supporting Processes ................................................ 55

vii


4.6.1 Scheduling ................................................. 55
4.6.2 Calling Processes ........................................ 56
4.7 Message Passing Communication ............................... 59
4.8 Interrupt Latency ............................................ 67
4.9 Model Checking ............................................ 68

5 Conclusion .............................................. 71

5.1 Efficient Kernel Support .................................... 71
5.2 Supporting Model Checking ................................ 72
5.3 Future Work ................................................ 73
5.4 Final Remarks ............................................... 74
List of Tables

1 Measurements of execution times for different implementations of action length .............................................. 49
2 Performance of assembly language and oberon system call implementations 52
3 Standard static memory usage .............................................. 53
4 Round-robin and earliest deadline first schedulers .............................................. 56
5 Comparison of called process and procedure call (Activation) .............................................. 58
6 Comparison of called process and procedure call (Termination) .............................................. 58
7 Performance of message passing for normal and SELECT communication 60
8 State space generated by LF model checker versus SPIN model checker 68
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Producer-consumer system in LF</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>Data structures used for block allocation</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>Representation of states in the LF model checker</td>
<td>22</td>
</tr>
<tr>
<td>4</td>
<td>Memory layout on IBM personal computers</td>
<td>27</td>
</tr>
<tr>
<td>5</td>
<td>Layout of a process activation record</td>
<td>28</td>
</tr>
<tr>
<td>6</td>
<td>Comparison of executing under different versions of the kernel</td>
<td>30</td>
</tr>
<tr>
<td>7</td>
<td>Producer-consumer system in LF with added sender identification</td>
<td>39</td>
</tr>
<tr>
<td>8</td>
<td>Installing an interrupt handler</td>
<td>42</td>
</tr>
<tr>
<td>9</td>
<td>Examples used to measure effect of action length on performance</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>The null system call</td>
<td>51</td>
</tr>
<tr>
<td>11</td>
<td>Procedure call compared to calling processes</td>
<td>57</td>
</tr>
<tr>
<td>12</td>
<td>Data transmission rate of normal and SELECT communication</td>
<td>59</td>
</tr>
<tr>
<td>13</td>
<td>Code used to measure performance of guarded communication</td>
<td>61</td>
</tr>
<tr>
<td>14</td>
<td>Code used to measure performance of conditional receive in multiple processes</td>
<td>62</td>
</tr>
<tr>
<td>15</td>
<td>SELECT communication under various conditions</td>
<td>65</td>
</tr>
<tr>
<td>16</td>
<td>Complex messages with and without kernel support</td>
<td>66</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

An important part of software development is to ensure that specifications are satisfied. Unfortunately, this issue is often neglected in industry, especially when deadlines are approaching. At best, systematic testing techniques are used to detect as many defects as possible before a product is released.

Formal methods have the potential to improve the situation, but there are still many practical problems to be solved. One successful method is known as model checking. The basic idea is to develop an abstract version of the software that can be subjected to formal analysis. This analysis is based on automatic state space exploration, a technique that can show up subtle defects that would normally be missed during traditional testing. Even so, model checking is still not widely accepted in industry. One reason is that a verified model does not guarantee that the corresponding implementation is perfect.

In principle, this problem can be attacked by applying state space exploration directly to the implementation code, but the amount of detail can be overwhelming—even for relatively simple systems. Furthermore, model checking is complicated by many features of programming languages that are in general use. As an experiment, we have designed a special-purpose programming language, LF [34], which is meant to support model checking directly. It is a simple, process-based language that is based on CSP [18]. LF is designed to support the development of small to medium scale embedded software such as a router or network monitor. This application area is particularly attractive because (1) such systems are often reasonably small, (2) embedded software
is often concurrent and model checking is particularly suitable for detecting concurrency defects, and (3) embedded software is difficult and expensive to test because of limited hardware resources.

As a concurrent language, LF requires a runtime support system. The kernel should provide functionality such as process scheduling, servicing of interrupts and inter-process communication.

1.1 Goals of the Project

The LF project aims to allow the verification of the implementation code of embedded software using model checking. This thesis deals with the design and implementation of a suitable kernel to support the LF language. Although the design of runtime support systems for CSP-based languages has been investigated before, there are two important considerations that make this project worthwhile:

- The design of the kernel supports a model checker directly. This means that sufficient information of every system state can be gathered during the execution of LF programs to allow automatic detection of software defects such as deadlock, buffer overflows and various problems related to synchronisation. The most important goal was to reduce the number of states generated.

- The kernel is efficient enough to support real embedded applications despite the constraints placed on the design by supporting model checking.

This thesis describes the problems encountered and the solutions that were devised to implement the LF runtime support.

1.2 Thesis Outline

Relevant issues regarding implementation of CSP-based languages are discussed in Chapter 2. In addition, the main concepts of the LF language are described in enough detail to allow discussion of the requirements to be met by the kernel. In Chapter 3 the design of the kernel is described and motivated. The focus is on the problems encountered and their solutions. The performance of the kernel was measured
and these results are presented in Chapter 4. In Chapter 5 the success of the kernel design is evaluated in terms of the goals stated above. Finally, some refinements and extensions are described as future work.
Chapter 2

Background

This thesis is concerned with the design of a runtime support system for a CSP-based language, LF. In general such languages require runtime support for processes communicating by message passing. Various such languages have been implemented for different purposes and with differing support requirements. LF is intended as an implementation language for embedded systems. As implementation language, it needs efficient support for communicating processes as well as access to devices and interrupts. Similar requirements are made of operating system kernels. Experience in operating system microkernel design is therefore relevant to the LF kernel.

LF is also intended for model checking at the machine code level, which allows the correctness of implementations to be verified. A general understanding of model checking and how it can be applied to LF is appropriate before considering support for the language.

In this chapter (1) an overview of the LF language will be provided, (2) the relevant literature on supporting CSP-based languages, including such support provided by microkernels, and (3) the basic techniques of the LF model checker will be discussed.

2.1 The LF Language

The LF language has been designed with model checking in mind. The syntax is largely based on Wirth’s Oberon language [30]. LF can be seen as a simplified version of Oberon with CSP-style concurrency added, as concurrency is often needed in embedded
software. A concurrent language also makes it possible to exploit one of the main strengths of model checking—the detection of defects in concurrent systems.

A simple example will be used to describe how LF is executed and verified. The example given in Figure 1 illustrates communication between two processes. Module initialization code is executed for each imported module, executed in the order they are imported, before execution of the main module starts with its body. In this case initialization of the imported output module Out (line 2) executes first, followed by the execution of lines 29–33. A port type Msg defines all messages that may be transmitted via the channel used in this module (line 3). Message types allow type checking to be performed on all communication commands at compile time. The variable port is declared to store a reference to the communication channel that connects the sender and receiver (line 4). The channel is created when the NEW command is executed (line 30)—it returns a reference to the channel via the parameter port. The processes Sender and Receiver are activated by executing the CREATE command (lines 31, 32). Processes created in this way execute concurrently.

When the Sender and Receiver processes are created, they are connected to a specific channel by specifying port as parameter. Process Sender transmits a sequence of integer values via the channel to process Receiver (lines 9–12). The last value is followed by the special message end to signal that the process has finished its job (line 13).

Process Receiver is a little more complex. It uses a WHILE loop with a nested SELECT command to receive either a numerical value (with symbol data attached, line 23), or the end signal (without data, line 24). The WHILE and REPEAT loops work exactly as in Oberon and most conventional programming languages. The SELECT command is used to allow reception of messages of different types via the same channel. A SELECT controls execution of different communication commands according to the type of messages available. Each WHEN clause in a SELECT command has a guard—in this case a simple communication command—which determines whether the command sequence following the THEN keyword may be executed or not. In the given example, the first guard of the SELECT is enabled if a message marked with symbol data is available. Similarly, the second guard is enabled when the end symbol is available, which then allows the process to proceed to its end command (line 27). Processes are terminated when reaching their end commands (lines 14, 27 and 33).

Some LF language features have been omitted from this initial discussion. Specifically,
MODULE ProdCons;
IMPORT Out;
TYPE  Msg = [data(INTEGER), end];
PORT  port : Msg;

PROCESS Sender(out : Msg);
VAR i : INTEGER;
BEGIN
  i := 0;
  WHILE i < 100 DO
    out ! data(i); INC(i)
  END;
END Sender;

PROCESS Receiver(in : Msg);
VAR x : INTEGER;
more : BOOLEAN;
BEGIN
  more := TRUE;
  WHILE more DO
    SELECT
      WHEN in ? data(x) THEN Out.Int(x)
      WHEN in ? end THEN more := FALSE
    END
  END;
END Receiver;
BEGIN
  NEW(port);
  CREATE Sender(port);
  CREATE Receiver(port)
END ProdCons.

Figure 1: Producer-consumer system in LF
specialised communication primitives and commands allowing low level access to hardware are among these. Additional features will be introduced as they are required in further discussions in the thesis.

2.2 Supporting CSP-based Languages

Several concurrent languages have been developed based on the CSP framework, originally specified by Hoare in 1978 [17]. The LF language is based on such a language, Joyce [8], and strongly influenced by occam [26]. Most languages inspired by CSP do not follow the CSP specification strictly. For example RBCSP, described by Roper and Barter [31], uses asynchronous buffered communication. Planet, a language aimed at distributed programming, allows shared memory between nested processes on the same processor [13]. Changes to the feature set also make different demands on the runtime support system.

The occam language was developed as programming language for the INMOS Transputer [26]. The Transputer hardware allows processes communicating by message passing to be supported directly. Due to the specialised hardware it was intended for, the occam language is aimed at constructing highly parallel systems of communicating processes. Three versions of occam have been developed, each new version adding additional functionality.

A short overview of the runtime support requirements of the most recent revision of occam, occam 3, will now be given. Occam is characterized by fine-grained concurrency; even individual commands are executable in parallel.

Interprocess communication happens by synchronous, unbuffered message passing supported by hardware channels between processors. The original occam allowed only base types as messages, while structured types are possible in the second revision. The final version allows complex message constructions, called protocols. Protocols allow constructions of messages as sequences of messages, but also more complex rules for the transmission of messages between processes. Uni-directional channels are used to identify communication partners, although such channels can be shared by only two processes.

Any occam process can be given a name and instantiated like a procedure call in a sequential language. Named processes can be instantiated either sequentially or in
parallel. Recursive calling of processes is not possible. Unlike CSP, occam does allow sharing of variables, although this is only possible under strict rules to prevent race conditions.

Non-determinism is specified using a guarded alternative command, ALT. The ALT command allows non-deterministic choice over several possible events. Each alternative must contain either a communication command or a special SKIP process and an optional boolean condition. An alternative is enabled if the optional boolean condition evaluates to true and the communication is possible or the special SKIP process, which is always enabled.

Occam allows low-level access to hardware as well as providing access to interrupts. Interrupts are indicated by the standard message passing interprocess communication facility.

Joyce is a language based on the Pascal syntax [7, 8]. Processes are instantiated similarly to procedure calls, and recursion is possible. No shared memory is allowed, which also excludes reference parameters and pointers. All processes execute concurrently, which prevents function processes as it is not possible to determine when or if a process will terminate and return its result.

Joyce is aimed at distributed systems suited to synchronous message passing. Message passing is the only method of sharing information between processes, as there is no shared memory. Dynamically allocated channels, data structures accessed by port variables, are used to connect communication partners, and can be shared between multiple processes. Each channel can define multiple alphabet symbols, which are type checked and act as separate communication channels. In addition to the basic types, messages consisting of structures, or the empty message, are also allowed.

A generalised guarded communication command, allowing both input and output commands as well as an optional boolean condition is available. This command can be used to specify behaviour for a process which depends on the availability of messages from other processes in the system. The boolean condition can be based on the value of the message to be received. This powerful receive command, known as conditional receive, allows reception of a message to be based on its contents.

The purpose of a language determines the focus of its runtime support system. Performance is not always important, such as for Planet [13], which was supported on
interpreted virtual processors while examining the suitability of the language to specifying distributed systems. Promela is the specification language for the SPIN model checker where again execution performance is of less importance than the performance of the model checker and efficient model specification [19, 20]. However, when languages are intended as implementation languages, performance of the support system becomes important.

Support for such languages has to be tuned to the specific requirements of the CSP model of concurrency. For example, CSP-based programs generally have many simple communicating processes cooperating to perform more complex tasks. This requires the runtime support to efficiently support many processes, and many context switches between these processes. In this framework, the support for message passing communication used for fine-grained interaction, needs to be efficient. CSP based languages also require some specialised support functions, such as support for the guarded communication primitives of CSP. One solution which provides good performance, is the use of specialised hardware, in the case of the occam language and the INMOS Transputer. On the Transputer, processes can be run on separate processors with interprocess communication supported by hardware. There has since been implementations supporting occam on general purpose hardware, with a runtime support system [38].

2.3 Microkernel Design

The basic support required by languages derived from CSP is (1) support for concurrent processes and (2) interprocess communication by message passing. These requirements correspond to the very basic support functions provided by most operating systems, generally provided by a kernel. More specifically, these basic functions correspond to those provided by microkernels, where the in-kernel features are kept to a minimum, implementing outside the kernel whatever possible. Microkernels usually also provide an abstract interface to the underlying hardware, which improves system structure and portability. The techniques developed in microkernel design, especially for providing efficient interprocess communication, should be well suited to the support system for a CSP-based language.

Various microkernels have been created and techniques developed to provide the required support functions [5, 6, 12, 14, 21, 22, 28, 35]. Microkernels usually support processes, interprocess communication, memory management and protection as well as
controlling and providing access to peripheral devices.

Early microkernels were developed from monolithic kernels by reducing the features and size. This so-called first generation includes kernels such as Amoeba [35], Chorus [6], L3 [21], mach [28] and V [12]. Performance was often a concern. The performance of interprocess communication in these first generation kernels was criticised, for example, by Liedtke [23] as the source of poor performance. Due to performance concerns, critical drivers and servers remained part of the kernel. This integration reduces the benefits from encapsulation, security and flexibility provided by the microkernel architecture [23]. However, with version V3 of the Chorus kernel, providing binary compatibility with UNIX, performance was comparable to well established monolithic kernels [6].

A radical redesign based on reduced abstraction and greater hardware dependence—the second generation—led to greatly improved performance from minimalistic designs. The second generation includes L4 [22, 23], the exokernel architecture [14] and the SPIN operating system [5].

In the exokernel architecture, the kernel only controls and manages access to resources. All standard kernel functions are then provided by application-level, untrusted libraries. Although promising performance results have been shown using this architecture, the focus of the following discussion will be on a more traditional kernel approach. The traditional kernel is more compatible with the design choices of the rest of the LF project—integrating a model checker and performing verification on compiled code.

The following sections will present the requirements of a CSP-based language in more depth and explain how these requirements can be met by a runtime support system.

2.4 Processes

In general a process can be seen as the abstract representation of a running program. This abstraction provides advantages when designing and implementing systems. For example, systems can be designed as separate, interacting processes where it becomes possible to argue about the requirements and expected behaviour of each process separately from the others. With a clear specification of the function and interface of a certain process, claims about the correctness of the implementation of the process can be evaluated in terms of this specification, isolated from the rest of the system.
Microkernels supporting processes generally provide memory management, protection, scheduling and interprocess communication.

2.4.1 Memory Management

Block allocation of free memory is a basic requirement of operating system kernels. Generally, facilities are available in the kernel to allocate and dispose of blocks of memory. Process creation and termination correspond to allocation and release of block structures, which store process state information. Standard solutions can be applied to tracking free blocks, using free lists or bit maps, combined with algorithms such as first-fit or best-fit. However, for the specific case of dynamic process creation and termination of CSP based languages, a more efficient solution has been proposed by Brinch Hansen [9], and applied in his implementation of SuperPascal.

Brinch Hansen’s strategy is based about some assumptions on the way in which blocks are allocated and released:

- only fixed-length contiguous memory blocks need to be allocated,
- once allocated, blocks will never be relocated,
- blocks will only be released when they are no longer referenced,
- blocks will need to be allocated and released in an unpredictable order,
- only a limited number of block sizes are required, and
- blocks of the same size are repeatedly allocated and released.

These assumptions do not hold for a general purpose operating system which allocates blocks for an unbounded number of processes of varying sizes. They do, however, hold for dynamic activation and termination of processes of a single block structured program.

Brinch Hansen’s memory allocation strategy intends only to solve the problem of efficient memory allocation for unbounded allocation of concurrently executing processes. The solution improves performance of the allocator by reusing previously allocated memory for requests of the same size. Data structures are used to maintain pools of all previously allocated blocks for each required size. Pools are identified by unique indexes assigned to each required size at compilation. This is possible as the method
Figure 2: Data structures used for block allocation.
is only applied to a fixed number of block sizes known at compile time. These sizes
include the sizes of all the process activation records defined that the system supports.
Initially all heap memory is free and the pools are empty. When a new allocation is
required, the corresponding pool is examined and two allocations are possible:

(a) If the pool is empty, a new block is allocated from the heap.

(b) If the pool is not empty, a block is allocated and removed from the pool.

When a block is no longer needed and needs to be released, it is added to the pool of
previous allocations for the specified size.

Figure 2 shows the memory allocator for three block sizes in the execution of a block
structured program after some allocations and disposing of process activation records.
In the figure pool number 1 holds three previous allocations, and pool number 3 two,
while pool 2 is empty. The previous allocations, along with any further active process
activation records, are stored in the allocated portion of the heap. The significant
difference from standard allocators is that, upon termination of a process, the memory
is not returned to the heap but rather stored in the separate pools. Each pool is then
accessed to reuse these blocks for subsequent allocations of the corresponding size. For
any allocation or disposal the unique pool identifier is used to identify the pool which
needs to be accessed. For example, the memory required by a new activation of a
process with pool identifier 3 will be allocated as the first of the two blocks in pool
number 3. Activating a process with pool identifier 2 will require a new allocation
from the heap as the pool is empty. Upon termination this block will be stored in its
pool for reuse by a later activation.

The allocation method intends to efficiently reuse previously allocated blocks for similar
reallocations. Once allocated, a block remains either in use or in the pool of previously
allocated blocks, which removes the need for standard hole management techniques in
the memory manager. Blocks can, however, only be reused when allocating blocks of
the same size. This can result in a situation where a block of a certain size cannot
be allocated, but where all memory is not currently in use. This happens when all
available heap memory has been allocated either for active use or reserved in the pools.
In this situation, it is necessary to recover available memory previously allocated to the
pools. This allocation strategy is highly efficient during normal operation—comparable
with the stack management of sequential procedures [9].
2.4.2 Protection

General purpose operating system kernels that support untrusted processes are required to provide memory protection to prevent malicious attacks on or accidental corruption of the internal data of a process.

Separate address spaces, implemented by either paging or segmentation, are usually employed to prevent such interference. This protection, however, comes at a cost. During context switching, additional operations need to be performed to manipulate the segmentation or paging hardware providing the separate address spaces.

In a paged environment, each virtual memory access requires additional memory references to load physical addresses from the page tables. These additional references are generally reduced by making use of a translation look-aside buffer (TLB). Whenever a context switch is done and page tables are altered, the TLB needs to be flushed. Apart from the additional task of flushing the TLB, a new process will start with an empty TLB. Initial memory references by the process will not be available in the TLB and will have to be translated using page tables. If processes are scheduled often with little work done between context switches, this penalty can be high.

2.4.3 Scheduling

The simplest solution to scheduling is to simply execute each task one after the other. This is known as non-preemptive, first-come-first-served scheduling. Scheduling only needs to be performed when a process can no longer continue or voluntarily yields control of execution. In general, processes cannot be allowed to occupy the processor indefinitely, because it makes it impossible to ensure that requirements of processes concerning timing will be met.

For most concurrent processes response time is also an important consideration. This means that the active process needs to be preempted after executing for a certain period. To prevent processes from executing for too long, a clock is generally used to interrupt the running process periodically. At each interruption the scheduler is invoked to either preempt the process or allow it to proceed. By increasing the frequency of the timer interrupt, improved response time can be gained, but at the cost of greater overhead.

Preemption means that a process can be interrupted at any point. This requires the
state of the process to be stored at the point of interruption. This state is restored when this process is rescheduled. This state includes the program counter, the stack and register values of the interrupted process. Preemptive scheduling presents the problem of race conditions which is not present with non-preemptive run to completion scheduling. Discussions of the implementation details and applications of various scheduling algorithms are provided in many operating system textbooks, such as [27, Chapter 4].

A simple, widely used algorithm to preemptively schedule processes is round robin scheduling [36]. With round robin scheduling a process is run for a given time interval before it is preempted and the next process is scheduled. Processes are scheduled from the front of the ready queue, and returned to the back of the queue whenever preempted.

In some cases, processes are not all of equal importance, and should be handled accordingly by the scheduler. Assigning priorities to processes according to importance allows priority scheduling. Whenever a process needs to be scheduled, the highest priority schedulable process is selected for execution.

Programs specified in CSP-derived languages should not make any assumptions about the specific scheduling algorithm implemented [17]. Non-determinism in a process is explicitly specified, and the programmer should ensure that valid behaviour is achieved regardless of the scheduling order. Therefore, an appropriate choice of scheduling algorithm depends only on the requirements of the systems implemented in the language.

In real-time systems timing is critical. In a real-time system, tasks executed in response to events or tasks performed periodically need to be completed within a certain time limit. Real-time systems can be categorised as either hard real-time or soft real-time [36, Chapter 2]. Hard real-time systems are systems where meeting every deadline is critical, while in soft real-time systems, occasionally missing a deadline is acceptable. The seminal paper on the subject by Liu and Layland [24] presents a formal basis for scheduling in real-time systems. Solutions to real-time scheduling are based on fixed and dynamic priorities. A schedule is said to be feasible if the scheduler can provide an execution order in which the deadlines of all the processes are met. It was shown that with a fixed-priority scheduler, feasibility can, in general, only be guaranteed with a maximum processor utilisation of 70 percent. Using dynamic-priority scheduling, full processor utilisation can be achieved.
2.5 Interprocess Communication

Cooperating processes require methods of sharing information. Two standard groups of solutions, either using shared variables, or message passing are used. Sharing of variables allows information sharing to be fast but requires additional synchronisation to control contention for shared data. When uncontrolled concurrent access to shared memory is allowed, race conditions can occur. Well-known solutions to synchronising of shared memory access are commonly provided. The standard text-book solutions include mutual exclusion with busy waiting, semaphores, and monitors [36, Section 2.2]. Using message passing prevents race conditions by combining sharing of information with synchronisation. However, because messages are transferred by copying data, efficiency is generally a concern. An advantage of designs based on message passing is that such designs translate well to distributed memory systems.

2.5.1 Message Passing

CSP is based on synchronous message passing between processes. This is blocking communication between processes without buffering. Messages are transferred by copying data from a sender to a receiver. Synchronisation of the sender with the receiver is performed by the interprocess communication facilities of the kernel.

Copying of data is generally an expensive operation and most implementations based on message passing contain various optimisations to improve performance. For the V microkernel, Cheriton specified a method of using registers to improve performance [11]. While this works well for a message sent to a blocked receiver, it does not improve network communication or when the sending process needs to be blocked where the contents of the message needs to be buffered. Another optimisation in the V kernel is in the design of its network protocol. In V, message passing is done as remote procedure call (send-receive-reply). The VMTP protocol was designed to handle this specific communication efficiently. In some kernels, message passing is closely coupled with memory management to allow optimisations based on the paging system such as copy-on-write [6, 21].

Although such optimisations improve the performance of interprocess communication in first-generation kernels, still significantly faster interprocess communication was provided by second generation kernels. The second-generation kernels achieved improved
performance by (1) reducing the abstractions provided by the kernel and (2) implementing these abstractions in a hardware-specific way [22].

2.5.2 Guarded Communication Commands

Originally CSP restricted guarded communication to only input commands [17]. This form of guarded communication allows simple and efficient implementation. The guarded communication command was, however, extended to allow both input and output commands in the revised specification of CSP [18]. Some algorithms can be more conveniently described with guarded output commands, such as the bounded buffer and dining philosopher algorithms of [17].

Efficiently implementing a generalised (both input and output) guarded command is, however, a difficult task. Buckley and Silberschatz specified criteria for implementing generalised guarded commands efficiently [10]. They further examined several suggestions under these criteria, showing failures in each, before presenting their proposed solution. Even this solution has been described by Raynal as ‘complicated and cumbersome’ [29].

Some efficient implementations have been provided for restricted versions of the general guarded command. Silberschatz provided an efficient solution with the limitation that processes have specific master or slave roles and that input-output guards are only allowed by a slave communicating with its masters [32]. Silberschatz later simplified this by only requiring that guarded commands be limited to ports owned by the specific process [33]. In general, efficient guarded communication can be achieved as long as communication between two guarded communication commands is disallowed.

2.5.3 Conditional Receive

Conditional Receive describes the ability of a process to decide whether or not to accept a message depending on its value. This is an extension of the guarded communication, which requires additional functionality from the runtime support system. Specifically, the runtime support needs to provide a method to examine the value of a message before copying it.

A simple, but inefficient solution was used in the implementation of Joyce. The solution involves repeatedly scheduling the process attempting the conditional receive until a
suitable message becomes available. This solution is described in detail in [8]; despite its inefficiency Brinch Hansen nonetheless considered it useful enough to include in Joyce.

2.5.4 Mobile Communication

Message passing performance depends on the size of the message, because message passing implies copying data. Performance improvements can therefore be gained by reducing the size of messages. One way to do this is by transferring a pointer to a data structure rather than transferring the structure itself, providing significant performance improvements for larger messages. Unfortunately by using this method, as for example in JCSP, a CSP implementation for Java, data is shared between processes which compromises security and safety [3].

The same advantages can be gained by using a safer variation. The difference is that once the pointer is transferred, both processes should not be allowed concurrent access to the same data. That is, once a message is sent, the local reference to the data should no longer be valid for the sender. This method, called mobile communication, has been implemented for occam [4]. Mobile communication describes messages passing where the data is moved from the sender to the receiver, leaving the source variable undefined after such a mobile send command. Unfortunately, this requires either runtime checks on all variable accesses or flow analysis by the compiler to ensure that undefined variables are not accessed.

2.6 Device Drivers

An important requirement of system software is the ability to access peripheral devices. The system kernel needs to facilitate this interaction between the system and devices. The interaction is managed by device drivers.

The purpose of device drivers is to hide the detail and peculiarities of the device and to provide a simplified interface to it. A driver is therefore specific to both the device itself as well as the kernel it needs to interface with. This also means that a developer needs detailed knowledge of both the system and the hardware itself when developing a driver. As such, development of device drivers is complex and error prone.
Another problem with programming device drivers is that they require low-level facilities to communicate with the hardware. This is often achieved by writing the driver, or a portion thereof, in assembly language. This adds even further to the complexity of device drivers. Using a high level language can ease the programming of drivers. However, this is only possible with languages which include low-level facilities to interact with hardware.

Different strategies are possible for device driver design. Device drivers can be developed as either (1) in-kernel device drivers, (2) dynamically loadable device drivers, or (3) user-level device drivers [25]. Each method has certain advantages and disadvantages.

*In-kernel drivers* are compiled and linked into the kernel image. Such drivers have access to the entire instruction set and all kernel resources. This makes it possible for in-kernel drivers to be efficiently implemented. Clients can efficiently access in-kernel drivers by making use of system calls. Development of in-kernel drivers however presents some problems. Testing and debugging of additions and changes require a reboot of the system at each test run. Also, the flexibility and configurability of the system and drivers are limited with in-kernel drivers. Any changes to include or exclude drivers, requires a rebuild and reboot of the kernel [36, Chapter 2].

*Dynamically loadable device drivers* are modular drivers that can be loaded or unloaded at runtime. They can be compiled separately and loaded to configure the system as needed. Once loaded, they can operate as in-kernel drivers with access to all kernel resources. Being able to load and unload modules, however, significantly simplifies testing and debugging. The kernel does not have to be rebuilt or rebooted to activate and test changes [36, Chapter 2].

Another possibility is to develop *drivers in user space*. These drivers are normal user processes which can be loaded, unloaded and changed during runtime. These processes can benefit from the same memory protection that any other user process enjoy. Also, development of such drivers would gain from any debug and development tools available for the system.

User-level driver processes require special support from the kernel during runtime. Driver processes require access to interrupts as well as low-level I/O facilities. Micro-kernels commonly provide access to interrupts using the standard interprocess communication facilities available to user processes. This method is provided by, for example,
Mach 3.0 and L3/L4 which allow development of device drivers in user space [15, 22]. Similarly, clients communicate with the user-level drivers by using the standard interprocess communication (IPC) facilities provided by the kernel. The separation of the driver and kernel is unfortunately usually less efficient than a driver in the kernel, that has direct access to all required resources, such as the memory manager.

2.7 The LF Model Checker

An explicit state model checker is used to check LF programs. This involves a systematic search of the state space, executing one process at a time.

Traditionally, model checking is used to check correctness properties of a verification model—an abstract rendition of a given program. The goal with the LF model checker is more ambitious: to check the executable code directly. To do this, the kernel contains a model checker that analyses the execution of LF programs. The model checker is an optional component of the kernel and is only required during development.

The main problem, of course, is the amount of detail contained in real programs. To have any chance of success, the LF language is as simple as possible although it is expressive enough to allow implementation of real embedded systems. The focus is on the essentials needed for the chosen application area while constructs that complicate model checking, such as pointers and objects, were deliberately left out. The LF system provides a suitable test bench for investigating the use of model checking at the machine code level.

Explicit state model checkers rely on algorithms and data structures that have been developed over the past two decades. An in-depth coverage of the algorithms and techniques used in SPIN, a powerful explicit state model checker, is presented in [19, 20]. Here, we provide an overview of the specific techniques relevant to this project.

State exploration starts from the program’s initial state, at each step exploring every possible scheduling alternative one at a time. The ordered list of processes scheduled at each step of these steps forms the current path. The model checker automatically falls back when no further paths lead from a given state. Consequently, it is necessary to store all states along the current path. In addition, this allows the detection of cycles, which is important to guarantee termination of the algorithm. It also helps to store as many states as possible from previous paths because this helps to avoid redoing work
The current system state is defined as the combined values of all data variables and the value of the location pointer of each process. The information is stored in a data structure known as the state vector. An L/F module starts executing with the initial process from the module body as explained in Section 2.1. Additional processes are usually created and this causes the state vector to grow because space must be allocated for the location pointer and data variables of each new process. Similarly, the state vector shrinks when processes terminate. The state vector is implemented as a small array. This rather rigid strategy implies that the model checker must be recompiled if the state vector is too small for a given verification task. However, this is an accepted approach for explicit state model checkers for reasons of efficiency.

A state cache is implemented in the L/F model checker in the form of a hash table using as much memory as is available. Each entry in the hash table represents a unique path. When the table fills up, states encountered along previous paths may be discarded to make room for new ones. This does not affect the result of a verification run, although some work will be repeated if a discarded state is revisited. This strategy works better when combined with partial order reduction techniques that reduce the probability of revisiting a given state. This was first reported in [6].

Compact state representation is essential because even small programs can generate huge state spaces. Only a tiny fraction of the potential state space of a program is reached in practice and it is possible to exploit this by encoding each unique state as a set of hash keys. The basic idea is to store all unique combinations of values assigned to the location pointer and local variables of each process in a table. These tables are then used to determine whether a new global state has been reached or not. The technique, known as collapse compression, is described in detail in [19, 18-20].

It is generally sufficient to check only the table associated with the active process because only this process can modify the state vector. Some exceptions such as when messages are transferred to an inactive process do occur. The set of hash keys for each process is pre-computed and stored as a hash table, one per process. For fast lookups, the tables are organized as relatively small hash tables, one per process as shown in Figure 3.
Figure 3: Representation of states in the LF model checker
process is then passed through another hash function associated with the state cache to determine whether the newly generated state has been visited before. If it is indeed a new state, it is inserted into the state cache. As long as the hash tables for each process do not fill up, this technique works well and makes it possible to store millions of states in the main memory of a typical workstation.

New states are generated by executing the LF code. For example, when a process executes an assignment such as “\(x := x + 1\)”, the location pointer and the data variable \(x\) are modified. LF commands are translated into equivalent machine code in the usual way, but the LF kernel ensures that entire sequences of machine instructions are executed as indivisible operations, called actions. Each action represents a transition from one global state to another as far as the LF model checker is concerned. This strategy makes model checking at the machine code level possible by reducing the number of states to be explored dramatically. Examining the effects of this strategy on performance and interrupt response times and how this can be optimised will be covered in Sections 4.2 and 4.8.
Chapter 3

The LF Kernel

LF is a language for developing embedded systems. To reduce costs, both memory and processing power are often limited. LF programs consist of concurrently executing sequential processes, communicating by means of message passing, and since this is not supported by general purpose hardware, a runtime support system is required. Just as any embedded software, this runtime support system needs to conserve system resources, and therefore needs to be small, fast and memory efficient.

To develop reliable embedded systems, LF programs should be verifiable at code level. A model checker is built into the system for this purpose. Integration of a model checker into the runtime support system creates new requirements and challenges. The advantages of being able to verify actual executable code, however, makes it a worthwhile objective.

The main influences on the design of the LF runtime support system are therefore:

- requirements of a model checker: Design support for processes to minimise the state space.

- restrictions when designing for embedded systems: On embedded systems resources such as memory and processing power are limited. The hardware on embedded systems often lack certain technologies such as a memory management unit.

- efficient support of an implementation language: Support for an implementation language should perform comparable with other implementation languages on
similar hardware. Also, implementing real systems requires sufficient access to hardware devices.

The requirements of integrating a model checker into the LF kernel restrict the choices available for the kernel design. The integrated model checker is, however, integral to the success of the LF project. Therefore, these requirements have been accommodated while limiting, as far as possible, their effect on overall performance and kernel efficiency. Throughout the design of the kernel, the requirements of integrating a model checker have been weighed against the requirements of efficient kernel support.

3.1 Supporting the LF Language

The basic requirements of LF systems during runtime are: (1) support for dynamically created processes, (2) synchronous message passing, and (3) access to peripheral devices.

All LF systems are constructed as systems of processes, cooperating with message passing as necessary. Each process has its own private variables which cannot be shared with concurrent processes so that race conditions are avoided. LF does, however, provide the ability to call a process, which then executes sequentially and allows variables to be shared between the called process and its caller. As in CSP, Processes are the main abstraction mechanism and the nesting of processes provides different levels of abstraction. To minimise scheduling and context switching every command is not a process, in contrast to occam. A process always consists of a number of sequentially executed commands. In most programming languages pointers are available to reference memory in a shared heap. Avoiding pointers removes the problem of concurrent access to shared dynamic data. Indiscriminate use of pointers increases the state space and therefore reduces the possibilities for model checking.

Synchronous communication is the only means by which concurrent processes can exchange information. Communication is synchronised using ports and channels, or indirect naming. Channels are dynamically allocated and message type checking is performed during compilation. Channel definitions can define multiple alphabet symbols allowing multiple types of messages to be transferred over a single channel, reducing the number of channels needed to be shared between LF processes. Message types may include multiple comma separated values, similar to parameters in procedure calls. Each
alphabet symbol is queued separately allowing multiple processes to be synchronised on different symbols of the same channel. Queues of senders or receivers are used to match corresponding communication commands. However, only senders or receivers can be queued at any time. This is because additional senders (or receivers) would not be queued when receivers (or senders) are waiting; instead, they will be matched up one-to-one.

Device drivers for LF systems are developed in the LF language. Facilities are required to communicate with hardware ports, access memory mapped devices, and to be notified of and react to interrupts.

3.2 The LF Kernel

The current kernel is designed to support a single processor and is responsible for allocating memory for new processes, reclaiming memory when processes terminate, scheduling processes, transmitting messages between processes and reacting to external events such as system calls and hardware interrupts. The kernel needs to provide this support in such a way that model checking at the code level is possible.

3.2.1 Memory Management

An LF program consists of one or more modules and these are loaded into memory by the boot loader. The rest of physical memory is available to be allocated at runtime for various purposes. The first LF kernel was designed for IBM personal computers because such machines are widely available and also because the Intel 386 family of embedded processors are used in many applications. Segmentation is used to create a single linear address space. Memory is organised as shown in Figure 4. Paging was avoided in the design of the LF kernel providing a performance advantage as well as improving the portability of the kernel to a wider range of hardware platforms. Paging would require the use of a memory management unit not always available in the processors that are popular in industry.

The executable code for the LF kernel is loaded in the lower part of memory. The model checker—an optional kernel module—is loaded only when needed. The rest of memory below the screen in the case of the IBM PC is available to store the executable code of an LF program. The kernel allocates blocks of physical memory above the
Figure 4: Memory layout on IBM personal computers
Figure 5: Layout of a process activation record

1MB boundary—above the area reserved for the screen—to store the essential state information of each process. Channel records are also stored in this area and, if the model checker is loaded, this memory area must be large enough to store its internal data structures. A special development platform with as much memory as possible is therefore required to use the model checker.

The only dynamic memory requirements of LF systems are the allocation of process activation and channel records. Brinch Hansen’s specialised memory allocation strategy, presented in Section 2.4.1, can be applied as long as the accompanying assumptions are satisfied for all dynamic memory requirements. This strategy is preferred to standard allocation strategies as it provides a simple, efficient algorithm with low overhead [9].

The assumptions the strategy is based on, hold for processes activation and termination in a block structured language such as LF. Channels can be allocated by a process and shared with its children. All channels created by a process will be released whenever the process terminates. Similar to process definitions, the number and unique types of channel records are fixed at compile time. The number of channel record sizes is therefore limited, and the channel records follow a similar pattern of allocation and release to that of processes. The assumptions of applying this strategy also hold for channel records. As there are no other dynamic memory requirements in the LF kernel, no general purpose allocator is implemented.

3.2.2 Support for Processes

LF programs are constructed as systems of concurrent processes. Support for processes is provided by the LF kernel. This includes support for storing the state of a process, scheduling and dynamic process creation and termination.
A process activation record, allocated at process creation, is used to store the state of a process. The basic layout of a process activation record is shown in Figure 5. While a process is executing, its state is stored in the hardware registers of the processor and its data variables in memory. When a process stops executing for some reason, the kernel keeps all information needed to restart it in its activation record. Some of this information is used by the scheduler, some for interprocess communication and some during model checking.

The scheduler is activated by external events, which may be a hardware interrupt or a system call. A specific process must be activated in reaction to each external event. For example, a specific device driver—implemented as a process in LF—must react to each hardware interrupt. The latency between the time when the hardware interrupt occurs and the device driver is activated depends on the length of code that needs to be executed before the scheduler can be activated. As mentioned before, the sequential code in a process is executed in atomic units, called actions. Each action ends with either a communication command or a system call that voluntarily yields the processor. The interrupt latency can be minimised by having very short actions. Unfortunately, this not only leads to an increase in the state space, as explored by the model checker, but also an increase in the overhead of process support.

The Scheduling Cycle

By only allowing processes to be scheduled at fixed points—between actions—the state space that needs to be explored by the model checker can be drastically reduced. This is achieved because it limits the number of possible interleavings of concurrent commands by reducing the number of scheduling points. Unfortunately, this may introduce a delay in the response to interrupts, as we have just mentioned.

To implement actions with fixed scheduling points, interrupts cannot be handled in the normal way. Normally, an interrupt would cause the currently executing process to be suspended and execution transferred to the interrupt handler. To achieve predictable scheduling, interrupts can only be allowed to suspend execution of a process at predefined locations. In the LF system, interrupts are permanently enabled, but not necessarily handled immediately as they happen. Only once the current action has completed, are pending interrupts serviced. Therefore interrupt handling is delayed while completing the current action.
Figure 6: Comparison of executing under different versions of the kernel
With this method of interrupt handling, the interrupt response time depends on the action length. The response might be delayed up to the time to execute the maximum length action as well as the time taken to activate the interrupt handler. Because of this limitation on response time, it was decided to limit the length of the actions processes are divided into. The purpose was to provide acceptable interrupt response latency. The disadvantages of shorter actions were, however, more states during verification and higher overhead during execution. This overhead of executing the code, as divided into actions by the compiler was found by Swart to be substantial. Swart measured this to be as high as 75 percent in some cases [34].

Although some examples performed much better, it was still deemed necessary to attempt to improve on this performance. I examined an alternative implementation, in which the lengths of actions are increased to the maximum possible. This means scheduling points are only added when absolutely necessary. Such points would occur when processes are blocked, terminate, or when no progress is possible without interaction of another process. A scheduling point also occurs when no guards in a SELECT evaluate to true. Another process then needs to be executed to (possibly) activate one of these guards. As shown in Figure 6, in the original implementation (on the left), execution of every LF command is interleaved by a call to the LF kernel. If no system call is required between commands, an additional call to the scheduler is inserted. The changes made to the execution can be seen in the right-hand pane of Figure 6, where actions have been extended to their maximum possible length. In some cases this does not result in a reduction of scheduling points, as can be seen from the execution of the Receiver process in Figure 6.

An improvement in the performance should be expected from the alternative, because of reduced overhead. Longer actions are also an advantage when model checking. Longer actions means fewer possible interleavings of processes, resulting in a reduced state space.

Clearly, the disadvantage of longer actions is a reduced response time to external events such as interrupts. From this point of view, the original implementation is preferred. It was decided that if the response time was acceptable for standard hardware, the alternative would be preferred for its advantages both to overall performance and model checking. Measurements of the response time and performance of both schemes are provided and discussed in Chapter 4.

To ensure that interrupts can be serviced under all circumstances, it may be necessary
to insert additional scheduling points. It was decided that such a limit on the length of actions should be enforced by the compiler. The complications towards implementing such a compiler, such as the difficulties in determining the maximum execution times of actions, falls outside the scope of this thesis. It is, however, an important component that will be required for the success of the LF project.

**Process Representation**

The process activation record (AR) stores all information relevant to a specific instance of a process. The process AR contains the information used by the kernel to identify a process, support dynamic allocation and disposal of memory, and maintain the current program counter for the process. This information is stored in the following record fields in the header of the AR:

- **Size**: the total size of the AR in bytes
- **Action**: the next action that will be executed when the process continues
- **Status**: current state of the process in terms of schedulability
- **ProcessID, PoolID**: used respectively to identify a process instance and the pool to allocate the AR from
- **ChildCount**: number of child processes
- **Parent, Child and Sibling**: references to the ARs of the parent process, the head of the list of child processes, and the next sibling in the list the process belongs to
- **Channels**: a reference to a list of all channel records owned by this process.

The header also contains record fields for synchronised interprocess communication. These record fields are:

- **BufferAddr**: stores the address (source or destination) of the message being transferred
- **Data**: used as temporary storage for a message value calculated by an expression in send command
• *Qnext:* stores a reference to the next message in the queue for a specific channel and alphabet symbol

The record fields used for maintaining information relevant to scheduling are:

• *Deadline:* set to the nearest deadline for the current process (when using a deadline driven scheduler)

• *Next:* the next process in the list of ready processes

The AR represents the complete current state of a process as it contains both the information stored by the kernel as well as all parameters and local variables of the process. This is advantageous for model checking where the current state of a process can be gathered from a single contiguous block of memory. Since procedure invocation, dynamic memory allocation and pointers are not available in LF, memory can be allocated for all local variables in the activation record of the process. To access variables and parameters, a register is set to point to the start of parameters in the current AR by the kernel whenever a new process is scheduled. Local variables and parameters can then be referenced relative to this register.

Channel records are allocated dynamically as required by a process executing a NEW command. The process executing the NEW command becomes the owner of the created channel record. The channel record is linked to the owner process by a linked list referenced by the *Channel* record field of the AR. This is used to keep track of memory allocated for channel records. All memory allocated to a process, including memory for channel records, is marked for release when a process terminates. Such channel records can be shared between a process and all its child processes. This means the channel records can only be safely released once all child processes have terminated.

**Creating Processes**

Since LF is a concurrent language, the creation of new processes is more complicated than creating activation records for procedures in a sequential language such as Oberon, because it is not possible to use a run-time call stack. When sequentially activating procedures, memory can be allocated linearly on a stack. This allows parameters to be passed on the run-time stack before making the call to the procedure. Unfortunately, because processes can be activated and terminated in parallel, a more complicated
memory management scheme is required. The technique described in Section 2.4.1 is used for this.

Dynamic process instantiation also complicates parameter passing applied to processes. Two alternatives were evaluated to support parameters. One alternative passes parameters to a process on the stack when making the system call to create the process. This requires the parameters to be copied to the activation record, once allocated, by the system call. The second alternative, which avoids the costly additional copying, but requires two system calls. The first system call is executed to allocate an activation record for a new process and return its address. This address is needed to copy the parameters of the newly created process from the activation record of its creator to the activation record of the newly created process. A second system call is then executed to register the new process as ready to be executed. For the LF kernel the second alternative is preferred because of the relatively cheap cost of system calls in LF, due to the simple memory model.

As an exception, to improve the performance of a common case, processes with few parameters can be created more efficiently. For such processes, parameters can be passed in registers to the system call instantiating the process. The kernel, aimed at the Intel x86 architecture, reserves two general purpose registers for this. This additional system call allows processes to be instantiated in a single call.

3.2.3 Calling Processes

Processes can be instantiated without the CREATE keyword as in the output commands in Figure 7, line 24. For example, "Out.Int(x)", calls the Int process in the Out module to display the value of x on screen. Processes instantiated in this way, known as called processes, are still separate processes, but do not execute concurrently with their callers. When a process is called, its parent process, the caller, is blocked until the called process completes. This means that there is no concurrency between such process pairs. This predictable scheduling makes it possible to share data between the caller and the callee without the possibility of race conditions. Shared data structures are accessible in LF by a nested process accessing variables in the parent process, passing reference parameters and global variables.

Another use of called processes is to implement function processes. Function processes are allowed to return a value upon completion. This value is returned in an implicit
reference parameter added to the parameter list explicitly specified.

The memory allocation strategy, allocating memory from pools of terminated processes, is especially important to speed up the creation of called processes, which are used as substitute for procedures. Support for called processes needs to be as efficient as possible.

Of course, called processes would not be required if procedure calls were available. Unfortunately, because processes are activated in parallel, ARs cannot be allocated on a stack. To support procedure calls, each process would require its own local stack. The size of this stack would naturally vary as successive procedures are activated, recursion makes it possible for the stack to grow unbounded. As with allocating ARs, the memory allocated on the stack would have to be cleared to prevent states being generated unnecessarily because of uninitialized stack values, which would add to the cost of procedure calls. Dynamically increasing and reducing the process descriptor size complicates gathering and restoring the states of processes. As a local stack was not otherwise needed by LF processes, it was decided not to support procedure calls.

A significant part of the increased overhead of called processes compared to procedure calls stems from the cost of scheduling the called process. This overhead can be reduced by bypassing the scheduler and directly activating a called process. Similarly, upon termination of the called process its parent can be activated immediately without any visit to the scheduler. However, by bypassing the scheduler, interrupts will not be handled until after the called process has completed. When a process is recursively activated, it will be difficult to determine the length of time interrupts will be delayed. This can be remedied by first examining whether interrupts are pending before activating a called process (or its parent). To activate a called process without the scheduler means that some scheduling functions are performed by the system call for called processes. This might conflict with the scheduling policy implemented by the scheduler as well as the model checker. Due to these concerns and the added complexity it was decided to omit such an implementation.

3.2.4 Inter-process Communication

When a process executes the NEW command, a channel record is allocated by the kernel and its address is returned. This address can be passed on when a new process is created to connect communication partners. Processes address channels indirectly
via port variables when they send or receive messages.

If a process executes a send command at a time when no receiver is ready, the address of the process activation record of the sender is stored in the channel record to indicate that it is ready to send a message. The sending process is then blocked until a suitable communication partner is ready to receive the message. If a process executes a receive command when no sender is ready, the address of the receiver’s activation record is entered in the channel record. When a suitable communication partner becomes available, the message is transferred from sender to receiver and both processes are marked as ready for execution. If a process executes a communication command (send or receive) and the channel record indicates that a suitable communication partner is already waiting, the message is transferred immediately and both processes are marked as ready for execution.

A single channel can define several message types. These message types are called alphabet symbols. Each symbol has an associated data type, which supports compile-time type checking of messages as discussed in Section 2.1. Symbols allow for different types of messages to be transferred between processes without sharing a separate channel for each message type. To efficiently synchronise communication partners, a separate queue of potential communication partners is maintained for each symbol defined for the channel record. Using separate queues makes it possible to determine if a suitable communication partner exists by simply checking the head of the queue for the specified symbol. If different symbols were queued in a combined queue, the entire queue would have to be examined to determine whether a message of the correct symbol type is available. A combined queue could contain several queued processes, as all waiting processes on each symbol would be added together. The separation of queues represents a significant reduction in the processing required when synchronising communication partners.

If more than one sender (or receiver) is waiting for communication, a queue of process activation records forms in the channel record. The queue cannot contain senders and receivers simultaneously. The reason is that a sender will never join a queue of receivers because it will immediately deliver its message to the first receiver in the queue. A similar argument explains why a receiver will not join a queue of senders.
Guarded Communication

The LF SELECT command allows a process to make a choice between multiple communication options. For example, to make a choice between two possible messages, message1 and message2, with different actions taken in response to each, one could write in:

```
1 SELECT
2   WHEN in ? message1 THEN (* action for message1 *)
3   WHEN in ? message2 THEN (* action for message2 *)
4 END
```

Each alternative in the SELECT command can also contain an optional boolean expression. The communication will only succeed if the boolean condition evaluates to true. For example, if the action depends on the current contents of a variable state, the previous example becomes:

```
1 SELECT
2   WHEN in ? message1 & state=1 THEN (* action for message1 in state 1 *)
3   WHEN in ? message1 & state=2 THEN (* action for message1 in state 2 *)
4   WHEN in ? message2 & state=1 THEN (* action for message2 in state 1 *)
5   WHEN in ? message2 & state=2 THEN (* action for message2 in state 2 *)
6 END
```

To support SELECT communication, the kernel provides system calls to examine whether a communication command can be completed successfully before a certain deadline. The SELECT command itself cannot be completed until one of the communication commands is successfully executed. To determine whether a communication command can be completed, the message queue for the specified symbol is examined by the kernel. This queue is constructed from messages sent, or processes accepting communication, when executing normal communication commands. As a result two SELECT communication commands cannot communicate with each other. It was decided that this limitation is acceptable, as implementing support for this generalised guarded communication would be overly inefficient. This restriction corresponds to the requirements for an efficient implementation of the generalised guarded communication commands discussed in Section 2.5.2. This functionality—allowing two SELECTs to communicate—is not essential for implementing embedded systems and was not included in the LF language [34].

When the end of a SELECT command is reached, and none of the guards evaluate to
true, a new process is scheduled. The SELECT command will be attempted again at a later stage when one of the communication commands can be completed. The simplest way to achieve this is simply to poll the SELECT command from time to time. A rescheduling is forced when the end of a SELECT is reached, and the process executing the unsuccessful SELECT command is returned to the queue of ready processes with its next action reset to the start of the SELECT. This is not very efficient as this process can be scheduled multiple times without any change to the message queues of the communication channels taking part in the SELECT command. The problem is magnified when several processes execute SELECT commands.

A more efficient solution is to block the process when a SELECT fails, and to reactivate it only when at least one of the relevant queues change. This method is implemented by the LF kernel by keeping an additional list, for each communication channel symbol, of processes waiting to complete a SELECT on the relevant channel symbol. This list identifies the set of processes to activate whenever there is a change to the queues of messages involved in the communication of the SELECT.

### 3.2.5 Complex Messages

LF message types are not limited to the basic types and signals seen so far. It is also possible to send structured message types such as arrays and records. Although these types of messages are relatively flexible, it is sometimes more convenient to construct messages from separate variables or constants (for example, “disk ! request(cylinder, block, count)” to request a disk access). Such combined messages, called complex messages, behave exactly as simple messages would when synchronising and blocking processes. Figure 7 shows communication between two processes, **Sender** and **Receiver**, transferring data and an identification number, *sourceid*, in a complex message. The complex message is sent as *data(i, id)* from **Sender** (line 11) and received into *x* and *sourceid* by **Receiver** (line 23).

Several options are possible to implement complex messages. Kernel support is not necessarily required. For any complex messages, consisting of separate variables, an equivalent message consisting of a single record could, of course, be substituted. The user is then responsible for constructing a record which contains all the variables required to be sent in a single message. This approach can be costly as variables need to be copied into the record before the message can be sent. Also, the message can only be received into a record structure, from which it will have to be copied again to its
1 MODULE ProdCons;
2 IMPORT Out;
3 TYPE Msg = [data(INTEGER, INTEGER), end];
4 PORT port : Msg;
5
6 PROCESS Sender(out : Msg; id : INTEGER);
7 VAR i : INTEGER;
8 BEGIN
9   i := 0;
10  WHILE i < 100 DO
11    out ! data(i, id); INC(i)
12  END;
13  out ! end
14 END Sender;
15
16 PROCESS Receiver(in : Msg);
17 VAR x, sourceid : INTEGER;
18   more : BOOLEAN;
19 BEGIN
20   more := TRUE;
21  WHILE more DO
22     SELECT
23       WHEN in ? data(x, sourceid) & sourceid=1 THEN
24         Out.Int(x); Out.String("from "); Out.Int(sourceid)
25       WHEN in ? end THEN more := FALSE
26     END
27  END Receiver;
28
29 BEGIN
30   NEW(port);
31   CREATE Sender(port, 1);
32   CREATE Receiver(port)
33 END ProdCons.

Figure 7: Producer-consumer system in LF with added sender identification
final destination.

A similar solution could be provided by the compiler, hidden from the user, also without kernel support. Complex messages can be combined in an intermediate buffer by the compiler and then be sent similarly to any other contiguous message. At the receiving end, the contents of each variable can then be recovered from the buffer.

Both these solutions require costly additional copying of data to support complex messages. However, by providing kernel support, complex messages can be supported more efficiently, avoiding the additional copying of data. The LF kernel provides support to efficiently transmit complex messages. This is implemented in combination with support from the LF compiler. For any complex message sent, or received, a list of data to be copied is constructed by the compiler. Each entry in the list consists of an address relative to the activation record base and the number of bytes to be copied. These lists are stored as part of the compiled code as a table of data values and therefore the memory overhead is kept to a minimum. Only a single copy of the table is stored along with the communication command that will need to access it. These lists are used by the kernel to copy each variable directly from the source to its destination.

3.2.6 Conditional Receive

The LF SELECT command is used to make a choice between several communication commands. These commands can also be guarded by a boolean expression. This guarded communication command becomes very powerful if boolean expressions are allowed which evaluate the contents of the message to be received. This allows messages to be accepted or rejected, based on their contents. For example, the SELECT in Figure 7, line 23, only accepts data messages from a source with sourceid=1.

Using this construct, a server can communicate with multiple clients without explicitly naming each client. The clients listen for a message from the server on a single shared communication channel. The conditional receive command allows a client to examine the contents of the message to determine whether or not the message should be accepted. Only if the message is intended for the specific client will the data be copied. Alternatively, without a conditional receive command, such a message would have to be copied to each client before it could be examined. If such a client received a message intended for another recipient, the client would have to re-send the message on the shared channel. By providing kernel support for conditional receive commands,
such communication is more efficient and also relieves the programmer of the burden of repeatedly having to implement it each time it is needed. Using a conditional receive ensures that the copying of the message data takes place only once, directly to the intended client. This is especially important for large messages.

Kernel support for conditional receive commands is provided by the same system calls as the general SELECT communication. The data contained in the message is accessed relative to a register returned as part of the polling communication system call used for all SELECT communication. This allows the conditional receive to be implemented with little additional overhead. Because the boolean expression depends on the value of the message sent, the boolean condition can only be evaluated after the system call to poll the communication channel has been executed.

3.2.7 Device Drivers

In embedded systems it is often necessary to communicate with peripheral devices. Device drivers need to be developed for such devices. As a language for developing embedded systems, LF needs to facilitate the development of these drivers. To this end, low-level commands were included in the LF language. Intrinsic functions are defined in the compiler and implemented by generating machine instructions to perform the operation directly where such functions are called. Such commands are, of course, hardware specific. For Intel hardware, intrinsic functions for reading and writing IO ports (PORTIN, PORTOUT) and a command for accessing memory mapped devices (AT) are provided [34].

Device drivers also require access to interrupts. Therefore to develop device drivers, it is necessary to be able to implement interrupt handlers in LF. Instead of the standard method of installing interrupt handlers to be activated by the processor in response to interrupts, the LF kernel allows interrupt handling using IPC. Processes execute as indivisible actions. Interrupt handlers cannot be activated at the point that the interrupt occurred as this would have to interrupt the action currently executing. Whenever an interrupt occurs, the kernel will present the interrupt as a signal available on a special interrupt channel. In LF, interrupts are handled by processes accepting interrupt signals on special interrupt channels. If no interrupt is currently pending when such a handler process executes a receive command on the interrupt channel it will be blocked. When the interrupt occurs, the handler process will immediately be placed in the ready queue and scheduled according to its priority. The priority is determined by the specific
MÖODULE KeyboardInterrupt;
IMPORT System;
PROCESS KeyboardHandler( keyboard : System.Interrupt );
BEGIN
  WHILE TRUE DO
    keyboard ? interrupt;
    (* handle interrupt *)
  END;
END KeyboardHandler;
BEGIN
  KeyboardHandler( System.Int );
END KeyboardInterrupt;

Figure 8: Installing an interrupt handler

interrupt that has occurred. If, however, no processes are ready to receive the interrupt message, the message will be queued on the interrupt channel, to be accepted as soon as the handler process returns to a state in which the interrupt can be handled. Only one such signal message will be queued for each interrupt. If another interrupt should occur before this queued signal has been received, the second interrupt will be lost. This makes it clear that the length of actions should be limited, as interrupts cannot be serviced until the current action has completed.

To install an LF interrupt handler, such as the keyboard interrupt handler in Figure 8, an LF program needs to do the following:

- Import the System module to access the system defined channel which represents the required hardware interrupt, System.Intl.

- Instantiate a process to handle the interrupts, KeyboardHandler (line 13).

The handler receives interrupt signals on the interrupt channel (line 7) and proceeds to perform the service required by the corresponding device (line 8). Also, to continue to receive and handle interrupts, the handler should in general execute handling of interrupts in a loop (lines 6–9).

Although it is intended that device drivers are generally developed as user-level LF processes, some in-kernel device drivers are appropriate. One reason for such in-kernel drivers is that some devices are needed by the runtime support system to perform its functions. An example of such a device is the timer driver.
Another reason for in-kernel drivers is to take advantage of having access to all kernel data structures and functions such as the ready queue and the memory manager. Having direct access to this data makes more efficient implementations possible. This might, for example, be appropriate for a network driver, for efficient buffering of network data. In all such cases an interface needs to be provided to the driver for use by LF modules. Access to peripheral devices is provided by message passing. Two-way communication with in-kernel drivers is possible by message passing on special shared device driver channels, while interrupts are delivered as signals to user processes on interrupt channels.

When model checking actual user code, devices and device drivers will always cause problems. The model checker needs to be able to control execution of the system and return it to previous states. This is impossible when interacting with a real hardware device. To successfully model check systems relying on devices, an abstract representation of the device needs to be provided internally to the system under verification. Providing access to device interrupts by message passing helps to make it possible to code a model of a device in software to replace an actual device during verification. If a model can be constructed to mimic the behaviour of a device then even interrupts can be simulated without any alteration to the device driver under verification. However, the feasibility of this approach falls outside the scope of this.

3.2.8 Scheduling

The LF kernel reacts to two kinds of external event: hardware interrupts and system calls. LF code executes with interrupts enabled, but the kernel only logs hardware interrupts when they occur—they are not serviced immediately. Instead, the kernel services interrupts only after completion of an action when the interrupt handler is scheduled.

Whenever the scheduler is invoked, the next process is selected based on the chosen scheduling policy. The default scheduling policy is a round robin scheduler. There is also a deadline-driven scheduler based on the _earliest deadline first_ (EDF) scheduling policy.

The scheduler is implemented as a loadable module, LFScheduler, which also serves as the interface to the model checker. From this point of view, the model checker can be seen as a special-purpose scheduler. The interface that needs to be provided by
a scheduler for LF consists of four procedures **Unblock**, **Block**, **Reschedule**, and **Update**.

*Unblock, Block* and *Reschedule* perform the expected functions of adding a process to the ready list, removing a process from the ready list, and scheduling the next process. The *Update* procedure is provided specifically for the model checker. The function of the update procedure is to allow notification of the model checker when a process activation record has changed that is not the currently executing process. This is needed, for example, to update the state of the receiving process whenever a send communication command has completed.

**Deadline-based Scheduling**

A deadline is associated with each event and the scheduler ensures that each process is activated soon enough to meet the deadline associated with its triggering event. Currently these deadlines are simply calculated as a predefined length of time after the triggering event. For more realistic deadlines, the LF language would have to be extended to allow the specification of timing constraints. To determine priorities, process records are stored in a heap structure, organised so that the top element has the closest deadline. A heap is an efficient representation of a priority queue, as both inserting and removing elements can be performed in $O(\log n)$ time in the number of elements in the heap.

### 3.3 Supporting Model Checking

Processes are executed one at a time, systematically exploring the entire state space. Generally, model checking is performed on an abstract model describing the system being verified. The LF model checker is different in that the verification is applied directly to the compiled implementation code. The same code is also used during normal execution. The model checker can be seen as a special purpose scheduler, replacing the normal scheduler by the model checker. The scheduler module therefore also defines the interface between the kernel and the model checker.

As model checking is applied directly to the compiled code of LF programs, steps had to be taken to minimise the state space generated. Processes are executed in atomic steps, *actions*. This reduces the state space by reducing the number of separate states
considered for each process, also reducing the number of alternative sequences in which they can be scheduled. Restricting the scheduling in this way significantly impacts on how interrupts can be handled. Because scheduling points are limited, and interrupts can only be serviced at these fixed points there may be a relatively long delay between the time an interrupt occurs and the time it is handled. This resulted in the method employed by the LF kernel of only marking interrupts as outstanding when they occur and delaying servicing until the next scheduling opportunity.

The model checker identifies unique states for each process by applying a hash function to the process AR. Any changes to the contents of the AR results in a new unique state. When an AR is allocated, the content of the memory area allocated is undefined. Uninitialized variables of a process therefore contain unspecified values. To avoid states being incorrectly identified as differing because of such uninitialized variables, the kernel fills all new ARs with zeroes. While important for model checking, this is a significant additional overhead during program execution.

For the Java Pathfinder (JPF) [37], a model checker for Java programs, a canonical heap representation is provided—ensuring dynamic memory will always be allocated in the same memory locations, independent of the order in which allocations are made, and avoiding additional states that may be generated because of this. However, the support provided to JPF aims to provide “efficient memory management rather than blistering speed [37].” The performance of allocating dynamic memory was important in the design of the LF kernel, which does not guarantee that all possible orderings will result in exactly the same allocation. The allocations for a specific process type are, however, independent of the order of execution or allocation of all other process types. All previous ARs for a certain type of process are stored in a pool, and reused on subsequent activations. Separate pools are kept for each process type, ensuring the location allocated to each process is maintained when changing the order in which different types of processes are activated. Changes in the order of activating processes of the same type can, however, lead to process ARs being allocated in different locations.

A single structure, the process activation record, is used to represent the entire state of a process, including its own local variables and information required by the kernel. The entire state of the process is stored in a single contiguous block of memory of a constant size. This makes it simple to gather the state of the current process when generating new states or when comparing it to previous states.
Chapter 4

Evaluation

The LF kernel is designed to support model checking and therefore differs from other kernels for CSP-based languages. LF programs can be checked for defects by simply linking an optional module—the model checker—into the kernel. To reduce the number of states to be explored, the kernel supports execution of short sequences of machine instructions as indivisible units. This strategy creates an optimisation problem: longer sequences lead to fewer states and less scheduling overhead, but reduce responsiveness to interrupts. It is therefore important to determine the optimal length of the indivisible sequences through measurement.

Supporting the model checker is important for the LF project, but in providing this support overall system performance should not be compromised. The support provided for LF programs should still be as efficient as possible. Performance was measured to determine:

- the overhead of actions: To make model checking simpler processes are divided into actions. The effect on performance of executing processes in these indivisible steps is evaluated.

- the cost of system calls: All support functions of the kernel are accessed by making system calls. The overhead of making these system calls is measured.

- how efficiently memory is used: Kernel memory usage and the performance of the dynamic allocator is evaluated.

- the latency in servicing interrupts: The disadvantage of executing LF systems in indivisible steps is that interrupt response is delayed. The effect of this delay on
handling of interrupts is examined.

- **the efficiency of process management**: The basic requirement of LF systems is process management which includes scheduling, creating and calling processes.

- **interprocess communication**: Support for LF interprocess communication is examined in terms of transfer rate and latency for simple message passing, guarded communication and conditional receive.

The LF project aims to aid in development of reliable embedded software. However, not only the correctness but also the performance of such systems is important. How successfully these goals are achieved will be fully decidable by implementing actual systems in LF. Success of any such implementations would require them to be efficient, while still amenable to model checking. However, until large case studies have been done, benchmarking measurements and evaluation of the requirements of the LF language and model checker serve as performance indicators. These results point out which features should be avoided, and which features are preferred. Limitations, especially on interrupt response time, are also considered.

### 4.1 Benchmark Methodology

The target system for the LF Kernel is the Intel386 EX embedded processor. Measurements were, however, made on an Intel386 SX at 33MHz, of which several were available in the department and allowed simple boot-loading from floppy disk. Measurements on this hardware are, however, representative as the performance of the Intel386 EX processor closely reflects the Intel386 SX CPU performance at the same speeds [1].

Performance measurements were made of the cost of performing important primitive operations required in supporting LF programs. The average cost of each operation is calculated from the total elapsed time, performing a large number of iterations. The number of iterations to obtain a representative value is typically in the order of millions\(^1\). Averaging the benchmark in this fashion removes the need for fine-granularity time-stamping. A single iteration could have been skewed by an external event such as an interrupt occurring.

---

\(^1\)Tests are repeated for a million iterations and measurements for start and end times are taken using a clock accurate to within 1ms. This gives an accuracy of \(\frac{2\text{ms}}{1,000,000} = 2\mu\text{s}\). For values in \(\mu\text{s}\) this error is at most 0.2%.
Such measuring requires that the operations executed repeatedly in a loop. The overhead of the loop itself is measured separately as an empty loop and eliminated from the total measurement. Generally, the average cost, as measured in this way, is considered more appropriate than the worst-case cost. However, for some tests the worst case will be considered when it is more relevant. For example, when considering an acceptable interrupt latency it is only the worst-case latency, resulting in loss of interrupts, that is of note.

4.2 Overhead of Actions

Model checking of LF programs at the machine code level is made possible by executing code as indivisible sequences of machine instructions, or actions. Executing processes divided into atomic actions simplifies model checking by reducing the number of scheduling points. This reduces the number of states to be explored by the model checker. This is the primary technique applied to reduce the state space towards making verification of actual implementation code possible. Processes consisting of actions can be seen as a transition system [34], moving from one state to the next by executing an action. Executing these processes, with runtime support, traverses the transition system. There is, unfortunately, some overhead associated in executing actions—control returns to the kernel at the end of each action. Lengthening the actions reduces the state space by reducing the number of states for each process as well as the number of possible interleavings of the processes in the system. Longer actions also reduces the overhead of interpretation as control is returned to the kernel less often. Unfortunately, long actions can cause servicing of interrupts to be delayed for a significant period of time.

In the initial implementation of the kernel for this thesis processes were divided into actions by executing an interpreter function—part of the kernel—after LF commands. The code generated by the compiler for execution with the interpreter function terminated actions after every LF command except simple assignments [34]. This allowed for a low interrupt response time, but also high overhead. Swart measured the effect of lengthening the actions as an optimisation during code generation for the compiler using the code examples of Figure 9. These results are summarised in Table 1 [34]. Swart found for these fragments of sequential code, performance could be improved by as much as three times (Tests 3 and 4) by removing all interpretation steps.
<table>
<thead>
<tr>
<th></th>
<th>Interpreter</th>
<th>No Interpreter</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1</td>
<td>1.375</td>
<td>45.5</td>
<td>2.13:1</td>
</tr>
<tr>
<td>Test 2</td>
<td>3.590</td>
<td>3.130</td>
<td>1.24:1</td>
</tr>
<tr>
<td>Test 3</td>
<td>3.595</td>
<td>1.229</td>
<td>2.91:1</td>
</tr>
<tr>
<td>Test 4</td>
<td>5.085</td>
<td>1.383</td>
<td>3.14:1</td>
</tr>
</tbody>
</table>

Table 1: Measurements of execution times for different implementations of action length

To improve performance, it was decided to execute LF systems with actions extended to the maximum possible length—processes can only be interrupted when they can no longer continue and thus scheduling is required. The compiler and kernel support was modified to implement actions in this manner. Whether the negative effect of such large actions on interrupt response is acceptable is examined in Section 4.8.

Independently of the intended action length, when an LF command results in a system call the action is terminated. This means that no additional interpretation step is added when the length of the action is reduced. This would indicate that less significant gains should be expected in general than is achieved for purely sequential examples such as in Figure 9. This was also experimentally measured, showing an average improvement in execution time of about a third for the systems we have developed in LF.

4.3 System Calls

The support provided to LF programs by the LF kernel is accessible through system calls. As a result, the performance of all the primitive operations performed in supporting LF programs is influenced by the overhead incurred in making system calls. It is therefore appropriate to attempt to minimise this overhead.

System call overhead is kept relatively low mostly by the simple memory configuration of the LF kernel. System calls are often implemented using traps, or software interrupts. These make use of the interrupt mechanism to pass control from the user to the kernel. This mechanism is generally used to allow implementation independence between the user and the kernel. Unfortunately, these trap instructions for entering and leaving the kernel are especially slow on many processors, including the Intel x86 family of processors [2]. The LF kernel is able to provide implementation independent system calls without making use of these instructions. Instead of executing a trap instruction, a process simply executes a jump to the required system call via a jump table. The
\begin{verbatim}
1    WHILE x < 20000000 DO (* Test 1 *)
2        x := x + 1;
3    END;
4
5    WHILE x < 20000000 DO (* Test 2 *)
6        x := x + 1; x := x; x := x; x := x; x := x; x := x;
7        x := x; x := x; x := x; x := x; x := x;
8    END;
9
10   WHILE x < 20000000 DO (* Test 3 *)
11        x := x + 1;
12        IF x > 0 THEN x := x END;
13    END;
14
15   WHILE x < 20000000 DO (* Test 4 *)
16        IF x > 0 THEN
17            IF x < 2000000 THEN
18                x := x;
19            ELSE
20                x := x;
21            END;
22        END;
23        x := x + 1;
24    END;
\end{verbatim}

Figure 9: Examples used to measure effect of action length on performance.
CHAPTER 4

PROCEDURE NullCall(Action : LONGINT);
BEGIN
  CurrentAR.Action := Action;
  IF LF8259.Interrupts # {} THEN
    Reschedule();
  ELSE
    Continue();
  END;
END MinimumSystemCall;

Figure 10: The null system call

jump table is provided as part of the kernel, and, indexed by the appropriate system

call number, contains the address of the code providing the required function. This is

possible because all code executes within a single shared address space.

Another reason for using traps is to make use of processor support for privilege levels. Generally only the

kernel is allowed to execute privileged instructions and therefore needs to execute at a higher privilege level. The interrupt mechanism can then be used to automatically switch to the required privilege level and back when entering and leaving the kernel. The LF kernel does not make use of privilege levels. Instead, protection is provided by the LF compiler. This is possible because all programs are coded in the LF language and compiled by the LF compiler.

LF system calls also gain performance from avoiding separate address spaces. When separate address spaces for processes are provided by making use of paging, each process and the kernel need to maintain their own page tables. Whenever the kernel is entered or left these page tables are switched. Apart from the overhead incurred due to the actual switching of address space, performance is also negatively affected because of the translation look-aside buffer (TLB) and the performance loss for memory accesses after an address switch. If required, separate address spaces could be provided by using segmentation with relatively low overhead compared to paging.

From the simplifications listed here, it may be expected that LF system calls can be made with relatively low overhead. A null system call, constructed specifically to measure the overhead of making system calls, was executed repeatedly as described in Section 4.1. The null system call represents the essential operations performed during every other system call. The operations performed by the null system call (Figure 10)

are:

• saving the program counter of the current process (line 3), and
• checking whether any interrupts are pending (lines 4–8).

  (a) If interrupts are pending control is passed to the scheduler, which ensures such interrupts are serviced as soon as possible.

  (b) If no interrupts are pending, control is returned to the process that made the system call.

As expected, an LF system call adds very little overhead. The overhead on the target hardware was measured to be only 3.6μs. For comparison with measurements of system calls in the Linux kernel, these measurements were repeated on a machine compatible with Linux distributions, a Pentium-III 800MHz. Measurements of system calls on the Linux kernel show overhead of 3.5-4μs, compared to only 6ns for the LF kernel. The large difference — LF is almost 600 times faster — shows the advantage of a small, simple, special-purpose kernel in its intended application area.

### 4.4 Implementing System Calls

Although the ultimate goal for LF is model checking at the code level, the main focus during the development of the LF kernel was efficiency. System calls, especially frequently used calls such as those used for interprocess communication, should be implemented as efficiently as possible. To optimise their performance, LF system calls are implemented in assembly language. This, unfortunately, leads to code that is difficult to maintain and port to different hardware.

Because it would greatly improve portability and maintenance, the feasibility of re-coding the kernel in the high-level language, Oberon, was examined. Some of the LF system calls were rewritten in Oberon and the performance difference measured. Tests were performed for simple communication as well as the calling of processes. The tests were repeated 1000000 times and the total time measured. The results, averaged over three repetitions, are given in Table 2.

<table>
<thead>
<tr>
<th>Simple Communication Calling Processes</th>
<th>Assembly language</th>
<th>Oberon</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:16.20</td>
<td>3:08.36</td>
<td>1:2.4</td>
<td></td>
</tr>
<tr>
<td>2:20.44</td>
<td>4:31.64</td>
<td>1:2</td>
<td></td>
</tr>
</tbody>
</table>
This result indicates that the use of assembly language is justified due to the performance gains achieved. Liedtke suggested that a microkernel achieves high performance by being optimised for the specific intended hardware [23]. Such a kernel is not portable but improves the portability of the rest of the system. Some reasons as to why LF system calls are especially suitable to implementation in assembly language are, that:

- parameters can be passed in registers (it is difficult to gain full advantage of this when Oberon is used as implementation language),
- the Oberon compiler does not generate highly optimised code, and
- assembly code can be optimised to avoid memory accesses as much as possible, because memory accesses are especially slow on the target hardware, the Intel386 EX processor (due to its 16-bit data bus).

### 4.5 Memory Management

Embedded software must generally be designed for systems with limited hardware resources due to technological or financial limitations. A kernel to support embedded software should therefore conserve system resources as much as possible. The LF kernel was designed to minimise memory usage. This requirement further justifies the simple linear address space shared by all processes. This removes the need for separate page tables and page directories to be stored for each process. This results in a significant saving in memory usage.

Table 3 presents the standard static memory requirements of the LF kernel. For simplicity, the partitioning of this memory and therefore the size allocated to each part is fixed. These rigid divisions restrict the memory used for each section to the area allocated to it. This is considered acceptable as these allocations can be customised to best suit the specific system being developed. For example, more memory could be made available for dynamic allocation of processes by allocating less memory to LF

<table>
<thead>
<tr>
<th>Kernel Code Size</th>
<th>80KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel Stack Size</td>
<td>16KB</td>
</tr>
<tr>
<td>LF User Program Code Size</td>
<td>(up to) 440KB</td>
</tr>
</tbody>
</table>

Table 3: Standard static memory usage
user code. The divisions are restricted by the IBM PC mapping of display memory at addresses following 640K on the target hardware. The portion below 640K is generally too small to contain both code and dynamic storage requirements. Because of this, all memory below 640K is used for static allocation, while memory above 1MB is available for dynamic allocation.

The only dynamic memory requirements of LF programs are the allocation (and disposal) of process activation- and channel records. The method described in Section 2.4.1 aims to efficiently reuse memory allocated for parallel processes. The allocator aims at removing the need for hole management as required by general purpose allocators and depends on reusing memory previously allocated for instantiating processes of the same size. A disadvantage is that it is possible for a piece of memory to be allocated to a pool, but not in use. If previously allocated blocks are of a size not required for a period these blocks will remain unused, but stored in that specific pool during this period. This is only a problem when there is little unallocated memory available. In such situations, it is possible to recover unnecessary memory allocated to pools and recycling this memory for general use. The execution of this maintenance task is, unfortunately, unpredictable and can adversely affect interrupt latency.

The degree to which memory is reused depends on the system of processes being supported. The method applied to LF memory management is based on a solution devised by Brinch Hansen for implementing SuperPascal. Brinch Hansen measured the efficiency of memory reuse for some standard examples implemented in SuperPascal. In general it was found that new blocks were only allocated up to the maximum number of processes simultaneously active in parallel. These include executable processes as well as processes that are blocked. It was also found that if no processes were activated in parallel, that the memory used was the same as for static allocation [9].

Similar results were measured for the small to medium sized examples implemented in LF. This allocation method is especially well-suited to called processes in LF. Called processes are instantiated like procedure calls, running to completion and then terminate. When a process is repeatedly called to perform an operation, the same activation record can be reused for each instantiation. The first execution will allocate a new block, which will be released to the pool upon termination. Subsequent instantiations can then reallocate the same block from the pool and return it to the pool when terminating. In this case all allocations, except the initial, will be from the pool of available blocks.
4.6 Supporting Processes

The usual kernel-related measurements produced nothing surprising. The time required to create a new process is only $41\mu s$, compared to several milliseconds for the Unix fork command on the same hardware. Process creation includes allocation of a new activation record and initialisation of all local variables of the process (required by the model checker).

The operations performed during standard process creation involves two system calls. The first system call is made to allocate memory, initialise local variables and return the address of the allocated process activation record. Parameters are copied to this activation record by the executing process. The second system call is made to activate the new process. This includes adding the process to the list of ready processes (a call to the scheduler) before returning control to the process executing the system call.

For processes with few parameters (less than 8 bytes), a simplified method is available which takes advantage of passing parameters in registers. When parameters can be passed in registers, process creation is achieved by a single system call. Parameters are stored in registers before making the system call, which allocates the process activation record before storing the parameters into it. Similar to the general case, other variables are cleared and the process is added to the list of ready processes. Two general purpose registers are available for such parameters. Using this method, the cost of process creation reduces to $35\mu s$. This reduces even further to $30\mu s$ when allocation can be done from a pool of previously allocated blocks.

4.6.1 Scheduling

In the standard kernel configuration, a round robin scheduler is used. An alternative scheduling implementation is available, because real-time scheduling is important for some embedded applications. A deadline-driven scheduler is implemented as a step towards real-time scheduling. Although this does not represent a full real-time scheduler, this provides insight into the performance implications of using such a scheduler.

Performance measurements, provided in Table 4, show a significant increase in overhead for the deadline-driven scheduler compared to using the round robin algorithm when the number of processes increases. The often-used technique of implementing a dynamic priority queue using a heap representation is applied to the ready queue
for the deadline-driven scheduler. The increase in overhead is due to heap operations required in constructing and maintaining the priority queue. These operations can be performed in $O(\log n)$ time in the number of processes. This logarithmic increase can be seen in Table 4. In contrast, with the round robin scheduler all scheduling operations are performed in constant time, because a simple FIFO queue is used.

### 4.6.2 Calling Processes

LF provides a mechanism whereby processes can be called, which is intended as a substitute for procedure calls absent from the process based language. Procedure calls are not supported because LF processes execute without a separate stack. Figure 11 shows a comparison between the performance of procedure calls and called processes. The graph indicates the ratio of the time taken when making the call—either process or procedure—and without the call. The graph represents these ratios against an increasing number of processor instructions in the body of the process or procedure. Measurements were first taken of loops executing a portion of code representing a certain number of instructions. Each test was then repeated, executing the same code, but contained in an Oberon procedure. Similarly, the test was repeated for LF processes. As the code generation of the LF compiler is based largely on that of the Oberon compiler, almost identical code is generated in both cases. Differences are limited to specific register allocations and the performance of the code is similar. The ratios provide an indication of the overhead incurred in making the respective procedure and process calls.

The graph shows that in comparison to procedure calls, there is a high cost associated with calling a process. This is most noticeable for low numbers of instructions in the body for the call.

Compared to a procedure call, shown on the right in Tables 5 and 6, the operations performed when calling a process (on the left) are significantly more complex. This is clear

<table>
<thead>
<tr>
<th>Processes</th>
<th>Round-Robin</th>
<th>EDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$4\mu s$</td>
<td>$4\mu s$</td>
</tr>
<tr>
<td>16</td>
<td>$4\mu s$</td>
<td>$68\mu s$</td>
</tr>
<tr>
<td>256</td>
<td>$4\mu s$</td>
<td>$104\mu s$</td>
</tr>
</tbody>
</table>
Figure 11: Procedure call compared to calling processes
### Table 5: Comparison of called process and procedure call (Activation)

<table>
<thead>
<tr>
<th>Called Process</th>
<th>Procedure Call</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocate memory by making a system call.</td>
<td>25 instructions</td>
<td>2 instructions</td>
</tr>
<tr>
<td>Pass parameters to activation record allocated and returned by the system call.</td>
<td>2 instructions per parameter</td>
<td>1 instruction per parameter</td>
</tr>
<tr>
<td>Activate process by making a second system call, blocking the parent while activating the child process.</td>
<td>30 instructions</td>
<td>1 instruction</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Called Process</th>
<th>Procedure Call</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A system call is made to terminate the process. This call frees all memory allocated to the process and activates the parent of the terminating process.</td>
<td>30 instructions</td>
<td>2 instructions</td>
</tr>
<tr>
<td>Execute the scheduler which schedules the highest priority process.</td>
<td>N/A</td>
<td>1 instruction</td>
</tr>
</tbody>
</table>

### Table 6: Comparison of called process and procedure call (Termination)

from the instruction counts shown for each operation to be performed. As expected, calling a process showed greater overhead than a procedure call, but as this overhead is constant, the relative cost reduces as the size of the process (or procedure) increases. The experimental results of Figure 11 suggest that this higher overhead is acceptable when performing relatively complex tasks. Therefore, it would be acceptable to, for example, call a process to perform a sorting algorithm, but not a simple arithmetic calculation. Similarly, a procedure call also introduces an unnecessary overhead if used to perform a very basic task.

Alternatives to this way of calling processes were discussed in Section 3.2.3. Procedure calls were eliminated due to the absence of a local stack for processes. Some improvements in performance can be achieved by bypassing the scheduler when activating a called process. The child process could simply be activated immediately and run to completion. In this way calling processes in this way significantly complicates calculating the length of time spent executing an action. This would make it difficult to ensure sufficient interrupt response rate. An obvious solution would be to interrupt such a called process once the action executes for too long. However, as these called processes
can be instantiated recursively, the execution time can no longer be determined beforehand, and preemption after a specified time slice would have to be implemented. This violates the assumption that scheduling points are fixed which currently ensures that all possible actual execution traces are evaluated by the model checker.

### 4.7 Message Passing Communication

Processes communicate by means of message passing. When using message passing to provide interprocess communication, as opposed to using shared memory and semaphores or monitors, performance is always a concern. The performance of the message passing support provided by the kernel was measured to evaluate its efficiency. The LF kernel supports simple send-receive communication (normal communication) as well as more complex message-based communication constructs. These more complex cases consist of three types: (1) guarded communication (SELECT communication), (2) conditional receive, and (3) complex messages.
<table>
<thead>
<tr>
<th>Message Size (bytes)</th>
<th>Normal Transfer Rate</th>
<th>Normal Latency</th>
<th>SELECT Transfer Rate</th>
<th>SELECT Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N/A</td>
<td>56µs</td>
<td>N/A</td>
<td>84µs</td>
</tr>
<tr>
<td>4</td>
<td>69KB/s</td>
<td>57µs</td>
<td>46KB/s</td>
<td>85µs</td>
</tr>
<tr>
<td>16</td>
<td>266KB/s</td>
<td>59µs</td>
<td>180KB/s</td>
<td>87µs</td>
</tr>
<tr>
<td>256</td>
<td>2677KB/s</td>
<td>93µs</td>
<td>2039KB/s</td>
<td>123µs</td>
</tr>
<tr>
<td>1024</td>
<td>4866KB/s</td>
<td>205µs</td>
<td>4233KB/s</td>
<td>236µs</td>
</tr>
<tr>
<td>16384</td>
<td>6618KB/s</td>
<td>2418µs</td>
<td>6341KB/s</td>
<td>2523µs</td>
</tr>
</tbody>
</table>

Table 7: Performance of message passing for normal and SELECT communication

Performance of the basic message passing was examined from two perspectives, transfer rate and latency. Figure 12 deals with data transmission rates, and both normal and SELECT communication are shown. The form of SELECT command used is the simplest form of SELECT with only a single WHEN command and an empty guard. In this case no alternatives need to be examined and there is no boolean expression to evaluate. Predictably, the performance of such a SELECT command is very similar to that of simple message passing communication.

Figure 12 shows the transfer rates for both types of communication steadily improve as the size of the messages increases. For large messages the limiting factor becomes the transfer rate of the processor and memory. With a size of 16KB the message transmission rate is very close to the maximum transfer rate for the processor of 6667 KB/s. (This maximum transfer rate is measured by copying a large block of memory (64KB) in a tight loop.) The other important performance measure is latency between sending a message and reception of the message by the receiver. Latency was measured, again for both normal and select communication. These results are provided in Table 7. As above the simplest form of the SELECT command was used.

Table 7 shows the message passing latency increasing linearly with the size of the messages, as the portion of time spent transferring data increases. The overhead of synchronising communication partners for both types of communication can be seen for zero message size. The overhead of synchronising SELECT communication remains somewhat higher, at 84µs, than for simple communication at 56µs.

Complex Communication

SELECT commands are generally more complex than examined in the previous two examples. The performance of SELECTs deteriorates when making use of guards or
Figure 13: Code used to measure performance of guarded communication.

shared channels. The performance of such SELECT commands depends on the number of alternatives that need to be evaluated before communication can be successfully completed. Performance is influenced by the number of alternatives in a single SELECT command, but even more influential is the number of processes performing SELECTs on a shared channel.

Performance of the different versions of the SELECT command was investigated separately. The influence of each is measured by varying the number of alternatives for a given type of communication. Figure 13 shows the code used to measure the effect of the number of guards in a single SELECT command. The effect of additional guards was measured by adding Guard processes or additional WHEN commands to the SELECT in each Guard process and sending messages corresponding to each alternative in Measure.

In the first two tests, measuring the performance of guarded communication, separate alphabet symbols are used to identify the intended target of a message. Therefore alternatives can be identified in the communication part of the WHEN command without a boolean condition, for example the choice made between msg0 and msg1 in Figure 13. The evaluation of a true boolean condition simply increases the time spent on each
1  PROCESS Condition0;
2  BEGIN
3    WHILE TRUE DO
4       SELECT
5       WHEN a ? msg(m) & m = 0 THEN;
6       END;
7       END;
8  END Condition0;
9
10  PROCESS Condition1;
11  BEGIN
12    WHILE TRUE DO
13       SELECT
14       WHEN a ? msg(m) & m = 0 THEN;
15       END;
16       END;
17  END Condition1;
18
19  PROCESS Measure;
20  BEGIN
21    CREATE Condition0;
22    CREATE Condition1;
23    (* Start timer *)
24    (* Repeat test 1000000 times *)
25    a ! msg(0);
26    a ! msg(1);
27    (* Stop timer *)
28  END Measure;

Figure 14: Code used to measure performance of conditional receive in multiple processes.
alternative by a constant amount. Similarly, conditional receive within a single SELECT will only increase the cost of evaluating each alternative, although by a greater amount.

Conditional receive makes it possible for several processes, sharing a common channel, to identify which messages are intended for which process. Figure 14, shows a channel shared between SELECTs in two different processes used to evaluate the effect of conditional receive. In the second two tests, conditional receive was used to identify the intended target process. Measurements were taken by varying the number of Condition processes and the number of WHEN commands in each SELECT.

In total, four cases were examined:

- **Case 1:** Separate alphabet symbols are used for each message type with no boolean condition required. Alternatives are specified in a single process.

- **Case 2:** Separate alphabet symbols are used for each message type with no boolean condition required. Alternatives are specified in multiple Guard processes.

- **Case 3:** A single alphabet symbol is shared, requiring conditional receive to identify the target. Alternatives are specified in a single process.

- **Case 4:** A single alphabet symbol is shared, requiring conditional receive to identify the target. Alternatives are specified in multiple Condition processes.

**Case 1** shows that for a single SELECT command the cost of support increases linearly in the number of alternatives (Figure 15(a)). This is to be expected, as alternatives are evaluated one after the other until communication can be completed successfully.

**Case 2** shows that by using separate symbols in separate processes additional alternatives (after the second) can be added without extra cost (Figure 15(a)). This is to be expected as each symbol uses a separate queue as described in Section 3.2.4. Each process is therefore only activated when a message intended for that specific process becomes available.

**Case 3** is similar to case 1, but with a higher rate of increase due to the overhead of evaluating the condition for the conditional receive (Figure 15(b)). When adding another process to this case, a similar rate of increase can be seen. Performance is, however, reduced due to the overhead of scheduling between these two processes.
**Case 4** shows much greater overhead than the other cases (Figure 15(b)). The problem is that multiple processes perform conditional receive commands on the shared channel. For each message sent, these processes need to be scheduled and the condition evaluated until the messages can be successfully transferred. This is necessary because the result of the condition cannot be determined without evaluating the contents of the message.

From a performance perspective, the best way to use SELECTs is by using separate symbols for each alternative. This method can be supported with relatively low overhead whether used in a single process or with a channel shared between processes. Similarly, when conditional receive is used in a single process (or only a few processes), the overhead is acceptable. Sharing a channel among several processes using conditional receive should, however, be avoided as far as possible.

**Simple versus Complex Messages**

**Complex messages** are messages constructed from several variables. Generally, simple messages can be transferred as a single continuous buffer from the source process to the destination. Additional support for complex messages is provided by the kernel. It is, however, possible to achieve similar functionality making use of a record as an intermediate buffer and using simple message passing. Using records requires all variables to be copied to a record before sending and again extracted upon receipt. These two alternatives were compared in memory usage and performance.

The two alternative approaches were used to transfer similar messages and measured to compare the performance. Messages of 256 bytes were used. The number of components making up the message was varied. The results are graphed in Figure 16. Generally, messages transferred with kernel support performed better. For example, when transferring two 128 byte components this advantage is 30 percent.

Kernel support does, however, come at a cost. To avoid additional buffering, components are copied one by one. This is achieved by reading addresses and sizes from tables constructed at compile time. Accessing these tables require two additional memory accesses per component, loading the source and destination from the tables. These memory accesses become costly when the components become very small. This explains the performance reversal at the high end of the graph. At 64 components, each component is 4 bytes, which result in three memory transfers without kernel support (to the record before sending, from process to process, and from the received record).
Figure 15: SELECT communication under various conditions
Figure 16: Complex messages with and without kernel support
With kernel support two table accesses and a single transfer is required for each component. Similar results apply for 128 two-byte components, and for 256 single-byte components the overhead of kernel support becomes greater than the cost of intermediate buffering. These results suggest that kernel support is justified, as in most cases performance increases can be achieved. Also, in cases where very small transfers are required, records can still be used to achieve better performance.

### 4.8 Interrupt Latency

A good interrupt response rate is important to the successful implementation of many embedded systems. This response rate is limited by the use of actions in LF programs to improve model checking performance. The response rate is limited by the length of the longest actions. When moving from very short actions (after every LF command) to maximum length (only when necessary to schedule), it was known that response rate to events deteriorated. After experimenting with an initial implementation (using very short actions), the overhead of this strategy was found to be too high. It was decided to reconfigure the kernel and compiler and attempt the opposite extreme and increase actions to the maximum possible length to improve efficiency. Even with this configuration it was found that, in general, actions tended to be very short, much shorter than required by the interrupt requirements of most devices. However, for interrupt response the worst case is more significant than the general case, as it is in the worst case that interrupts are lost. To improve interrupt response time, actions that become too long, can be divided into smaller actions by adding additional, forced scheduling points.

A serial driver was implemented to test the response rate to interrupts. Interrupt response was tested with the serial port configured to transfer at the maximum rate of roughly 6600 characters per second (57600 bps). This corresponds to 6600 interrupts per second. In the test the serial driver was combined with an additional process. Adding even more processes would increase the total load on the system, but because processes servicing interrupts execute at a higher priority, the response time is not influenced. The response time is still only limited by the length of the action executing when the interrupt occurs. Thus, during measurements only a single process was used, executing a single action repeatedly. The length of the action was increased until interrupts were lost.
<table>
<thead>
<tr>
<th>Example</th>
<th>LF Model Checker</th>
<th>SPIN Model Checker</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProdCons.LF</td>
<td>208 states</td>
<td>1139 states</td>
</tr>
<tr>
<td>Peterson.LF</td>
<td>158 states</td>
<td>69787 states</td>
</tr>
<tr>
<td>DiningPhil.LF</td>
<td>5268 states</td>
<td>88 states</td>
</tr>
<tr>
<td>MutualExcl.LF</td>
<td>20 states</td>
<td>46 states</td>
</tr>
</tbody>
</table>

Table 8: State space generated by LF model checker versus SPIN model checker

In such an experiment only the worst case is of importance, because this is the case that leads to interrupts being missed. The worst case will occur when the interrupt occurs right at the start of executing an action. It was found that the driver started to miss interrupts when the length of the action increased above 800 clock cycles. This translates to roughly 150 processor instructions on the target system, an Intel386 SX at 33MHz. However, examining several small- and medium-sized programs implemented in LF, it was found that very few actions were longer than this required minimum. Less than two percent of all actions were too long.

Ensuring action execution times are kept below a specified minimum is not always a trivial task, as actions can contain loops, which may have to be partially or completely unrolled. Support for limiting actions to a certain length could eventually be provided by the LF compiler. Without such support from the compiler it is still possible for the user to ensure that limitations are met by manually adding rescheduling points.

### 4.9 Model Checking

The size of programs that can be verified by model checking is limited by the amount of memory available to store the states of the state space generated. The size of the state space depends on the size and complexity of the program being verified. The problem of a state space explosion is magnified when, as in LF, actual implementations are verified as opposed to simplified models. The main countermeasure to this problem is the execution of portions of code as indivisible actions, which greatly reduces the state space.

Successful model checking of LF programs, in general, depends on the degree by which the state space can be reduced using this technique. Some initial case studies have been performed with the LF model checker which show positive results. The state space traversed by the LF model checker compared to the state space traversed by the
SPIN model checker \cite{19, 20} typically showed significant savings. For example, the producer-consumer system described in Section 2.1, when translated to Promela and verified by SPIN shows a state space reduction of 10 times. The results from some of the case studies performed are provided in Table 8. Simple, well-known examples were implemented in LF and translated to Promela. All examples were then verified for freedom of deadlock by their respective model checkers. The examples included the simple producer-consumer example used in the discussion in Section 2.1; Peterson’s algorithm for mutual exclusion; a dining philosophers example, as well as an early algorithm proposed as solution to the mutual exclusion problem, but which contained a flaw. Both the LF and Promela implementations of this algorithm were model checked to determine whether the property of mutual exclusion is maintained.

Most examples show a significant reduction in states as compared to SPIN. The comparison is, however, without applying partial-order reduction, a technique usually applied to significantly reduce the state space that needs to be examined. This was done as no partial order reduction was available for the LF model checker. The savings in this comparison are mostly due to the technique used in LF of dividing processes into actions, and only allowing scheduling points between actions. This is especially effective if large sections of code can be executed without influence from other processes. LF processes transfer information by message passing. Peterson’s algorithm for mutual exclusion shows the significant gains that can be achieved. A reduction of 400 times is achieved. Similar advantages can be replicated in SPIN by adding atomic sections to sections of the model. For LF, however, the atomic actions are part of the implementation. Model checking applied to such an implementation will examine all possible behaviours the implementation could achieve. Any properties verified by model checking directly apply to the implemented code. In general-purpose systems, such claims cannot be guaranteed of models simplified by reducing the scheduling points. Generally processes can be preempted at any point during their execution and therefore all possible schedulings should be examined by the model checker.

The gains shown by the LF model checker over SPIN are less significant in examples that rely heavily on interprocess communication. This was shown by the Producer-Consumer example. When verifying the MutualExcl.LF program against the requirement of providing mutual exclusion, both model checkers required only a few states to discover the error. Although the gains were less significant, the LF model checker still required only about half the number of states, 20 compared to 46 for the SPIN model checker, before reaching an error state.
However, in some cases LF programs produce more states as seen in the Dining Philosophers example. The main reason for the increase in states is the differences between how channels are treated in LF and Promela. In Promela, synchronous channels (as used in this example) do not store information about the order in which messages are sent and received. In LF, on the other hand, this information forms part of the channel records as well as the ARs. Therefore, when $n$ messages are stored in a channel, one could expect the number of states to increase by a factor of $n!$, as happens in the case of the dining philosophers.
Chapter 5

Conclusion

The aim of the LF project is to provide an environment and a set of tools to help in the development of reliable embedded software. The LF language is a type-safe concurrent language with interprocess communication based on message passing. Kernel support is provided for processes, communication, and supporting device drivers. A model checker is integrated into the kernel to verify programs at a machine-code level. This thesis is concerned with the design and implementation of the kernel.

The goals of this project as mentioned in Chapter 1 were to provide kernel support in such a way as to (1) be efficient enough to support real embedded applications, while (2) allowing model checking to be applied directly to the implementation code. The degree to which these objectives have been met is discussed in this chapter and some ideas for future work are presented.

5.1 Efficient Kernel Support

From the measurements and evaluations of the previous chapter, the kernel has in general met the first goal. The basic requirements for supporting processes and simple message passing by system calls were implemented efficiently (Sections 4.6 and 4.7).

Interrupt response is acceptable, although limited by designing for model checking. The serial port, a device with a high interrupt rate, successfully transferring at its maximum rate under different loads is a case in point (Section 4.8).

LF provides specialised message passing support which comprises generalised guarded
communication, conditional receives and complex messages. By limiting guarded communication to allow only one side of the communication to be guarded, efficient implementation is possible. Without guarded communication non-determinism could not be specified in processes. This makes it the most useful of these three types of specialised communication. In Section 4.7 it was shown that the two other types of communication—complex messages and conditional receive—are not essential. Conditional receive for a single process can be emulated by local queuing. For multiple processes, either separate symbols can be used or the processes can be connected in a ring by sharing channels. Complex messages can be transferred without kernel support by making use of records.

Support for these specialised types of message passing is efficient except for the case of conditional receive performed by several processes sharing a single channel. It seems worthwhile to attempt alternative methods of achieving a similar result.

Providing kernel support for specialised message passing, although more efficient, increases the kernel size. A more complicated kernel is more difficult to maintain and also to port to different hardware. Generally, the specialised communication commands are useful and improve efficiency. The impact on kernel size is also not excessive, with the kernel size remaining below 100KB and system calls totalling less than 30. For these reasons, it seems sensible to include support for these specialised types of message passing communication.

Called processes are intended to provide for the structuring and abstraction generally associated with procedure calls. However, called processes have significantly higher overhead than procedure calls. Therefore called processes should only be used where the function to be performed is relatively complex, such as a sorting algorithm, and the relative overhead is acceptable or if performance is not critical, such as during initialisation. This somewhat limits the possible uses of called processes, but does provide a very necessary function. It is therefore important to further examine possibilities for improving the performance of support for called processes as discussed in Section 3.2.3, or possibly even adding support for procedure calls as alternative.

5.2 Supporting Model Checking

The LF kernel with integrated model checker has been used to verify actual implementation code. The requirements of the integrated model checker, for the most part,
has no significant negative effects on efficient kernel support. Whether the effect of actions, a requirement for successful model checking, on interrupt performance in real embedded software is acceptable still needs to be demonstrated. A compiler that can be adjusted to ensure all actions can execute within certain limits should make it possible to support any device (within the processing capabilities of the target processor).

To support message passing efficiently, channel structures were used to contain the queues required to synchronise communication partners. Although this allows for efficient implementation, this channel structure becomes part of the current state. Currently these channel records are stored as part of the state vector by the LF model checker. The problem is that the queues need to be updated as new states are being generated and restored again whenever the model checker backtracks. This can have a significant impact on the size of the state space explored, as shown in Section 4.9.

5.3 Future Work

The kernel implementation has so far proven to achieve the goals originally set out. However, there are some opportunities for improvements. There are also some problems relating to model checking of embedded software that still need to be resolved. Some of these problems and possible improvements are:

- **An alternative to the current implementation of ordering the messages in channel records** is necessary. The current implementation adds significantly to the size of the state space and therefore limits the size of programs that can be verified. Although the runtime system needs to maintain the message order, the model checker may ignore this. Therefore, it should be possible to sort the messages and thus reduce the size of the state space that needs to be explored.

- **Device drivers** are an important part of many embedded systems. Therefore, to be able to verify such applications, the model checker will have to be able to handle systems containing device drivers. Model checking cannot be directly applied to device drivers as the device driver is dependent on the device and the current state of the device.

Device drivers interact with hardware devices which alter the internal registers and memory of such devices. These values represent the internal state of such a device. For a driver to function correctly the internal state should match
the state as represented by the driver. Model checking requires previous states to be restored when backtracking. Restoring the internal state of a device is not currently part of the model checker and is not always possible or practical. Therefore, for device drivers to be verifiable, abstraction will need to be applied to emulate the functioning of such a device, including reading and writing of I/O ports and interrupts.

- Providing a **real-time scheduler** will extend the range of embedded applications that can be implemented in LF. Real-time embedded systems require a real-time scheduler which can guarantee that deadlines will be met. The initial experimentation with a deadline-driven scheduler will have to be extended to provide real-time scheduling. Protection against priority inversion such as the priority inheritance protocol needs to be implemented. Also, the language will have to be extended to allow specification of deadlines for responding to events. Implementing real-time scheduling in the kernel will have to be combined with a real-time model checker if the design objective of verifying actual implementation code is to be maintained. In real-time model checking, time constraints specified for the execution of the real-time system should also be considered. Only execution sequences which both function correctly and meet time constraints are accepted as valid.

### 5.4 Final Remarks

As a design and implementation task, I found the construction of the LF kernel a highly enjoyable task. In the process I also gained useful experience in constructing a rather complex system. Factors such as concurrency and interfacing with hardware posed interesting problems. The possibility of model checking at the code level, was also an exciting prospect. From initial results this seems to be a reachable goal.

As an implementation language for embedded systems, I found the language and environment expressive and intuitive. In the development of small control and monitoring applications the language proved to provide simple and elegant solutions. This is especially apparent when concurrent operations are required.

From a personal perspective, I would certainly look to LF when developing any simple monitoring or controlling application. For example, when I wanted to use an old radio tuner card and 386 machine to listen to radio transmissions, I immediately decided
to use LF. Using LF to implement a driver and user interface proved to be very easy. Such experiences proved the suitability of LF in developing control and monitoring applications to me. If the ability to verify any such application is also available, the advantage over alternatives without such facilities is clear.

The final goal for LF is to allow development of real-world embedded systems, while providing the possibility to verify the correctness of such systems using the LF model checker. Although this goal has as yet not been achieved, progress so far has been promising.
Bibliography


process Communication. ACM SIGOPS Operating Systems Review, 18(4):12–20,
October 1984.


[14] D. R. Engler, M. F. Kaashoek, and J. O’Toole. Exokernel: An operating system archi-
ctecture for application-level resource management. In Symposium on Operating 


ings of 1993 International Workshop on Object-Orientation in Operating 


September 1996.


