A compiled parallelism programming language : MultitaskC

Hugo Delchini
Email: hugo@delchini.fr

October 24, 2012

Abstract

Computing applications are often written using a high-level computer language for programming and an operating system for execution handling. A family of programming languages (CPL for "Compiled Parallelism Languages") offers features and advantages similar to a programming language coupled with a multi-task operating system. We intend here to point out that CPLs bring several additional advantages in terms of performance, especially for embedded real-time applications. As a demonstration, we developed two versions of the same application, first using a CPL and then using a common language coupled with an operating system. We then compare the respective performances.

Contents

1 Introduction **3**

2 Several development solutions **4**
2.1 Main characteristics of embedded real-time applications 4
2.2 A standalone programming language 4
2.3 A programming language with an operating system 5
2.3.1 VRTX 5
2.3.2 LynxOS and POSIX real-time extensions 8
2.3.3 The ADA programming language 10
2.3.4 VxWorks 10
2.3.5 PharOS 11
2.3.6 Features summary 11
2.4 Synchronous programming languages 11
2.4.1 Esterel 12
2.4.2 Other synchronous languages 13
2.5 Comparison of CPLs and "language/OS" 13
2.5.1 CPLs drawbacks 13
2.5.2 CPLs advantages 13

3 Compiled parallelism and MultitaskC **15**
3.1 The MultitaskC programming language 15
3.2 Compilation 16
3.3 Comparison with classical features 16
3.4 MultitaskC programming remarks 16

4 Evaluation **18**
4.1 Previous evaluations 18
4.2 Evaluation application description 18
4.3 Language/OS version 19
4.4 MultitaskC version 19
4.5 Results comparison 19

5 Conclusion **20**
5.1 Conclusions about comparisons 20
5.2 MultitaskC scope 20
5.3 Possible language extensions and further work 21

Chapter 1

Introduction

Software development for industrial applications is usually done using a high-level programming language and an operating system for execution control. The operating system offers services which can be used to implement an application on a computer.

Coupling high-level language programming with a multi-task operating system provides comfortable means for application design and implementation, so that it seems difficult to do without. This pair enables you to produce applications largely independent from the hardware, maintenance and evolution of the software is made easier.

A family of high-level programming languages - called CPLs for "Compiled Parallelism Languages" - which include operating system features (such as task definition, task synchronizations and communications between tasks) - provides the same advantages. It is admitted, from a theoretical standpoint, see [?], that these languages bring additional advantages in terms of portability, formal modeling and execution efficiency. This work is particularly focused on execution efficiency.

We intend to show that CPLs in practice really have significant advantages over the classical combination of tools, and that using CPLs enables you to devise better solutions to various problems.

In chapter 2 we describe various traditional tools and operating systems used to build applications, specially in the real-time and embedded domain. We will point out the main characteristics of common applications and how classical tools deal with their constraints.

In chapter 3 we introduce CPLs by describing several of the existing languages and we show their advantages in implementing embedded real-time applications. Then we present our CPL - MultitaskC - which is more particularly oriented towards high performance.

In chapter 4 we describe several embedded real-time applications, chosen because they are representative in terms of varied constraints. We implement two versions of each application : one using an operating system and a high-level programming language, and the other using MultitaskC. We then compare significant results in terms of time and memory-space performances.

Finally, in chapter 5 we conclude and suggest directions for further research.

Chapter 2

Several development solutions

There are several methods to program applications. Our purpose in this section is to list some of the most representative among those used for mono-processor or few-processors machines. Applications are often developed using a cross-compiler with binary codes being loaded on the target system via a suitable media such an USB link or a JTAG. Some real-time operating systems such as LynxOS make the development environment directly available on the target system.

We are now going to describe a few development environments. These are usable whether you use cross-compilation or target-compilation.

2.1 Main characteristics of embedded real-time applications

Data acquisition applications as described in [**?**] and motor-control applications that will be described here later share a set of characteristics. The main ones we will consider are :

- *They react to one or more input signals (event-triggered).* An application which is able to process an event without delaying the processing of further incoming events, is called a real-time application. In this case the response of the application can be considered as synchronous with stimulus occurrence.
- *They ensure precise clock-triggered processing.* Very often some processes must be executed at a precise rate. For example, for electric-motor control, you usually have to execute a current-control loop at a very high rate (typically 10 us cycle time). Rate's stability is very important for the overall stability of the system. It may also be important to control precisely the delay between two separate clock-triggered processes.
- *They fulfill non-triggered servicing.* Most applications need the possibility to specify whether a particular process needs to be executed as fast as possible or only when there is time to do it.
- *They support concurrent processing.* Applications often consist of several more or less independent tasks communicating via standard mechanisms of mutual exclusion and synchronization. For example, in a data acquisition application one task will do data readout and another one data transfers. This way independent hardwares can be operated simultaneously and the total dead time is reduced (provided asynchronous interfaces are available). For maximal efficiency the programmer should be able to specify each task individually as non-triggered, clock-triggered or event-triggered.
- *They interface with specific hardwares.* Real-time systems for apparatus control deal with varied hardwares whose characteristics have a profound impact on the software. Interfaces between the processor and the hardware are thus usually quite different from one platform to the other, and it is very important that the programming environment makes it easy to develop and use specific hardware interfaces.

2.2 A standalone programming language

This approach is the oldest one and many embedded applications have been programmed this way. It consists in using a high-level programming language, possibly complemented with assembler parts, to write a code that is executed independently of any operating system. The programmer is in charge of everything and cannot use any of the complex objects devised for application design and structuring (tasks, means of communication and

synchronization, ...). The execution environment boils down to the hardware and a possible debugger during the development phase.

The main point here is choosing the right programming language. A standard high-level programming language with many useful libraries, including if possible some libraries of basic operating system features, can be advantageous. However the choice may be limited by the availability of compilers for the hardware target. Whenever possible, a widely used standardized programming language makes a good choice.

Several languages meet these criteria. We will retain C programming language, as described in [KR:88] and its object oriented variant C++. Note that when using C++ the programmer should pay attention to the resources (such as memory) required by each objects, especially for constructors and destructors, as the underlying management of these resources can introduce an important overhead during execution. Actually all objects should be declared as static memory and no dynamic allocation should be used, at least after initializations. In a real-time application those hidden costs can become hazardous, as it is difficult to control them precisely enough to be sure that the application's constraints will always be respected.

Originally, the C programming language had been designed for operating system development. It is fit for low-level programming, and this why it makes a good choice for people interested, as we are, in industrial embedded real-time applications. The second important advantage is that C compilers are available for most processors. The well known "gcc" compiler from the GNU project for example can deal with most of the common hardwares (moreover, it has options to be ANSI-standard compliant). This compiler suite also offers C++ support, together with other popular languages such as FORTRAN. A standardized debugging interface is provided with "gdb", as well as binary tools for specific linking. Like all modern compilers it implements very high-level optimization techniques such as inter-procedural analysis, for example.

Many real-time embedded applications have been programmed without using an operating system for execution control. It is a long-standing programming method. Applications written this way consist mainly of interrupt-routines dedicated to event-triggered processing. For clock-triggered processing, an interrupt can be generated by a timer at the desired rate . Other interrupts can be used to indicate the availability of input-data or to start a data transfer.

High-level programming languages such as C don't dispense from writing some parts of the application in assembly language. Typically because some instructions or registers are processor-specific and cannot be dealt with by high-level languages. Unless coding and precise structuring rules are defined, the programmer is allowed to do whatever he wants. As a consequence diverging coding styles may become a problem, particularly when large teams are involved, but more generally it is a problem as far as maintenance is concerned. One can avoid such drawbacks however, since these simple programming tools make it easy to structure the application as a set of finite-state automata for example, or even Petri nets in case of multi-core architectures.

Such an approach, though quite basic, is widely used. The only point we'll keep in mind is the use of the C programming language. We are now going to give an overview of how to combine this programming language with an operating system.

2.3 A programming language with an operating system

Such a development environment has been commonly used for many years in the real-time community. More and more micro-controllers are able to run operating systems, as the resources needed to do so are more and more available. Let's have a look at some operating systems and other similar tools. In each case we consider the programming language used is C.

The operating systems were chosen because they are representative of what is commonly used in the industry. PharOS comes with its own programming language, but it is a language very similar to C. We mention OS9 which is an early real-time operating system but we'd like to focus on more recent ones. Note that VRTX is the ancestor of VxWorks.

We will also do an overview of the ADA programming language, explain below why we consider it as a language coupled with an operating system. In the section dealing with the evaluation of development environments we consider PharOS with the PsyC programming language. In [?] we use VRTX and LynxOS with the C programming language. We have used C and C++ with VxWorks in an other evaluation that will be described shortly.

2.3.1 VRTX

All the information we present here is taken from [?] and [?].

VRTX ("Versatile Real Time eXecutive") is a real-time fixed-priority operating system designed for the development of industrial applications with constraints in terms of performance and reactivity. It consists of

a customizable kernel that can be configured to fit exactly the needs of each application. We've done all the evaluation tests using a version customized for the AMD29000 micro-processor, but there are other version for common targets such MC68000 and others.

Developments were done in C using a cross-compiler on a PC. Application code is downloaded onto the target via an industrial bus called VME. An on-target environment for VRTX is also available but we didn't use it. The main characteristics of this operating system are that it is a mono-user system with multi-tasks capability and a fixed priority scheduler. There are three execution levels : supervisor mode, user mode and a mode for interrupt routines. In the version used in [?], the operating system shares the physical address-space for data and instructions with user tasks. There is no memory segmentation or virtualization, neither memory protection via a MMU.

Configuring VRTX to support paginated memory is possible but this is not the option by default. For performance considerations we preferred to have simple memory-management because page-tables handling during contexts commutations introduces an additional system latency. This feature however could be interesting for safety reasons, as we will see with PharOS and LynxOS. System code can be flashed into ROM, which is a common custom for embedded applications. Note that this has an impact on performances because ROM access-time is usually much slower than RAM access-time.

System calls do not conform to any standard but the C compiler we use conforms to ANSI C. The kernel is available as an assembly source file describing in a big data area the binary code of VRTX. This can be used to link the operating system with additional software such as a startup code specific to the mother-board (board support package), or various drivers. As the board-specific software has to be written by the users, it is possible to configure the kernel for any particular use. Several initializing informations must be defined, such as the maximum number of tasks, the stack-size, etc. Writing this piece of software is not easy, and once it has been done you still have to develop the entire application.

VRTX can be configured with different hooks to handle specific processing related to tasks management. There is one hook for task creation, one for task commutation and one for task deletion. In our implementation we also used hooks to manage the AMD29050 floating-point coprocessor, in order to backup and restore the FPU registers during context switching. If memory protection and segmentation is used, other hooks are available to handle the MMU/MPU contexts during the same varied task operations. For performance considerations, in our evaluation we dismissed MMU/MPU management but we used the FPU for motor control.

The operating system handles dynamic memory-allocation as well as basic input/output functions over a serial line if the related driver are included in the configuration software. Standard C library functions such as printf/scanf and malloc/free must be also implemented by the user.

The AMD29000 version of VRTX reserves several registers of the processor to the operating system. Such a feature improves execution efficiency by reducing the number of registers to save/restore during context switching, without being detrimental because the hardware implements a large set of registers (192). As regards the number of registers dedicated to each task, note that although a large number of registers helps the C compiler to do optimizations, it implies a significant cost during task commutations. Since the MC68000 processor architecture implements much less registers, no registers are reserved to the system in the MC68000 version of VRTX. We are now going to review some features of the operating system, being understood that many of the preceding characteristics can be found in other real-time operating systems.

Tasks. As we said before, VRTX is a multi-tasks operating system. A VRTX task corresponds to the execution of binary instructions by the processor. The operating system handles time sharing between all the ready tasks of a given priority level. A task can create other tasks but there are no ancestor or sibling relationship between them. A task can get no information about it's ancestor. Task creation and deletion is transparent to the system except as regards the priority of context switching. When a task terminates executing for instance, there is no implicit meeting between the task and the task which created it. Each task is allotted a priority when it is created. Priorities enable the system to determine which task among the set of ready tasks will run at any given time on the processor (scheduling). VRTX is using a fixed-priorities scheme, unlike time-sharing multi-tasks operating systems such as UNIX. In these systems, the priority of any given task varies with time. The priority of an intensively time-consuming task will be lowered to avoid processor starvation for the other tasks. Its priority will be raised again later to its initial value when the system considers it should be allocated some processor time anew. This is called a "fair operating system". Like many fixed-priority operating systems, VRTX is an unfair one : the highest-priority task may get hundred per cent of the processor time.

There are two scheduling modes in VRTX. In the first mode, all the ready tasks of highest priority are executed in a round robin manner, each task running for a maximum time called a quantum (typically 10 milliseconds). The operating system gets processor control at the end of each quantum via a hardware clock

interrupt (timer). In this mode the system is fair inside a given priority level. In the second mode, the quantum is infinite so that the running task must relinquish the processor (either explicitly or via a system call) to allow execution an other task.

We have mentioned previously that in our VRTX configuration there is only one address-space shared by the user tasks and the system. Another feature to be mentioned is that a private stack is allocated to each task so that it is possible to implement a function-call mechanism with parameter passing as well as local dynamic allocation of variables in a function. However there is no stack-memory protection in our configuration.

Let's review shortly now basic tasks-related functions. *Create* : creates the task, specifying priority, entry point, and execution mode (supervisor or user). *Delete* : immediate cancellation of a task (a task can also terminate itself). *Suspend* : the task goes to the non-ready queue (a task can also suspend itself). *Resume* : once suspended, a task can be resumed by another one. *Priority* : changes the priority of the current task. *Information* : recovers the address of the system internal task-control block containing the relevant informations.

VRTX is an unfair operating system because it is designed for real-time application control. A high-priority task can monopolize the processor, releasing it only when necessary. Other system calls allow a task to lock context-switching or to define windows of execution with limited quantum. Once the operating system is locked, it behaves as a mono-task system with only one task running. The operating system can also be unlocked, of course. Last point, there is one task of lowest priority in the operating system - the idle task - that is executed when no other task is ready to run. The default idle task can be replaced at configuration time by a user idle task which can be used to implement a low-priority activity.

Communications between tasks. VRTX provides four types of objects to deal with communications between tasks. These standard objects, which can be found in other operating systems, allow the user to implement simple (producer/consumer for example) or complex communication schemes. Communication tools are listed below (note that it is possible to specify a time-out for blocking system-calls). *Mailbox* : a mailbox is a 32-bits memory where the user can post a value (non-blocking action), an error being raised if the mailbox is full. Symmetrically, to pend on a mailbox is a blocking action if the mailbox is empty. If more than one task is pending, the highest-priority task or the next task in the same priority level will be served. *FIFO* : a FIFO is a set of 32-bits values. The user specifies the FIFO size when he creates the FIFO, and similarly to a mailbox, posting to a FIFO is not blocking while pending on a FIFO is blocking if the FIFO is empty (but it is possible to check the case before pending). A user can ask for the current FIFO size then jam a new element into it. There is a system call to get a copy of the next element in the FIFO without removing it. Two policies for reactivating a task pending on a FIFO are available : either the systems activates the task of highest priority (as for mailboxes) or it activates all the tasks waiting for that FIFO in sequential order of arrival. At FIFO creation, the user can specify the policy to be applied. *Flags set* : a flag set consists of 32 flags, each being one bit in a 32-bits word. In order to check the flag, the user gives a mask and a parameter indicating if he requires an "and" or an "or" wait condition. If the condition is satisfied, checking the flag is not blocking. Checking a flag never changes its value, there is a system call to do that. More than one task can be awakened on a flag condition at the same time, task-execution being done according to priorities. *Counting-semaphores* : a counting-semaphore can have a value between 0 and 65535. It is a well-known concept introduced by Dijkstra to handle mutual exclusion while accessing to a shared resource. The user can increment a counting-semaphore (non-blocking action). He can also decrement it : if the value is zero, the calling task is blocked. Task-awakening options are the same as for FIFOs.

Lastly, VRTX provides a mechanism called "shared memory" data between tasks. This is possible because, as was said before, there is only one address-space in the system. Combined use of semaphores and shared memory allows the user to implement communications schemes as complex as necessary.

Interrupts. Some key points concerning interrupts in general. The concept of processor interruption by peripherals is a basic mechanism for operating system design. System calls are usually implemented through software interrupts that are similar to hardware-generated ones, but are raised by a specific processor instruction. When an interrupt occurs, the processor execution flow is restarted at an address called an interrupt-service routine address. Execution mode is automatically set to privileged and further interrupts are disabled. Several registers are saved either on a stack or into additional special registers that can be afterwards saved in memory if necessary.

This mode of operation allows the operating system to manage all resources coherently : save/restore task-contexts, handle communications, etc. all this can be done without interruption during critical phases. The operating system also controls peripherals via interrupts, such as for instance a peripheral telling the processor

that data is available by raising an interrupt. In real-time programming interrupts are very often used to trigger a process, for example a data-acquisition one. Interrupts can also occur at specific rates for clock-triggered actions.

As regards interrupts, two different operating-systems behaviors are possible. Interrupts may themselves be executed atomically while the operating system is running in privileged mode and system calls are then atomic. They execute without being interrupted ; if an other interrupt occurs, the corresponding handler execution is delayed. These operating systems are said to be non-preemptive in privileged mode. This behavior has a detrimental impact on operating system latency, and is not suitable for real-time applications. VRTX is preemptive in privileged mode, so operating system implementation is more difficult but interrupt latency is better. VRTX is designed to remain as much as possible open to interrupts, thus minimizing the delays due to handlers.

In VRTX, a special stack to backup registers during the interrupt-routine prologue is available. Such a routine can be activated by hardware without indirection by the interrupt driver, as it is the case in many real-time operating systems. This reduces interrupt latency. An interrupt-routine can communicate with the task level according to fixed priorities, for example by posting a value in a mailbox in order to trigger the task-level process. However, such a mechanism must comply to a system protocol : ie, if any system-call is used inside an interrupt-routine, the interrupt-routine entry and exit must be signaled to the system. At the end of an interrupt routine, control is given back to the operating system in supervisor mode so that rescheduling can take place. This protocol is time-consuming but actually less than the protocol implying interrupt drivers in many other operating systems. Note also that for some specific applications, it is possible to do without this protocol in order to get better interrupt latency.

We are now going to describe an other operating system dedicated to real-time applications.

2.3.2 LynxOS and POSIX real-time extensions

Before describing the LynxOS operating system, we are going to give some general information about the POSIX standard ("Portable Operating System Interface", X standing for similarities with UNIX). Information listed here is essentially taken from [?], [?] and [?].

POSIX reference number is 1003 in the IEEE directory ("Institute of Electrical and Electronics Engineers") which intends to establish industrial standards. POSIX is also referenced as the international standard ISO 9945 ("International Standards Organization"). An ISO standard contains the description of all the programming interfaces between applications and the operating systems that implements it. These interfaces specify data types, type names, function names and semantics. As a POSIX-compliant application can be considered operating-system independent, and as C is also a standardized system-independent programming language, we'll assume POSIX and C to be our reference framework. Most UNIX implementations implement the POSIX interface, and this is also true for VxWorks and LynxOS. There are several subsections in POSIX, the basic features are in 1003.1, real-time extensions are in 1003.4 and real-time threads extensions in 1003.4a.

The LynxOS operating system is intended to be POSIX and ANSI-C conforming, which is an interesting approach. The ANSI-C part is based on GNU compilers and libraries. As regards the operating system interface, LynxOS includes all of POSIX (including real-time extensions). Besides, LynxOS is compatible with the two main UNIX flavors : "System V" and BSD. LynxOS is a multi-tasks and multi-users operating system. Execution levels are taken into account as in VRTX but in addition, LynxOS implements a notion of user as UNIX does. LynxOS is available for several targets such as IBM PC, Motorola VME-boards based on MC68000 or PowerPC, SUN Spark workstations, several HP embedded platforms, IBM PowerPC based computers, etc. The operating system is delivered in binary code together with a collection of object files allowing the user to customize it. The complete development environment is located directly on the target system.

The kernel size may range from 200Kb to more than 1000Kb (all extensions included). It can be stored in ROM, with a minimal ROM file system. Alternatively, it can be booted via a network. These two possibilities allow easy deployment of applications designed for minimal hardware configurations, as well as easy implementations of the same software on multiple hardware targets.

Although LynxOS integrates a lot of UNIX inspired standards, its kernel is really different because it is designed for real-time applications. Its kernel design is very similar to VRTX, with the difference that the development environment is located on the target. Our evaluation system is based on a MC68040 VME board with LynxOS installed on a local hard disk.

Memory management is paginated and can be also virtual. Each process has its own private address-space. Like any user process, the kernel has its own address-space ; however, kernel pages are always resident in core memory. When virtual memory management is not enabled, all the processes pages are also resident in core memory. If virtual memory is to be used, a swap area must be defined on hard disk to store the pages that cannot temporarily stay in core memory. Users processes and the kernel cannot access each other address-spaces,

a memory segmentation that constitutes a feature of safety.

Processes and activities. The basic entity for execution in LynxOS and UNIX is the same : the process. However, when the process of UNIX has a time-dependent priority, the process of LynxOS has a fixed priority. The scheduling of its processes is based on queues of different priorities, with a round-robin allocation of quantums inside a given priority queue, as in VRTX. The quantum can differ from one priority level to the other, so, as regards scheduling LynxOS is fair only for tasks with the same priority (this is close to real-time processes handling in the "System V.4" version of UNIX).

When virtual memory is used, LynxOS swapping mechanism is handled by a system process which itself has a priority as any other process. This means that the relative priorities of the processes have an influence on memory management. A process with a priority higher than the memory-management process will never be swapped when a process with a lower priority can be swapped. In addition, a process can lock pages in core memory without modifying its priority. This feature must be used with caution because it may lead to memory starvation. A LynxOS process can be seen as a unit of resources encapsulation, such as file descriptors, memory space, etc. The execution of a program is implemented as a "thread". Any LynxOS process consists of one or more threads that are running in the process address-space.

As for VRTX tasks, each thread owns a private stack for function calls and automatic variables allocation. The thread is given a fixed priority when created. A LynxOS process always has an initial thread, a specific feature not included in the POSIX recommendations. A LynxOS process with its initial thread is equivalent to a UNIX original process. As any thread can monopolize the processor, LynxOS scheduling is not necessarily fair. LynxOS threads implementation conforms to POSIX, for more informations about POSIX threads see [?].

We are now going to list the main operations available for thread control under POSIX. *Create* : thread creation. The operating system assigns a unique thread identifier that is usable for further operations. Some parameters set priority, stack size, scheduling type (FIFO, ROUND ROBIN, DEFAULT), etc. There are defaults values for these attributes. *Self termination* : a thread triggers its own termination, returning an exit code that can be an address in the stack. As stack memory is not released after termination, this data area remains accessible by other threads. *Termination by another thread* : a special exit code is returned. *Detach* : releases all the resources of a terminated thread. Several scheduling algorithms are possible : DEFAULT scheduling combines a fixed priority and a specific quantum for each priority level, ROUND ROBIN and FIFO scheduling are similar except that there is only one global quantum for all priority levels, finite for ROUND ROBIN and infinite in the second case.

Communications. LynxOS implements two sets of communication tools, one for inter-processes communications and the other for threads. Inter-processes communication tools are implemented as a mix of UNIX "System V" and BSD flavors, LynxOS including IPCs ("Inter Process Communications") and STREAMS from "System V", and UNIX domain sockets from BSD. POSIX standard changed recommendations in 1003.4 Draft 9 concerning the "System V" IPCs. The use of communications tools (semaphores, shared memory, message queues) is made easier. For threads running in a process, additional simple tools have been designed so that the user can implement more efficiently complex communication schemes.

Besides shared memory inside a process address-space, LynxOS implements POSIX tools. *Rendezvous* : a thread can wait for the termination of another thread via a rendezvous. When the termination occurs, the waiting thread gets the exit code of the terminated thread. *Mutexes* : a mutex is similar to a binary semaphore but only the thread owning the mutex can release it. A thread is blocked when it tries to get a mutex that has been taken by another thread. This is used when mutual exclusion between threads is needed, for instance when shared resources must be accessed in a critical section. *Condition variables* : combined to mutexes, it allows complex synchronizations between threads. Once a thread owns a mutex, it can wait on a condition variable. The mutex is released while the condition is false. When a thread signals the condition variable, the mutex is automatically taken and the waiting thread is awakened. It is possible to wakeup selectively one thread among a set of threads waiting on the same condition with the same mutex. One can also wake up all the waiting threads at the same time. A thread can also use inter-processes communication tools, and LynxOS includes tools that are not required by POSIX, such as fast binary and counting-semaphores or shared memory.

Software signals similar to the UNIX ones can be sent to processes or threads. A signal sent to a process is randomly taken by one of the process's threads accepting that signal. If none of the threads accepts the signal, it stays pending for a thread to accept it. A signal can also be sent to one particular thread of a process.

Interrupts. LynxOS does not allow direct catching of hardware interrupts by an application. To the designers of LynxOS this direct mode would have lead to the implementation by the users of services that should be

handled by the operating system. This direct mode (as we have seen in VRTX) is better as regards interrupt latency but it necessitates writing more code to be implemented. With LynxOS, the only way to handle interrupts is to use system drivers. Only drivers can attach a hardware interrupt to a handler functions. Usually, interrupt routines are as short as possible, their main duty being to trigger one or more system threads to deal with more time-consuming processing. These threads are executed in kernel space. As they are themselves scheduled, the user can manage interrupts priorities.

A particular counting-semaphore type is used by the interrupt handler to communicate with the system-threads. There is only one kernel handler that catches all hardware interrupts, prepare the execution context, and then executes the driver handler if one is installed. This makes it easy to maintain system coherency but introduces interrupt latency.

2.3.3 The ADA programming language

All the information listed here comes from [?] and [?]. The ADA programming language is a high-level language which integrates a lot of features. One of its field of application is real-time development and we will describe here ADA as a programming language allowing to specify applications with tasks communicating asynchronously. There are two standards describing ADA : ANSI and ISO.

Tasks. It is possible to declare tasks in an ADA program. Execution of the tasks is either controlled by a kernel which is linked with the executable code of the application, or by an operating system running independently on the host and not included in the ADA compiler itself. In the first implementation, the ADA compiler builds an executable by compiling the source files and linking the objects with a micro-kernel. In the second case, the compiler generates the extra code needed by the host operating system to create the tasks.

A "PRIORITY" pragma is available to assign priorities to ADA tasks. These priorities have different significations depending on the host operating system. No scheduling algorithm is specified in the ADA standard, not even for tasks of the same priority. An ADA program consists of tasks similar to UNIX processes, LynxOS threads or VRTX tasks. This is why we can see ADA as an operating system coupled to a high-level programming language.

Communications in ADA. The language includes essentially one communication tool : the binary rendezvous with data passing. An ADA task can accept a rendezvous with another task via the "accept" operator. The rendezvous is realized when at least two tasks are waiting on the "accept" operator. If there are more than two tasks, priorities may influence how the two tasks will be selected. A rendezvous is blocking as long as only one task is trying to accept it. It is optionally possible to transfer data when a rendezvous is realized, similarly to the parameter passing mechanism of the function call.

The "select" operator makes it possible to wait for more than one rendezvous, according to the state of local variables of the task. In that case, only one of the possible rendezvous will be realized ; if none is possible, the user can specify a default action to take place. If more than one rendezvous is possible, tasks are selected to realize the rendezvous according to priorities - the highest priority being selected. If all the tasks have the same priority, it is the scheduling implementation of the underlying operating system that will decide. It is also possible to declare shared-memory segments in ADA which can be combined with rendezvous to implement complex communication-schemes. Usually, simple rendezvous with data exchange will suffice.

Interrupts. It is possible to associate a hardware interrupt to an ADA rendezvous. In that case only one task is allowed to wait for this rendezvous. When the interrupt is taken, the corresponding rendezvous is realized and the waiting task is awakened. The task can get read-only data when the rendezvous occurs. An interrupt can be seen as the highest-priority task of the system, with a priority always greater than any real tasks. This way, the rendezvous associated to an interrupt is at the highest priority. The scheduling algorithm of the underlying operating system is not involved in interrupt rendezvous. Interrupt-handling implementation by the host operating system is not specified in the language standard. One can use a direct handler without an interrupt driver or any other implementation technique.

2.3.4 VxWorks

VxWorks is a multi-tasks operating system very similar to VRTX, obviously because VRTX was the kernel included in the initial versions of VxWorks (VxWorks stands for "VRTX Works"). Tasks management and communication tools are thus almost identical. VxWorks comes with a very user-friendly configuration tool for board support-package and kernel features. Its version 5 is called "Tornado" and version 6 based on

"Eclipse" is called "Workbench". This configuration tool is also an Integrated Development Environment for the programmer, allowing to develop in C, C++, and other gcc-supported languages (cross-compilation).

In version 6 the VxWorks kernel introduced Real Time Processes implementing memory segmentation in the POSIX way. This can be useful for applications needing safety features. Interrupts are handled in the same way as LynxOS with an interrupt driver. The operating system is available on mainstream hardwares such as IA32 and PowerPC. Note that during the development phase of an application, the user can include a shell with a minimal C interpreter that allows to load binaries and to call functions with parameters from the shell. Once the application is ready, the user can remove that shell from his VxWorks configuration. VxWorks also offers a very good instrumentation tool for kernel and applications called WindView. This can really help the user to improve performances and robustness of the application.

2.3.5 PharOS

More informations about PharOS can be found in [?]. The PharOS operating system has been developed for safety-critical applications. It is mainly composed of the PsyC programming language and a real-time kernel. The PsyC language, basically similar to C, includes the possibility to describe tasks and provides "Temporal variables" and "Message queues" as communication tools. The particularity of this operating system is that it is clock-triggered. The user can define clocks in PsyC sources and then all processes must occur within temporal windows defined in tasks. Any treatment must end before its deadline and scheduling is done according to the Earliest Deadline First algorithm.

Interrupts can be handled with PharOS : they are time-tagged when they occur and associated with a process of the clock-triggered space. Event-triggered processing is thus converted to clock-triggered. There is an automatic memory segmentation, with MMU or MPU hardware to do the spatial partitioning. This is similar to the partitioning realized in VxWorks or LynxOS. Spatial or temporal partitioning violations can be handled by the operating system within task groups. When a group of tasks is defined in an application, if one task of that group has a failure, the entire group is restarted at the entry point, with an indication that it is a failure-recovery startup. PharOS is available for IA32, PowerPC, MC68000 and ARM targets.

2.3.6 Features summary

We are going here to summarize the useful characteristics of the development and execution tools previously described. Except in the case where one uses a standalone programming language, they all offer the possibility to specify an application as a structure of tasks that can communicate and synchronize. This is a very common design scheme for real-time applications, and it is necessary to follow it if one wants to have a good development tool.

The operating systems described are preemptive in privileged mode, except PharOS in micro kernel mode. As practically all real-time applications they are using interrupts, an almost mandatory characteristic to reduce system latency.

With or without cross-compilation, it is possible to program in a high-level programming language and debugging tools are available. Assembly language can be used in special cases to access particular registers or instructions, its usage being minimized to make portability easier. Note that operating system portability is not a simple matter, even if one uses a standardized high-level language, because various features are hardware dependent.

The family of programming languages we are now going to describe provides a good answer to the portability problem, as well as several other advantages.

2.4 Synchronous programming languages

Synchronous programming languages with compiled parallelism (such as Esterel) appeared in the early eighties. They implement the synchronous product of deterministic finite-state machines defined by [?]. We will introduce several of these languages starting with Esterel, which is one of the firsts. After that description, we will compare the OS/language approach to the CPLs's one. Then we will introduce MultitaskC, the language we have conceived. The aim of MultitaskC is to provide the programmer with the same capabilities as an operating system coupled to a high-level language, but with compiled parallelism for greater efficiency. In the evaluation phase, we use MultitaskC to represent all CPLs as it implements the general paradigm of the concept. Informations about Esterel, Lustre and Argos come from [?].

2.4.1 Esterel

Esterel was developed by a team common to the "Centre de Mathematiques Appliquees" of "Ecole des mines" and INRIA ("Institut National de Recherche en Informatique et Automatique"). It is a programming language that mainly allows the user to specify and validate behaviors. This explains why, for example, it does not offer the complex data structures that can be found in familiar languages. With Esterel one can specify an application as a set of communicating tasks.

An Esterel program describes a reactive system, meaning a system which reacts to external stimuli and emits other stimuli as a consequence. The result of an Esterel compilation is a finite state machine coded in a high-level language called host language (C,ADA,...). Automata transitions are implemented in a host language engine. After compilation, the execution of this engine on the hardware target realizes the behavior described in the original Esterel code. The execution consists in chaining transitions by the automata engine. The use of a host language allows easier portability.

One reactive system usually does not implement an entire application. An application is made of several reactive systems that are activated by a main program triggering transitions in each engines while handling other functions (initializations, interrupts,...). The language syntax is very close to PASCAL and ADA though the underlying execution model is completely different. There is a C front end for Esterel called "Reactive C" that may be easier to use.

In an Esterel program it is possible to define local variables, to call host language functions, to define instructions sequences, to have control statements such as loops, tests and common computing expressions with the usual operators. Users can define modules that can be included in other programs, so that application structuring is possible. There are no global variables in Esterel.

Tasks and communications. The "||" operator allows the user to specify independent tasks. Communications between tasks - like external communications - is made by signals. This is the only communication tool in Esterel, there is no shared memory for example. A signal can be raised externally or internally for the reactive system. If a task raises a signal ("emit() operator") then all pending tasks (with "do halt watching signal") are awakened. Waiting on a signal can be blocking but a task can also just test the state of the signal (with "present"). A value may be associated to a signal and modified at emission of the signal. The task taking a signal into account can get its value. As this is also true for external signals, values can be transmitted to the outside world. The interface between the reactive system and the outside world is simple. The signal emission from the system to the outside world is translated into a function call in the host language, and the developer has to implement a function for each output signal. For each input signal, the Esterel compiler builds an input function that should be called by the user to raise the signal internally. A signal can be both for input and output, in which case there are an input and an output function.

Execution of a single reactive system can be seen like this : a main program raises signals to the reactive system by calling associated functions, then it calls the automata engine for the next transition that will trigger signals emissions. This execution scheme can be quite efficient because there is only one context to handle for all the reactive systems.

Parallelism. Task handling in Esterel is completely different from task handling in the operating systems we've described before. When several tasks are described in an Esterel program, parallelism is so to say compiled. After compilation there is only one task which realizes the execution of all the original tasks. As a consequence it is not possible to create or destroy tasks dynamically. A multi-tasks operating system simulates parallel execution of tasks by scheduling dynamically the runtime, but with languages like Esterel, scheduling is statically evaluated at compile time. Each Esterel task is considered as a finite state machine and parallelism is compiled by executing the synchronous product of all the input state machines. The result is a state machine whose execution will simulate the concurrent execution of all the operand state machines. In MultitaskC we also use this principle. Finite state machines are well known mathematical objects that can be used to formalize behaviors. A synchronous product gives out two things : a formal model of program execution that can be used for proof of properties and a compilation of the parallelism.

Interrupts. With Esterel as with any operating systems it is important to be able to handle interrupts. A natural way to associate a signal to an interrupt is to call the signal input function in the interrupt handler. The signal occurrence is then taken into account at execution level. This is quite similar to the way operating systems handle interrupts, but there is one significant difference because here interrupts can be allowed all the time (except during interrupt handling itself), and thus latency is minimized.

2.4.2 Other synchronous languages

Quite a lot of synchronous languages have been implemented, we are now going to briefly describe two of them.

Lustre. The result of a Lustre compilation, as for Esterel, is a finite state machine coded in a high-level programming language for the host. Lustre being a declarative language, a Lustre program is made of operators declarations (nodes with inputs and outputs). Elementary operators can be combined to build more complex operators. Lustre is based on the data-flow paradigm. This means that a Lustre program is a network of operators. For example, an operator may have two inputs and one output, output which can be connected to other operators, and so on. Any operator of the network can be executed if all its inputs are valid. This execution model integrates parallelism if all the operators ready at a given time can be executed concurrently. A Lustre program execution is cadenced by at least one clock. Each clock tick triggers the execution of all the ready operators. The compilation of operators is realized by a synchronous product as in Esterel.

Argos. Argos is almost equivalent to Esterel. The main difference is that it implements a graphical syntax. An Argos program is made of graphical finite-state machines that can be combined with a parallel operator. The result of the combination of automata is an automata that can be in turn combined with others. Using a synchronous product, an Argos compilation produces an automata coded in a host language.

2.5 Comparison of CPLs and "language/OS"

All the programming solutions we have described allow the user to build real-time applications using classical features (communicating tasks). We are now going to compare the CPLs and OS/language approaches. After this comparison we will try to verify some points in the evaluation section.

2.5.1 CPLs drawbacks

A limitation of CPLs comes from the synchronous product used to compile parallelism. Potentially, the number of states and transitions of the result grows exponentially with operand sizes. External stimuli can also increase this complexity. The limitation is largely theoretical and is rarely reached in practice. However such a limitation does not exist with an operating system. We will describe later in MultitaskC how to avoid this exponential complexity problem by using a particular way to implement the synchronous product. Similarly, the memory space requirement of a synchronous product is potentially exponential. A drawback that again is irrelevant with an operating system, even though an operating system needs extra memory space for its own data.

An other limitation concerns the dynamic creating/deletion of tasks, allowed by an operating system but not by a CPL. In a CPL based on a synchronous product calculated at compile time, tasks and communication tools have to be described statically. We will also present later the particular runtime synchronous product that we have implemented in MultitaskC in order to make the dynamic handling of objects possible in a CPL.

2.5.2 CPLs advantages

CPLs allow the user to write programs completely independent from an operating system. An application written with a CPL is more easily portable, as it depends only on a standard C library. Using a host programming language is also better for portability. In case of hardware upgrading things are also easier with a CPL. For example, if one wants to add an arithmetic co-processor, with an operating system the context of each task may be impacted by the necessity to take into account the FPU context. This requires a modification of the operating system and has a negative impact on performance. With a CPL the integration is taken care of transparently by the host language compiler.

A CPL allows the programmer to specify an application by using the standard features available with operating systems, such as tasks and communication tools. So the programmer doesn't have to integrate new paradigms. We will see later that for any given hardware, replacing an application based on an operating system by its CPL-based equivalent boosts the performances. Another difference relates to interrupt handling, which is very important for real-time applications. In a real-time operating system many sequences must be executed with interrupts disabled. This is absolutely necessary to provide system coherency, during scheduling for example, when one must deal with linked lists. Some system calls must also be executed atomically from the task level. With a CPL, it is possible to keep interrupts enabled all the time, except of course during the execution of the handlers prologues and epilogues. There is no need to have critical section because there is only one task context and only two execution levels : the application level and the interrupt level.

With an operating system, communications between interrupt and task execution levels are costlier than with CPLs. In an operating system, signaling a counting-semaphore will trigger scheduling and a lot of other actions to activate the task level. With a CPL, the interrupt routine can be reduced to the incrementation of a simple counter (without the need of a critical section) just to tell the task level it has to start some new processing. Interrupt routines being thus as short as possible, more time is available for task level processing.

In multi-tasks operating systems, each task needs a data structure usually called a context, consisting of all the processor registers, optionally some MMU/MPU descriptors, and a stack memory-address. Whenever a task is chosen by the scheduler, the previous task's context must be saved and the new task one must be restored. This is called context commutation. There is no dynamic scheduling nor context commutations with the compiled parallelism of CPLs. No context commutation is needed either for communications and synchronizations, since everything is statically handled at compile time, so there is overall no cost at runtime. This is why with a CPL it is possible to control very finely the processor execution time. Thus, unlike with an operating system, a task can release the CPU even for a very short time to another task without any loss of efficiency.

The final target compiler, when it compiles the host language program that is the result of the synchronous product, gets a unified vision of all the application tasks. This allows its optimizer to do global time and space optimizations, a feature that is inaccessible when one uses an operating system because in that case, at compile time, the system code is out of the compiler's reach and the various tasks codes are dealt with separately.

Chapter 3

Compiled parallelism and MultitaskC

3.1 The MultitaskC programming language

MultitaskC's syntax is an extension of the C language syntax. We will consider that the reader is familiar with this programming language as described in [**?**].

The first MultitaskC extension of C's syntax concerns the possibility to describe independent tasks. What we call a task here is a C statement (or compound statement) in parallel with at least another C statement (or compound statement). We have implemented the same paradigm for parallelism compilation as in other CPLs, and we are targeting mono-processor architecture. The result of a MultitaskC compilation is a finite-state machine coded in pure C. Execution of this automata simulates the parallel execution of all the tasks. Common communication and synchronization tools are implemented : rendezvous and the Esterel synchronous broadcast. Use of shared memory is supported via global variables.

The user has the possibility to declare meetings or rendezvous at the beginning of a function, the scope of the rendezvous being the entire function. The declaration follows this grammar :

```
meeting_declaration :  meeting meeting_list ';'
meeting_list :  'id' ':'  'constant'
meeting_list :  meeting_list ',' 'id' ':'  'constant'
```

Here are some valid rendezvous declarations. The constant indicates the number of tasks that are needed for the rendezvous to occur (respectively 2, 3 and 4) :

```
meeting rdv:2;
meeting rdv1:3,rdv2:4;
```

All C instructions are included in MultitaskC, such as "for()", "while()", or "if()" control statements. Expressions for tests and statements are also the same, as well as all operators and declarations. All expressions are executed atomically, and there is an operator to render a sequence atomic : "group". For example, the "for()" loop is atomic :

```
group { for(i=0;i<j;i++) func_call(i); }
```

For more complex instructions the following rules are added to the standard C grammar (where "id" stands for a block identifier or a rendezvous) :

The "join()" and "when()" instructions allow to communicate with rendezvous (that must be declared before use). With the declaration "meeting rdv:3;", three tasks must have reached the rendezvous to realize it, either with the "join()" instruction or with the "when()" one. "join()" blocks the task until the rendezvous is realized. "when()" allows to test the status of a rendezvous ; if it is realized the "then" part is executed, the "else" part if not. Note that there is an implicit rendezvous at the end of an "execute and" block, as the two tasks must be terminated for the block to terminate.

The "break()" instruction combined with "block()" allows a task to interrupt another task, or itself. "block()" is used to give a name to a statement or a compound statement, and then when a "break()" of that name is executed all the tasks in the block with that name are resumed and execute to the end of the block.

```
statement :  conc_statement
statement :  sync_statement
statement :  block_statement
statement :  break(id);
conc_statement :  execute statement and statement
sync_statement :  when(id) statement
sync_statement :  when(id) statement else statement
sync_statement :  join(id)
block_statement :  block(id) statement
```

3.2 Compilation

The compilation of a MultitaskC source code with N tasks (at least two) consists in the translation of the N tasks into an intermediate representation of N finite-state machines. Then a synchronous product of these N state machines is done by the MultitaskC compiler and a C code is generated as a result. There are two possible synchronous products.

Compile-time synchronous product. With this option, the MultitaskC compiler evaluates a static synchronous product where all communications and interleaving is done at compilation time. In that case, the result is a C code with maximum performance but the complexity may explode.

Runtime-synchronous product. With this option, the MultitaskC compiler generates the tables of statements, data structures and a generic engine necessary to evaluate the synchronous product at runtime. In this case the complexity of the code generated grows linearly with the number of tasks. The execution is slower than with previous option but the complexity cannot explode. Note that dynamic tasks creation or deletion should also be possible, although it is not yet implemented.

3.3 Comparison with classical features

The use of multi-tasks operating systems has induced common habits for application design and coding. It is possible to keep most of these habits with MultitaskC.

Scheduling. It is very usual to build an application with a fixed-priority scheduler so that each task has a priority. A high-priority task must be able to run as fast as possible and keep the processor until it explicitly releases it. This is the case, as we said before, for instance for VRTX, LynxOS and VxWorks. It is possible to do this fine scheduling with MultitaskC by using rendezvous. This way, a high-priority task can allow other lower-priority tasks to progress only when it is idle. An example will be given in the application for motor control. By adjusting the number of control rendezvous in the low-priority tasks one can obtain a precise scheduling, and this is done at no execution cost with compiled parallelism.

Communications. Varied communication schemes can be implemented with rendezvous and synchronous broadcasts. In [?] we have implemented message queues similar to the VRTX ones, as well as counting-semaphores. With simple preprocessor macros one can implement communications that follow the syntax used for their operating systems equivalents, so that the source codes are not very different.

3.4 MultitaskC programming remarks

The "group" instruction can be used to reduce the size of the synchronous product result, by grouping control statements for example. Error cases can be grouped to form atomic processing because, when an error occurs, it is not necessary that the corresponding actions should be interleaved with other tasks.

One may use the runtime synchronous product during the development phase of an application to get quicker results, then, when the application is ready, switch to compiled synchronous product to get better performances. Note that the two products have the same semantics but the runtime results are different because in one case everything is compiled whereas in the other case the necessity to evaluate communications and interleaving introduces some software latency.

It is possible to build an application with MultitaskC under an operating system, such as UNIX for example. This way, the developer can simulate the target hardware under UNIX and compile for the final target only when the application is ready.

Chapter 4

Evaluation

In this section we refer to an evaluation of a data-acquisition system in [?], an evaluation made in 2006 of a motor control application for a 3-axis cobot (ie collaborative robot), and finally a recent and more detailed evaluation of an application for the motor and force-feedback control of a "steer-by-wire" apparatus.

4.1 Previous evaluations

The first evaluation concerned a data-acquisition application (see [?]), comparing two versions under VRTX and LynxOS with a MultitaskC version. The principal results are that, with MultitaskC interrupt latency is better and it becomes possible to recover the dead-time due to ADC conversion, while this is impossible with any operating system. Recovering the ADC dead-time is possible because of there is no cost associated to context switching in MultitaskC. When an event occurs in some part of the detector, the hardware of the data-acquisition system is triggered (ie ADC conversions are started) and then, the data readout software can operate. As the time of conversion may be important, waiting for its completion results in a significant dead-time. In order to recover this dead-time, one must try to do something useful during the ADC conversion. For example, one might do filtering of the raw data from the previous event before sending them to the final receiver. However, the total conversion time may vary with the event complexity and it is essential to be able to start the readout process with minimal delay. This is possible with MultitaskC because switching from a data-filtering task to the data-readout task is extremely fast. With an operating system on the contrary, context switching generally introduces a significant dead-time.

The second evaluation concerned an application for impedance and 3-axis control of a cobot, comparing a version under VxWorks configured for a PC104-IA32 motherboard with a MultitaskC version. A peripheral board was dedicated to implement a current control-loop with a proportional corrector. The main processor board implemented a position and speed control-loop that had to be as fast as possible in order to get a realistic feeling of the cobot. The VxWorks version was able to run with a 2 milliseconds cycle time whereas the MultitaskC version achieved a 250 microseconds cycle time. Besides giving a better feeling, this gain was interesting for the calculation of the speed from the positions given by the position sensor. It is usual in motor control applications to use only a position sensor, because avoiding the use of a speed sensor allows to reduce cost, weight and congestion. Speed can be deduced from positions and cycle time, but it has to be filtered because typical cobot operation is done at very low speed. Speed filtering with a median filter gives better results but needs computations that take more time. In this case, as the MultitaskC version was 8 times faster than the VxWorks version, its speed elaboration was really much better.

4.2 Evaluation application description

The last evaluation concerned an application for a steer-by-wire electrical motor control with force feedback. The system consists in a wheel with a position sensor and a 24V motor. An H-bridge power board is driven by a PWM (Pulse Width Modulation) generator integrated in a Freescale dual core PowerPC processor. The processor also integrates an ADC for the current sensor of the motor and a quadrature position logic to get the value of the position sensor. As usual, the current control-loop is realized by a dedicated hardware. In this application, the motor control is distributed between two tasks running on one processor core, one task for position/speed control and the other for current control. A second core is dedicated to communications via an Ethernet link with a car simulator that realizes the force feedback through the exchange of position data.

The two control loops must of course run as fast as possible and the network activity must match the rate of information exchange generated by the position/speed loop. Once again, for better speed evaluation the current loop should typically run at 10 microseconds cycle time and the position/speed one at 100 microseconds.

4.3 Language/OS version

The operating system for this application is PharOS. We have implemented the two control tasks as agents, one with a 100 microseconds clock and the other one with a 34 microseconds clock for the current control. Below 34 microseconds for the current agent's clock, there is a deadline overrun and the system stops. For current set point, the two tasks communicate via a simplified temporal variable.

One important feature is that the typical time of conversion of the ADC is about 25 microseconds. We have implemented two versions of the current loop, one that waits for the ADC conversion at each cycle and the other one that reads the data only if they are ready but keeps the previous value if not. With PharOS there is no difference between the two versions because the best cycle time is always longer than the time of ADC conversion. As a consequence no dead-time recovery is possible.

4.4 MultitaskC version

In the MultitaskC version we also have two tasks, one task with a 100 microseconds cycle time and the other one with either a cycle time of 26 microseconds for the version waiting for the ADC to be ready at each cycle, or of 9 microseconds for the version that is not waiting for ADC conversion. Communication between the two tasks is realized via a shared memory.

For this application, we did a fine tuning of the synchronous product. The current loop have a higher priority than the position/speed one. This is implemented through a scheduling rendezvous : the low-priority task frequently asks the high-priority one to grant it progression in its execution flow. The high-priority task grants the scheduling rendezvous only when it is idle, ie waiting for the ADC conversion or waiting to reach the next deadline. This fine scheduling produces a better interleaving of the synchronous product expressions than the default interleaving, leading to better performances of the system.

4.5 Results comparison

The MultitaskC version is better in both cases. Recovery of dead-time becomes possible, and more useful time is available. This time can be used for example, to do a better filtering for speed evaluation. Or to increase safety - while keeping the motor control at optimal rate - by cross-checking redundant sensors for failure recovery. It is also possible to communicate with a redundant processor that becomes active when the local processor fails. These features are very important for steer-by-wire applications, where a force feedback of very good quality and a very high level of safety are mandatory.

Chapter 5

Conclusion

5.1 Conclusions about comparisons

As a conclusion we'll summarize here several points that follow from the results of the comparisons listed above. They concern different types of applications, each application being programmed with different methods. We think that it is possible to generalize those points and consider that some of the results are pertinent whatever the context. Note that some of the points concern all CPLs and not merely MultitaskC.

In [?] one should remember that the time taken to react to an interrupt depends only on the hardware, thus, as compared to an operating system, MultitaskC cannot bring direct benefits in this respect. However, using a CPL makes it possible to keep interrupts always open during execution, when an operating system needs to disable interrupts at certain times to insure system coherency (as for example, during scheduling because this operation must be realized atomically).

We have seen that MultitaskC avoids generating the dead time due to the inner workings of an operating system. In particular, in an operating system the handling of context switching is quite time consuming. Furthermore, the fact that all the communications objects used for data exchange and tasks synchronization are created dynamically results in a system latency that may be prohibitive for some applications. The time it takes to activate a task in response to an interrupt is also shorter with MultitaskC than with an operating system.

With MultitaskC, an other benefit is the fact that the optimizer of the target compiler has a vision of all the tasks of the application, brought together in one source function. It allows to scale up inter-procedure optimizations to the inter-task level. A global optimization that can be done using any target processor. Lastly, one should not forget that it is possible to implement an application with MultitaskC and still use most of the traditional programming concepts and customs. Thus any programmer already familiar with the services of an operating systems and high-level language programming can use MultitaskC easily with only a short period of adaptation.

Concerning dead time recovery, we've shown in [?] that is is possible to apply the principles both with MultitaskC and with an operating system, but in that latter case, the potential gain is much more limited because of the system software latency.

The better overall efficiency of MultitaskC in dealing with interrupts and in handling the parallelism of tasks makes it possible to devote more processing time to hardware specific features, leading to a benefit in the performances of the application, as compared to applications based on operating systems.

Finally, see [?], note that the MultitaskC version of an application is independent of the processor target. It is true that one can keep versions for different operating systems quite similar, but this is much more easily done with a CPL. MultitaskC brings easier portability, and - provided all hardware dependent code is encapsulated - one can even reuse the code generated to target different processors.

5.2 MultitaskC scope

We have shown that with MultitaskC it possible to keep the programming practices that come from the use of traditional operating systems and languages. We have also demonstrated that MultitaskC brings real advantages in terms of execution efficiency and portability. It was also shown that, compared to operating systems, MultitaskC allows an increased use of hardware features which leads to better performances. With MultitaskC, as there is no software latency the performance is limited only by the hardware.

It is obvious however that such a programming language is not suitable for all problems. We are now going to describe the main characteristics of the problems for which MultitaskC can provide an interesting solution.

MultitaskC is particularly suited to develop embedded applications on targets with few resources and small memory configurations. The main benefit is to be able to specify the application as communicating tasks when an operating system is not usable. If time constraints are not important, runtime synchronous product can be used. The only restriction then is the size of the application, a limit that depends only on the hardware.

5.3 Possible language extensions and further work

MultitaskC programs modeled as communicating processes allow verification of properties with automated proof systems. This is the principal purpose of the synchronous languages we have described before (Esterel, ...). As said before, tools that have been developed at INRIA and at LaBRI for the automatic checking of properties on the results of synchronous product, are available for different CPLs. At INRIA, these tools are based on an standardized automata format called OC for "Object Code" (introduced by [**?**]). Esterel and Lustre for example, can produce results in OC format so that verification tools based on OC can be used indifferently for the two languages.

It would be interesting also for MultitaskC to produce OC code as a result of compilation, in order to be able to use the existing tools for automated proof. We think such an upgrading is possible, even though there are some semantic differences between MultitaskC and the other languages of the Esterel family.

Tasks priorities handling is currently possible with MultitaskC. However, it is the programmer's responsibility to slice low-priorities tasks and to organize the relinquishment of processing by high-priorities ones. This could partially be done automatically by the compiler if we add a notion of priority in the language. The possibility to do a fine tunning of the synchronous product must be kept, but the programmer should be given an easier way to express it. Low-priority tasks could be sliced automatically to insure maximum interleaving and then the "group" statement could be used to tune the granularity.

The fact that MultitaskC makes it possible to recover dead time, thus leading to performances being limited only by the hardware, can be used to increase safety. For example, in the motor control application described before, one can increase the safety by recovering the dead time of ADC conversion to communicate with a redundant processor. This is what we would like to investigate in the future. Also, again for safety and other critical aspects of an application, it would be interesting to see how to use the execution model generated by the synchronous product, as well as how to combine those two developments together.