# A compiled parallelism programming language : MultitaskC

Hugo Delchini
Email: hugo@delchini.fr

October 10, 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

# 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 that 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" - that include operating system features (such as tasks definition, task synchronizations and communications between tasks) provides the same advantages. It is admitted, from a theoretical standpoint, see [6], 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 the last chapter 4 we describe an embedded real-time application, chosen because it is representative in terms of constraints. We implement two versions of this application : one using an operating system and a high level programming language and the other using MultitaskC. Then we compare representative results in terms of time and memory space performances.

Finally, 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 [5] 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 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). The rate's stability is very important for the overall system's stability. 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 profound impact on the software. Interfaces between the processor and the hardware thus are 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 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 these 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 has 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 common hardware (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 in case of large team work, and more generally 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.

They were chosen because they are representative of what is commonly used in the industry. PharOS comes with its own programming language, but the language is 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 development environments evaluation we consider PharOS with the PsyC programming language. In [5] we use VRTX and LynxOS with the C programming language. Also 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 [10] and [1].

VRTX ("Versatile Real Time eXecutive") is a real-time fixed priority operating system designed for industrial applications development with constraints in terms of performance and reactivity. It consists of a customizable kernel that can be configured to fit exactly the application's needs. We've did all the evaluation tests on 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 : 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. The version used in [5] of the operating system shares the physical address space for data and instructions with user tasks. There is neither memory segmentation or virtualization nor 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 pages' 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 usage for embedded applications. Note that this has an impact on performances because ROM access time is usually really slower than RAM one.

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 startup code relative to the mother-board support (board support package) and drivers. The board specific software has to be written by users and this allows to configure the kernel for a particular use. Several initialization informations must be defined, such as maximum number of tasks, stack-sizes, etc. Writing this piece of software is not easy and when it is 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 tasks commutation and one for task deletion. In our implementation we also used hooks to manage the AMD29050 floating point co-processor, in order to backup and restore the FPU registers on context switching. Other hooks can be used to handle the MMU/MPU contexts during the same task operations, if memory protection and segmentation is needed. For performances issues, in our evaluation we dismissed MMU/MPU management but used the FPU for motor control.

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

In the AMD29000 version, several processor registers are reserved for the operating system. This feature isn't detrimental because the hardware implements a big amount of registers (192) but improves execution efficiency because it reduces the number of registers to save/restore during context switching. As regards the number of registers dedicated to each task, note that although a big number of registers helps the C compiler do optimizations it implies a significant cost during tasks commutations. Since there are much less registers in the MC68000 processor architecture, no registers are reserved for system use in the MC68000 version of VRTX. We are now going to review some features of the operating system, been understood that a lot of the preceding characteristics can be found in other real-time operating systems.

**Tasks.** As we said before, VRTX is a multi-task operating system. A VRTX task corresponds to the execution of binary instructions by the processor. The operating system handles time sharing between all ready tasks in a given priority. A task can create other tasks but there are no ancestor or sibling relations between them. A task can get no information upon it's ancestor. Tasks are created and deleted without any incidence except context switching with respect to priorities. There is no implicit meeting between a task and the task which created it at the end of execution, for example.

Each task is allotted a priority when created. Priorities enables 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. An intensive time-consuming task's priority will be lowered to avoid processor starvation for the other tasks. Later its priority will be raised again to its initial value when the system considers it should anew be allocated processor time. 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 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 for 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 executing an other task.

We have mentioned previously that in our VRTX configuration there is only one address space shared by user tasks and system. An other 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 or 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 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 it needs to do so. 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. Finally, 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 that 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 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 is 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 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 it is empty (but it is possible to check the case before pending). A user can ask for the current FIFO size and 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 is made of 32 flags, each 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 "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 wakened on 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-wakening options are the same as for FIFOs.

Lastly, VRTX provides a mechanism called "shared memory" to share 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 point 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 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 allows interrupts as much as possible, so delays due to handlers are minimized.

In VRTX, a special stack to backup registers during 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 to trigger the task-level process. However, such a mechanism must comply to a system protocol : ie, when any 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 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 Lynx 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 [8], [11] and [12].

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 application programming interfaces for the operating systems that implement it. These interfaces specify data types, types names, functions names and semantic. As POSIX-compliant applications 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 aimed to be POSIX and ANSI-C conforming, which is a interesting approach. The ANSI-C part is based on GNU compilers and libraries. As regards the operating system interface, LynxOS includes all 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, with execution levels similar to VRTX but implementing in addition a notion of user similar to UNIX. 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 plus 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 with all possible extensions. It can be stored in ROM with a minimal ROM file system, and it can also be booted via a network. These two possibilities allow easy deployment of applications on 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 its 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 also be virtual. Each process has its own address space which is private. Like any user process, the kernel has its own address space but kernel pages are always resident in core memory. When virtual memory management is not enabled, all process pages are also resident in core memory. In order to have virtual memory enabled, a swap area must be defined on hard disk to store the pages that temporally cannot stay in core memory. As users processes and kernel cannot access each other address spaces, such a memory segmentation is a feature of safety.

**Processes and activities.** The basic execution entity of LynxOS and UNIX are the same : the process. However, when the process of UNIX has a time-dependent priority, the process of LynxOS has a fixed priority. Its scheduling of 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 "System V.4" version of UNIX).

When virtual memory is used, LynxOS swapping mechanism is handled by a system process which has itself a priority as any other process. This means that the relative priorities of processes have an influence on memory management. A process with 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. It is given a fixed priority when created. A LynxOS process always has an initial thread, a specific feature not specified in POSIX recommendations. The ensemble of 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 [12].

We are now going to list the main operations available for threads 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 in the first case and infinite in the second.

**Communications.** LynxOS implements two sets of communication tools, one for inter-process communications and the other for threads. Inter-process 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 "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 an other 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 it owns a mutex, a thread 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, as well as all the waiting threads at the same time. A thread can also use inter-process 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 which accept that signal. If none of them accepts the signal, it stays pending for a thread to accept it. A signal can also be sent to a particular thread of a process.

**Interrupts.** LynxOS does not allow an application to catch a hardware interrupt by itself. To LynxOS designers this direct mode would have lead to the implementation by users of services that should be handled by the operating system. We have seen in VRTX that this direct mode is better as regards interrupt latency but implies 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 handle more time-consuming tasks. 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 an 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 a driver handler if installed. This allows easy system coherency but introduces interrupt latency.

### 2.3.3 The ADA programming language

All the information listed here comes from [3] and [9]. 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 which allows 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 not included in the ADA compiler but running on the host. 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.

The user can use the "PRIORITY" pragma to set priorities to ADA tasks. These priorities have different significations depending on the host operating system. Neither is the scheduling algorithm is specified in the ADA standard nor the way to schedule two tasks of same priority. An ADA program is made of tasks similar to UNIX processes, LynxOS threads or VRTX tasks. This is why we consider 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 parameter passing for a function call.

With the "select" operator it is possible to wait for more than one rendezvous, according to the state of local variables of a task. 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 will be used. 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, rendezvous with data exchange 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, with a priority always greater than any real tasks. This way, the rendezvous associated to an interrupt is at highest priority. The scheduling algorithm of the underlying operating system is not involved in interrupts rendezvous. Interrupt-handling implementation for the host operating system is not specified in the language standard. It can be done either by a direct handler without an interrupt driver or by any other implementation.

### 2.3.4 VxWorks

VxWorks is a multi-task operating system very similar to VRTX, obviously because VRTX was the kernel included in VxWorks in initial versions (VxWorks stands for "VRTX Works"). So tasks management is almost the same and communication tools also. VxWorks comes with a very user friendly configuration tool for board support package and kernel features. In version 5, it is called "Tornado" and in version 6 is is based on "Eclipse" and called "Workbench". This configuration tool is also an Integrated Development Environment for the programmer with cross compilation. The user can use C programming language but also C++ and other languages that the compiler can handle (gcc based).

Note that from version 6, VxWorks kernel introduced Real Time Processes that is an implementation of memory segmentation similar to POSIX ones. This can be useful for applications that need safety features. Interrupts are handled the same way than LynxOS with an interrupt driver. The operating system is available on main hardware such as IA32 and PowerPC. Note that in the development phase of an application, the user can include a shell with a minimal C interpreter that allow to load binaries and call functions with parameters from the shell. Once the application is ready, the user can remove the shell from VxWorks configuration. VxWorks also offers a very good instrumentation tool with possible kernel and application instrumentation, it is called WindView and 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 [4]. The PharOS operating system is being developed for safety critical applications. It is mainly composed of the PsyC programming language and a real-time kernel. The PsyC language is very similar to C and includes the possibility to describe tasks and also provides communication tool as "Temporal variables" and "Message queues". The particularity of this operating system is that it is time triggered. The user can define clocks in PsyC sources and then all processes must occur in temporal windows defined in tasks. All treatment must end before a deadline and scheduling is made according to Earliest Deadline First algorithm.

Interrupts can be handled with PharOS but are time tagged when they occurs and associated with a process in time triggered space. Thus event triggered processing is converted to time triggered ones. There is an automatic memory segmentation using MMU or MPU hardware to do spatial partitioning. This is similar to the partitioning done in VxWorks or LynxOS. Spatial or temporal partitioning violations can be handled by the operating system with task groups. If groups are defined in an application, when a task of that group has a failure, the entire group is restarted at 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 summary the characteristics that seem useful in the development and execution tools we have described. Except for the standalone programming language, they all offer the possibility to specify an application structured as several tasks that can communicate and synchronize. This is a very common design scheme for real-time applications so it is almost necessary to have it for a good development tool.

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

With or without cross compilation, it is possible to program in a high level programming language and debugging tools. Assembly language can be can in special cases to access particular registers or instructions and its usage can be very minimized to ease portability. Note that operating system portability is not simple as it is hardware dependent for various features even if it is coded in a standardized high level language.

An answer to this portability problem that brings several other advantages can be a programming language family we are going to describe.

### 2.4 Synchronous programming languages

Synchronous programming languages with compiled parallelism appeared in early 80 for Esterel, implementing the synchronous product of deterministic finite state machines defined by [2]. We will introduce several of these languages starting with Esterel which is one of the first. Then after the description, we will compare the OS/language approach to CPLs. Following this comparison, we will introduce the language we have defined which intend to give to the programmer the same features than an operating system coupled to a high level language but with compiled parallelism for efficiency. We will use our language to represent CPLs in our evaluations as it implements the same paradigm. Informations here comes from [6] regarding Esterel, Lustre and Argos.

#### 2.4.1 Esterel

Esterel was developed by a common team of the "Centre de Mathematiques Appliquees" of "Ecole des mines" and of INRIA ("Institut National de Recherche en Informatique et Automatique"). It's a programming language that mainly allow the user to specify behaviors and validate them, this is why it does not offer complex data structures for example that you can find in common languages. It is possible with Esterel to specify an application as a set of communicating tasks.

An Esterel program describes a reactive system, this means a system which react to external stimuli and as a reaction emit other stimuli. The result of an Esterel compilation is a finite state machine coded in a high level language call host language (C,ADA,...). Automata transitions are implemented in a host language engine. This engine execution after compilation for the hardware target realize the behavior described in the original Esterel code. This execution consists in chaining transitions by the automata engine. The use of a host language allow easier portability.

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

It is possible in an Esterel program to define local variables, to call host language functions, to define instructions sequences, to have control statements such as loops and tests and common computing expressions with usual operators. The user can define modules that can be included in further programs so applications can be structured. There are no global variables in Esterel.

**Tasks and communications.** The "||" operator allows the user to specify independent tasks. Communications between tasks is made by signals as for external communications. It is the only communication tool in Esterel and for example there is no shared memory. A signal can be externally or internally raised for the reactive system. It can be used for internal communications only, a tasks can raise a signal ("emit() operator") and then all pending tasks (with "do halt watching signal") can be awaken. Waiting on a signal can be blocking but a task can just test the state of a signal (with "present"). It is possible to associate a value to a signal and then emission can modify it. A task can get the signal value when taking it in account. This is also true for external signals so values can be transmitted with the outside world. The interface between the reactive system and the outside world is simple. Signal emission from the system to the outside world is translated into a function call in the host language so the developer must 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 this case there are an input and an output function.

The 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 call 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 reactive systems.

**Parallelism.** Tasks handling is completely different in Esterel than for the operating systems we've described before. The parallelism is compiled, so to say, if several tasks are described in an Esterel program, after compilation there is only one task which realize the execution of all the original tasks. So it is not possible to create or destroy tasks dynamically in Esterel. A multi-task operating system simulates parallel execution of tasks with dynamic runtime scheduling but in languages such as Esterel, scheduling is statically evaluated at compile time. Each Esterel task is considered as a finite state machine and parallelism is compiled by a synchronous product of all the input state machines giving in turn a state machine which when executed simulates the concurrent execution of operand state machines. We will also use this principle for our programming language. Finite state machines are well known mathematical objects that can be used to formalize behaviors. So the synchronous product gives two things : a formal model of program execution that can be used for properties proof and parallelism compilation.

**Interrupts.** As for operating systems it is important to be able to handle interrupts with Esterel. Obviously, a signal can be associated to an interrupt by calling the signal input function in the interrupt handler. The signal occurrence is then taken in account at execution level. This is quite similar to operating systems way to handle interrupts but there is one significant difference because here interrupts can be allowed at any time (except during interrupt handling itself) so latency can be minimized. This is a important point for efficiency in real-time applications.

### 2.4.2 Other synchronous languages

A lot of other synchronous languages have been implemented and we are going to describe two of them rapidly.

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

## 2.5 Comparison of CPLs and "language/OS"

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

### 2.5.1 CPLs drawbacks

A limitation of CPLs is given by the synchronous product which is used to compile parallelism. The size of the result in terms of states and transitions number is potentially exponential of the operands sizes. The external stimuli can also increase this complexity. This limitation is theoretical and in practice it is rarely reached. However this limitation does not exists with an operating system. We will introduce later the possibility to avoid this exponential complexity with a particular way to implement the synchronous product in our language. In the same way, the space complexity of a synchronous product is potentially exponential. This is also irrelevant for an operating system even if it also consumes memory space for its own data that is not consumed with a CPL.

Another point is that an operating system allows dynamic creating/deletion of tasks. This is not possible for a CPL based on compile time synchronous product where tasks and communication tools must described statically. We will also introduce later the runtime synchronous product we have implemented in our language and that can be a solution to handle dynamic objects in a CPL.

### 2.5.2 CPLs advantages

CPLs allows the user to write programs completely independent from an operating system. An application written with a CPL has a better portability and will only depend eventually on a standard C library that is always available with today compilers and also needed with an operating system. The use of a host programming language is also better for portability. Even when you have to upgrade existing hardware, for example to support an arithmetic co-processor, it's easier with a CPL. With an operating system, the context of each task may be impacted to save/restore FPU context. This has an impact on performance and need an operating system modification. With a CPL the same integration is just handled by the host language compiler, in a case the compiler gives FPU emulation and in the other generates real FPU instructions.

A CPL allow the user to specify an application using standard features also given by operating systems such as tasks and communication tools. So the programmer doesn't have to integrates new paradigms. We will see later that in the same time, CPLs brings more performance than operating systems with the limit given by hardware only. Another difference is about interrupts which is very important for real-time applications. In a real-time operating system, there are a lot of sequences executed with interrupts disabled. This is absolutely necessary to provide system coherency during scheduling for example where there are some linked lists to be managed. Some system calls must also be atomically executed from task level. Even if the operating system is preemptive in kernel mode, it is not possible to be always the case. With a CPL, it is possible to have interrupts enabled all the time except of course during the execution of handlers prologues and epilogues. There is no need to have critical section toward interrupts because there is only two execution levels : application and interrupts, and only one task context.

With an operating system, the communication between interrupts and tasks execution levels has a bigger cost than with CPLs. With an operating system, signaling a counting semaphore will trigger scheduling and a

lot of actions to have the task level activated for processing. With a CPL, the interrupt routine can be reduced to a simple counter incrementation without critical section just to indicate task level that some processing must start. This also involves that with a CPL, interrupt routines can be as short as possible so there is more time for task level.

In multi-task operating systems, each task must have a data structure usually called context made of all processor registers plus a stack and optionally some MMU/MPU descriptors. When a task is chosen by the scheduler, the previous task context must be saved and the new task one restored, this called context commutation. This operation is not necessary with compiled parallelism because there is only one context, so no dynamic scheduling and context commutations. This is also the same for communications and synchronizations, every thing is statically handled at compile time and there is no cost at runtime. With a CPL, as there is no context commutation cost, it is possible to tune with a very fine grain processor execution time. So it is possible for a task to release CPU for very short periods that can be used by other tasks. This is not the case with an operating system.

The result of the synchronous product in the host language gives the final target compiler a vision of all application tasks. This can allow the target compiler optimizer to do global time and space optimizations that is not possible with an operating system because tasks code and data are completely separated.

# Chapter 3

# Compiled parallelism and MultitaskC

## 3.1 The MultitaskC programming language

MultitaskC syntax is a C language extension, this is the most popular programming language so the choice is obvious to us. We consider that the reader is familiar with this programming language as described in [7].

A first extension of MultitaskC is the possibility to describe independent tasks. We will call task a C statement or compound statement in parallel with at least another C statement or compound statement. We have implemented the same paradigm as in other CPLs for parallelism compilation and we are targeting mono-processor architecture. The result of an MultitaskC compilation is a finite state machine coded in pure C. Execution of this automata is simulating the parallel execution of all the tasks. Common communication and synchronization tools are implemented : rendezvous and an implementation of the Esterel synchronous broadcast. There is also the possibility to use share memory via global variables.

The user has the possibility to declare meetings or rendezvous at the beginning of a function and the scope of the rendezvous is the entire function. The declaration is following this grammar :

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

Here are few valid rendezvous declarations, the constant indicates the number of tasks 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()", "if()" control statements. Expressions for tests and statements are the same, also all operators and declarations. All expressions are executed atomically and there is an operator to let a sequence be 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 standard C grammar where "id" stands for a block identifier or a rendezvous :

```
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
```

The "join()" and "when()" instructions allows to communicate with rendezvous that must be declared before use. With this declaration "meeting rdv:3;", tree tasks must have reached the rendezvous to realize it whether

with the "join()" instruction or with the "when()" one. "join()" blocks the task until the rendezvous is realized and "when()" allows to test a rendezvous status and 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, the two tasks must be terminated for the block to terminate.

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

## 3.2 Compilation

An MultitaskC compilation of a source code with at least two tasks consist in the translation of the two tasks in two finite state machines with an intermediate representation. Then a synchronous product of these two state machines is done by the MultitaskC compiler and C code is generated as a result. If more than two tasks are present in source code, this is the same. 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 a compilation time. So the result is C code with a maximum performance but complexity may explode.

**Runtime synchronous product.** With this second option, the MultitaskC compiler generates statement tables, data structures and a generic engine to evaluate the synchronous product at runtime. In this case generated code complexity is linear according to the number of tasks. So execution is slower than with previous option but complexity cannot explode. Note that also dynamic tasks creation or deletion should be possible even if not currently implemented.

## 3.3 Comparison with classical features

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

**Scheduling.** It is very common to build an application on top of a fixed priority scheduler so each task has a priority. A high priority task must be run as fast as possible and keep processor until it explicitly release it. This is a common implementation as we said before, for instance in VRTX, LynxOS and VxWorks. This is possible to do fine scheduling with MultitaskC by using rendezvous, a high priority task can allow other less priority ones to progress only when it is idle. We will seen an example in the evaluation application for motor control. The number of control rendezvous in low priority tasks allows fine scheduling and with compiled parallelism this is at no cost.

**Communications.** Communication schemes can be implemented with rendezvous and eventually synchronous broadcast. In [5] we have implemented message queues similar to VRTX ones and also counting semaphores. It is possible with simple preprocessor macros to implement communications with a syntax close to the operating systems versions so source code is not very different.

## 3.4 MultitaskC programming remarks

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

The runtime synchronous product can be used during application development to get faster results if it's possible. Then, when the application is ready, compiled synchronous product can be used for better performances. Note that the two products have the same semantics but runtime result is different because in one case everything is compiled and in the other one, some software latency is introduced to evaluate communications and interleaving.

It's possible to build an application with MultitaskC under UNIX for example. So the developer can simulate target hardware under UNIX and when the application is ready, compile for the final target.

# Chapter 4

# Evaluation

In this section we are going to point out evaluations made in [5] for data acquisition system and another evaluation made during 2006 on an 3 axis cobot with motor control and a more recent and more detailed one on a steer-by-wire application with 1 axis motor control and force feedback.

## 4.1 Previous evaluations

The first evaluation was made in [5] on data acquisition systems with VRTX and LynxOS compared to MultitaskC. The main results are that interrupt latency is better with MultitaskC and also that it is possible to recover ADC conversion dead-time with MultitaskC but not with any operating system. The dead-time recover is possible because of zero cost context switching in MultitaskC. When an event occur in a part of a detector the hardware data acquisition system is triggered and mainly ADC conversions starts and then data readout software can operate. The conversion time can be significant and the naive way to wait for it is resulting in dead-time. The idea is to recover dead-time by doing something useful during ADC conversion. Of example in a particle physic data acquisition system, the user can do data filtering on the previous event during ADC conversion of current event. But as the conversion time may vary depending on the event complexity, it is essential to be able to start readout very quickly, this is possible with MultitaskC because switching from a data filtering task to a data readout one is very fast. With an operating system, this is possible but context switching may introduce more dead-time.

A second evaluation was made with a 3 axis cobot and impedance control, one version with VxWorks on a PC104 IA32 motherboard. An peripheral dedicated hardware board was implementing a current control loop with a proportional corrector. The main processor was implementing a position and speed control loop that should be as fast as possible to get realistic feeling on the cobot. The VxWorks version was able to run with a 2 milliseconds cycle time and the MultitaskC version was running at 250 microseconds cycle time. More than giving better feeling, the speed gain was interesting for speed elaboration based on position sensor. It is a common thing in motor control to use a position sensor only without a speed sensor to reduce cost ,weight and congestion. The speed sensor can be interpolated from positions and cycle time. But this has to be filtered because of typical very low speed during cobot operation. Speed filtering with a median filter is giving really better results but need some computations that take some time. And the faster speed elaboration is the better as usual in signal processing. So in this case the MultitaskC version was 8 times faster than the VxWorks one and thus speed elaboration was really better.

## 4.2 Evaluation application description

The last evaluation was made for an electrical motor control application with force feedback steer-by-wire. The system is made of 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 a ADC for motor current sensor and a quadrature position logic to get position sensor value. Usually, as in previous application, current control loop is made by dedicated hardware. In our application, we will use one core for motor control with two tasks, one for position/speed control and the other one for current control. The other core will be dedicated to communications via an Ethernet link with a car simulator that will be coupled by position exchange for force feedback.

The two control loop must as usual run as fast as possible and the network activity must run at the same rate than the position/speed loop because of position information exchange. Typically, the current loop should run at 10 microseconds cycle time and position/speed one at 100 microseconds again for better speed evaluation.

## 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 the fastest clock. The fastest clock for current agent is 34 microseconds otherwise there is a deadline overrun and the system stops. Communication between the two tasks for current set point is made via a simplified temporal variable.

One important thing is that the ADC conversion time for current loop is typical about 25 microseconds. We have implemented two version of the current loop, one that is waiting for ADC conversion at each cycle and another one is reading conversion only is it is ready and keep the previous value if it is not. With PharOS there is no difference between the two version because best cycle time is always longer than ADC conversion. This involves that no dead-time recovery is possible here.

## 4.4 MultitaskC version

The MultitaskC version is also made of two tasks, one with a 100 microseconds cycle time and the other one can run up to 26 microseconds for the version that waits for the ADC to be ready at each cycle. And up to 9 microseconds for the version that is not waiting for ADC conversion. Communication between the two tasks is made by a shared memory.

We have used fine synchronous product tunning in this application. The current loop have a higher priority than the position/speed one. This is implemented by using a scheduling rendezvous, the low priority task is asking the highest one to grant progression frequently in its execution flow. The highest priority task is granting the scheduling rendezvous only when it is idle : waiting for ADC conversion or waiting to reach next deadline. This fine scheduling is tuning the synchronous product expression interleaving so performance are better than with the default interleaving.

## 4.5 Results comparison

The MultitaskC version in both case is better. It is possible to recover dead-time and for example do better filtering for speed evaluation or safety. For example, it is possible with the MultitaskC version to check redundant sensors for failure recovery and this while motor control is made at ideal rate. It is also possible to communicate with a redundant processor that will become active in case of failure of the local one. This is very important for a steer-by-wire application, very good quality force feedback is mandatory and safety also.

# Chapter 5

# Conclusion

## 5.1 Conclusions about comparisons

The conclusion points listed here are given by several comparison results. They are based on multiple applications programmed with different methods. We think that it is possible to generalize several conclusions and consider that some results are pertinent whatever the context. Note that some points in this conclusion are concerning all CPLs rather than just MultitaskC.

As in [5] we can remind that the time taken to react to an interrupt is just hardware dependent and for that there is no benefit with MultitaskC than with operating systems. We just what to point out that with a CPL, it is possible that interrupts are always open during execution. This is different with an operating system in which interrupts must be disabled for some times to insure system coherency. As an example, during scheduling interrupts are disabled to permit this operation atomically.

As we sensed at the beginning of this work, MultitaskC allows to recover dead time consumed by an operating system. Particularly, in an operating system, the context switching handling is really time consuming and all communications objects used for data exchange and tasks synchronization are done dynamically involving a system latency that can be prohibitive for some applications. Tasks activations in response to an interrupt is also better with MultitaskC than with an operating system.

The fact that, with MultitaskC, the optimizer of the target compiler has a vision of all tasks in one source function is also a benefit for applications. This can be compared to inter procedural optimizations, but there we can talk of inter tasks optimizations. This global optimization can be done with any target processor. All these benefits should be considered with keeping in mind that it is possible to implement an application with MultitaskC using most traditional programming habits and concepts. So the programmer which is usually familiar with operating systems services and high level programming languages can use MultitaskC easily with a short period of adaptation.

These response time improvements and the static compilation of parallelism handling brought by MultitaskC allows to get a better benefit of hardware specific features than with operating systems. We've shows that in [5] concerning dead time recovery. With operating systems it is possible to implement this kind of principles but the limitation comes really early compared to MultitaskC due to system software latency.

Note that in [5], the MultitaskC version of the application are similar whatever the processor target is. The operating system versions also also quite similar but less than the CPL ones. MultitaskC brings easier portability and even the generated code may be reusable for targeting other processors if any hardware dependent code is encapsulated.

## 5.2 MultitaskC scope

We have shown that with MultitaskC it possible to keep programming habits that comes from the use of traditional operating systems and languages. We have also demonstrate that MultitaskC brings actual advantages in terms of execution efficiency and portability. It was also shown that MultitaskC allows the use of hardware features with more performances than Operating systems or in other words that the performance limit with MultitaskC is given by hardware without software latency.

However, it is obvious that such a programming language is not suitable to solve all problems. We are going to raise the main characteristics of problems for which MultitaskC can offer an interesting solution.

Our language should be particularly suitable 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 restrictions are then the size of the application that are limited by hardware only.

### 5.3 Possible language extensions and further work

MultitaskC programs modeled as communicating processes can allow to verify properties with automated proof systems. This is the main purpose of synchronous languages we have described before (Esterel, ...). As we said before, for other CPLs, there are tools that have been developed at INRIA and also at LaBRI for automatic checking of properties on synchronous product results. At INRIA, these tools are based on an standardized automata format called OC for "Object Code" (introduced by [6]). Esterel and Lustre can for example produce results in OC format and then verification tools based on OC can be used and considered as common for the two languages.

It should be interesting to produce OC code as a result of an MultitaskC compilation so it will be possible to use existing tools for automated proof. We think that it is possible to do so even if there are semantic differences between MultitaskC and other languages of the Esterel family.

Tasks priorities handling is possible currently with MultitaskC. However, the programmer is responsible for slicing low priorities tasks and processing relinquish in high priorities ones. This can be partially done automatically by the compiler if we add a priority notion in the language. The possibility to do fine tunning of the synchronous product must be kept but the programmer should have an easier way to express it. A low priority task could be sliced automatically with maximum interleaving and then the "group" statement can be used to tune granularity.

The fact that with MultitaskC it is possible to recover dead time and have performances that are only limited by hardware, could be advantageously used for safety. For example, in the motor control application we described before, one can use ADC conversion dead time to communicate with a redundant processor for safety purpose. This is a way we would like to investigate in future works. It can also be combined with the possibility to use the execution model generated by synchronous product for safety and critical aspects of an application.

# Bibliography


[1] Advanced Micro Devices. The amd29050 microprocessor reference manual. 1991.

[2] A. Arnold and M. Nivat. Comportements de processus. colloque AFCET, Paris, 1982.

[3] G. Booch. Software engineering with ada. 1986.

[4] V. David, J. Delcoigne, E. Leret, A. Ourghanlian, P. Hilsenkopf, and P. Paris. Safety properties ensured by the OASIS model for safety critical real time systems. In *Proc. of the 17th Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP'98)*, volume 1516 of *Lecture Notes in Computer Science*, pages 45–59, Heidelberg, Germany, 1998. Springer.

[5] H. Delchini. Conception, développement et évaluation d'un langage de programmation adapté aux applications industrielles : llC. 1995.

[6] N. Halbwachs. Conception de système réactifs, les langages synchrones. IMAG/LGI Grenoble, 1991.

[7] B. Kernighan and D. Ritchie. The ansi c programming language. Prentice Hall Software Series, 1988.

[8] Lynx RTS. LynxOS reference manual. 1993.

[9] Norme ANSI. Reference manual for ada. 1983.

[10] J. Ready. The VRTX-32/29000 user's guide. Mentor Graphics, 1990.

[11] J. M. Rifflet. La programmation sous UNIX. Ediscience International, 1993.

[12] J. M. Rifflet. La communication sous UNIX, chap. 3 : POSIX et les threads. Ediscience International, 1994.