A compiled parallelism programming language : MultitaskC

Hugo Delchini
Email: hugo@delchini.fr

October 2, 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 more advantages in term of performances 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.

Chapter 1

Introduction

Industrial computing applications development is usually done by using a high level programming language and an operating system for execution control. An operating system offers services that the developer uses to implement an application on a computer.

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

A high level programming language family (that we will call CPL for "Compiled Parallelism Languages") that include operating system features (as tasks definition, task synchronizations and communications between tasks) gives, as we will see later, the same advantages than said before. It is admitted, from a theoretical point of view, as in [6], that these languages brings additional advantages in terms of portability, formal modeling and execution efficiency. This work is particularly focused on execution efficiency.

We intend to show that the CPLs advantages versus the traditional "operating system/programming language" are significant and that they can be better to solve various problems.

Our evaluation will be done by first do a description of various traditional tools and operating systems used to build applications, specially in real-time and embedded space, this is in chapter 2. We will point out the main characteristics of common applications and classical tools allowing their design and implementation with respect of their constraints.

Then, in chapter 3 we will introduce CPLs by a description of several existing languages and show that they have advantages for the implementation of embedded real-time applications. We will also introduce our CPL which is high performance oriented.

In the last chapter 4 of this work, we will describe an embedded real-time application we choose because it is representative in terms of constraints. We will implement two versions of the application : one with an operating system and a high level programming language and the other one with the CPL we have defined. We will then compare representative results mainly about time and space performances.

Finally, chapter 5 concludes and suggests 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 them that are representative of the ones used in industry for mono-processor or few processors machines. Applications are often developed using a cross compiler and binaries are loaded on the target system via an appropriate media such an USB link and a JTAG. Some real-time operating systems such as LynxOS allows the development environment to be used directly on the target system.

We are going now to describe few development environment for applications. These solutions are applicable whether you use cross compilation or not.

2.1 Main characteristics of embedded real-time applications

Data acquisition applications described in [5] and motor control applications we will describe later for evaluations always have a set of characteristics. The main ones we will consider are :

- *The reaction to one or more input signals (event triggered).* This is the case of a lot of industrial applications, they react to stimuli that triggers processing as data input or triggering other stimuli. When the application runs and is always ready to react to each stimulus without the introduction of a delay for following stimuli, it can be said to be a real-time one. In this case one can consider that the response of the application is synchronous with stimuli occurrences.

- *Time triggered processing.* Very often in real time application, there are processes that must be executed at a precise rate. For example, in an electric motor control task, you usually have to implement a current control loop at a very high rate (typically 10 us cycle time). The stability of the rate is very important for the stability of the system and also it is better to be able to control the jitter between two time triggered processes.

- *Non triggered processing.* Most applications need the possibility to specify that a particular process need to be done as fast as possible or when there is time to do it.

- *Several more or less independent processing.* Applications are often made of more that one independent tasks communicating via standard mechanisms for mutual exclusion and synchronization for example. In a data acquisition system one task can do data readout and another data transfers for centralization. This can allow to operate for example independent hardware simultaneously and so reduce dead time (an asynchronous interface between two buses). The possibility for the programmer to specify tasks is also convenient for application design. Tasks can be non triggered, time triggered or event triggered.

- *The use of specific hardware.* In real time systems for motor control as an example, one is going to use a particular electric motor with characteristics that have an impact on the control software. Interfaces between the microprocessor and an H-bridge for current control is usually a Pulse Width Modulation module that can be very different from one hardware platform to another. The hardware can be also specifically developed for the application. The programming environment should make easier the use of particular hardware.

2.2 A standalone programming language

This first approach is the oldest one. One can consider that a lot of embedded applications have been programmed with it. It consist in using a mix of a high level programming language and eventually assembler parts that is executed without the control of an operating system. All must be written by the programmer that is not allowed to use complex objects for application design and structure (tasks, means of communication and synchronization, ...). Execution environment is reduced to hardware and eventually during the development phase, a debugger.

Actually, the main point here is the choice of the programming language. Choosing a standard high level programming language with as much as possible libraries that can eventually give some basic operating system features can be advantageous. The choice can be induced by the availability of a compiler for the hardware target. It can be reasonable to use a standardized programming language that is widely used.

Several languages conforms to these criteria. The one we will retain is the C programming language described in [7] or eventually the object oriented variant : C++. Note that concerning C++, the programmer should take care of objects resources such as memory, especially for constructors and destructors that can introduce an overhead during execution. Actually, all objects should be declared with static memory and no dynamic allocation should be used at least during execution, after initializations. Control of these costs can be hazardous when it is necessary in real-time applications to handle them precisely in order to be sure to suit all the constraints of the application.

Originally, the C programming language has been designed for operating system development. It is suitable for low level programming and this confirms that this can be a good choice because as said before, we are interested by industrial embedded real-time applications. Another good argument for the choice of this programming language is the availability of compilers for most processors. For example, the well known "gcc" compiler from GNU project can handle most common hardware and have options to be conform to ANSI standard. This compiler suite can also offer C++ support and even other common languages such as FORTRAN. There is a standardized interface for debugging with "gdb" and binary tools for specific linking. Modern compilers are implementing very high level optimization techniques such as inter-procedural analysis for example.

Real-time embedded applications have been programmed very often without the use of an operating system for execution control. This programming method is used for a long time and applications written with it are mainly made of interrupts routines for event triggered processing. One interrupt can occur at a specific rate generated by a timer for time triggered processing. Other interrupts can indicate the availability of data or readiness for starting a data transfer.

Even if one uses a programming language such as C, some parts of the application must be developed in assembly language. Typically because some instructions or registers are specific to the processor and cannot exist at C level. Except by defining coding and precise application structure rules, the programmer is allowed to do whatever he wants and coding style could be a problem for maintenance and distributed programming in a programmers team. With this simple programming tool, it is possible to structure the application as a set of finite state automata for example or even Petri nets for multi-core architectures.

This approach is quite basic but also used widely but the only thing we will retain is the use of the C programming language. We are now going to overview the use of this programming language along with an operating system.

2.3 A programming language with an operating system

Such a development environment is commonly used for several years in real-time space. More and more micro-controllers are able to run operating systems as there is more and more resources needed to do so. Lets see a sample of operating systems and other similar tools.

For each operating system we consider using the C programming language. We've chosen these mainly because they are representatives of the ones commonly used in industry. PharOS comes with its own programming language as we will see but it very close to C. We just want to cite OS9 that is an early real-time operating system but we'd like to concentrate on more recent ones. Note that VRTX is the VxWorks ancestor.

We will also do an overview of the ADA programming language. We will explain below why we consider this language as a language coupled to an operating system. In the section about development environments evaluation we will use PharOS with the PsyC programming language. In [5] we have used VRTX and LynxOS with the C programming language. Also we have used C and C++ along with VxWorks in another evaluation we are going to describe shortly.

2.3.1 VRTX

All informations we present here is taken from [10] and [1].

VRTX ("Versatile Real Time eXecutive") is a real-time fixed priority operating system designed to allow industrial applications development with respect of constraints in terms of performance and reactivity. It is made of a customizable kernel one can configure to fit exactly the application needs. We've made all the evaluation test on the version for the AMD29000 micro-processor but there are other version for common targets such MC68000 and others.

Developments in C programming language is using a cross compiler on a PC. Applications are downloaded on the target via an industrial bus called VME. It is possible to get a on target environment for VRTX but this is not the case for our system. 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 is sharing the physical address space with user tasks for data and instructions. There is no particular memory segmentation or virtualization and no memory protection via an MMU.

It is possible to configure VRTX with paginated memory but it is not available by default. For performances issues we prefer to have simple memory management because the pages tables handling during contexts commutations can introduce system latency. This feature could however be interesting for safety as we will see with PharOS and LynxOS. The system can be flashed in ROM which is a common thing for embedded applications. Note that this have an impact on performances because ROM access time is usually really slower than RAM one.

System calls are not conforms to any standard but the C compiler we use conforms to ANSI C. The kernel is available as 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 drivers an startup code relative to the mother-board support. The board specific software has to be written by users and this allows to configure the kernel for a particular use. One must give several initialization information such as the maximum number of tasks, the size of stacks, etc. The development of 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 hooks to handle specific processing for tasks management. There is one hook for tasks creation, one for tasks commutation and one for tasks deletion. In our implementation we have hooks for the AMD29050 floating point co-processor, to backup and restore FPU registers on context switching. For memory protection and segmentation, one can use these hooks to handle MMU/MPU contexts for the same tasks operations. In our evaluation, for performances issues, MMU/MPU contexts is not used but FPU contexts are used for motor control.

The operating system handles dynamic memory allocation and basic input/output functions on a serial line but drivers must be given for this 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 operating system use, this is possible because of the big amount of registers (192). This is done to improve execution efficiency by reducing context switching number of registers to save/restore. Obviously, the number of registers can be a benefit for a task because of C compiler optimizations but it implies a big cost for tasks commutations. In the MC68000 version of VRTX, there are no reserved registers for system use as there are less of them. After this introduction we are going to review some features of the operating system. Note 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 cannot have information on what task created it. Tasks are created end deleted without any incidence except context switching with respect to priorities. For example, there is no implicit meeting between a task and her mother at end of execution.

Each task has a priority defined at its creation. The priorities allows the system to choose which task among the ready ones will run on the processor (scheduling). VRTX is using fixed priorities in opposition to time sharing multi-task operating systems as UNIX. In such a system, the priority of a given task is varying along the execution. An intensive time consuming task priority will be lowered to avoid processor starvation for the other tasks. Then, the priority will be raised to initial value when the system consider that the task should get more processor execution time, this is considered as a fair operating system. As a lot of fixed priority operating systems, VRTX is an unfair one : a high priority task can get a hundred per cent of processor execution time.

There are two scheduling modes in VRTX. In the first, the ready tasks in the highest priority are executed in a round robin manner each for a maximum of time called quantum (typically 10 milliseconds). The operating system is getting processor control at the end of each quantum via a hardware clock interrupt (timer). This mode make the system fair for a given priority level. In the second mode, the quantum is infinite and the running task must relinquish processor explicitly or via a system call to allow another task execution.

We have seen previously that in our VRTX configuration there is only one address space shared by tasks and system. However, each task has a private stack that allows it (with a high level programming language or not) to implement function call mechanism with parameter passing or local variable dynamic allocation in a function. There is no stack memory protection in our configuration.

Here is a short description of common tasks operations. *Create* : crate the task and specifying priority, entry point, and execution mode (supervisor or user). *Delete* : immediate cancellation of a task, a task can terminate itself. *Suspend* : the task goes to non ready queue, a task can suspend itself. *Resume* : after a suspend, a task can be resumed by another one. *Priority* : change current task priority. *Information* : get the system internal task control block address with informations.

VRTX is an unfair operating system because it intended to be used for real-time application control. A high priority task can monopolize the processor and release it only when it has to do so. More system calls allow a task to lock context switching and quantum limited execution windows. Once the operating system is locked, there is only one task executed as with a mono-task one. The operating system can be of course unlocked. The operating system also has an idle task that has a low priority and is executed when there is no other task to execute. The default idle task can be replaced at configuration time by a user provided idle task which can be used to implement a low priority activity.

Communications between tasks. VRTX gives for communications between tasks four object types. They are common objects that can be found in other operating systems and allow the user to implement classical communication schemes (producer/consumer for example) or more complex ones. Communications means are (in case of a blocking system call, the user can give a maximum amount of time) listed after. *Mailbox* : a mailbox is a 32 bits memory, the user can post a value in and this is a non blocking action, there is an error if the mailbox is full. Pending on a mailbox can be a blocking action if it is empty. If more than one task is pending, the highest priority or next in same priority level will be served. *FIFO* : a FIFO is a set of 32 bits values. The user specifies the FIFO size at creation time and as for mailboxes, posting to a FIFO cannot be blocking. Pending a FIFO is blocking if it is empty but it is possible to check for it before. The user can ask for the current FIFO size and also jam a new element. There is a system call to get a copy of the next element in the FIFO without removing it. At FIFO creation, the user can tell the system the way blocked tasks are awaken. Two ways, the highest priority task is activated (as for mailboxes) or it is the first task waiting for the FIFO which is activated and so on in blocking order. *Flags set* : a flag set is made of 32 flags, each bit in a 32 bits word. For the consult action, the user gives a mask and a parameter to indicate a "and" or "or" wait condition. If the condition is satisfied, consulting the flag is not blocking. Consulting a flag never changes its value, a system call allows this. Task awaking of flags is made according to priorities and more than one task can be awaken at a time. *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 for a shared resource. The user can add one to a counting semaphore which is never blocking. He can also decrement it and if the value is zero, the calling task is blocked. As for a FIFO, there are two strategies for tasks awaking.

There is also another way to share data between tasks : shared memory as we said that there is only one address space. The combined use of semaphores and shared memory allows the user to implement as complex communications schemes as possible.

Interrupts. Just several key point concerning interrupts in general. The processor interrupt concept by peripherals is a mechanism on which an operating system design is based. System call are usually implemented with software interrupts that are similar to hardware generated ones but raised by a specific processor instruction. When an interrupt occurs, the processor execution flow is restarted at an address called an interrupt service routine. The execution mode is automatically set to privileged and further interrupts are not allowed. Several registers are also saved either on a stack or in additional special registers that can be in turn saved in memory.

This mode of operation allows the operating system to manage all resources with coherency : save/restore tasks contexts, handle communications, etc. Without being interrupted in critical phases. The operating system also drives peripherals using interrupts, for example a peripheral telling the processor that data is available by

using 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 a specific rate for time triggered actions.

One can consider two operating systems behaviors regarding interrupts. Some are not interruptible while the operating system is in privileged mode and then system call are atomic. They execute without being interrupted and if an interrupt occurs, the corresponding handler execution is delayed. These operating systems are said to be non preemptive ones in privileged mode. This behavior has an 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. In VRTX implementation interrupts are allowed as much as possible so handlers delay is minimized.

In VRTX, the user can use a special stack to backup registers at interrupt routine prologue. Such a routine can be activated by hardware without indirection via a interrupt driver as it is the case in many real-time operating systems. This is reducing interrupt latency. An interrupt routine can communicate with task level, for example, post a value in a mailbox to trigger a task level process according to fixed priorities. However, this mechanism can be used only with conformance to a protocol with the operating system : it is necessary to signal interrupt routine entry and exit if system calls should be used in it. At the end of interrupt routine, control must be given back to operating system in supervisor mode and then scheduling can occur that will produce one or more context switches. This protocol is a little time consuming but really less than in other operating systems with interrupt drivers. For a specific application, this protocol can be avoided to get better interrupt latency.

We are now going to describe another operating system suitable for real-time applications.

2.3.2 LynxOS and POSIX real-time extensions

Before describing the Lynx operating system as we did for VRTX, we are going to point out several informations about POSIX standard ("Portable Operating System Interface" X for similarities with UNIX). Information listed here is given essentially by [8], [11] and [12].

POSIX reference number is 1003 in IEEE directory ("Institute of Electrical and Electronics Engineers") which intend to establish industrial standards. POSIX is also referenced as an international standard of ISO 9945 ("International Standards Organization"). This standard is based on application programming interfaces descriptions of the operating system implementing it. These interfaces specifies data types, types names, function names and semantic. We consider POSIX used with the C programming language which is also standardized. Application developed with POSIX conformance can be considered as operating system independent. Most UNIX implementations implements POSIX interface but this is also the case for VxWorks and LynxOS for example. There are several subsections in POSIX, the basic features are in 1003.1 and real-time extensions are in 1003.4 and 1003.4a for real-time threads extensions.

The LynxOS operating system aim to be POSIX and ANSI-C conforming which is a interesting approach. For ANSI-C it is based on GNU compilers and libraries. For operating system interface LynxOS includes all POSIX even real-time extensions. On another hand, LynxOS is also compatible with the two main UNIX flavors : "System V" and BSD. LynxOS is a multi-task and multi-user operating system with the same execution levels than in VRTX but a user notion similar to UNIX is also implemented. LynxOS is supported for several targets such as IBM PC, Motorola VME boards based on MC68000 or PowerPC, SUN spark based workstations, several HP embedded platforms, IBM PowerPC based computers for example. The operating system is delivered in binary plus a collection of object files to allow the user to customize it. The complete development environment is located on the target system directly.

The kernel size can go from 200Kb up to more than 1Mb with all possible extensions. It can be stored in ROM with a minimal ROM file system and it can be booted via a network. These two possibilities allow application deployment on minimal hardware configurations and multiple targets to share easily the same software configuration.

Even if LynxOS integrates a lot of UNIX inspired standards, the kernel is really different mainly because it is designed to target real-time applications. The kernel design is very close to VRTX one but the development environment is on target. Our evaluation system is based on a MC68040 VME board with LynxOS installed on a local hard disk that allow real-time application implementation.

Memory management is paginated and can also be virtual. Each process has its own address space which is private. The kernel has also its own address space but kernel pages are always resident in core memory. If virtual memory management is not enabled, all process pages are also resident in core memory. If it is enabled, a swap area can be defined on a hard disk to store pages that cannot be in core memory temporally. Such a memory segmentation brings a safety feature as processes and kernel cannot access each other address spaces.

Processes and activities. The basic execution entity of LynxOS has the same name than UNIX one : process. But here a process has a fixed priority as opposed to time dependent priority under UNIX. Scheduling is made with different priority queues and a round robin quantum for a given priority as in VRTX. The quantum can differ from one priority level to another so LynxOS is fair for time scheduling only for tasks with the same priority (this is close to real-time processes handling in "System V.4" version of UNIX).

If virtual memory in engaged, processes pages can swapped, LynxOS swapping is handled by a system process who has a priority as all processes. This means that processes priorities have an influence on memory management. A process with a higher priority than the memory management process will never be swapped. Processes with a lower priority can be swapped. In addition to this mechanism, it is possible for a process to lock pages in core memory without influence of priority. This must be used with caution because it can lead to a memory starvation. A LynxOS process can be considered as a resource encapsulation unit such as file descriptors, memory space, etc. A program execution is implemented as a "thread". A LynxOS process is made 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 and also a priority given at creation. A process always has an initial thread, which is particular to LynxOS and not specified in POSIX recommendations. A LynxOS process with its initial thread is equivalent of a process in the original UNIX way. As for VRTX, a thread can monopolize the processor resource and scheduling is not necessary fair as usual in real-time operating systems. 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 on POSIX. *Create* : thread creation. The operating system assign a unique thread identifier that can be used for further operations. Some parameters can set priority, stack size, scheduling type (FIFO, ROUND ROBIN, DEFAULT), etc. There are defaults values for these attributes. *Self termination* : thread termination by itself with an exit code that can be an address in the stack. This data area is accessible by other threads so stack memory is not released after termination. *Termination by another thread* : a special exit code is given. *Detach* : release all resources of a terminated thread. Several scheduling algorithms are possible and the default one is based on fixed priorities with a quantum for a given level. Each priority level can have its own quantum. FIFO and ROUND ROBIN scheduling are the same but with one global infinite quantum for the first and a fixed quantum for each priority level for the second.

Communications. There are two communication tools types, one suitable for inter-process and the other one for threads. For inter-process communication the tools are implementations of UNIX "System V" and BSD flavors. LynxOS includes IPCs ("Inter Process Communications") and also 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, messages queues) is made easier. For threads running in a process, there are more simpler tools designed for efficiency the user can combine to implement complex communication schemes.

Additionally to shared memory in the process address space, LynxOS implements POSIX tools. *Rendezvous* : a thread can wait for another thread termination via a rendezvous. When it occurs, the waiting thread get the exit code of the other thread. *Mutexes* : it is similar to a binary semaphore but only the owning thread can release the mutex. A thread is blocked when it tries to get a mutex taken by another thread. This is used for mutual exclusion toward a shared resource that must be accessed in a critical section. *Condition variables* : combined to mutexes, it allows complex synchronizations between threads. After awning a mutex, a thread can wait on a condition variable. During this, 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 awaken. It is possible to wakeup one thread among a set of thread waiting on the same condition with the same mutex. It is also possible to wakeup all the waiting threads at the same time. A thread can use processes communication tools and LynxOS includes tools that are not required by POSIX such as fast binary and counting semaphores and shared memory.

Software signals similar to UNIX ones are also available in LynxOS and can be sent to process or threads. A signal sent to a process is taken randomly by one of its threads which accept that signal. If none of them accept it, it is pending waiting for a thread to accept it. A signal can be sent to a particular thread in a process.

Interrupts. LynxOS does not allow an application to catch a hardware interrupt itself. According to LynxOS designers this direct mode leads the user to implement services that should be handled by the operating system. In VRTX, we have seen that this direct mode is better for interrupt latency but with the cost of more code to implement. The only way to handle interrupts with LynxOS is to use system drivers. Only drivers can attach a

hardware interrupt to a handler functions. Usually, interrupt routines are as short as possible and they trigger more time consuming handlers in system threads. These threads are executed in kernel space and as they are scheduled, the user can manage priorities for interrupts.

A particular counting semaphore type is used by an interrupt handler to communicate with system threads level. There is 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 informations listed here come from [3] and [9]. The ADA programming language is a high level language that integrates a lot of features. One of the target application field is real-time development and we will describe here ADA as a programming language that allows to specify applications with asynchronous communicating tasks. There are two standards describing ADA : ANSI and ISO.

Tasks. It is possible to declare tasks in an ADA program. Execution of them is either controlled by a kernel linked with the application executable or by a host operating system not included in the ADA compiler. In the first implementation, the ADA compiler builds an executable by sources file compilation and link with a micro-kernel. In the second one, the compiler generate code for task creation in the host operating system.

The user can use the "PRIORITY" pragma to set priorities to ADA tasks. These priorities can have different significations depending on the host operating system. The scheduling algorithm is not specified in ADA standard and neither the way to schedule two tasks at same priority level. 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 mainly 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 the two tasks selection. A rendezvous is always blocking when only one task is trying to accept it. It is optionally possible to transfer data when a rendezvous is realized, similarly to a parameter passing for a function call.

It is possible with the "select" operator to wait for more than one rendezvous according to the state of local variables in a task. Only one rendezvous will be realized and if none is possible, the user can specify a default action to be taken. 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 are in the same priority level, it is the scheduling implementation of the underlying operating system that will decide. It is also possible to declare shared memory segments in ADA that can be combined with rendezvous to implement complex communication schemes. But usually, rendezvous with data exchange is enough.

Interrupts. It is possible to associate a hardware interrupt to an ADA rendezvous. Only one task then is allowed to wait for this rendezvous to occur. When the interrupt is taken, the corresponding rendezvous is realized and the waiting task is awaken. The task can get some read-only data when the rendezvous occur. An interrupt can be seen as the highest priority task always greater than real tasks. In 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 use a direct handler without an interrupt driver or another 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.