Throughput optimization by software pipelining of conditional reservation tables

Master of Science Thesis – Secure and Dependable Computer Systems

Thomas Carle

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, August 2011

Distributed Real-Time Scheduling Optimization using Software Pipelining techniques

Thomas Carle

Examiner: Jan Jonsson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden August 2011

1. Preface

This Master of Science Thesis report details the work performed during an internship at INRIA Rocquencourt, a French public research laboratory in computer science and automatic control. The team in which the internship took place is called AOSTE, which stands for ”models and methods for the Analysis and Optimization of Systems with real-Time and Embedding constraints”.

I would like to thank the whole team for allowing me to work with them as well as for the welcome and help they gave me throughout the internship. I would especially like to thank my tutor in the team, Dr Dumitru Potop-Butucaru, for his help, advice and for the precious time he gave me.

2. Abstract

In the world of embedded real-time applications, the optimization of schedules has been since long a major concern. Indeed such applications are often ruled by hard real-time constraints, meaning that they must compute a correct result in terms of logical computations, but also that this result must be computed before a deadline. In case the result is not computed before the deadline, the consequences to the system can be dramatic, equivalent to or worse than a wrong logical computation.

My master thesis work concerns embedded control systems, which are systems in which the software controls a physical process. For such systems, control engineers provide the software development teams with a discretized automatic control specification usually described in Matlab/Simulink, SCADE or other equivalent formalisms. Such applications always have a cyclic/periodic execution model alongwith real-time constraints such as latency/makespan and throughput. These constraints must be respected, as well as the functional specification. The general problem of automatically generating optimal code in this context is NP-complete or undecidable (depending on the formalism used), but various techniques exist for generating efficient implementations. Our work starts from existing techniques allowing the generation of optimized code for one cycle of computation. Such techniques allow the fulfillment of the latency constraints. The present work proposes techniques that optimize the throughput of applications at a constant latency. Thus, it completes existing implementation techniques.

Our work is based on the representation of static real-time schedules using conditional reservation tables defining the mapping of computations to computing and communiation resources in time according to the execution conditions. This approach is validated by many industrial standards such as ARINC 653 for avionics and AUTOSAR for automotive industry.

On such conditional reservation tables we apply software pipelining techniques inspired from existing work on code optimization for microarchitectures with instruction-level parallelism (superscalar, VLIW). Our solution to the problem is one (new) kind of software pipelining of ”modulo scheduling” type that respects the real-time requirements of our problem. The originality of the solution we provide, when compared to classical software pipelining techniques, is that:

1. It defines a scheduling model that better supports and takes advantage of the execution conditions attached to each computation, resulting in more compact generated schedules in the end.

2. As in traditional software pipelining, a dependency analysis must be performed. Nevertheless, in our technique, the real-time context allows the optimization of this analysis, which should allow for a shorter analysis time.

I started my research work by studying a corpus of research articles dedicated to embedded control systems, and more precisely to synchronous systems, and to software pipelining techniques. Those articles were the starting point of a detailed bibliography that I did in order to get good knowledge on the subject, and especially on existing software pipelining techniques. A particular interest was given to modulo scheduling techniques – software pipelining techniques that allow the definition of an optimized schedule of operations by performing simple computations and a dependency analysis – and to predicated execution for conditional branches of applications. This bibliographical study allowed the understanding of the state of the art in this domain, and also put into light the limitations inherent to these techniques, especially when it comes to the real-time constraints that were mentioned earlier.

At that point I was able to define a formal model for the representation of pipelined schedules, and to use it in order to define pipelining algorithms. Such algorithms are the key to solving the problem, as they take a (non-pipelined) schedule in input, and output an optimized pipelined version of the schedule that respects all the constraints we want to fulfill. Moreover, alongside the algorithms was developed a memory management technique that tackles all variable reuse problems that arise during pipelining.

The formal model for pipelined schedules is embodied in an intermediate representation for both pipelined and non-pipelined implementations, which I defined in order to be able to automate code generation. I then implemented the algorithms using those models, into a prototype written in Objective CaML. The algorithms were tested on real-life cases and showed good results.

At the end of the internship, I was able to describe my work and my achievements in a research paper that will be submitted soon, and to present them to the rest of my research team, as well as to other researchers from both the real-time and the software pipelining communities.

3. Table of contents

Table of contents

Figures

4. Introduction

4.1. The AOSTE research team

I conducted my research in the AOSTE research team of INRIA (http://www-sop.inria.fr/aoste/) . The team develops correct-by-construction implementation techniques for real-time embedded systems. The AOSTE team conducts research in covering the entire development cycle, from high-level modeling to the actual mapping of applications on particular hardware architectures. The price to pay for covering the entire design cycle is focusing work on specific models of concurrency like data-flow process networks and synchronous reactive languages, which give the theoretical basis of its developments.

On the specification side, the team defined a generic "logical time" approach based on logical clocks and constraints between them, which has been implemented in the OMG UML MARTE profile (http://www.omg.org/omgmarte/). On the implementation side, the team defined the AAA methodology (Algorithm/Architecture Adequation http://www.syndex.org/) which imposes that the scheduling action be coupled to a mapping of computations and communications to processing units and communication media described by an architecture model.

The AAA methodology is embodied in the SynDEx tool for the design of distributed real-time embedded applications. From a high-level description of the application, given in a hierarchical data-flow formalism close to SCADE or simulink, SynDEx is able to synthesize a distributed implementation that targets a given hardware architecture. It is based on the synchronous paradigm that we will present in the theory section.

4.2. Motivation and problematics

Embedded systems design brings together research and engineering communities that used to be only loosely connected. This new interaction helps bring forth common problems that are central to more than one community. This cross-fertilization ideally results in the development of common formalisms and general modeling, analysis, and code generation techniques.

My work followed this paradigm for a specific problem: the efficient execution of cyclic computations over synchronous architectures comprising several computing and communication resources . Instances of this problem are present at several levels of the embedded design cycle. At low level, compilers are expected to improve code speed by taking advantage of micro-architectural instruction level parallelism[1]. To minimize synchronization overhead, pipelining compilers usually rely on reservation tables to represent an efficient (possibly optimal) static allocation of the computing resources (execution units and/or registers) with a timing precision equal to that of the hardware clock. Executable code is then generated that enforces this allocation, possibly with some timing flexibility. But on VLIW architectures , where each instruction word may start several operations, this flexibility is very limited, and generated code is virtually identical to the reservation table. The scheduling burden is mostly supported here by the compilers, which include software pipelining techniques [2] designed to increase the throughput of loops by allowing one loop cycle to start before the completion of the previous one.

A very similar picture can be seen in the system level design of safety-critical realtime embedded systems. The timing precision is here coarser, both for starting dates, which are typically given by timers, and for durations, which are characterized with worst-case execution times (WCET). However, safety and efficiency arguments[3] lead to the increasing use of tightly synchronized time-triggered architectures and execution mechanisms, defined in well-established standards such as TTA, FlexRay[4], ARINC653[5], or AUTOSAR[6]. Systems based on these platforms typically have hard real-time constraints, and their correct functioning must be guaranteed by a schedulability analysis. In this paper, we are interested in statically scheduled systems where resource allocation can be described under the form of a static reservation table which constitutes, by itself, a proof of schedulability. Such systems include:

- Periodic time-triggered systems[7; 8; 9; 10; 11] that are naturally mapped over ARINC653, AUTOSAR, TTA, or FlexRay.
- Systems where the scheduling table describes the reaction to some sporadic input event (meaning that the table must fit inside the period of the sporadic event). Such systems can be specified in AUTOSAR, allowing, for instance, the modeling of computations depending on engine rotation events [12].
- Some systems with a mixed event-driven/time-driven execution model, such as those synthesized by SynDEx[13].

To facilitate the synthesis and implementation of such systems from high-level specifications, implementation techniques[7; 8; 13; 11; 10] often produce a scheduling table that implements exactly one cycle of the embedded control algorithm.[1] Given the time-triggered execution policy, this means that cycles of the control algorithm cannot overlap. Depending on the nature of the controlled physical system or computing resource limitations, this restriction may not be acceptable (as the system becomes non-schedulable).

To work around this limitation, we define pipelining techniques adapted to this system-level real-time scheduling framework. We start from reservation/scheduling tables defining the (possibly distributed) non-pipelined time-triggered implementation of *embedded control applications*. We define algorithms that synthesize pipelined implementations where a new computation cycle can begin before the previous one has completed, subject to resource and inter-cycle data dependency constraints. The algorithms optimize the *throughput* of the system, but each computation cycle is executed exactly as specified by the input reservation table, so that all *latency* guarantees are preserved. The *functionality* of the system is also preserved. The pipelined implementation is represented using a *pipelined reservation table*.

We allow the use of *conditional scheduling tables* where each operation can be guarded by an activation condition, allowing a natural modeling of control applications having several (nominal or degraded) execution modes. Our algorithms give the best results on specifications without temporal partitioning, like the previously-mentioned AUTOSAR or SynDEx applications and, to a certain extent, applications using the FlexRay dynamic segment. For partitioned applications like those mapped over ARINC 653, TTA, or FlexRay (the static segment), our algorithms currently cannot exploit conditional control information, but allow pipelining and synthesize a new partitioning of the computation and communication cycles.

The remainder of the report is structured as follows. Section 5 provides a comparison with existing work. Section 6.1 and 6.2 use intuitive examples to define our

1 One hyper-period, in the case of multi-periodic specification

model of time-triggered system implementation. Section 6.2 extends this model to allow the representation of pipelined implementations and defines the mapping of pipelining-specific constructs to executable implementation code. It also provides a larger example involving conditional scheduling tables, a sporadic implementation model, and memory constraints. Section 6.7 provides the pipelining algorithms, and the data dependency analysis they use. Section 6.8 gives experimental results, and Section 7 concludes.

5. Related work review

An important part of my research work was to established a detailed bibliography relative to the two domains that the problem crosses: the world of embedded control systems, and the world of software pipelining. This study was crucial, as in any research, in order to be able to understand correctly the problem and its application domains, but also to get knowledge about what techniques and theories already exist, in order to be able to decide if they can be applied, adapted, or are not fit to the problem. This section is a quick state of the art in the domains of synchronous programming – a paradigm that allows to define and implement reactive systems such as those concerned in this thesis – and in software pipelining. A particular focus is given to a particular software pipelining technique called modulo scheduling, because it is from that technique that I derived the models used to solve the problem. An explanatory axample is also given in section 5.2.1. The end of the section is devoted to other optimization techniques close to software pipelining and especially to the retiming technique.

5.1. Reactive systems/Synchronous programming

Reactive systems [16], by opposition to transformational systems, is the term used to designate computing systems that interact continuously with their environment, in a timely manner. In other words, such systems cannot make their environment wait, and thus have to react quickly to any event. More precisely, they are activated by input events coming from their environment, and react to them within specified delays by producing an output. Such systems are very common in industry, and are often highly critical, which implies a high degree of control in their design to cope with their hard real-time constraints. These systems share some common features that need to be taken care of while designing them. First of all, they are often concurrent systems because:

- they run in parallel with their environment,
- they often include several sub-systems running in parallel
- they are often implemented in a distributed way for performance, fault-tolerance or functionality.

Second, they are submitted to hard real-time constraints induced by the environment. Failing in meeting those constraints may lead to catastrophic consequences and makes the system useless. Finally, those systems are dependable which means we must design them carefully, and be able to verify them thoroughly. As a consequence, engineers should be able to, and are encouraged to use formal methods to design and verify them.

The synchronous paradigm [14][15] meets all the aforementioned constraints. The behavior of a synchronous systems is divided in execution cycles. The synchronous model is based on the assumption that each cycle is executed without interaction with or interference from the exterior world. All interactions between the system and its environment happen between execution cycles. In addition to this atomicity hypothesis, the synchronous model also requires that the computation of each cycle is functionally deterministic and bounded in number of operations (so that worst-case execution time guarantees can be computed). As a consequence, we consider that time is discrete, divided into several execution instants that correspond to the reactions of the system.

The main implication of the synchronous hypothesis is that every synchronous program can be viewed as a (synchronous) finite-state automaton machine (FSM) in which the states are valuations of the memory state, and the transitions correspond to the reactions. By consequence, every synchronous formalism has a natural interpretation in two well-studied mathematical models: explicit FSMs and digital synchronous netlists at RTL level. This makes (automated) analysis and verification easier, but also helps us describe our system clearly and efficiently by providing a notion of deterministic concurrency. Synchronous formalisms define a system of activation conditions (also called "logical clocks") specifying under which conditions parts of the system will be activated or not when an execution instant is reached. This allows us to slow some parts of the system compared to the quickest clock, known as global clock or tick given by the succession of execution cycles.

High-level specification languages and graphical formalisms such as Esterel, Signal/Polychrony, Lustre/Scade allow a simple specification of complex systems. All such specifications can be given an interpretation in mathematical models (synchronous finite state machines and synchronous digital netlists) that allow the application of efficient verification techniques. Our work is concerned with code generation, and we will rely on a representation level which is much simpler and flexible than the high-level formalisms, allowing complex code transformations, yet preserving some of the high-level structural information to allow the definition of efficient analysis and code generation algorithms. Our representation of such systems [17] is obtained by drawing and decorating the various computations and communications of the different system parts with relevant information such as activation conditions (logical clocks), data dependencies, etc.

5.2. Software pipelining

Software pipelining [18] is a microcode generation discipline that was developed in the 1980's and 1990's and that targeted highly-parallel architectures (such as super-scalar computers for example). Due to the advances in the design and production of CPUs, it is now a common optimization used in most compilers. It is a family of code generation techniques that aim at optimizing the average-case throughput of loops on parallel architecture machines.

Software pipelining techniques were soon developed for the VLIW (Very Long Instruction Word) architectures, that is to say machines composed of many functional (execution) units that can operate independently in parallel, where each instruction word of the software contains multiple machine instructions to be executed simultaneously by the various functional units.

The main idea of software pipelining is to re-organize an executable code in order to achieve a maximal use of the parallelism of the platform. Reorganization must preserve the function of the code. However, it can move operations within the code while taking care that data dependencies are preserved. Thus, the notion of dependency graph is central to all SP techniques[18].

Software pipelining is mostly used to optimize the execution of loops, by allowing a new iteration of a loop to start before the end of a currently executing one in order to use hardware units that would otherwise remain unused. Multiple iterations of a loop are therefore active at the same time. In practice, each new iteration will be started at a fixed interval called the initiation interval (II) [19][20]. By doing so, we can eventually detect a recurring stable pattern in the execution, that hopefully involves several iterations of the loop. This pattern is called the kernel of the loop, and it includes all of the operations of a cycle, although these operations can be from different iterations. The length of this kernel will always be the length of the II.

Most techniques differ on the heuristic they use in order to determine the kernel of the loop iterations, which is the steady-state of operations that we reach when the pipeline is loaded. In practice, finding the smallest possible II allows us to reduce the time between the end of the various iterations (and thus to maximize the throughput of the program), as when an iteration finishes, the next ones have already been started.

5.2.1. Example

A simple example [19] can be used to illustrate: in this example, a pseudo machine code corresponding to the following loop is displayed :

```
for(i=0; i<n; i++){
        a[i]+=1;
};;
```

where array a has been initialized before.

The first operation reads the current value of a[i], then 1 is added to it, and the result is written back in a[i]. Note that the add operation takes two clock cycles. The hardware architecture is composed of a Read unit, two ALUs (Arithmetic and Logical Unit) and a Write unit, all connected to a central memory.

In the original, non-pipelined schedule, one cycle takes four clock cycles to execute. Thus, as only one cycle can execute at a time, the throughput of the system is ¼ result per clock cycle.

Clock cycle	Read unit	ALU1	ALU2	Write unit
1	Read			
2		Add		
3		Add		
4				Write

Original schedule for one cycle of the loop. Throughput : ¼ result per clock cycle.

Cycle mod II	Read unit	ALU1	ALU2	Write unit
0	Read s = 1	Add s = 2	Add s=3	Write s=4

Modulo resource reservation table for the example program.

Clock cycle	Read unit	ALU1	ALU2	Write unit
1	Read_1			
2	Read_2	Add_1		
3	Read_3	Add_1	Add_2	
4	Read_4	Add_3	Add_2	Write_1
5		Add_3	Add_4	Write_2
6			Add_4	Write_3
7				Write_4

Pipelined schedule : each line is a VLIW instruction word. II = 1. Kernel described at time 4.
Throughput : 1 per clock cycle (once the steady state is reached).

5.2.2. Modulo scheduling

One of the most commonly used software pipelining techniques is called modulo scheduling [20][21]. In this technique, the goal is to first determine the kernel length by computing the constraints that arise when pipelining the original loop.

When pipelining a loop, operations that were initially executed in a sequential way are now executed in parallel, which increases the requirements in terms of resources. As the hardware architecture is fixed, the number of computing resources of each kind imposes constraints on the initiation interval (and thus, on the length of the kernel). Indeed, as no functional unit can be over-committed at any time, the number of operations requiring a functional unit at each instant must not exceed the number of available functional units of this type.

$$\boxed{ResMII = Max_{i \in FU\ types}\left(\frac{\#\,cycles\ requiring\ FU\ type\ i}{\#\,units\ of\ type\ i \in processor}\right)}$$

[21] Minimum II due to resource constraints.

Moreover, another type of constraints arises from dependence cycles that are caused by recurrences inside the loop. When computing in a sequential way, there is no problem due to these dependencies, as any result needed during an iteration of the loop, but generated by a former iteration, has already been computed. If the pipelining process is done without care, it could happen that an operation that needs a result from a former computation be scheduled before that former computation, and that the result be not ready when needed. This could lead to brutal termination of computations, or to corrupted results. To avoid such problems, another limitation on the length of the II is computed. For all dependence cycles in the dependence graph, the length of the cycle is divided by the number of loop iterations the dependence spans (i.e. how many iterations of the loop are between the iteration that produces the data and the iteration that uses it). The maximum of all these ratios is another minimum boundary for the II. In case there is no recurrence, this minimum boundary is 0.

$$\boxed{RecMII = Max_{all\ dependence\ cycles} \frac{(length\ of\ cycle)}{(iteration\ distance)}}$$

[21] Minimum II due to recursion constraints

Once both have been computed, modulo resource reservation tables [20] are used, in order to effectively avoid resource conflicts. These tables map the scheduled operations to the resources in time, taking into account the cyclic behavior of the pipelined system. The goal is to create a table without conflict, and with the smallest modulo index possible, i.e the smallest II that would satisfy the constraint. Technically, a table is built with $max(ResMII, RecMII)$ rows, and $\#\ functional\ units$ columns. Then, operations are scheduled in the cells keeping in mind that each cell has three potential states : empty, no-conflict, and full. An empty slot means no operation has yet been scheduled in the slot. A no-conflict slot means that no control path exist in the dependence graph between the operation that is being scheduled, and the ones that already are in that slot. It is the case when scheduling operations from different paths of a conditional branch. On the contrary, when a control path exists between the operations that is being scheduled and an operation that already is in that slot, the slot is said to be full for that operation. In order to get the tightest schedule, each operation must be scheduled using the following technique : try to find no-conflict slots available for this operation. If there are, schedule the operation in the earliest available slot compared to the starting date of the operation. If not, schedule it in the earliest empty slot compared to the starting date of the operation.

When pipelining, as no dependence exist between the various iterations of the loop and there is enough functional units to avoid any resource conflict, it is possible to exploit parallelism and start one iteration of the loop every clock cycle. During the prologue, that is, the three first lines of the table, the pipeline is loading. The steady state is reached at clock cycle number four, and is repeated until the last iteration is launched. When the last iteration has been launched, the pipelined loop enters the epilogue phase that is pictured from line 5 to 7 in the table. During that phase, the pipeline unloads itself, and the computation ultimately ends. In the classic execution, one result is obtained every four clock cycles, whereas with pipelining, once the steady-state is reached, one result is issued every clock cycle. The gain in throughput is thus 300%, or said otherwise, the program execution is four times faster.

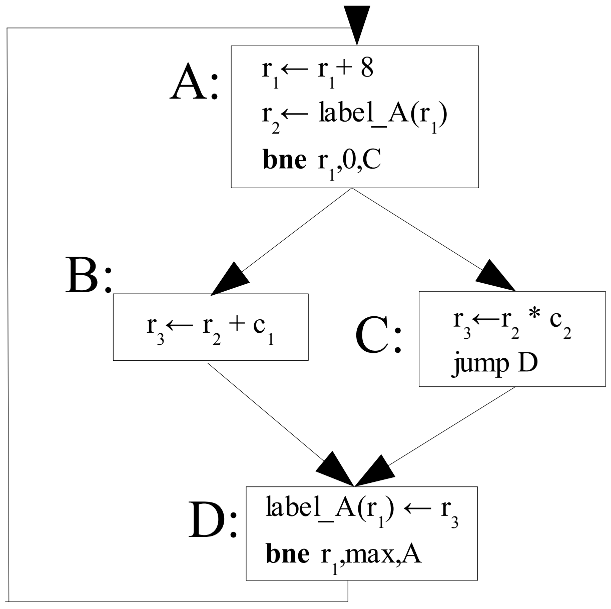
5.2.3. More advanced work

Other techniques [18] use an incremental modification of the loop body to construct the kernel (URPR/GURPR techniques [23]: Global UnRolling, Pipelining, and Rerolling for example), or use various scheduling techniques and check if repeating patterns appear (kernel recognition techniques).

Although software pipelining techniques have been developed for some years, and are now mature, it is still very hard to find efficient algorithms for loops that include conditional branches. However, some techniques were developed early on in order to cope with this problem. For example, [24][19] present a technique called hierarchical reduction that allows modulo scheduling to be used even when there are conditional branches in the loop. In short, both conditional branches are scheduled independently, and the shortest one is padded with NOPs. Then the scheduler treats the if-then-else as one operation. This technique has several disadvantages: padding paths with NOPs may slow the execution, considering the if-then-else as one operation gives an over estimation of the real resource requirements, and code motion becomes complicated. Another technique [21] is to modulo schedule all possible paths separately, and try to optimize the transition code between them. In this case, no guarantee can be given on the periodicity of the system schedule, as many different kernels exist, and the efficiency of the pipelined implementation is correlated to the efficiency of the transition code, and to the number of successive iterations without change in the executed conditional path. Indeed, the more the paths are changed, the more the system has to execute transition code, and change the kernel. On the contrary, the less the paths are changed, the more successive iterations of the same kernel will be executed. Thus, for loops where the probability of execution of the various possible paths are close to one another (e.g. one conditional branch with 50% chance that the condition is true), this technique clearly is suboptimal.

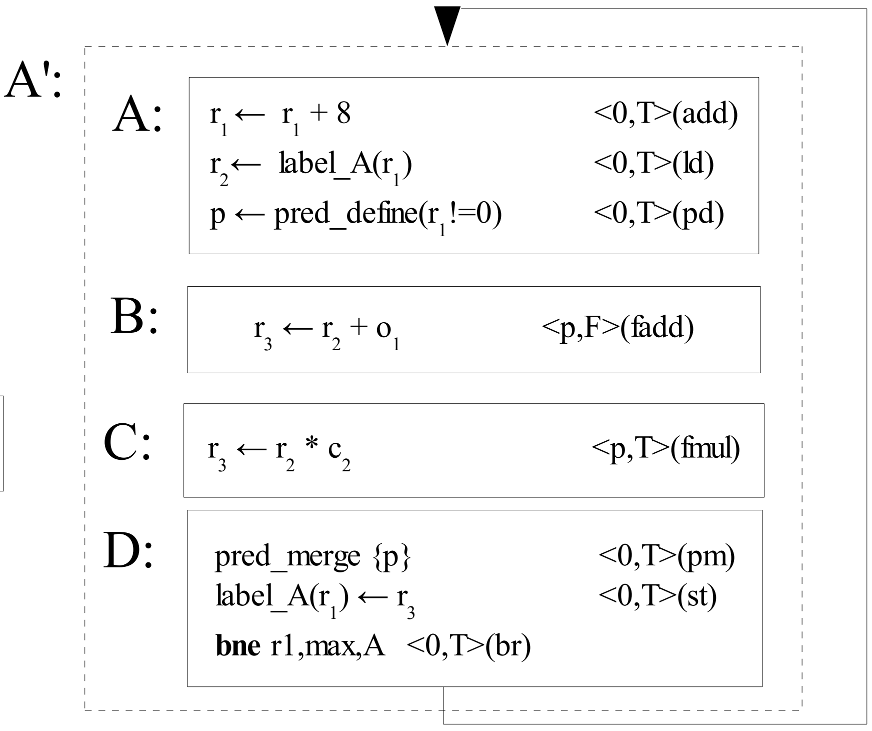
Other improvements in the field of software pipelining use predication to solve the problem of conditional branches [22]. The main idea is to use if-conversion [20] together with predication on conditional statements, that is, to replace the conditional statement by an operation that sets or clears a predicate, and to schedule both paths with a flag on each.

Example from [7] :



A:
$r_1 \leftarrow r_1 + 8$
$r_2 \leftarrow$ label_A(r_1)
bne r_1,0,C
B:
$r_3 \leftarrow r_2 + c_1$
C:
$r_3 \leftarrow r_2 * c_2$
jump D
D:
label_A(r_1) $\leftarrow r_3$
bne r_1,max,A

Fig1: Example loop before if-conversion



A':
A:
$r_1 \leftarrow r_1 + 8$ <0,T>(add)
$r_2 \leftarrow$ label_A(r_1) <0,T>(ld)
p ← pred_define(r_1!=0) <0,T>(pd)
B:
$r_3 \leftarrow r_2 + o_1$ <p,F>(fadd)
C:
$r_3 \leftarrow r_2 * c_2$ <p,T>(fmul)
D:
pred_merge {p} <0,T>(pm)
label_A(r_1) $\leftarrow r_3$ <0,T>(st)
bne r1,max,A <0,T>(br)

Fig2: Example loop after if-conversion

Then, all instructions are modulo scheduled as one branch. Before executing the former conditional branches, the flags are compared against the value of the predicate. If they match, the instruction is executed. If not, the instruction is nullified, that is to say it does not execute. As we see in the example, the conditional branch operation present in the non if-converted version of block A is replaced by a predicate define operation under the same condition. Then blocks B and C are both scheduled, but B only executes when predicate p is false, and C only executes when predicate p is true. In block D, an operation is added in order to merge the predicate after it has been used. As we can see, the schedule A,B,C,D in its if-converted version can be seen as just one block A', and thus can be scheduled as one branch.

One huge side-effect of that technique is that the resource constraints along both paths that are if-converted are summed. In other words, the compiler has to allocate distinct resources for each of the operations of both paths at each instant. This can require a lot of resources, or otherwise make the kernel of operations longer. Moreover, with predication, this is counter-productive, as only operations from one conditional path will execute at each iteration, meaning that resources assigned to the operations of the other path will remain inactive. To counter this side effect, [26] proposes a technique that removes the assignment of resources to operations whose predicate evaluates to false at run-time. The solution lies in assigning multiple operations to the same resource at the same time, as long as the predicates of these operations are exclusive with one another. This way, only one operation can have its predicate set to true at a given time, and the over-subscription of the resource is only virtual : only one operation can be executed at a time. This reduces the need for extra resources, and prevents the under-utilization of the resources of the system. Nevertheless, this technique requires heavy modifications of the hardware in order to support the over-committing of resources, and is thus not yet put into practice on a large scale. Still, in the context of our problem, it is possible to adapt this state of the art technique, as the conditional reservation tables allow the overcommitment of resources as long as the execution conditions of simultaneous computations on each resource are exclusive.

Another software pipelining technique [28] was developed recently for high performance multimedia embedded systems. It allows the synthesis of a pipelined schedule from a dependence graph, and gives a formalism for the description of the process. Although different from the problem treated here, it is noticeable because it is, to our knowledge, the first attempt to use software pipelining in the real-time context.

Other techniques derive from the retiming [29] method and can achieve noticeable improvements in the throughput of digital networks or be applied to software optimization by using a synchronous representation of the application. The goal of these optimizations is to cleverly move registers (memories) in the high-level representation graph of the application, in order to obtain a quicker execution time, and thus a better throughput.

6. Results

The final theoretical results obtained during the thesis are described in [25]. The objective is to define formalisms that allow the description of the hardware architecture on which the application will be running, as well as pipelined and non-pipelined reservation tables. Algorithms that allow us to pipeline a non-pipelined table will also be presented here. The architecture and implementation models designed for this paper are at a low abstraction level that allows an easy modeling of existing implementations, but makes the pipelining algorithms more complex. The techniques we describe here can easily be integrated at the end of the development cycle of applications.

The remainder of this section will present and illustrate the results described in the paper, by following the same outline.

6.1. Architecture model

The execution architectures' topologies will be described using sequential execution resources, memory blocks, and their interconnections. An architecture topology is thus a bipartite graph *A=<P,M,C>*, where *P* contains "processors", that is to say computation and communication devices capable of independent execution, such as CPU cores, accelerators and DMA controllers. Each processor can execute only one operation at a time, and has its own sequential or time-triggered program. This is natural on actual CPU cores, and models the fact that the cost of control of a DMA by another processor is negligible. *M* contains RAM blocks, that is to say sets of disjoint untyped memory cells. The set of all these cells in the system is called *Cells*. Each memory cell in fact represents an implementation variable in memory that can be of arbitrary type, and thus occupy an arbitrary space. Nevertheless, we focus mainly on enforcing the non-corruption of the data stocked in these memory cells, and it is assumed that new memory cells can be created at will for pipelining purposes, but the size of the memory cells is taken into account for memory access durations as well as for the lifetime analysis used to reduce the number of newly created cells during pipelining.

Finally *C* is composed of pairs of one processor and one memory block. Each (P,M) pair present in C denotes the fact that processor P has direct access to memory block M. All processors connected to the same memory block M can access it at any time, so the algorithms will have to ensure that no concurrent read-write or write-write access will occur, in order to protect data integrity inside the memory block. It is here assumed that the input schedule of the pipelining algorithm ensures this property. The algorithm will preserve it if it is already present in the input.

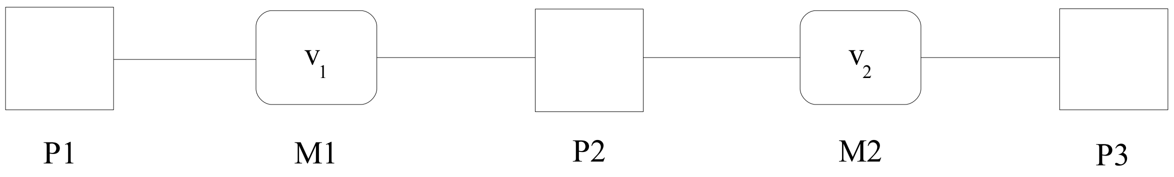


Fig3: A simple example architecture composed of three processors interconnected via two RAM blocks.

In this example, P={P1,P2,P3}, M={M1,M2}, C={(P1,M1),(P2,M1),(P2,M2),(P3,M2)}, and Cells={v_1,v_2}.

6.2. Implementation model

We focus on embedded control applications that we implement using the time-triggered paradigm. Our implementations all have a periodic non-preemptive execution model and are embodied by reservation tables, that is to say finite time-triggered activation patterns. Such a table describes exactly one cycle of execution of the system, and is thus periodically triggered every time its end is reached in order to run the system indefinitely. In the normal execution mode, the period at which the table is triggered corresponds to its length, meaning that no two different cycles can execute at the same time. Our goal is to reduce the triggering period, in order to make overlap cycles execution.

We chose to model reservation tables by a triple S = <p,O,Init>, with p being the activation period of cycles, O is the set of scheduled operations, and Init is the set of all initial values of the memory, that is, the initial state of the memory. If a memory cell is used by an operation before it has ever been written by another, the operation will use the initial value corresponding to the same variable, and present in Init. For a memory cell m, Init(m) can be either a constant or *nil*.

As p is the activation period of execution cycles, and cycles cannot overlap, p is equal the length of the reservation table len(S), that is to say the duration of one cycle of execution of the system. Each operation o in O has the following attributes:

- $\mathrm{In}(o) \subseteq \mathit{Cells}$ is the set of all variables (memory cells) that operation o uses as inputs,
- $\mathrm{Out}(o) \subseteq \mathit{Cells}$ is the set of all variables that are written by operation o,
- $\mathrm{Guard}(o)$ is the execution condition of operation o. It is a predicate defined over the values of memory cells. The set of variables needed for the computation of Guard(o) is GuardIn(o),
- $\mathrm{Res}(o) \subseteq P$ is the set of independently executing resources involved in the computation of o. This set can contain more than one resource, meaning that some operations need to be executed on multiple resources at the same time (cf knock control example),
- $t(o)$ is the start date of operation o,
- $d(o)$ is the duration of operation o. More precisely, it is a time budget determined through worst case execution time (WCET) analysis so that it cannot be exceeded by the execution of o.

During the execution, when t(o) is reached, if Guard(o) evaluates to true, all resources of Res(o) are used exclusively by o during d(o) time units. This is analogous to the use of predicates in classical software pipelining, with the important difference that guards naturally enable the overcommitting of resources, as long as they are exclusive to one another. This means that our technique does not imply an increase in resource needs or in the length of the kernel, contrarily to the predication techniques. The memory cells in GuardIn(o) are read at the beginning of the execution of o, but it is here assumed that the computation of the guard takes no time, or is included in the WCET time budget. As some variables can be read and updated by the same operation, the sets In(o) and Out(o) are not necessarily disjoint. Moreover, in our model, we consider that all variables inside In(o) and Out(o) are used during the whole execution of the operation. This is an important choice that we made in order to be able to perform lifetime analysis on the various variables handled by the system.

Time	P1	P2	P3
0	A@true		
1		B@true	
2			C@true

Fig4: Scheduling table for a simple application running on the architecture of Fig3.

Time	P1	P2	P3	
0	A@true iteration 1			Prologue
1	A@true iteration 2	B@true iteration 1		
2	A@true iteration 3	B@true iteration 2	C@true iteration 1	Steady-state
3	A@true iteration 4	B@true iteration 3	C@true iteration 2	
	...	...	...	

Fig5: Pipelined execution of the application.

This model is enough to describe non-pipelined reservation tables. Figure 4 gives the non-pipelined scheduling table for a very simple application running on the architecture described in figure 3. The length of the cycle is 3 time units, and thus the throughput is 1/3. It contains three operations defined as such: Operation A has no input, but writes its output in memory cell v_1. Thus we have: $\mathrm{In}(A)=\emptyset$ and $\mathrm{Out}(A)=\{v_1\}$. The same way we have: $\mathrm{In}(B)=\{v_1\}, \mathrm{Out}(B)=\{v_2\}, \mathrm{In}(C)=\{v_2\},$ and $\mathrm{Out}(C)=\emptyset$. the guards of each operation is true in this example, which means A, B, and C will execute at each cycle no matter what. The guards are represented in figure 4 with the notation "@true". Each operation has a dedicated processor: Res(A)={P1}, Res(B)={P2}, Res(C)={P3}, and the timings are the following: t(A)=0, t(B)=1,t(C)=2, and d(A)=d(B)=d(C)=1. Finally, there is no need for initialization of the variables, so the initial values of v_1 and v_2 (stored in Init) are: Init(v_1)=Init(v_2)=*nil.*

For pipelined implementations, we will incrementally extend this formalism in order to be able to have a compact but sound description format. A pipelined reservation table will describe the behaviour of the system during one pipelined cycle, that is to say the equivalent of the kernel of operations in traditional software pipelining: the smallest set containing all operations of a non-pipelined cycle and that repeats itself indefinitely, as long as new iterations are started. Indeed, the kernel describes exactly the repeating part of the execution of the system, but also describes the prologue phase, when the pipeline is loading, if we simply inhibit some operations. In order to be sound, our description model must be able to account for the prologue phase, otherwise it would not be able to describe the whole execution of the system. This is why in the pipelined implementation model, each scheduled operation o has an extra attribute called the starting index and noted fst(o). This index tells in which pipelined cycle the operation is scheduled for the first time. This allows the system to start running progressively, by scheduling operations effectively only when it makes sense. For example, an operation that has fst(o)=n will be scheduled effectively for the first time at the (n+1)th repetition of the pipelined cycle (indexes start at 0). Thus, an operation o executed in the pipelined cycle m belongs to the computation iteration m-fst(o).

Figure 5 describes the four first time units of the pipelined execution of the system. We see that during the prologue phase (time 0 and 1), the pipeline is loading, and operations are scheduled progressively (B is scheduled for the first time at time 1, and C at time 2). When C is scheduled for the first time, all operations are scheduled at the same time: the steady-state is reached, which means the kernel repeats itself in time as long as new iterations are launched. Graphically, the kernel of operations is easy to determine: it is any row of the table after time 1. The pipelined reservation table of figure 6 can be deduced from these observations. It represents the kernel of operations, and starting indexes are attributed to each of the operations: indeed, this table is of length 1, which means the new periodicity and throughput of the application is 1. If we do not use starting indexes, all operations will be scheduled at each time unit during the execution, just as if there was no prologue. We need to be able to reconstruct the execution in figure 5, just by repeating periodically the pipelined cycle of figure 6. In order to deal with the prologue phase, we give the following starting indexes to the operations: fst(A)=0, meaning that A will be scheduled when the system starts to execute. fst(B)=1 so operation B will not be scheduled in the first pipelined cycle of the execution: it will not appear at time 0, but will be scheduled at time 1. In the same fashion, fst(C)=2, so operation C will be scheduled for the first time in the third iteration of the pipelined cycle. Nevertheless, that instance of operation C, that will be scheduled at time 2, belongs to the first execution cycle of the system.

Time	P1	P2	P3
0	A@true fst(A)=0	B@true fst(B)=1	C@true fst(C)=2

Fig6: Pipelined reservation table of the application.

Now, we have a model that describes non-pipelined and pipelined implementations, and that allows the formalization of well-formed properties of such implementations. Indeed, some behaviours are prohibited for the systems we implement, and we need to define correctness properties for our models.

6.3. Well-formed properties

The problem here lies in the fact that a syntactically correct specification can model an incorrect implementation, or can be unimplementable. An incorrect implementation can arise due to non-determinism induced by data races or exceeded time budgets, and an unimplementable specification can be caused for example by the scheduling of an operation on a processor that is not connected to RAM blocks needed by the operation.

To avoid this last problem, we can formalize the following correctness property :

$$\forall o \in O, \forall m \in \mathrm{In}(\mathrm{o}) \cup \mathrm{Out}(\mathrm{o}), \exists P \in \mathrm{Res}(\mathrm{o}), \exists \mathrm{Block} \in M, (m \in \mathrm{Block}) \wedge ((P, \mathrm{Block}) \in C)$$

with M and C defined in the architecture description.

Nevertheless, some correctness properties, including this one, are of no importance to us because we assume that the input implementation models to our algorithms are correct, and thus respect these properties. Most of the well-formed properties will be preserved by our methods, because the internal state (allocation and scheduling) of each cycle is left unchanged. The only properties that we will formalize are correctness properties that take care of problems inherent to pipelining, that is to say problems that do not exist in the non pipelined implementations and that arise when we pipeline them.

First of all, we say that two operations o_1 and o_2 with $\mathrm{Res}(o_1) \cap \mathrm{Res}(o_2) \neq \emptyset$ are non-concurrent, denoted $o_1 \perp o_2$, if either their executions do not overlap in time $(t(o_1)+d(o_1) \geq t(o_2) \vee t(o_2)+d(o_2) \geq t(o_1))$ or if they have exclusive guards $(\mathrm{Guard}(o_1) \wedge \mathrm{Guard}(o_2) = \mathit{false})$. Using this notation we can formalize two well-formed properties that are supposedly respected by the input reservation tables, and that must be respected by our output pipelined models: the fact that no two operations can use the same processor at the same time (including the prohibition of preemptive behaviours), and the absence of data races.

Sequential processors: $\forall o_{1,} o_2 \in O, (\mathrm{Res}(o_1) \cap \mathrm{Res}(o_2) \neq \emptyset) \Rightarrow o_1 \perp o_2$.

Absence of data races: $\forall m \in \mathit{Cells}, ((m \in \mathrm{Out}(o_1)) \wedge (m \in \mathrm{In}(o_2) \cup \mathrm{Out}(o_2))) \Rightarrow o_1 \perp o_2$ (if a memory cell is written by an operation and used (read or written) by another operation, then these two operations are non-concurrent).

6.4. Dependency graph and maximal throughput

Given that our pipelining approach does not change scheduling decisions inside computation cycles, the transformation of Fig. 4 into Fig. 6 only depends on the throughput of the pipelined system. In turn the (maximal) pipelined throughput is determined by the dependencies between successive execution cycles. As we aim for periodic pipelined schedules, we can represent these dependencies as a Data Dependence Graph (DDG) - a formalism that is classical in software pipelining based on modulo scheduling techniques [2].

In this section we formalize DDGs and we explain how it limits the throughput of the maximal pipelined throughput can be computed from it. The actual computation of the dependency graph and the pipelined implementation will be detailed in Section 6.7.
Given an implementation model S =< p;O; Init >, the DDG associated to S is a directed graph:

$$DG =< O; V >$$

where $V \subseteq O \times O \times N$. Ideally, the elements of V are all triples (o_1; o_2; n) such that there exists an execution of the implementation and a computation cycle k such that operation o_1 is executed in cycle k, operation o_2 is executed in cycle k + n, and o_1 must be executed before o_2, for instance because some value produced by o_1 is used by o_2. In practice, any V including all the arcs defined above (any over-approximation) will be acceptable, leading to correct (but possibly sub-optimal) implementations.

The DDG represents all possible dependencies between operations, both inside a cycle (when n = 0) and between successive cycles at distance $n \geq 1$. Given the statically scheduled implementation model, with fixed dates for each operation, the pipelined schedule must respect unconditionally all these dependencies. This is classical in software pipelining based on modulo scheduling. The originality of our work comes from applying such a technique to an implementation model allowing a resource to be allocated at the same time to operations with exclusive activation conditions, and from the technique used for computing the dependency graph, which uses timing information to limit the computation of dependencies to triples (o_1; o_2; n) where o_1 and o_2 can actually overlap in time.

To define the constraints, we start with some definitions. Recall that a (non-pipelined) schedule table S =< p;O; Init >, represents statically scheduled periodic implementations where each operation is assigned a fixed starting date in every computation cycle. For each operation $o \in O$, we denote with $t_n(o)$ the date where operation o is executed in cycle n, if its guard is true. By construction, we have $t_n(o) = t(o) + n * p$. In the pipelined implementation of period p', this date is changed to $t'_n(o) = t(o) + n * p'$.

Given $(o_1; o_2; n) \in V$ with $n \geq 1$, the pipelined implementation must satisfy, for all $k \geq 1$:

$$t'_{k+n}(o_2) \geq t'_k(o_1) + d(o_1)$$

From here, we obtain:

$$p' \geq \frac{t(o_1) + d(o_1) - t(o_2)}{n}$$

Our objective is to construct pipelined schedules that satisfy these constraints for all $(o_1; o_2; n) \in V$, and which are well-formed in the sense of Section 6.3.

6.5. Code generation issues

Given that our pipelining approach does not change scheduling decisions inside computation cycles, the transformation of Fig. 4 into Fig. 6 only depends on the throughput of the pipelined system, which is determined by the analysis of Section 6.6. However, these figures mainly describe the scheduling of operations, whereas pipelining implies significant changes in memory allocation and the execution mechanism. We deal with these issues here.

We start with a notation: In the pipelined system, the maximal number of simultaneously-active computation cycles is $max\_par = \lceil len(S) / len(\overline{S}) \rceil$, where S is the non-pipelined scheduling table, and $\overline{S}$ is its pipelined version. Note that max_par can also be computed as $1 + max_{o \in O} fst(o)$. In the initial scheduling table of our example, both A and B use memory cell v1. In the pipelined table A and B work in parallel, so they must use two different copies of v1. We say that the replication factor of v1 is rep(v1) = 2. Each memory cell v is assigned its own replication factor, which must allow concurrent computation cycles using different copies of v to work without interference. Obviously, we can bound rep(v) by max_par. We use a tighter margin, based on the observation that most variables (memory cells) have a limited lifetime inside a computation cycle. We set rep(v) = 1 + lst(v) − fst(v), with

$$fst(v) = min_{v \in \text{In}(o) \cup \text{Out}(o)} fst(o)$$
$$lst(v) = max_{v \in \text{In}(o) \cup \text{Out}(o)} fst(o)$$

Through replication, each memory cell v of the non-pipelined scheduling table is replaced by rep(v) memory cells, allocated on the same memory block as v, and organized in an array $\overline{v}$, whose elements are $\overline{v}[0],...,\overline{v}[\text{rep}(v)+1]$. These new memory cells are allocated cyclically, in a static fashion, to the successive computation cycles. More precisely, the computation cycle of index n is assigned the replicas $\overline{v}[n \bmod \text{rep}(v)]$ for all v. The computation of rep(v) ensures that if n1 and n2 are equal modulo rep(v), but n1 6= n2, then computation cycles n1 and n2 cannot access v at the same time.

For systems like our simple example, where no information is passed from one computation cycle to the next, this static allocation allows for a simple code generation, which consists in replacing v with $\overline{v}[(\text{cid} - \text{fst}(o)) \bmod \text{rep}(v)]$ in the input and output parameter lists of every operation o that uses v. Here, cid is the index of the current pipelined cycle. It is represented in the generated code by an integer. When execution starts, cid is initialized with 0. At the start of each subsequent pipelined cycle, it is updated using the code:

```
cid:=(cid + 1) mod R
```

where R is the least common multiple of all the values rep(v).

When a computation cycle may use values produced by previously-started computation cycles,[2] code generation is more complicated, because a computation cycle may access memory cells different than its own. The code generation problem is complicated by the fact that it is impossible, in the general case, to statically determine which cell must be read (because the cell was written at an arbitrary distance in time). Thus, we need a dynamic mechanism to identify which cell to read. If more static pipelined implementations are needed, different pipelining techniques should be designed, either limiting the class of accepted non-pipelined systems, or allowing the copying of one memory cell onto another, which we do not allow because it may introduce timing penalties. Our memory access mechanism is supported by a new data structure which associates to each memory cell v of the non-pipelined scheduling table an array src(v) of length rep(v), and allocated on the same memory block as v. In this context, code is generated as follows:

- At execution start, all the values of src are initialized with 0 (pointing to the initial values of the memory cells).
- At the start of each pipelined cycle, for each cell v of the initial scheduling table, assign to src(v)[(cid − fst(v)) mod rep(v)] the value of src(v)[(cid − fst(v) − 1) mod rep(v)]. This assignment indicates that the value of v initially used during computation cycle cid is that used (but not necessarily produced) during computation cycle cid − 1 and stored in memory cell $\overline{v}[\text{src}(v)[(\text{cid} - \text{fst}(v) - 1) \bmod \text{rep}(v)]]$.
- When an operation o of the non-pipelined scheduling table reads v, its counterpart in the pipelined table will read $\overline{v}[\text{src}(v)[(\text{cid} - \text{fst}(o)) \bmod \text{rep}(v)]$. The same is true for cells used by the computation of execution conditions.
- When o writes v in the non-pipelined table, there are 2 cases:
  - If o also reads v, then the counterpart of o in the pipelined table will write the same memory cell it reads (as defined above).

2 This is necessary to represent systems having an internal state

- If not, then o writes the memory cell normally assigned to this computation cycle by the replication process ($\bar{v}[x]$, where x = (cid − fst(o)) mod rep(v)). An operation is added after o and on the same execution condition to set src(v)[x] to x.

The last aspect of memory management is initialization. In our case, v1 requires no initialization, so that none of its replicas do. In the general case, if Init(v) 6= nil, we need to initialize $\bar{v}[0]$ with Init(v), but not the other replicas.

6.6. The knock control example

We complete this section with a larger example that illustrates several key points of our approach, including the use of conditional scheduling tables and the pipelining of sporadic systems. Knock control is one of the functions of the engine control unit (ECU) of gasoline spark-ignition engines. At each rotation of the engine, it chooses for each cylinder an ignition time that maximizes power output while keeping engine-destructive knocks (autoignition events) at an acceptable level.

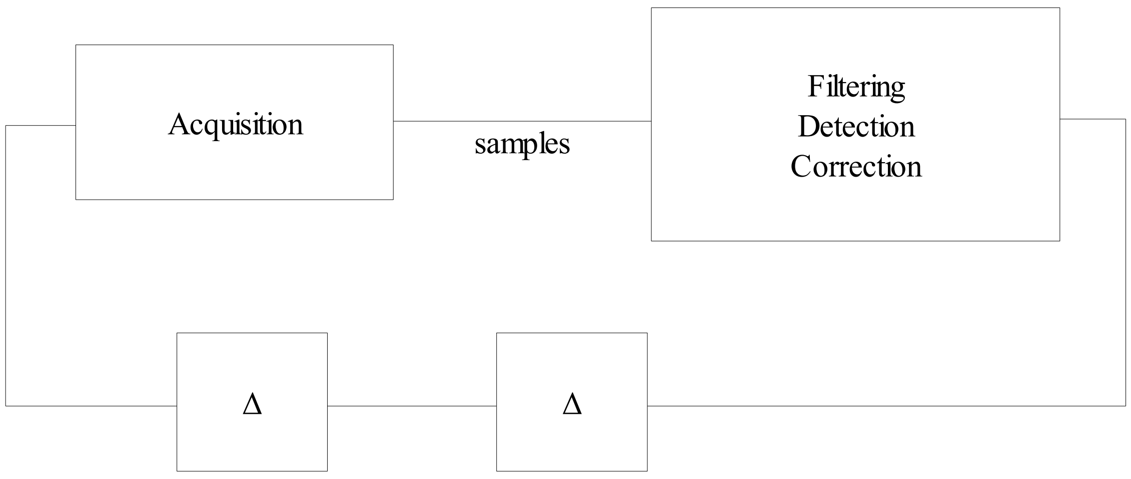


Fig7: High-level representation of the knock control function

We provide in Fig. 7 a high-level description of the knock control functionality. The model is based on an industrial case study and on the description of [12]. It leaves unrepresented the other ECU functions and degraded functioning modes, which are not essential to our presentation. The behavior is as follows: One computation cycle is triggered at each rotation of the engine crankshaft. The cycle starts with the acquisition of knock noise data. Acquisition is performed over a knock acquisition window where autoignition can occur. It is performed using a vibration sensor sampled at 100kHz, and the samples are stored in a buffer. The samples are used by the filtering, detection, and correction (FDC) function to adjust the ignition time (not figured here) and the position and size of the acquisition window. The configuration data produced by the computation cycle of index n controls the acquisition of cycle n+2. This delayed feedback is realized using two unit delays (labeled). Acquisition is performed by a specialized device (AD) of the ECU, whereas the FDC function is computed by the ECU microcontroller (µC).

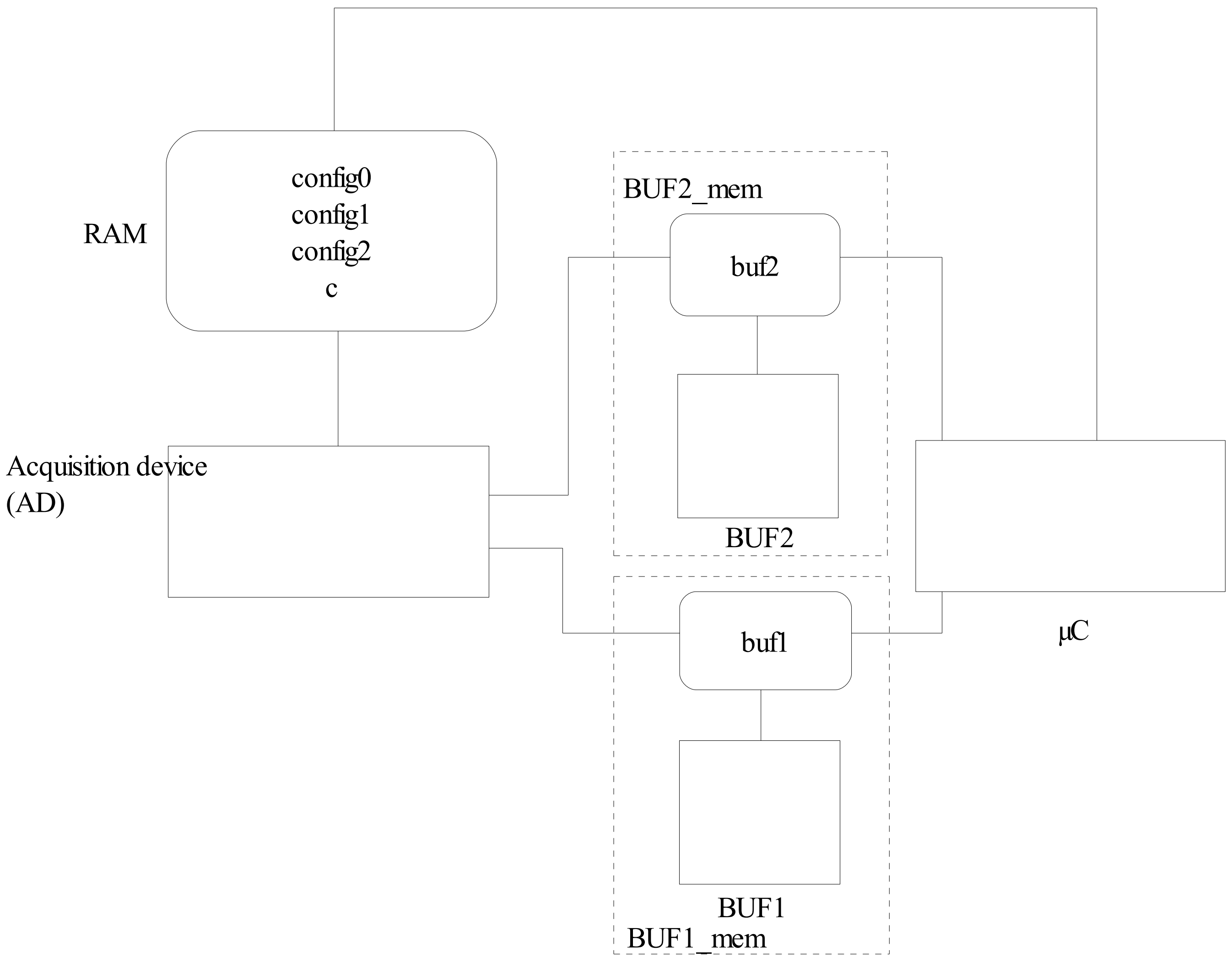


Fig8: Engine Control Unit (ECU) architecture

Rotation units	AD		BUF1		BUF2		µC	
0	book@true							
1	Acq1 @ c	Acq2 @ c	Acq1 @ c			Acq2 @ c		
2								
3			FDC1 @ c			FDC2 @ c	FDC1 @ c	FDC2 @ c
4								
5								

Fig9: Non-pipelined scheduling table for the knock control

For reasons related to the physics of engines and to computing resource limitations in the ECU, the successive computation cycles must sometimes be pipelined, by allowing the acquisition and FDC operations of successive cycles to be executed in parallel. Such a pipelining can be directly constructed using our approach, using the code generation scheme of the previous section. However, our code generation may conflict with memory constraints or pre-existent implementation choices. We will assume here that the system designers have already fixed the maximal number of buffers to 2, placed them at fixed places in memory, and written the protocol that alternates the use of the buffers in both acquisition and FDC. To model such an implementation, the two buffers are best represented as in Fig. 8, with two memory cells (buf1 and buf2) on separate memory blocks (BUF1_mem, resp. BUF2_mem). Each memory block has its own memory controller (BUF1, resp. BUF2) that ensures exclusive access and makes memory cell replication impossible during pipelining.

In this implementation model, the scheduling table of one computation cycle is represented in Fig. 9. Memory cell c is a Boolean used in guards to determine which buffer to use in the current computation cycle to pass data from the acquisition function to FDC. In the beginning of each computation cycle, operation book flips the value of c by executing "c:= ¬ c". In cycles where the new value is true, buffer buf1 is used. Otherwise, buf2 is used. If we denote with cn the value of c used in guards throughout computation cycle n, we have $c^n = \neg c^{n-1}$ for all n positive. Operation book also implements the "book keeping" function of the unit delays. The scheduling table represents both activation scenarios, corresponding to different initial values of c. In order not to introduce special memory access operations, we split the acquisition and FDC operations in two. Both Acq1 and Acq2 perform acquisition. But the first writes its samples in buf1 and is executed on condition c, while the second writes them in buf2 and is guarded by ¬ c. Each of the Acqi and FDCi operations use two resources: One of CD and µC and one of the memory controllers BUF1 and BUF2.

The operation durations must be interpreted here as upper WCET bounds in an engine rotation referential. More precisely, each duration gives the maximal rotation (in degrees) of the engine crankshaft during the execution of the operation. For the acquisition operation, this is the maximal acquisition window size. The FDC function runs on a microcontroller, and its duration is characterized with a classical WCET (in real time). Conversion to the engine rotation referential is performed by assuming the maximal engine rotation speed.

The algorithms of the next section determine that successive computation cycles can be at best pipelined as pictured in Fig. 10. To do so, they determine that $c^n = \neg c^{n-1}$ for all n, thus allowing the acquisition and FDC operations of successive computation cycles to be executed in parallel. Note that a resource can be allocated to two operations at the same dates if their guards are exclusive. Like in Fig. 9, we represent here both activation scenarios, corresponding to different initial values of c.

The corresponding pipelined scheduling table is provided in Fig. 11. The system is schedulable if the length of this table is smaller than the engine rotation interval between successive triggers of computation cycles. In turn, this is given by the number of cylinders and structure of the engine. If the system is schedulable, the code generation technique of Section 6.5 can be used to automatically generate the book keeping memory cells and code. Thus, we automatize the analysis of [12] and also allow automatic code generation.

Rotation units	AD		BUF1		BUF2		μC	
0	$book^0$ @ true							
1	Acq_1^0 @ c^0	Acq_2^0 @ $\neg c^0$	Acq_1^0 @ c^0			Acq_2^0 @ $\neg c^0$		
2								
3	$book^1$ @ true		FDC_1^0 @ c^0			FDC_2^0 @ $\neg c^0$	FDC_1^0 @ c^0	FDC_2^0 @ $\neg c^0$
4	Acq_2^1 @ $\neg c^1$	Acq_1^1 @ c^1		Acq_1^1 @ c^1	Acq_2^1 @ $\neg c^1$			
5								
6	$book^2$ @ true			FDC_1^1 @ c^1	FDC_2^1 @ $\neg c^1$		FDC_2^1 @ $\neg c^1$	FDC_1^1 @ c^1
7	Acq_1^2 @ c^2	Acq_2^2 @ $\neg c^2$	Acq_1^2 @ c^2			Acq_2^2 @ $\neg c^2$		
8								
	...		...		…		...	

Fig10: Pipelined execution of the knock control

Rotation units	AD		BUF1		BUF2		μC	
0	book@true		FDC_1 @ c^{n-1} fst= 1			FDC_2 @ $\neg c^{n-1}$ fst= 1	FDC_1 @ c^{n-1} fst= 1	FDC_2 @ $\neg c^{n-1}$ fst= 1
1	Acq2 @ $\neg c^n$	Acq1 @ c^n		Acq1 @ c^n	Acq2 @ $\neg c^n$			
2								

Fig11: Pipelined scheduling table for the knock control

6.7. Pipelining algorithms

Our algorithms determine if it is possible to pipeline the execution of the system, and if so, the new period of the system. This is achieved by a dependency analysis that computes the DDG of the application by using the timing information present in the initial schedule. All these dependencies then allow the algorithms to determine the new throughput of the system, and thus the pipelined reservation table.

Time	P1	P2	P3
0	A@true		
1		B@true	
2			C@true
3	D@true		

Fig12: Dependency analysis example

Time	P1	P2	P3
0	A@true fst(A)=0		C@true fst(C)=1
1	D@true fst(D)=1	B@true fst(B)=0	

Fig13: Dependency analysis example, pipelined

6.7.1. Dependency analysis

In our setting, there are 2 sources of dependencies:

1. "Classical" data dependencies between operations of successive cycles. These dependencies must be preserved by any pipelining.
2. Dependencies that facilitate the computation of a periodic or sporadic pipelined implementation.

To explain the source of the second dependency type, consider the nonpipelined scheduling table of Fig. 12. Resource P1 has an idle period between operations A and B where a new instance of A can be started. However, to preserve a periodic execution model, A should not be restarted just after its first instance (at date 1). Indeed, this would imply a pipelined throughput of 1, but the fourth instance of A cannot be started at date 3 (only at date 6). The correct pipelining starts A at date 2, and results in the pipelined scheduling table of Fig. 13. Note that the pipelined system is strictly periodic, of period 2, because every instance of D is bound to its slot of size 1 between two instances of A (and vice-versa).

Determining if the reuse of idle spaces between operations is possible requires a complex analysis which basically checks, for each integer n smaller than the length of the initial table, whether a pipelined scheduling table of length n can be constructed. This complex computation can be avoided when such idle spaces between two operations are excluded from use. This can be done by creating a dependency between any two operations of successive cycles that use a same resource and have non-exclusive execution conditions.

Excluding such idle spaces from pipelining also has the advantage of supporting a sporadic execution model. In sporadic systems the successive computation cycles can be executed with the maximal throughput specified by the pipelined table, but can also be triggered arbitrarily less often, for instance to tolerate timing variations, or to minimize power consumption in systems where the demand for computing power varies.

Dependency analysis is performed by Function 1. The function takes as input a scheduling table and a Boolean flag stating whether the analysis should include dependencies of the second type. It produces a unique integer. When fast_pipelining_flag is true, the idle spaces are excluded from pipelining and the result gives the length of the pipelined scheduling table. When fast_pipelining_flag is false, idle spaces can be used, and the result gives a lower bound on the distance between successive starts of computation cycles. Respecting this lower bound ensures that data dependencies are satisfied (but not dependencies of the second type). Moreover, this function can be called many times to construct incrementally the DDG of the application: the goal is to make sure that all data dependencies have been found and included in the graph.

Function 1 works as follows: it takes two schedules in input, and concatenates them without pipelining using the Concat2Copies function. It returns a new schedule S'. The list of all starting and ending of operations is constructed using the BuildEventList function. This function returns a sorted list in terms of dates of events. Function1 then computes all data dependencies between the operations of the concatenated schedule. During Phase 1 scheduling table S' is progressively transformed to allow the identification of the data dependencies between operations. The result needs not be consistent as a scheduling table. It is simply used as an internal data structure similar to a static single assignment (SSA) form. The transformation proceeds as follows: For each memory cell v of the initial table and every operation o producing v, we introduce a new cell v_o. We also introduce a new cell v_{init} representing the initial value of v. With these notations, there is a bijection between the possible productions of v and its versions vo and v_{init}.

The remainder of the transformation is performed by a symbolic execution of S', realized by the traversal of list l. A new data structure *curr* is used to identify at each point of the list traversal the possible producers of each memory cell. For each cell v of the initial table, *curr*(v) is a set of pairs v_o@C, where vo is a version of v and C is the condition on which the value of v is that corresponding to vo. Condition C is a predicate over memory cell versions. The predicates of the elements in *curr*(v) provide a partition of *true*. Initially, *curr*(v) is set to {v_{init}*@true*} for all v. This changes upon treatment of completion events (lines 23-31 of Function 1).

Dependencies are identified in lines 15-16 (for classical data dependencies) and 18- 21 (for the dependencies of the second type). The code involves in both cases predicate comparisons. A third predicate comparison is used in line 29. These comparisons are handled by a SAT solver that also considers a Boolean abstraction of the operations of the algorithm. In our knock control example, the Boolean abstraction of the book operation provides the information that $c_n = \neg c_{n-1}$.

These dependencies are added to the DDG under the form : (o_i;o_j;n) where o_i and o_j are data dependent and n is the number of iterations that separate the execution of o_i and o_j. In practice, n is just the number of times Function 1 has been called, because we call it each time using an incrementally constructed pipelined schedule as first input, and the original schedule as second input.

To do so, Function 1 is first called with the two first non-pipelined cycles of execution of the application. From that point, all inter-cycle dependencies that exist between these two successive cycles are computed. In practice, if operations o_1 from cycle 1 and o_2 from cycle 2 are data dependent, we add (o_1;o_2;1) to the graph. Then, these two cycles are pipelined using *min_dist* - that is, the number of time units that can be substracted to the timings of the second cycle without violating dependencies -, and used as input to the next call of Function 1, the second input schedule being the initial schedule. The new DDG input is the DDG output of the last call of Function 1. The difference with the first call of Function 1 is that now when the two schedules are concatenated, the second schedule does not start at the end of the execution of the initial cycle, but at the end of the pipelined version of the execution of the two first cycles of execution. Moreover, we only need to compute the dependencies between operations from the first cycle and the newly concatenated cycle. The same analysis process is performed until the nth initial schedule being concatenated at the end of the pipelined execution version of the n-1 first cycles starts executing after the end of the execution of the 1st cycle. At that point, we are certain that the DDG is complete, as all possible dependencies from one given cycle to all following cycles have been determined. In practice we actually have determined the conditional DDG of the application by using the timings of the application.

With this DDG, Function 4 is able to determine an optimized throughput for the application which respects all data dependencies as well as the periodic behaviour of the application, by using the formula determined in Section 6.4.

Algorithm1: Dependency Analysis

```
Input: S1 : current  scheduling table
       S2 : initial non-pipelined scheduling table
       iteration : integer stating how many time the function
has been called
       DDG : currently constructed DDG
       fast_pipelining_flag : boolean
Output: min dist : integer
        DDG : incremented DDG
1: /* Phase 0: Preliminaries */
2: S' := Concat2Copies(S1, S2)
3: l := BuildEventList(S')
4: /* Phase 1: Compute the dependencies */
5: for all o operation in S' do
6:      Out(o) := {$v_o$ | v ∈ Out(o)}
7: for all v ∈ Cells do
8:      curr(v):={$v_{init}$@true}
9:while l not empty do
10:     e := head(l) ; l := tail(l)
11:     if e = start(o) then
12:             Assume Guard(o) = $g_o$(v1, . . . , vk).
Replace it by:
                $\bigcup_{v'_i@C_i \in curr(v_i),\, i=\overline{1,k}} (C_1 \wedge \ldots \wedge C_k) \wedge g_o(v'_1, \ldots, v'_k)$
13:          for all o′ ∈ Completed, v ∈ In(o), v′ ∈ Out(o′) do
14: /* Classical data dependencies */
15:             if v′@C ∈ curr(v) and C ∧ Guard(o) ≠ false
then
16:                DDG := DDG ∪ {(o′, o, iteration)}
17: /* Dependencies of the second type */
18:             if fast_pipelining_flag then
19:               if Res(o) ∩ Res(o′) ≠ ∅ then
20:                 if Guard(o) ∧ Guard(o′) ≠ false then
21:                   Depend := Depend ∪ {(o′, o, iteration)}
22:     else
23: /* e = end(o) */
24:     Completed := Completed ∪ {o}
```

```
25:       for all vo ∈ Out(o) do
26:              new curr(v) := vo@Guard(o)
27:             for all vo′@C′ ∈ curr(v) do
28:                C″ := C′ ∧ ¬Guard(o)
29:                if C″ ≠ false then
30:                  new curr(v) := new curr(v) ∪ {vo′@C″}
31:             curr(v) := new curr(v)
32: /* Phase 2: Compute the minimal distance */
33: Depend := {(o′, o, iteration) ∈ DDG | t(o) > len(S1) ≥ t(o′)}
34: min_dist := $min_{(o', o, iteration) \in Depend}(t(o) - t(o') - d(o'))$
35: return DDG, min_dist
```

Algorithm2: BuildSchedule

```
Input:        S  : Non-pipelined schedule
              p' : new period of the system
Output:       S' : Pipelined schedule of the application
/* Compute operation dates and start indices */
4:     for all o in O do
5:            fst(o):= $\lfloor \frac{t(o)}{p'} \rfloor$
6:            t'(o):=t(o) - fst(o) * p'
/* Assemble the pipelined schedule */
7:     S' := AssembleSchedule(S, p', fst, t′)
8:     return S'
```

6.7.2. Pipelining routines

Algorithm 3: PipelineSchedules

```
Input: S1 : non-pipelined scheduling table
       S2 : non-pipelined scheduling table
       counter  : integer
       min_dist : integer
Output: S       : non-pipelined scheduling table
        $S_{counter}$     : integer
1:   S := S1
2:   for all o in S2 do
3:       t(o) := t(o) + len(S1) - min_dist
4:       S := AddOpToSchedule(o,S)
5:   $S_{counter}$ := len(S1) - min_dist
6:   return (S,$S_{counter}$)
```

The pipelining functions are simple drivers using the results of the dependency analysis. Function 2 takes a non-pipelined scheduling table and the Data Dependency Graph computed by several iterations of Function 1, and builds a pipelined schedule. It starts by determining the new period of the system, using the formula given in section 6.4. It then determines the start indices and new start date of every operation. Then, it calls AssembleSchedule to fully assemble the pipelined scheduling table. This function implements the complex memory allocation scheme described in Section 4.1, and we shall not provide pseudocode for it here.

Function 3 is used to construct incrementally the first input schedule of function 1. It basically concatenates the current schedule with one instance of the initial non pipelined schedule, and pipelines them using *min_dist* that has been computed by Function 1. It outputs the new current schedule, as well as the starting date of the last cycle iteration that it contains. This way, we can know if the last iteration starts after the end of computation of the first cycle contained in the schedule. This is the condition on which we can stop computing the DDG, as it is complete when that condition becomes true.

Function 4 is the top-level pipelining driver. The function starts by performing the data analysis, in order to define the new period of the system. If *fast_pipelining_flag* = true, the output scheduling table is simply produced by a call to Function 2. If not, we have to find the pipelined scheduling length by testing all the values between the minimal new period and the non-pipelined length. For each length j, the pipelined scheduling is assembled, but the result may not respect the well-formed properties defined in Section 6.3. We do not provide here the code of function ConsistentSchedule, which closely follows the definition of these properties.

Algorithm 4: Pipelining driver

```
Input: S1 : non-pipelined schedule table
       fast_pipelining_flag : boolean
Output: S : pipelined schedule table
1:   new length := 0
2:   DDG:= ∅
3:   i := 1
4:   S2 := S1
5:   cover := false
6:   while ¬cover do
7:       (DDG,min_dist) := DependencyAnalysis(S2, S1,i,DDG,
fast_pipelining_flag)
8:       (S2, $s_i$) := PipelineSchedules(S2, S1, i, min_dist)
9:       cover := ($s_i$ > len(S1))
10:      i := i + 1
11:  for all (o;o';i) in DDG do
12:      p':=max(p, $\lceil \frac{t(o)+d(o)-t(o')}{i} \rceil$ )

13:  if fast_pipelining_flag then
14:      S:= BuildSchedule(S1,p')
15:      return S
16:  else
17:      for j:= p' to len(S1) do
18:          new_length:=j
19:          S := BuildSchedule(S1,new_length)
20:          if ConsistentSchedule(S) then
21:              return S
```

6.8. Experimental results

We have applied our pipelining algorithms on several examples, and the results are synthesized in Table 1.

	Scheduling table length		
example	initial	pipelined	gain
cycab	1482	1083	27,00%
ega	84	79	6,00%
knock	6	3	50,00%
simple	3	1	66,00%

Table 1: experimental results

The largest and most typical example we use is the embedded control application of the CyCab electric car [22]. The CyCab control application we use allows it to be driven manually, or in an autonomous “platooning” mode where it follows the vehicle in front of it, letting it make the speed and direction change decisions. The embedded software runs on a platform composed of 3 micro processors connected by a CAN bus. Our pipelining technique allows a significant reduction of 27% in cycle time. This reduction means that the application can be significantly complexified while maintaining I/O latency.

The second example is an adaptive equalizer. This filter is normally part of a larger control application, but we considered it here in isolation. The particularity of this example is that it has already been carefully designed to exploit the parallelism of the execution platform (it can be seen as “manually pipelined”). The cycle length reduction after application of our technique is not very large, but it is still significant in spite of the very optimized starting point.

The third example is the knock controller, based on an industrial case study and presented in Section 4.2. We also add a line for our toy example. The comparison is interesting, because this example allows for an ideal pipelining with a resource usage of 100%.

7. Conclusion

We have defined a pipelining approach allowing the optimization of the throughput of periodic and sporadic real time systems defined through scheduling tables. The pipelined system preserves all the latency guarantees of the non-pipelined system. Our approach includes a model for the representation of non-pipelined and pipelined systems, a code generation technique, and pipelining algorithms. By using a very general system model, our approach can be applied at implementation level, allowing a simple integration in existing design flows. We have applied our technique, with good results, on real-life systems and system models with various implementation types.

For the future, many problems remain. One of them is the extension of the framework to fully cover memory constraints and the exploitation of execution guards over partitioned architectures. Using the n-synchronous formalism [27] should allow us to express and potentially exploit regular repetition patterns in the pipelining process. Another important goal is to integrate pipelining in the initial scheduling process, so that better trade-offs between latency, throughput, and resource usage can be obtained.

8. Bibliography

[1] J.L. Hennessy and D.A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 4th ed. edition, 2007.

[2] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. ACM Computing Surveys, 27(3), 1995.

[3] G. Fohler, A. Neundorf, K.-E. Årzén, C. Lucarz, M. Mattavelli, V. Noel, C. Von Platen, G. Butazzo, E. Bini, and C. Scordino. EU FP7 ACTORS project. Deliverable D7a: State of the art assessment. Ch. 5: Resource reservation in real-time systems. http://www3.control.lth.se/user/karlerik/Actors/d7a-rev.pdf, 2008. Online. Accessed 15 mars 2011.

[4] J. Rushby. Bus architectures for safety-critical embedded systems. In Springer, editor, Proceedings EMSOFT'01, volume 2211 of LNCS, Tahoe City, CA, USA, 2001.

[5] Airlines Electronic Engineering Committee. Arinc 653: Avionics application software standard interface, 2005.

[6] Autosar (automotive open system architecture), release 4. http://www.autosar.org/, 2009. Online. Accessed 15 mars 2011.

[7] P. Caspi, A. Curic, A. Magnan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to tta: a layered approach for distributed embedded applications. In Proceedings LCTES, San Diego, CA, USA, June 2003.

[8] W. Zheng, J. Chong, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli. Extensible and scalable time-triggered scheduling. In Proceedings ACSD, St. Malo, France, June 2005.

[9] A. Monot, N. Navet, F. Simonot, and B. Bavoux. Multicore scheduling in automotive ecus. In Proceedings ERTSS, 2010.

[10] P. Eles, A. Doboli, P. Pop, and Z. Peng. Scheduling with bus access optimization for distributed embedded systems. IEEE Transactions on VLSI Systems, 8(5):472–491, Oct 2000.

[11] D. Potop-Butucaru, A. Azim, and S. Fischmeister. Semantics-preserving implementation of synchronous specifications over dynamic tdma distributed architectures. In ACM, editor, EMSOFT '10 Proceedings of the tenth ACM international conference on Embedded software, pages 199–208, 2010.

[12] C. André, F. Mallet, and M.-A. Peraldi-Frati. A multiform time approach to real-time system modeling; application to an automotive system. In Proceedings SIES, Lisbon, Portugal, July 2007.

[13] T. Grandpierre and Y. Sorel. From algorithm and architecture specification to automatic generation of distributed real-time executives. In Proceedings MEMOCODE, 2003.

[14] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), 2003.

[15] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9), 1991.

[16] D. Harel and A. Pnueli. On the development of reactive systems, in Logics and models of concurrent systems, NATO Advanced Study Inst. *On Logics and Models for Verification and Specification of Concurrent Systems*, New York: Springer Verlag, 1985.

[17] D. Potop-Butucaru, R. de Simone, Y. Sorel, J.-P. Talpin. Clock-driven distributed real-time implementation of endochronous synchronous programs. EMSOFT '09.

[18] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3),1995.

[19] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. *Proceedings of the SIGPLAN '88*, 1988.

[20] N. J. Warter, G. E. Haab, J. W. Bockhaus, K. Subramanian. Enhanced Modulo Scheduling For Loops With Conditional Branches. *Proceedings of the 25th Annual International Symposium on Microarchitecture* , pp.170-179, 1992.

[21] M. G. Stoodley, C. G. Lee. Software pipelining loops with conditional branches. *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture* ,1996.

[22] N. J. Warter, D.M. Lavery, W.W. Hwu. The benefit of predicated execution for software pipelining. *Proceeding of the Twenty-Sixth Hawaii International Conference on System Sciences* , 1993.

[23] B. Su, S. Ding, J. Wang, J. Xia. GURPR—a method for global software pipelining. *Proceedings of the 20th annual workshop on Microprogramming,* 1987.

[24] A. Aiken, A. Nicolau, S. Novack. Resource-constrained software pipelining. *Transactions on Parallel and Distributed Systems,* vol.6, no.12, 1995.

[25] T. Carle, D. Potop-Butucaru. Throughput optimization by software pipelining of conditional reservation tables. INRIA research report RR-7606, 2011.

[26] M. Smelyanskiy, S. A. Mahlke, E. S. Davidson, H.-H. S. Lee. Predicate-aware scheduling: a technique for reducing resource constraints. *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization,* 2003.

[27] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. N-synchronous kahn networks: a relaxed model of synchrony for real-time systems. *In Proceedings POPL'06, pages 180–193. ACM Press*, 2006.

[28] Y-S Chiu, C-S Shih, and S-H Hung. Pipeline schedule synthesis for real-time streaming tasks with inter/intra-instance precedence constraints. *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2011.

[29] C. E. Leiserson, and J. B. Saxe. Optimizing synchronous systems. *22nd Annual Syposium on Foundations of Computer Science (SFCS)*, 1981.