Throughput Optimization by Software Pipelining of Conditional Reservation tables

Thomas Carle, Dumitru Potop-Butucaru

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Throughput Optimization by Software Pipelining of Conditional Reservation Tables

Thomas Carle — Dumitru Potop-Butucaru

N° 7606

Avril 2011

Embedded and Real Time Systems

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7606--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
PARIS - ROCQUENCOURT

Throughput Optimization by Software Pipelining of Conditional Reservation Tables

Thomas Carle , Dumitru Potop-Butucaru

Theme : Embedded and Real Time Systems
Équipe-Projet AOSTE

Rapport de recherche n° 7606 — Avril 2011 — 20 pages

Abstract: Reservation tables are used at various levels in embedded systems design to represent the allocation of resources in cyclic computations. They model system-level static real-time task schedules in fields like automotive or avionics, but also model the cycle-accurate ordering of instructions at microarchitectural level, as used in software pipelining. To optimize system throughput, successive execution cycles can be pipelined, subject to resource constraints and inter-cycle data dependencies. In this paper we take inspiration from software pipelining to define system-level pipelining techniques for static task schedules given under the form of scheduling/reservation tables. We allow the use of conditional tables where each operation can be guarded (predicated). Our algorithms optimize system throughput while maintaining the end-to-end latency guarantees defined by the input scheduling table. We demonstrate the approach on real-life examples of task scheduling problems.

Key-words: embedded systems, real-time, distributed applications, scheduling,computation cycles, code generation, software pipelining

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Optimisation du débit de sortie par pipelinage logiciel de tables de réservation conditionnelles

Résumé : Les tables de réservation sont utilisées à différents niveaux dans le design des systèmes embarqués, afin de représenter l'allocation des ressources dans le cas de calculs cycliques. Elles modélisent l'ordonnancement statique de taches temps-réel au niveau du système dans des champs d'application tels que l'automobile ou l'avionique, mais aussi l'ordre d'execution des instructions d'un cycle de calcul au niveau microarchitectural, comme dans le cas du pipelinage logiciel. Pour optimiser le débit de sortie du système, des cycles d'exécution successifs peuvent être pipelinés, en prenant garde aux contraintes dues aux ressources et aux dépendances de données inter-cycles. Dans cet article, nous nous inspirons du pipelinage logiciel pour définir des techniques de pipelinage au niveau du système pour les ordonnancements statiques de taches donnés sous la forme de tables de réservation/d'ordonancement. Nous autorisons l'utilisation de tables conditionnelles où l'exécution des opérations peut être soumise à la valeur d'un prédicat. Nos algorithmes optimisent le débit de sortie du système, tout en maintenant les garanties sur le temps de réponse définies dans la table d'ordonnancement initiale. Nous illustrons notre approche par des exemples tirés de problèmes réels d'ordonnancement de taches.

Mots-clés : systèmes embarqués, temps réel, applications distribuées, ordonnancement, cycles de calcul, génération de code, pipelinage logiciel

1 Introduction

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.

Our paper follows 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 real-time 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]

[1] One hyper-period, in the case of multi-periodic specifications.

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 paper is structured as follows. Section 2 provides a comparison with existing work. Section 3 uses intuitive examples to define our model of time-triggered system implementation. Section 4 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 5 provides the pipelining algorithms, and the data dependency aanalysis they use. Section 6 gives experimental results, and Section 7 concludes.

2 Related work

One main inspiration in our work came from the software pipelining community[2], and especially from the predication-based approaches to the software pipelining of loops with conditional branches[14, 15, 16]. These approaches allow an independent, possibly predicated (conditional) control of the various computing resources at instruction level. This model parallels the one we shall use at system level, but significant differences remain. The first one is the objective. Software pipelining techniques optimize the processing *speed* (throughput) of loops while preserving the computing *function* [17]. In addition to function, our approach also preserves existing real-time *end-to-end latency* guarantees, and allows a *periodic* execution. To this end our approach preserves all scheduling decisions of the input scheduling table, whereas software pipelining techniques rely on more complex code transformations.

The periodic time-triggered execution model also means we have the exact value of system throughput before and after pipelining. Optimal solutions always exist, which is not always the case in software pipelining of loops with conditional branches [17].

In the real-time scheduling community, we already mentioned in the introduction existing work on the implementation of systems based on static reservation/scheduling table models [7, 8, 9, 10, 12, 13]. In this context, our objective of relaxing the frontiers between execution cycles to improve schedulability and efficiency is not new. We are aware of two existing solutions. Unlike our approach, which works at a time-triggered *implementation* level and assumes no knowledge of the process generating this implementation, these approaches intervene at a synchronous dataflow *specification* level. This specification is re-organized to allow the generation of better real-time schedules using existing synthesis tools. In one approach, specification re-organization is semi-automatic [7]. The drawback is that expert human intervention and actual changes in the specification itself are needed.

In the second approach, reorganization is based on an automatable retiming technique [18, 19]. Retiming techniques move operations across the barrier between 2 execution cycles, in one sense or the other. This is allowed whenever the move does not change the input/output latency of the initial specification, computed as a number of execution cycles (not necessarily in real time). Our approach does not have this restriction. For instance, we can pipeline *stateless* systems where no value is passed from one execution cycle to the next. Our algorithm can exploit conditional behaviors to optimize pipelining, whereas retiming techniques work in a pure dataflow context.

More generally, given the synchronous architectures we consider, our work is related to previous approaches for the distributed and real-time implementation of synchronous formalisms [20].

3 Implementation model

We define here the formalism we use to model non-pipelined implementations. Inspired from [21, 13], our formalism remains at a significantly lower abstraction level. The models of [21, 13] are fully synchronous: Each variable has at most one value at each execution cycle, and moving one value from a cycle to the next can only be done through explicit *delay* constructs. In our model, each variable (called a memory cell) can be assigned several times during a cycle, and values are by default passed from one execution cycle to the next. One operation can read and write on a given memory cell at the same time.

This lower abstraction level allows the simple modeling of existing implementations, but complexifies the pipelining algorithms, as we shall see in the following sections.

3.1 Architecture model

We model execution architectures using a very simple language defining *sequential execution resources*, *memory blocks*, and their *interconnections*. Formally, an architecture model is a bipartite undirected graph $\mathcal{A} =< \mathcal{P}, \mathcal{M}, \mathcal{C} >$, with

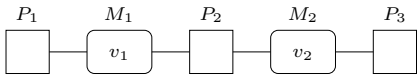

Figure 1: Simple architecture

$\mathcal{C} \subseteq \mathcal{P} \times \mathcal{M}$. The elements of $\mathcal{P}$ are called *processors*, but they model all the computation and communication devices capable of independent execution (CPU cores, accelerators, DMA controllers, etc.). We assume that each processor can execute only one operation at a time. We also assume that each processor has its own sequential or time-triggered program. This last assumption is natural on actual CPU cores. On devices such as DMAs and accelerators, it models the assumption that the cost of control by some other processor is negligible.

The elements of $\mathcal{M}$ are RAM blocks. We assume each RAM block is structured as a set of disjoint *cells*. We denote with *Cells* the set of all memory cells in the system, and with $Cells_M$ the set of cells on RAM block M. For clarity reasons, we do not provide a full treatment of memory allocation issues. In particular, we assume that we can (statically) create new cells on a given memory bank for pipelining purposes. However, we do account for memory cell sizes in both memory access durations and in the lifetime analysis used to minimize the creation of new cells during pipelining. We also provide, in Section 4.2, a mechanism that manipulates the architecture model to represent specific memory allocation constraints.

Finally, $(P, M) \in \mathcal{C}$ specifies that processor P has direct access to memory block M. All processors directly connected to a memory block M can access M at the same time. Therefore, care must be taken to prohibit concurrent read-write or write-write access by two or more processors to a single memory cell, in order to preserve functional determinism (we will assume this is ensured by the input system, and will be preserved by the pipelined one).

The simple architecture of Fig. 1 has 3 processors (P_1, P_2, and P_3) and 2 memory blocks (M_1 and M_2). Each of the M_i blocks has only one cell v_i.

3.2 Implementation model

On such architectures, we execute time-triggered implementations of embedded control applications with a *periodic non-preemptive* execution model. We represent such an application with a scheduling/reservation table, which is a finite time-triggered activation pattern. This pattern defines the computation of one period (also called an *execution cycle*). The infinite execution of the embedded control system is the infinite succession of periodically-triggered execution cycles.

Formally, a reservation/scheduling table is a triple $\mathcal{S} =< p, \mathcal{O}, Init >$, where p is the *activation period* of execution cycles, $\mathcal{O}$ is the set of *scheduled operations*, and *Init* is the *initial state* of the memory.

The activation period gives the (fixed) duration of the execution cycles. All the operations of one execution cycle must be completed before the following execution cycle starts. The activation period thus sets the *length* of the scheduling/reservation table, and is denoted by $len(\mathcal{S})$.

	Non-pipelined scheduling table		
time	P1	P2	P3
0	A@true		
1		B@true	
2			C@true

Figure 2: Simple scheduling table

The set $\mathcal{O}$ defines the operations of the scheduling table. Each scheduled operation $o \in \mathcal{O}$ is a tuple defining:

- $In(o) \subseteq Cells$ is the set of memory cells whose data is used as input (read) by o.
- $Out(o) \subseteq Cells$ is the set of cells written by o.
- $Guard(o)$ is the execution condition of o, defined as a predicate over the values of memory cells. We denote with $GuardIn(o)$ the set of memory cells used in the computation of $Guard(o)$.
- $Res(o) \subseteq \mathcal{P}$ is the set of processors used during the execution of o.
- $t(o)$ is the start date of o.
- $d(o)$ is the duration of o. The duration is viewed here as a time budget the operation must not exceed. This can be statically ensured through a worst-case execution time analysis.

All the resources of $Res(o)$ are exclusively used by o after $t(o)$ and for a duration of $d(o)$. The sets $In(o)$ and $Out(o)$ are not necessarily disjoint, to model variables that are both read and updated by an operation. For lifetime analysis purposes, we assume that input and output cells are used for all the duration of the operation.[2] The cells of $GuardIn(o)$ are all read at the beginning of the operation, but we assume the duration of the computation of the guard is negligible (zero time).

To cover cases where a memory cell is used by one operation before being updated by another, each memory cell can have an *initial value*. For a memory cell m, $Init(m)$ is either *nil*, or some constant.

The simple scheduling table pictured in Fig. 2 uses the architecture of Fig. 1. It has a length of 3 and contains 3 operations (A, B, and C). Operation A reads no memory cell, but writes v_1, so that $In(A) = \emptyset$ and $Out(A) = \{v_1\}$. Similarly, $In(B) = \{v_1\}$, $Out(B) = In(C) = \{v_2\}$, and $Out(C) = \emptyset$. All 3 operations are executed at every cycle, so their guard is *true* (guards are graphically represented with "@true"). The 3 operations are each allocated on one processor: $Res(A) = \{P_1\}$, $Res(B) = \{P_2\}$, $Res(C) = \{P_3\}$. Finally, $t(A) = 0$, $t(B) = 1$, $t(C) = 2$, and $d(A) = d(B) = d(C) = 1$. No initialization of the memory cells is needed (the initial states are all *nil*).

[2] The memory access model where an operation reads its inputs at start time, and writes its outputs upon completion can be represented on top of our model.

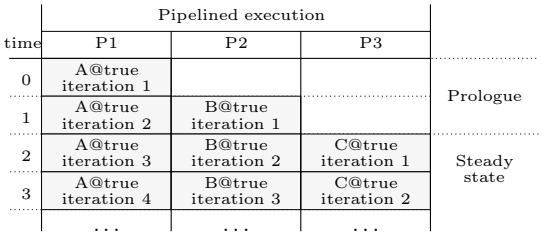

Figure 3: Pipelined execution

3.3 Well-formed properties

The formalism above provides the syntax of our implementation models, and allows the definition of operational semantics. However, not all syntactically correct specifications model correct implementations. Some of them are non-deterministic due to data races or due to operations exceeding their time budgets. Others are simply un-implementable, for instance because an operation is scheduled on processor P, but accesses memory cells on a RAM block not connected to P. A set of correctness properties is therefore necessary to define the well-formed implementation models.

However, some of these properties are not important in this paper, because we assume that the input of our pipelining technique is already correct. Our pipelining techniques will preserve most correctness properties because they preserve all allocation and scheduling choices *inside each execution cycle*. We only formalize here two correctness properties that will need attention in the following sections.

We say that two operations o_1 and o_2 are *non-concurrent*, denoted $o_1 \perp o_2$, if either their executions do not overlap in time ($t(o_1) + d(o_1) \geq t(o_2)$ or $t(o_2) + d(o_2) \geq t(o_1)$), or if they have exclusive guards ($Guard(o_1) \wedge Guard(o_2) = false$). With this notation, the following correctness properties are assumed respected by input (non-pipelined) implementation models, and must be respected by the output (pipelined) model:

Sequential processors. No two operations can use a processor at the same time. Formally, for all $o_1, o_2 \in \mathcal{O}$, if $Res(o_1) \cap Res(o_2) \neq \emptyset$ then $o_1 \perp o_2$.

No data races. If some memory cell m is written by o_1 ($m \in Out(o_1)$) and is used by o_2 ($m \in In(o_2) \cup Out(o_2)$), then $o_1 \perp o_2$.

4 Pipelined implementations

In this section, we define our model of pipelined implementation. This model builds over the non-pipelined one, enriching it with structural information describing specific implementation choices. In turn, these choices impact the scope and efficiency of the pipelining techniques defined in Section 5.

For the example in Fig. 2, an execution where successive cycles do not overlap in time is clearly sub-optimal. Our objective is to allow the pipelined execution of Fig. 3, which ensures a maximal use of the computing resources. In the

	Pipelined scheduling table		
time	P1	P2	P3
0	A@true $fst(A) = 0$	B@true $fst(B) = 1$	C@true $fst(C) = 2$

Figure 4: Pipelined scheduling table

pipelined execution, a new instance of operation A starts as soon as the previous one has completed, and the same is true for B and C. The first two time units of the execution are the *prologue* which fills the pipeline. In the *steady state* the pipeline is full and has a throughput of one computation cycle (of the non-pipelined system) per time unit. If the system is allowed to terminate, then completion is realized by the *epilogue*, not pictured in our example, which empties the pipeline.

We represent this pipelined implementation using the *pipelined scheduling table* pictured in Fig. 4. Its length is 1, corresponding to the throughput of the pipelined system. The operation set contains the same operations A, B, and C, but there are significant changes. The start dates of B and C are now 0, as the 3 operations are started at the same time in each *pipelined cycle*. To avoid confusion, we reserve the name *computation cycle* for full computations, as specified by the initial scheduling table. A computation cycle spans over several pipelined cycles, but each pipelined cycle starts exactly one computation cycle.

To account for the prologue and epilogue phases, where operations progressively start to execute, each operation is assigned a *start index* $fst(o)$. If an operation o has $fst(o) = n$ it will first be executed in the pipelined cycle of index n (indices start at 0). Due to pipelining, the instance of o executed in the pipelined cycle m belongs to the computation cycle of index $m - fst(o)$. For instance, operation C with $fst(C) = 2$ is first executed in the 3rd repetition of the table (of index 2), but belongs to the first computation cycle.

4.1 Code generation issues

Given that our pipelining approach does not change scheduling decisions inside computation cycles, the transformation of Fig. 2 into Fig. 4 only depends on the throughput of the pipelined system, which is determined by the analysis of Section 5. 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 $max\_par = \lceil len(\mathcal{S})/len(\overline{\mathcal{S}}) \rceil$, where $\mathcal{S}$ is the non-pipelined scheduling table, and $\overline{\mathcal{S}}$ is its pipelined version. Note that max_par can also be computed as $1 + \max_{o \in \mathcal{O}} fst(o)$.

In the initial scheduling table of our example, both A and B use memory cell v_1. In the pipelined table A and B work in parallel, so they must use two different copies of v_1. We say that the replication factor of v_1 is $rep(v_1) = 2$. 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

$$\begin{aligned} fst(v) &= \min_{v \in In(o) \cup Out(o)} fst(o) \\ lst(v) &= \max_{v \in In(o) \cup Out(o)} fst(o) \end{aligned}$$

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 $\overline{v}[0], \ldots, \overline{v}[rep(v)-1]$. 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 $\overline{v}[n \bmod rep(v)]$ for all v. The computation of $rep(v)$ ensures that if n_1 and n_2 are equal modulo $rep(v)$, but $n_1 \neq n_2$, then computation cycles n_1 and n_2 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 $\overline{v}[(cid - fst(o)) \bmod rep(v)]$ 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:

$$cid := (cid + 1) \bmod R$$

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,[3] 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 $\overline{v}$. 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
  $src(v)[(cid - fst(v)) \bmod rep(v)]$ the value of
  $src(v)[(cid - fst(v) - 1) \bmod rep(v)]$. 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 $\overline{v}[src(v)[(cid - fst(v) - 1) \bmod rep(v)]]$.

[3] This is necessary to represent systems having an internal state.

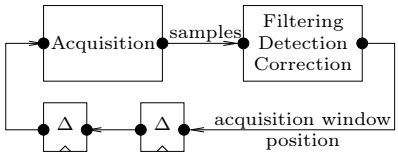

Figure 5: High-level representation of the knock control function

- When an operation o of the non-pipelined scheduling table reads v, its counterpart in the pipelined table will read $\overline{v}[src(v)[(cid - fst(o)) \bmod rep(v)]]$. 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).
  - If not, then o writes the memory cell normally assigned to this computation cycle by the replication process ($\overline{v}[x]$, where $x = (cid - fst(o)) \bmod rep(v)$). 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, v_1 requires no initialization, so that none of its replicas do. In the general case, if $Init(v) \neq nil$, we need to initialize $\overline{v}[0]$ with $Init(v)$, but not the other replicas.

4.2 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.

We provide in Fig. 5 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 Δ). Acquisition is performed by a specialized device (AD) of the ECU, whereas the FDC function is computed by the ECU microcontroller (μC).

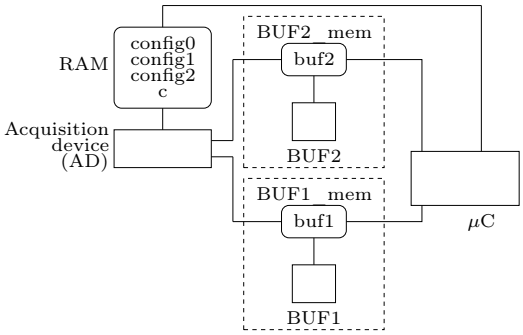

Figure 6: Engine control unit (ECU) architecture

rotation units	AD	BUF1	BUF2	μC
0	book@true			
1	Acq_1 @c / Acq_2 @$\neg c$	Acq_1 @c	Acq_2 @$\neg c$	
2				
3		FDC_1 @c	FDC_2 @$\neg c$	FDC_1 @c / FDC_2 @$\neg c$
4				
5				

Figure 7: 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. 6, 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. 7. 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 "$c{:=}\neg c$". In cycles where the new value is *true*, buffer $buf1$ is used. Otherwise, $buf2$ is used. If we denote with c^n 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

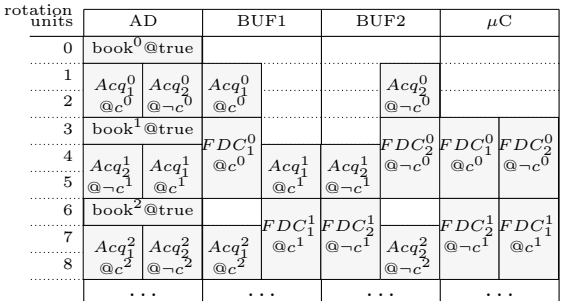

Figure 8: Pipelined execution of the knock control

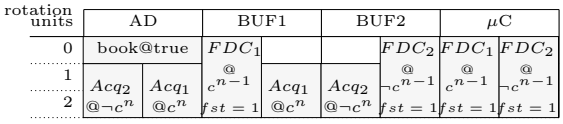

Figure 9: Pipelined scheduling table for the knock control

operations, we split the acquisition and FDC operations in two. Both Acq_1 and Acq_2 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 $\neg c$. Each of the Acq_i and FDC_i 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. 8. 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. 7, we represent here both activation scenarios, corresponding to different initial values of c.

The corresponding pipelined scheduling table is provided in Fig. 9. 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 4.1 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.

	Non-pipelined scheduling table		
time	P1	P2	P3
0	A@true		
1		B@true	
2			C@true
3	D@true		

Figure 10: Dependency analysis example

	Pipelined scheduling table		
time	P1	P2	P3
0	A@true $fst(A) = 0$		C@true $fst(C) = 1$
1	D@true $fst(D) = 1$	B@true $fst(B) = 0$	

Figure 11: Dependency analysis example, pipelined

5 Pipelining algorithms

In this section, we provide the actual pipelining algorithms that determine if and when a new computation cycle can be started while another one is still active. The core of this computation is a dependency analysis which determines when an operation o of cycle $n+1$ cannot be executed before operation o' of cycle n. These dependencies are then used by pipelining algorithms to determine the new system throughput and to construct the pipelined scheduling table.

5.1 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 non-pipelined scheduling table of Fig. 10. Resource P_1 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. 11. 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).

The function works as follows: It concatenates without pipelining two instances of the input scheduling table $\mathcal{S}$. On the resulting scheduling table, it determines the dependencies between all operations. The resulting dependency set is formed of pairs (o', o) where o' must be completed before the execution of o. From this set are then deleted the pairs where both operations belong to the same instance of $\mathcal{S}$. Then, the result of the function is computed as the minimum for all remaining pairs (o', o) of the distance between o' and o $(t(o) - t(o') - d(o'))$.

The *Concat2Copies* function produces $\overline{\mathcal{S}} =< 2 * n, \overline{O}, nil >$ where $\overline{O} = \{o_1, o_2 \mid o \in O\}$, o_1 has the same fields as o, and o_2 differs from o only in its start date, which is $t(o) + n$. Note that we lose all initialization information in the process, meaning that our analysis does not exploit knowledge on the initial state. The *BuildEventList* function takes a scheduling table and produces the list of all operation start and end events, ordered by increasing date. More precisely, the list contains $start(o)$ and $end(o)$ for all operation o of the scheduling table. The list is ordered by increasing event date using the convention that the date of $start(o)$ is $t(o)$, and the date of $end(o)$ is $t(o) + d(o)$. Moreover, if $start(o)$ and $end(o')$ have the same date, the $start(o)$ event comes first in the list.

During Phase 1 scheduling table $\overline{\mathcal{S}}$ 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 v_o and v_{init}.

The remainder of the transformation is performed by a symbolic execution of $\overline{\mathcal{S}}$, 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 v_o is a

Function 1 DependencyAnalysis

Input: $\mathcal{S}$: non-pipelined schedule
1: $fast\_pipelining\_flag$: boolean
Output: min_dist : integer
2: /* Phase 0: Preliminaries */
3: $\overline{\mathcal{S}} := Concat2Copies(\mathcal{S})$
4: $l := BuildEventList(\overline{\mathcal{S}})$
5: /* Phase 1: Compute the dependencies */
6: **for all** o operation in $\overline{\mathcal{S}}$ **do**
7: $Out(o) := \{v_o \mid v \in Out(o)\}$
8: **for all** $v \in Cells$ **do**
9: $curr(v) := \{v_{init}@true\}$
10: $Depend := \emptyset$
11: **while** l not empty **do**
12: $e := \text{head}(l)$; $l := \text{tail}(l)$
13: **if** $k = start(o)$ **then**
14: Assume $Guard(o) = g_o(v_1, \ldots, v_k)$. Replace it by:

$$\bigvee_{v'_i@C_i \in curr(v_i), i=\overline{1,k}} (C_1 \wedge \ldots \wedge C_k) \wedge g_o(v'_1, \ldots, v'_k)$$

15: **for all** $o' \in Completed$, $v \in In(o)$, $v' \in Out(o')$ **do**
16: /* Classical data dependencies */
17: **if** $v'@C \in curr(v)$ and $C \wedge Guard(o) \neq false$ **then**
18: $Depend := Depend \cup \{(o', o)\}$
19: /* Dependencies of the second type */
20: **if** $fast\_pipelining\_flag$ **then**
21: **if** $Res(o) \cap Res(o') \neq \emptyset$ **then**
22: **if** $Guard(o) \wedge Guard(o') \neq false$ **then**
23: $Depend := Depend \cup \{(o', o)\}$
24: **else**
25: /* $e = end(o)$ */
26: $Completed := Completed \cup \{o\}$
27: **for all** $v_o \in Out(o)$ **do**
28: $new\_curr(v) := v_o@Guard(o)$
29: **for all** $v_{o'}@C' \in curr(v)$ **do**
30: $C'' := C' \wedge \neg Guard(o)$
31: **if** $C'' \neq false$ **then**
32: $new\_curr(v) := new\_curr(v) \cup \{v_{o'}@C''\}$
33: $curr(v) := new\_curr(v)$
34: /* Phase 2: Compute the minimal distance */
35: $Depend := \{(o', o) \in Depend \mid t(o) > len(\mathcal{S}) \geq t(o')\}$
36: $min\_dist := \min_{(o',o) \in Depend}(t(o) - t(o') - d(o'))$
37: **return** min_dist

version of v and C is the condition on which the value of v is that corresponding to v_o. 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 25-33 of Function 1).

Dependencies are identified in lines 17-18 (for classical data dependencies) and 20-23 (for the dependencies of the second type). The code involves in both cases predicate comparisons. A third predicate comparison is used in line 31. 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 $c^n = \neg c^{n-1}$.

5.2 Pipelining routines

The pipelining functions are simple drivers using the results of the dependency analysis. Function 2 takes a non-pipelined scheduling table and a new scheduling length and builds a pipelined scheduling. It starts by determining 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 2 BuildSchedule

Input: $\mathcal{S}$: non-pipelined scheduling table
new_length : integer
Output: $\overline{\mathcal{S}}$: pipelined schedule table
/* Compute operation dates and start indices */
for all o in $\mathcal{O}$ **do**
$fst(o) := [\frac{t(o)}{new\_length}]$
$t'(o) := t(o) - fst(o) * new\_length$
/* Assemble the pipelined schedule */
$\overline{\mathcal{S}} := AssembleSchedule(\mathcal{S}, new\_length, fst, t')$
return $\overline{\mathcal{S}}$

Function 3 is the top-level pipelining driver. The function starts by performing the data analysis. 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 min_dist and the non-pipelined length. For each length i, the pipelined scheduling is assembled, but the result may not respect the well-formed properties defined in Section 3.3. We do not provide here the code of function *ConsistentSchedule*, which closely follows the definition of these properties.

6 Experimental results

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

Function 3 PipeliningDriver

Input: $\mathcal{S}$: non-pipelined schedule table
fast_pipelining_flag : boolean
Output: $\overline{\mathcal{S}}$: pipelined schedule table
min_dist := *DependencyAnalysis*($\mathcal{S}$, *fast_pipelining_flag*)
if *fast_pipelining_flag* **then**
new_length := *min_dist*
$\overline{\mathcal{S}}$:= *BuildSchedule*($\mathcal{S}$, *new_length*)
return $\overline{\mathcal{S}}$
else
for *i* := *min_dist* to *len*($\mathcal{S}$) **do**
new_length := *i*
$\overline{\mathcal{S}}$:= *BuildSchedule*($\mathcal{S}$, *new_length*)
if *ConsistentSchedule*($\overline{\mathcal{S}}$) **then**
return $\overline{\mathcal{S}}$

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

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 [23] 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.

Acknowledgement. The authors wish to thank Albert Cohen for having introduced them to the field of software pipelining.

References

[1] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 4^{th} 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] N.J. Warter, D. M. Lavery, and W.W. Hwu. The benefit of predicated execution for software pipelining. In *HICSS-26 Conference Proceedings*, pages 497–506, 1993.

[15] D. Milicev and Z. Jovanovic. A formal model of software pipelining loops with conditions. In *Proceedings of the 11th International Symposium on Parallel Processing*, IPPS '97, pages 554–558, Washington, DC, USA, 1997. IEEE Computer Society.

[16] N. J. Warter, G. E. Haab, K. Subramanian, and J. W. Bockhaus. Enhanced modulo scheduling for loops with conditional branches. In IEEE Computer Society Press, editor, *MICRO 25 Proceedings of the 25th annual international symposium on Microarchitecture*, pages 170–179, 1992.

[17] H.-S. Yun, J. Kim, and S.-M. Moon. Time optimal software pipelining of loops with control flows. *International Journal of Parallel Programming*, 31(5):339–391, October 2003.

[18] C. Leiserson and J. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.

[19] L. Morel. *Exploitation des structures régulières et des spécifications locales pour le developpement correct de systèmes réactifs de grande taille*. PhD thesis, Institut National Polytechnique de Grenoble, 2005.

[20] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

[21] D. Potop-Butucaru, R. De Simone, Y. Sorel, and J.-P. Talpin. Clock-driven distributed real-time implementation of endochronous synchronous programs. In ACM, editor, *EMSOFT '09 Proceedings of the seventh ACM international conference on Embedded software*, pages 147–156, 2009.

[22] C. Pradalier, J. Hermosillo, C. Koike, C. Braillon, P. Bessière, and C. Laugier. The CyCab: a car-like robot navigating autonomously and safely among pedestrians. *Robotics and Autonomous Systems*, 50(1), 2005.

[23] 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.

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399