Cyclic Task Scheduling with Storage Requirement Minimization under Specific Architectural Constraints: Case of Buffers and Rotating Storage Facilities

Sid Touati

UNIVERSITE DE VERSAILLES SAINT-QUENTIN EN YVELINES

Cyclic Task Scheduling with Storage Requirement Minimisation under Specific Architectural Constraints: Case of Buffers and Rotating Storage Facilities

Sid-Ahmed-Ali TOUATI

N° HAL-INRIA-00440446

December 2009

Rapport de recherche

Cyclic Task Scheduling with Storage Requirement Minimisation under Specific Architectural Constraints: Case of Buffers and Rotating Storage Facilities

Sid-Ahmed-Ali TOUATI*

Thème : Optimisation de programmes
Équipe-Projet ARPA - Laboratoire PRiSM

Rapport de recherche n° HAL-INRIA-00440446 — December 2009 — 27 pages

Abstract: In this report, we study the exact and an approximate formulation of the general problem of one-dimensional periodic task scheduling under storage requirement, irrespective of machine constraints. We rely on the SIRA theoretical framework that allows an optimisation of periodic storage requirement [18]. SIRA is based on inserting some storage dependence arcs (*storage reuse* arcs) labeled with *reuse distances* directly on the data dependence graph. In this new graph, we are able to bound the storage requirement measured as the exact number of necessary storage locations. The determination of storage and distance reuse is parametrised by the desired minimal scheduling period (respectively maximal execution throughput) as well as by the storage requirement constraints - either can be minimised while the other one is bounded, or alternatively, both are bounded [6, 14]. This report recalls our fundamental results on this problem, and proposes new experimental heuristics. We typically show how we can deal with some specific storage architectural constraints such as buffers and rotating storage facilities.

Key-words: Task scheduling, Storage requirement, Periodic scheduling, Task parallelism

* Sid.Touati@uvsq.fr

UFR des sciences
45 avenue des Etats Unis,
78035 Versailles cedex
Téléphone : +33 1 39 25 41 12 — Télécopie : +33 1 39 25 46 04

Ordonnancement périodique de tâches avec minimisation du besoin de stockage sous contraintes architecturales spécifiques: cas des tampons et des mécanismes de stockage rotatifs

Résumé : Dans ce rapport, nous étudions la formulation exacte et simplifiée du problème d'ordonnancement périodique uni-dimensionnel de tâches, avec minimisation du besoin de stockage, sans considération des contraintes de ressources. Nous nous appuyons sur la plateforme théorique appelée SIRA [18]. SIRA se base sur l'insertion d'arcs valués dans le graphes de tâches afin de controller le besoin périodique en stockage. Dans le graphe étendu, nous pouvons borner le besoin en stockage en fonction de la période souhaitée pour l'ordonnancement [6, 14]. Ce rapport rappelle nos résultats et apporte de nouvelles heuristiques expérimentales sur le problème d'ordonnancement sous contraintes de stockage. Typiquement, nous étudions les résultats dans des cas particuliers d'architectures de stockage avec des tampons (files d'attente) ou avec des mécanismes de stockage rotatifs.

Mots-clés : Ordonnancement de tâches, besoin en stockage, ordonnancement périodique, parallélisme de tâches

Contents

1 Introduction

This research report addresses the problem of storage minimisation in cyclic data dependence graphs (DDGs), which is for instance applied to the practical problem of periodic register allocation for innermost loops on Instruction Level Parallelism (ILP) processors[18]. The massive introduction of ILP processors since the last decades makes us re-think new ways of optimising register/storage requirement in assembly codes before starting the instruction scheduling process under resource constraints. In such processors, instructions are executed in parallel thanks to the existence of multiple small computation units (adders, multipliers, load-store units, etc.). The exploitation of this fine grain parallelism (at the assembly code level) asks to revisit the old classical problem of register allocation initially designed for sequential processors. Register allocation has not only to minimise the storage requirement, but has also to take care of parallelism and total schedule time. In this research report, we do not assume any resource constraints (except storage requirement); Our aim is to study the trade-off between memory (register pressure) and parallelism in a periodic task scheduling problem. Note that this problem is abstract enough to be considered in other scheduling disciplines that worry about conjoint storage and time optimisation in repetitive tasks (manufacturing, transport, networking, etc.).

Our theoretical framework treats three general cases for storage mapping:

1. In a *classical* storage mechanism, each storage unit is independent and can be addressed using a distinct label. For instance, in a processor, each storage unit has a number (label) used to uniquely address/access it (this is the case of processor registers). By default, this report assumes a classical storage mechanism.

2. In a *rotating* storage mechanism, all storage units are grouped inside a *storehouse*. Such storehouse cyclically permutes storage units at each scheduling period. Consequently, a label used to access a generic

(logical) storage unit would address distinct physical storage units from one period to another. Some processors have such rotating storage mechanism, namely rotating register files. A detailed definition of this storage mechanism and a special solution for it have already been provided in [18].

3. With the presence of *buffers*, each task has its own pool of storage locations. No storage sharing is possible between tasks.

Usual task scheduling problems deal with precedence constraints having non-negative latencies. This seems a natural way for modelling scheduling problems, since tasks delays are generally non-negative quantities. However in some cases, we need to consider arcs latencies that do not only model task latencies, but also model other precedence constraints [1, 9]. For instance in storage optimisation problems, our generic machine model allows the consideration of some access delays into/from storage locations. In this case, arc latencies may be non-positive leading to difficult scheduling constraints in case of limited resources. We show that the usage of non-positive arc latencies, considered to optimally minimise storage requirement, may lead further to infeasible scheduling problems when considering resources constraints. We have a solution for this situation using circuits retiming [10] based on integer linear programming, that we experiment massively.

This report is in continuation on our previous work on register allocation [6, 14]. In that research result, we showed how to minimise the storage requirement without any constraint on the shape of the reuse graph. The current report shows how can we deal with situation where the shape of the reuse graph is restricted (for instance, the reuse graph must be hamiltonian, or must be reflexive, etc.). This is done by first fixing reuse arcs to fix the shape of the reuse graph before minimising the reuse distances.

Our report is organised as follows. Section 2 explains the formal problem and the exact definition of periodic storage requirement. Section 3 recalls our concept of reuse graphs that we use for storage optimisation [18]. The problem of optimal periodic scheduling under storage constraints is described with integer linear programming in Section 4: this problem is similar to the register optimisation problem studied in [18, 6, 14], except that register allocation requires an additional treatment (loop unrolling) not necessary in the general case. Section 5 presents simplified sub-problems and polynomial heuristics. Section 6 provides additional information on our heuristics. Section 7 studies the problem of non-positive circuits generated by the use of non-positive latencies. Before concluding, Section 8 presents the results of our experimental evaluation of the heuristics.

2 Tasks Model

We consider a set of l generic tasks (instructions inside a program loop) $T_0, \ldots, T_{l-1}$. Each task T_i should be executed n times, where n is the number of loop iterations. n is an unknown, unbounded, but finite integer. This means that each task T_i has n instances. The k^{th} occurrence of task T_i is noted $T\langle i, k\rangle$, which corresponds to task executed at the k^{th} iteration of the loop, with $0 \leq k < n$.

The tasks (instructions) may be executed in parallel. Each task may produce a result that is read/consumed by other tasks. The considered loop contains some data dependences represented with a graph $G = (V, E, \delta, \lambda)$ such that:

- V is the set of the generic tasks of the loop body, $V = \{T_0, \ldots, T_{l-1}\}$.
- E is the set of arcs representing precedence constraints (flow dependences or other serialisation constraints). Any arc $e = (T_i, T_j) \in E$ has a latency $\delta(e) \in \mathbb{Z}$ in terms of time steps (processor clock tics for instance) and a distance $\lambda(e) \in \mathbb{Z}$ in terms of number of loop iterations. The distance $\lambda(e)$ means that the arc $e = (T_i, T_j)$ is a dependence between the task $T\langle i, k\rangle$ and $T\langle j, k + \lambda(e)\rangle$ for any $k = 0, \ldots, n - 1 - \lambda(e)$.

In the case of a data dependence graph between loop instructions, it is constructed by the compiler, and we have generally $\delta(e) \in \mathbb{N}$ and $\lambda(e) \in \mathbb{N}$. That is, all the arcs are valued with non-negative numbers (because the dependences are constructed from a sequential program). In a general periodic task scheduling problem, we do not need to restrict ourselves, and we can consider that $\delta(e) \in \mathbb{Z}$ and $\lambda(e) \in \mathbb{Z}$ as studied in [1, 9].

In our model, we make a difference between tasks and precedence constraints depending whether they refer to data to be stored or not:

1. V^R is the set of tasks producing data to be stored.

2. E^R is the set of flow dependence arcs through storage units. An arc $e = (T_i, T_j) \in E^R$ means that the task $T\langle i, k\rangle$ produces a result stored and read/consumed by $T\langle j, k + \lambda(e)\rangle$. The set of consumers (readers) of a generic task T_i is then the set:

$$Cons(T_i) = \{T_j \in V \mid e = (T_i, T_j) \in E^R\}$$

Example 1 *Figure 1 is an example of a data dependence graph (DDG) where bold circles represent V^R the set of generic tasks producing data to be stored. Bold arcs represent flow dependences (each sink of such arc reads/consumes the data produced by the source). Tasks that are not in bold circles are tasks/instructions that do not require our storage facilities (they may require other non considered storage, or simply they do not produce any data). Non-bold arcs are other data or precedence constraints different from flow dependences. Every arc e in the DDG is labelled by the pair $(\delta(e), \lambda(e))$.*

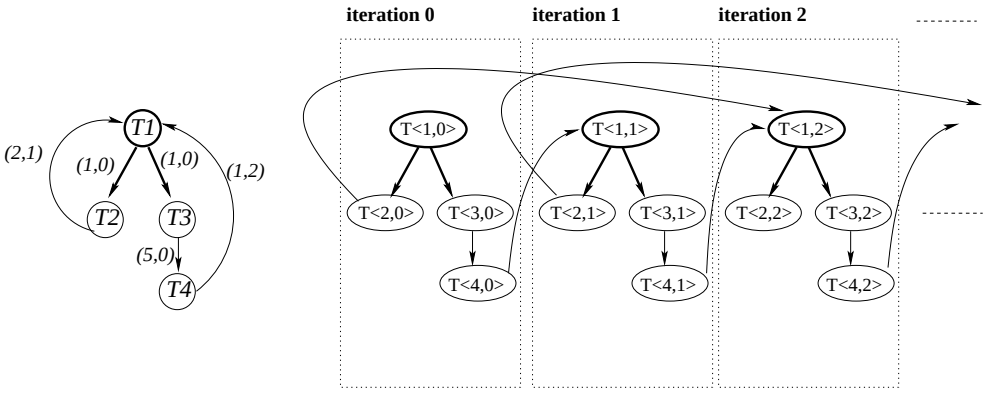


Figure 1: Example of Data Dependence Graphs with Generic Tasks

In our generic processor model, we assume that the reading and writing from/into storage units may be delayed from the starting time of task execution. Let assume $\sigma(T\langle i, k\rangle) \in \mathbb{N}$ as the starting execution time of task $T\langle i, k\rangle$. We thus define two delay functions δ_r and δ_w in which

$$\begin{array}{ll} \delta_w : & V^R \rightarrow \mathbb{N} \\ & T_i \mapsto \delta_w(T_i) |\ 0 \leq \delta_w(T_i) \\ & \text{the writing time of data produced by } T\langle i, k\rangle \text{ is } \sigma(T\langle i, k\rangle) + \delta_w(T_i) \\ \delta_r : & V \rightarrow \mathbb{N} \\ & T_i \mapsto \delta_r(T_i) |\ 0 \leq \delta_r(T_i) \\ & \text{the reading time of the data consumed by } T\langle i, k\rangle \text{ is } \sigma(T\langle i, k\rangle) + \delta_r(T_i) \end{array}$$

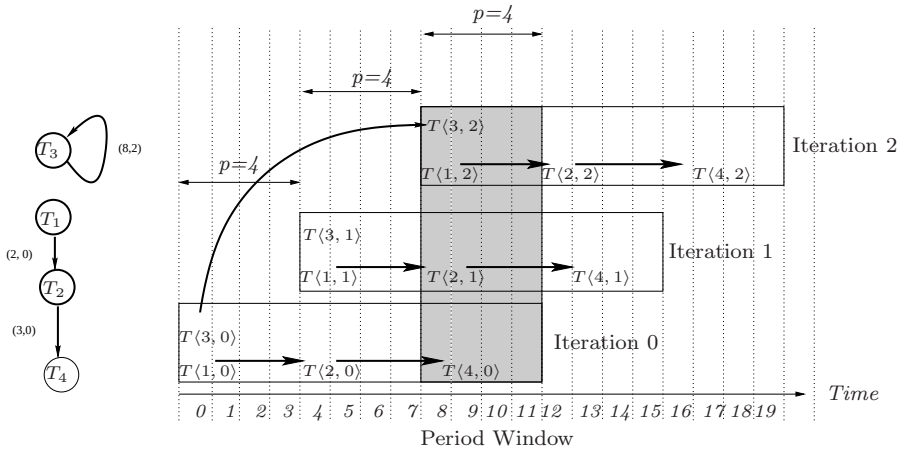
These two delays functions depend on the target processor and model almost all regular hardware architectures (VLIW, EPIC/IA64 and superscalar processors). These delay function have not necessary a relationship with the arc latency δ. The next section recalls the definition of periodic task scheduling problem in case of one dimensional schedule.

2.1 The Periodic Scheduling Problem

Instruction or task scheduling in our case is the process of assigning an integral execution date to each task occurrence. A schedule is considered as an integral function noted σ which must at least satisfy the precedence constraints defined by the DDG $G = (V, E, \delta, \lambda)$:

$$\forall e = (T_i, T_j) \in E, \forall k \in [0, n - 1 - \lambda(e)] :\ \sigma\big(T\langle i, k\rangle\big) + \delta(e) \leq \sigma\big(T\langle j, k + \lambda(e)\rangle\big) \quad (1)$$

However, since n the number of task occurrences is unknown and unbounded, we should not consider any shape of scheduling functions, even if they meet the constraints defined above. We should only look for *periodic*



(a) Another DDG example (b) Example of Periodic Schedule

Figure 2: Example of a Periodic Schedule: $\sigma_0 = 0, \sigma_2 = 4, \sigma_3 = 0, \sigma_4 = 8$

schedules since our aim is to generate a final compact code (a loop). A periodic scheduling function σ is associated with a unique integral period p (to be computed). The scheduling period p is integral and common to all generic tasks because it simplifies the code generation of the final loop. Other multi-dimensional periodic scheduling functions may be employed (with multiple periods, or with rational periods), but with the expense of huge code size. In our scope, a periodic scheduling function with a unique period p assigns to each generic task T_i an integral execution date for only the first task occurrence $T\langle i, 0\rangle$ that we note $\sigma_i = \sigma(T\langle i, 0\rangle)$. The execution date of any other occurrence $T\langle i, k\rangle$ becomes equal to $\sigma(T\langle i, k\rangle) = \sigma_i + k \times p$. By reporting this definition into Equation 1, we get new periodic scheduling constraints that should be satisfied by σ:

$$\forall e = (T_i, T_j) \in E : \sigma_i + \delta(e) \leq \sigma_j + \lambda(e) \times p \tag{2}$$

Classically, by adding all such inequalities over any circuit C of the DDG G we find that p must be greater than or equal to $\max_C \left\lceil \frac{\sum_{e\in C} \delta(e)}{\sum_{e\in C} \lambda(e)} \right\rceil$, that we will denote in the sequel as the absolute Minimal Execution Period MEP. The circuit that defines MEP is called *the critical circuit* - non necessarily unique. Computing MEP of a cyclic graph is a well known polynomial problem [4, 8]. The usual problem of periodic instruction scheduling looks for a schedule with a minimal period which satisfies additional constraints (resources, bounded storage requirement, etc.). In this report, we study the problem of periodic scheduling under data dependence and storage constraints.

Example 2 *Figure 2 illustrates an example of a periodic schedule. Figure 2(a) is a new DDG example. Bold lines represent flow dependences and bold circles represent tasks producing a result to be stored. Figure 2(b) illustrates a schedule σ with a period $p = 4$. As can be seen, all DDG arcs are satisfied in this schedule. For instance, the arc $e = (T_1, T_2)$ with $\delta(e) = 2$ and $\lambda(e) = 0$ is satisfied because $\forall i, \sigma(T\langle 1, i\rangle) + 2 \leq \sigma(T\langle 2, i\rangle)$. Also, the arc $e' = (T_3, T_3)$ with $\delta(e') = 8$ and $\lambda(e') = 2$ is satisfied because $\forall i, \sigma(T\langle 3, i\rangle) + 8 \leq \sigma(T\langle 3, i + 2\rangle)$, etc. Note that the period $p = 4$ is minimal because the critical circuit in our example is (T_3, T_3) resulting in $MEP = \lceil \frac{8}{2} \rceil = 4$.*

2.2 Storage Requirement

When considering a periodic schedule σ with an integral period p, any generic task $T_i \in V^R$ corresponds to n task occurrences, each one producing a data at time $\sigma(T\langle i, k\rangle) + \delta_w(T_i)$, with $k = 0, \ldots, n-1$. Such data should be stored inside a register until its last reading. Each flow dependence $e = (T_i, T_j) \in E^R$ means that the task occurrence $T\langle j, k + \lambda(e)\rangle$ reads the data produced by $T\langle i, k\rangle$ at time $\sigma_j + \delta_r(T_j) + (\lambda(e) + k) \times p$. The last reading time of a data produced by $T\langle i, k\rangle$ is called the *death* date and is equal to:

$$d_\sigma(T\langle i, k\rangle) = \max_{e=(T_i,T_j)\in E^R} \big(\sigma_j + \delta_r(T_j) + (\lambda(e) + k) \times p\big) \tag{3}$$

Every consumer that reads a data at its death time is called a *killer* of the data. Then, every data produced by $\sigma(T\langle i, k\rangle)$ is alive during a contiguous interval between the production date and the death date. It is called *lifetime*

interval and is equal to:

$$LT_\sigma(T\langle i,k\rangle) =]\sigma(T\langle i,k\rangle) + \delta_w(T_i), d_\sigma(T\langle i,k\rangle)]$$

As can be seen, lifetime intervals are considered as left open, because in our model the data produced at time t is considered alive one step later at time $t+1$. Note that this is not a limitation of the model, but it is a choice that does not alter any of our formal results. During the lifetime interval, the considered data is said *alive* and should reside inside a storage location (register) during the whole interval. The register assigned to store this data is not free to store another data during this lifetime interval. Let $alive^t_\sigma$ be the set of alive data at time $t \in \mathbb{N}$ when considering a schedule σ

$$alive^t_\sigma = \left\{T\langle i,k\rangle | T_i \in V^R, 0 \leq i < l,\ 0 \leq k < n,\ t \in LT_\sigma(T\langle i,k\rangle)\right\}$$

The storage requirement of the DDG G when considering the periodic schedule σ is equal to the maximal number of simultaneously alive data at any time. Formally, it is equal to:

$$SR_\sigma(G) = \max_{t\in\mathbb{N}} \left|\{alive^t_\sigma\}\right| \tag{4}$$

Example 3 *Figure 3(a) illustrates the lifetime intervals of the tasks previously shown in Figure 2. For instance, the task T_3 produces many lifetime intervals, one for each task occurrence. $LT_\sigma(T\langle 3,0\rangle) =]1,8]$, $LT_\sigma(T\langle 3,1\rangle) =]5,12]$, etc.*

Another remark concerns the delay between the schedule date and the lifetime intervals. For instance, the task occurrence $T\langle 3,0\rangle$ is schedule at date $t=0$ but its result is alive two steps later. This is because $\delta_w(T_3) = 2$. Consequently, the lifetime interval of $T\langle 3,0\rangle$ starts at time 2 instead of time 0.

Now, the storage requirement of our example is the maximal number of data simultaneously alive. It is equal to $SR_\sigma(G) = 4$, as can be calculated during the dates $8, 11, \cdots$. Since our schedule is periodic, it is sufficient to calculate the periodic storage requirement inside a period window, as shown in Figure 3(b).

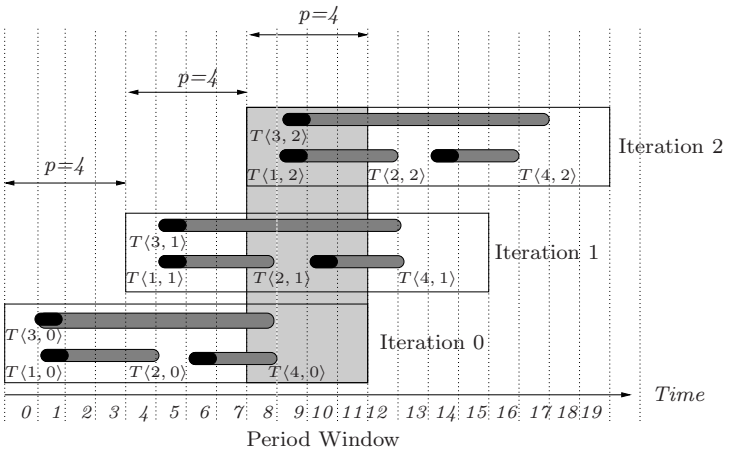
When considering unbounded resources and unbounded storage/register facilities, we can easily compute an optimal periodic schedule, *i.e.*, with a minimal scheduling period $p = MEP$ [4]. We consider now R a limited storage capacity (a finite number of available registers) and all the schedules that have a maximum of R simultaneously alive variables. Any register/storage allocation will induce new storage dependencies in the DDG; Hence register pressure has influence on the expected p even if we assume unbounded resources. What we want to analyse here is the minimum p that can be expected for any periodic schedule using at most R storage units. We will denote this value as MEP_R and we will try to understand the relationship between MEP_R and R. Let us recall the reuse graph concept that model storage allocation in a set of generic tasks.

3 Recall on Reuse Graphs

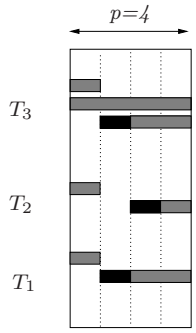
We give in this section more intuitions about the new arcs that we add between two tasks in order to restrict the parallelism and hence restrict the whole storage requirement. These arcs represent possible register reuse between tasks. This can be viewed as a variant of general storage mapping functions [19, 20].

Let us consider the DDG of Figure 4(a). This DDG has two generic tasks writing into storage locations, T_0 and T_3 respectively. The consumers of these tasks are $\{T_1, T_2\}$ for T_0 and $\{T_4, T_5\}$ for T_3. The DDG does not contain any circuit. So theoretically, if there are unbounded resources, the task parallelism is unbounded too (this is because there is no cyclic precedence constraints restricting the parallelism). Now, it there is an unbounded parallelism, then the number of storage units required to keep all results is unbounded too: all iterations of this DDG can be run in parallel and each distinct task instance requires a storage unit. If we want to bound the storage requirement by allowing storage sharing between tasks, we should restrict the task parallelism. This means that we should add new arcs into the DDG in order to serialise the lifetime intervals of some tasks. For instance, if we want to make task T_j to reuse the storage unit freed by task T_i, we should serialise the lifetime interval of $T\langle j,k\rangle$ after the lifetime interval of $T\langle i,k\rangle$. That is, we should guarantee that $LT_\sigma(T\langle i,k\rangle)$ finishes before the starting of $LT_\sigma(T\langle j,k\rangle)$ for any schedule σ. This is equivalent to guarantee $d_\sigma(T\langle i,k\rangle) \leq \sigma(T\langle j,k\rangle) + \delta_w(T_j), \forall\sigma$.

The above inequality can be guaranteed by simply inserting an arc e from the killer of $T\langle i,k\rangle$ to $T\langle j,k\rangle$ with a latency $\delta(e) = -\delta_w(T_j)$. The problem here is that the killer of a task is not defined before fixing a schedule. When a task produces a data that is read by multiple consumers, and when the periodic schedule has not been fixed yet, we cannot know which consumer would be a killer of the data and hence we cannot know in advance when



(a) Periodic Lifetime Intervals



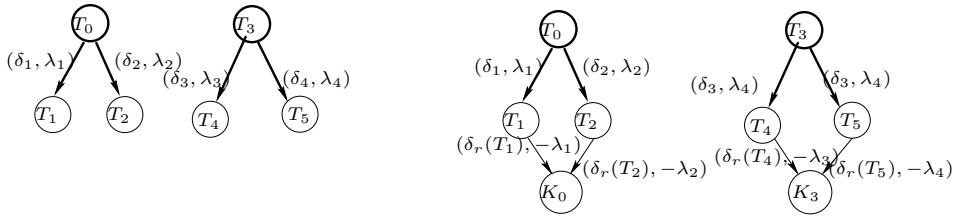
■ indicates the production of the task result

(b) Lifetime Intervals inside the Period Window

Figure 3: Storage Requirement in Periodic Task Schedules: $\delta_r = 0$ for all tasks, $\delta_w(T_1) = 2$, $\delta_w(T_2) = 3$, $\delta_w(T_3) = 2$

a storage unit could be freed. We use a trick which defines for each task T_i a fictitious killing task K_i. We insert an arc from each consumer $T_j \in Cons(T_i)$ to K_i to reflect the fact that this killing task is scheduled after all the consumers of T_i (see Figure 4(b)). The latency of this arc is set to $\delta_r(T_j)$ because of the reading delay. We set its distance to $-\lambda$, where λ is the distance of the flow dependence between T_i and its consumer T_j. This is done to model the fact that the virtual task occurrence $K\langle i, k+\lambda-\lambda\rangle$, *i.e.* $K\langle i, k\rangle$, is scheduled after the death date of the data produced by $T\langle i, k\rangle$. The occurrence/iteration number k of the killer of $T\langle i, k\rangle$ is only a convention and can be changed by circuit retiming [10] without changing the fundamental mathematical problem.

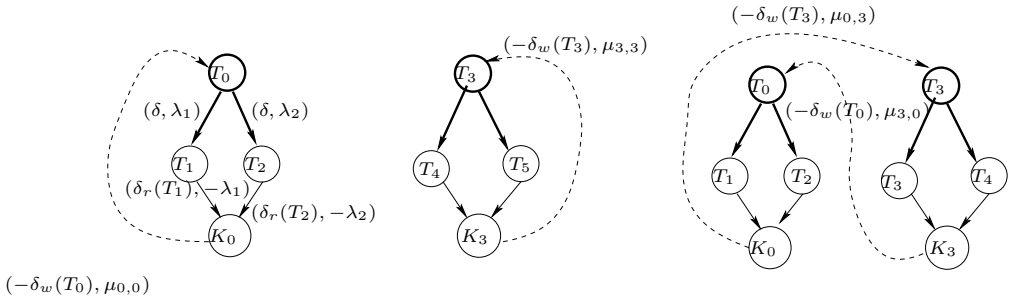
Now, we can serialise the lifetime interval of any tasks at the DDG level. For instance, if we want to guarantee that the task $T\langle 0, k+\mu_{0,0})\rangle$ writes into the same storage unit previously used by $T\langle 0, k\rangle$, we should add a *storage* dependence from $K\langle 0, k\rangle$ to $T\langle 0, k+\mu_{0,0}\rangle$ (for some $\mu_{0,0}) \in \mathbb{N}$, $\mu_{i,}$) is the reuse distance between two generic tasks T_i and T_j). This is equivalent to inserting a storage dependence in the DDG from K_0 to T_0 with a distance equal to $\mu_{0,0}$. Since the writing of T_0 is delayed by $\delta_w(T_0)$, the latency of the introduced storage dependence is set to $-\delta_w(T_0)$. Consequently the DDG becomes cyclic because of storage limitations (see Figure 5(b) where the storage dependences are dashed). The introduced dependences, also called *Universal Occupancy Vector* [19], must in turn be counted when computing MEP_R the new minimum scheduling period resulting from fixing a storage capacity to R available units (registers).



(a) A DDG Example

(b) The DDG with its Killing Tasks

Figure 4: Example of a DDG with Killing Tasks



(a) First Storage Reuse Decision

(b) Another Storage Allocation Scheme

(c) First Reuse Graph

(d) Second Reuse Graph

Figure 5: Killing Tasks and Reuse Graphs

Figure 5(a) presents a first reuse decision where each generic task reuses the register freed by itself. This is illustrated by adding a storage dependence from K_0 (respectively K_3) to T_0 (respectively T_3) with an appropriate distance $\mu_{0,0}$ as we will see later. Another reuse decision (see Figure 5(b)) may be that the generic task T_0 (respectively T_3) reuses the register freed by T_3 (respectively T_0). This is illustrated by adding a storage dependence from K_0 (respectively K_3) to T_3 (respectively T_0). Two schemes do not have the same impact on MEP_R:

intuitively, it is better that the operations share registers instead of using two different pools of registers. For this simple example with two tasks, we have only two choices for reuse decisions. However for a general loop with l tasks producing data, we may have an exponential number of possible reuse decisions.

There are three main constraints that the resulting DDG must meet. First it must be schedulable by periodic scheduling. Second, the storage requirement must be lower than or equal to the storage capacity. Third and last, the critical ratio (MEP_R) must be kept as lower as possible in order to save task parallelism. The next section recalls the formal definition of the problem using the concept of reuse graphs.

Formal Results on Reuse Graphs

The storage reuse relationship between the generic tasks is described by defining a new graph called *a reuse graph*. Figure 5(c) shows the first reuse decision where T_i (respectively T_j) reuses the register used by itself $\mu_{i,i}$ ($\mu_{j,j}$ respectively) iterations earlier. Figure 5(d) is the second reuse choice where T_i (respectively T_j) reuses the register used by T_j (respectively T_i) $\mu_{j,i}$ (respectively $\mu_{i,j}$) iterations earlier. The resulting DDG after adding the killing tasks and the storage dependences to apply the register reuse decisions is called the *DDG associated with a reuse graph*: Figure 5(a) is the associated DDG with Figure 5(c), and Figure 5(b) is the one associated with Figure 5(d). In the next section, we give a formal definition and model of the storage allocation problem based on reuse graphs. We denote by $G^{\rightarrow r}$ the DDG associated with the reuse graph.

A storage allocation consists of choosing which task reuses which released register. We define:

Definition 1 (Reuse Graph) *[15, 18] Let $G = (V, E, \delta, \lambda)$ be a DDG. The reuse graph $G^{reuse} = (V^R, E^{reuse}, \mu)$ is defined by the set of tasks V^R, the set E^{reuse} of arcs representing storage reuse relationship, and arc distances $\mu \in \mathbb{Z}$. Two tasks are connected in G^{reuse} by an arc $e = (T_i, T_j) \in E^{reuse}$ iff $T\langle j, k + \mu_{i,j}\rangle$ reuses the register freed by $T\langle i, k\rangle$.*

We call E^{reuse} the set of reuse arcs and $\mu_{i,j}$ a reuse distance. Given $G^{\text{reuse}} = (V^R, E^{\text{reuse}}, \mu)$ a reuse graph, we report the storage reuse relationship to the DDG $G = (V, E, \delta, \lambda)$ by adding a storage dependence from K_i to T_j iff $e = (T_i, T_j)$ is a reuse arc. The distance of this dependence is $\mu_{i,j}$. The introduction of these extra arcs into G produces the DDG $G^{\rightarrow r}$ associated with the reuse graph G^{reuse}.

A reuse graph must obey some constraints to be valid:

1. The resulting associated DDG $G^{\rightarrow r}$ must be schedulable, *i.e.*, it has at least one periodic schedule;
2. Each task must reuse only one freed register, and each register must be reused by only one task.

Lemma 1 *[15, 18] Let $G^{reuse} = (V^R, E^{reuse}, \mu)$ be a valid reuse graph for a DDG $G = (V, E, \delta, \lambda)$. Then:*

- *The reuse graph consists of only elementary and disjoint circuits;*
- *Any task $T_i \in V^R$ belongs to a unique circuit in the reuse graph.*

First, let us assume a reuse graph for a DDG G. If such reuse graph is valid, we can build a periodic storage allocation for G, as explained in the following theorem. We require $\mu(G^{\text{reuse}})$ registers, in which $\mu(G^{\text{reuse}})$ is the sum of all $\mu_{i,j}$ distances in the reuse graph G^{reuse}.

Theorem 1 *[15, 18] Let $G = (V, E, \delta, \lambda)$ be a DDG and $G^{reuse} = (V^R, E^{reuse}, \mu)$ be a valid reuse graph. Then:*

$$\forall \sigma \text{ a periodic schedule for } G^{\rightarrow r} : \; SR_\sigma(G^{\rightarrow r}) \leq \mu(G^{reuse})$$

From all above, we deduce a formal definition of the problem of optimal periodic storage allocation with maximal execution throughput. We call it Periodic Scheduling with Storage Minimisation (PSSM).

Problem 1 (PSSM) *Let $G = (V, E, \delta, \lambda)$ be a DDG and p a desired scheduling period. Find a valid reuse graph such that the associated DDG $G^{\rightarrow r}$ is schedulable with a period equal to p while $\mu(G^{reuse})$ is minimal.*

This problem is NP-complete [15]. In practice, the problem of register allocation is sightly different. The processor has R a finite number of registers and we should find a time-optimal schedule such that the storage requirement is below the limit R. In this case, the problem can be re-stated as : Find a valid reuse graph such that $\mu(G^{\text{reuse}}) \leq R$ while the associated DDG $G^{\rightarrow r}$ is schedulable with a period equal to $p = MEP_R$. It is straightforward to see that PSSM can be iteratively used to solve the problem of register allocation. However, the problem of minimising storage requirement has other practical benefits such as minimising code size (minimise the loop unrolling degree [3]). This benefit is out the scope of the current contribution.

The following section gives an integer linear formulation for the PSSM problem.

4 Exact Problem Formulation with Integer Linear Programing

In this section, we give an integer linear model for solving PSSM. It is built for a fixed desired period p. Our PSSM exact model uses the linear formulation of the logical implication ($\Longrightarrow$) by introducing binary variables [15].

Basic Variables

- An integral schedule variable $\sigma_i \in \mathbb{Z}$ for each task $T_i \in V$, including σ_{K_i} for each killing node K_i. We assume a finite upper bound L for such schedule variables (L sufficiently large, $L = \Sigma_{e \in E} \delta(e)$);
- A binary variables $\theta_{i,j}$ for each pair of tasks $(T_i, T_j) \in V^R \times V^R$. It is set to 1 iff (T_i, T_j) is a reuse arc;
- $\mu_{i,j} \in \mathbb{Z}$ reuse distances for all pairs of tasks $(T_i, T_j) \in V^R \times V^R$;

Linear Constraints

- Data dependences (see Equation 2):

$$\forall e = (T_i, T_j) \in E : \ \sigma_i - \sigma_j \leq -\delta(e) + p \times \lambda(e)$$

- Killing dates for consumed data:
  $\forall T_i \in V^R, \ \forall T_j \in Cons(T_i) | e = (T_i, T_j) \in E^R$: $\sigma_{K_i} \geq \sigma_j + \delta_r(T_j) + p \times \lambda(e)$
- There is a storage dependence between K_i and T_j if (T_i, T_j) is a reuse arc:

$$\forall (T_i, T_j) \in V^R \times V^R : \theta_{i,j} = 1 \Longrightarrow \sigma_{K_i} - \delta_w(T_j) \leq \sigma_j + p \times \mu_{i,j}$$

- If there is no register reuse between two tasks T_i and T_j, then $\theta_{i,j} = 0$. The storage dependence distance $\mu_{i,j}$ must be set to 0 in order to not be accumulated in the objective function.

$$\forall (T_i, T_j) \in V^R \times V^R : \theta_{i,j} = 0 \Longrightarrow \mu_{i,j} = 0$$

The reuse relation must be a bijection from V^R to V^R:

- a register can be reused by one task: $\forall T_i \in V^R : \sum_{T_j \in V^R} \theta_{i,j} = 1$
- a task can reuse one released register: $\forall T_i \in V^R : \sum_{T_j \in V^R} \theta_{j,i} = 1$

Objective Function We want to minimise the storage requirement:

$$\text{Minimise} \sum_{(T_i, T_j) \in V^R \times V^R} \mu_{i,j}$$

The above integer linear program is also useful if we are faced to the inverse problem: how to compute MEP_R the minimal scheduling period given a storage capacity R ? Indeed, answering this question does not require to minimise the storage requirement at the lowest possible level, we can simply remove the objective function and we add a constraint:$\sum_{(K_i, T_j) \in E'} \mu_{i,j} \leq R$. MEP_R can be calculated iteratively using a binary search on p between MEP and an upper-bound L.

PSSM is an NP-complete problem, even for trees and chains [15]. As far as we know, polynomial instances have not been discovered yet. So in practice, we require some heuristics in order to be able to consider large DDG instances. In [6, 14], we designed an efficient polynomial heuristic that minimises $\sum \mu_{1,j}$ without any constraints on the shape of the reuse graphs (reuse arcs can be added between any tasks). That is, the heuristic designed in [6, 14] starts by first minimising $\sum \mu_{i,j}$ then fixing reuse arcs. The next section, contrary to [6, 14], shows how to first fix reuse arcs then to minimise $\sum \mu_{i,j}$. Starting by fixing reuse arcs is necessary in some situations (to be explained later) when the underlying architectural constraints impose to have particular shape for the reuse graph.

5 Problem Simplification : Fixing Reuse Arcs

Given a DDG, the number of possible reuse graphs is combinatorial. Our exact PSSM formulation models all possible reuse graphs, which produce very long processing time. In order to simplify the problem, we consider now the complexity of our storage minimisation problem when fixing reuse arcs, while letting their μ distances to be computed. Formally, this problem can be stated as follows.

Problem 2 (Fixed PSSM) *Let $G^{reuse} = (V^R, E^{reuse}, \mu)$ an incomplete reuse graph with already fixed reuse arcs, but the reuse distances remain to be computed. Let $G^{\rightarrow r} = (V, E, \delta, \lambda)$ an incomplete DDG associated with it, and p a desired scheduling period. Let $E' \subseteq E$ be the set of already fixed storage dependences (which correspond to the reuse arcs of G^{reuse}). Find a distance $\mu_{i,j}$ for each storage dependence $(K_i, T_j) \in E'$ such that $\sum_{(K_i,T_j)\in E'} \mu_{i,j}$ is minimal, and the resulted DDG $G^{\rightarrow r}$ has a valid schedule with a period equal to p.*

In the following, we assume that $E' \subseteq E$ is the set of these already fixed storage dependences (their distances μ have to be computed). The process of early fixing storage dependences greatly simplifies the integer linear program of Section 4. Consequently, the Fixed PSSM problem can be solved by the following integer program, assuming a given desired scheduling period p.

$$\begin{cases} \text{Minimise} & \sum_{(K_i,T_j)\in E'} \mu_{i,j} \\ \text{Subject to:} & \\ p \times \mu_{i,j} + \sigma_j - \sigma_{K_i} \geq -\delta_w(T_j), & \forall (K_i, T_j) \in E' \\ \sigma_j - \sigma_i \geq \delta(e) - p \times \lambda(e), & \forall e = (T_i, T_j) \in E - E' \\ \sigma_{K_i} - \sigma_j \geq \delta_r(T_j) + p \times \lambda(e), & \forall T_i \in V^R, \forall T_j \in Cons(T_i) | e = (T_i, T_j) \in E^R \end{cases} \tag{5}$$

System 5 contains $\mathcal{O}(|V|)$ variables and $\mathcal{O}(|E|))$ linear constraints, but it does not mean that its resolving complexity is polynomial. Indeed, even if System 5 is a good simplification of PSSM, its constraints matrix isn't totally unimodular yet. However, this simplification allows to solve the PSSM for larger DDGs (multiple hundreds of nodes) compared to the case of exact optimal PSSM where only small DDGs can be taken into account (since optimal PSSM is an NP-complete problem).

Furthermore, we can go further by simplifying System 5. Since p is a constant, we can do the variable substitution $\mu'_{i,j} = p \times \mu_{i,j}$ and System 5 becomes:

$$\begin{cases} \text{Minimise} & \sum_{(K_i,T_j)\in E'} \mu'_{i,j} \\ \text{Subject to:} & \\ \mu'_{i,j} + \sigma_j - \sigma_{K_i} \geq -\delta_w(T_j), & \forall (K_i, T_j) \in E' \\ \sigma_j - \sigma_i \geq \delta(e) - p \times \lambda(e), & \forall e = (T_i, T_j) \in E - E' \\ \sigma_{K_i} - \sigma_j \geq \delta_r(T_j) + p \times \lambda(e), & \forall T_i \in V^R, \forall T_j \in Cons(T_i) | e = (T_i, T_j) \in E^R \end{cases} \tag{6}$$

Theorem 2 *[15] The constraints matrix of the integer linear program of System 6 is totally unimodular, i.e., the determinant of each square sub-matrix is equal to 0 or to $\pm$ 1.*

Consequently, we can use polynomial algorithms to solve this problem [13]. This would allow us to consider huge DDGs (multiple thousands of nodes). We must be aware that the back substitution $\mu = \frac{\mu'}{p}$ may produce a non integral value for the distance μ. If we ceil it by setting $\mu = \lceil \frac{\mu'}{p} \rceil$, a sub-optimal solution may result[1]. It is easy to see that the loss in terms of number of storage requirement is not greater than the number of generic tasks that write into a storage location ($|V^R|$).

The current section shows that if reuse arcs are fixed in advance, then the PSSM problem becomes simplified, and the experimental section will demonstrate the practical efficiency of this method. The next section gives hints about strategies for fixing reuse arcs.

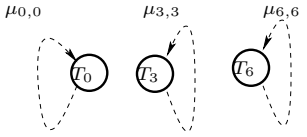
6 How to Fix Reuse Arcs ?

In this section, we provide some ways to automatically decide how to reuse fix arcs before solving the Fixed PSSM problem. Note that by fixing reuse arcs, we do not guarantee any optimality. This section provides heuristics that treat large problems which cannot be optimised with the optimal integer linear model. Later, we will study the experiment efficiency of our heuristics.

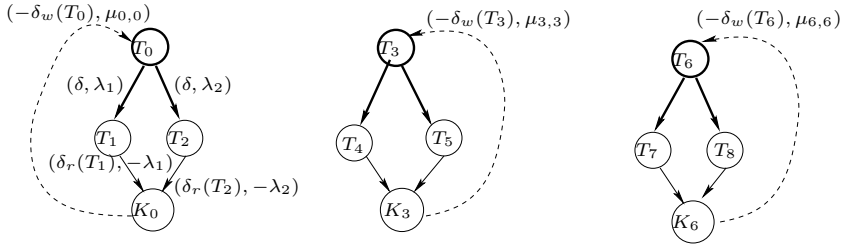
[1] Of course, if we have $p = 1$ (case of non cyclic DDG for instance), the solution remains equal to System 5.

6.1 Buffers

Our fixed PSSM problem is a generalisation of the buffer optimisation problem. Buffers are FIFO storage facilities that transport data between generic tasks. That is, a generic task writes its result in its own buffer that will be accessed (read) by further tasks. When buffer/FIFO structures are used as a storage memory, there is no storage sharing between generic tasks: each task has its own pool of storage locations. In our framework, this problem actually amounts to deciding that each generic task reuses the same storage location, possibly some iterations later. Consequently, when dealing with buffer, reuse graphs consist of nodes with reflexive arcs. Figure 6 provides an example of a DDG with its reuse graph in case of buffers.



(a) Example of Reuse Graph in Case of Buffers



(b) Example of Associated DDG in Case of Buffers

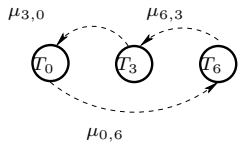
Figure 6: Example with Buffers

One of the advantages of using buffers is that their optimisation can be done easily and quickly thanks to the fixed PSSM model (will be shown later in our experimental section). But the main disadvantage of using buffers is that they require a larger storage requirement. Indeed, having a set of l generic tasks, the minimal number of required buffers is always greater than or equal to l, since we require at least one buffer per generic task (sharing storage locations between generic tasks is disabled).

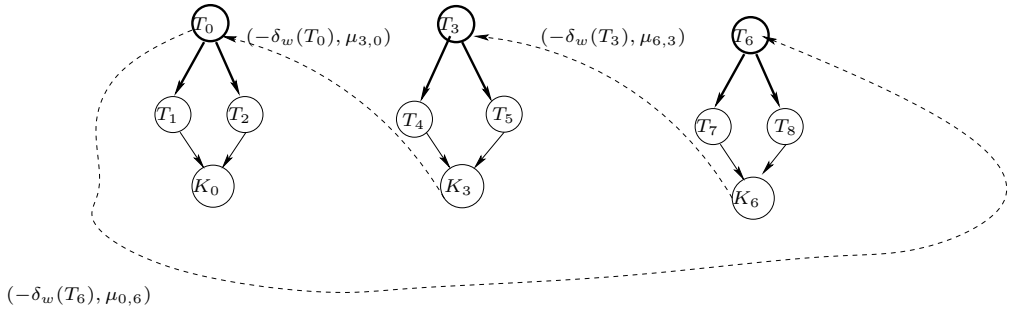
6.2 Rotating Storage Facilities

A rotating register file (RRF) is a hardware feature implemented inside some ILP processors in order to allow a compact code generation for loops. If RRF are used, the index number of each register in the processor is shifted at each loop iteration. At every new loop iteration, each logical register accessed by the program is renamed and mapped to a new physical register. Let R_i denotes a logical register numbered by i and accessed by the program and R'_j a physical register, then the mapping function is defined as: $R_i \mapsto R'_{(i+RRB) \bmod s}$ where RRB is an integral number (Rotating Register Base) and s the total number of physical registers. The number RRB is decremented at each loop iteration, which yields to a new mapping between logical and physical registers. Consequently and necessarily, the intrinsic reuse scheme between tasks describes now a Hamiltonian reuse circuit. The hardware dynamic behaviour of rotating register files does not allow other reuse patterns. PSSM in this case must be adapted in order to look only for Hamiltonian reuse circuits. Figure 7 provides an example of a DDG with its reuse graph in case of rotating storage facilities. In this example, we have fixed a Hamiltonian reuse circuit in an arbitrary way.

Given a DDG, the number of possible Hamiltonian reuse circuits is combinatorial too. Looking for the best Hamiltonian reuse circuit can be done by integer linear programming as studied in [18]. However, in order to solve a fixed PSSM problem, we should fix a Hamiltonian reuse circuit. The question is how to choose a good one ? A first practical approximation is to fix an arbitrary one by numbering the generic tasks from 0 to $l-1$. Or, we can use some sophisticated heuristics as explained in the following section.



(a) Example of Reuse Graph in Case of Rotatig Storage



(b) Example of Associated DDG in Case of Rotating Storage

Figure 7: Example with Rotating Storage Facilities

6.3 Exploiting Precedence Constraints Information

By analysing the data dependence and precedence constraints of the generic tasks, we can sometimes deduce partial lifetime relationships that would be satisfied by any schedule. For instance, we can sometimes deduce which tasks that can never have simultaneously alive results for all possible schedules. Or even better, we can deduce partial transitive order between lifetime intervals. For instance, if we can deduce that the lifetime interval of a $T\langle i,k\rangle$ is always before the lifetime interval of $T\langle j,k\rangle$ for any schedule, then we can safely decide that T_j reuses the storage location released by T_i with a distance $\mu_{i,j} \geq 0$. That is, we can fix a reuse arc between T_i and T_j without hurting task parallelism: since the lifetime interval of $T\langle i,k\rangle$ is guaranteed to always be before $T\langle j,k\rangle$ (no lifetime overlap), then $K\langle i,k\rangle$ (the virtual killer of $T\langle i,k\rangle$) would always be scheduled before $T\langle j,k\rangle$. Consequently, the fixed reuse arc $(T_i,T_j) \in E'$ introduces a redundant precedence relationship between $K\langle i,k\rangle$ and $T\langle j,k\rangle$. Furthermore, the $\mu_{i,j}$ distance of the reuse arc (T_i,T_j) can be minimised till zero without hurting task parallelism, because $T\langle i,k\rangle$ and $T\langle j,k\rangle$ belongs to the same iteration and can always share the same storage location without additional restriction on parallelism.

Now the problem is how to deduce a partial order between lifetime intervals in a set of repetitive task ? This is an open interesting and difficult question. In [16], we have formally answered this question in the case of acyclic schedules considering directed acyclic graphs (DAG). When considering a DAG $G = (V,E,\delta)$[2], the tasks are no longer repetitive, and the set of generic tasks represents the set of all tasks instances. Generalising our study on DAGs to general cyclic DDG of repetitive tasks is not straightforward. However, in the current report we take benefit of our former result in order to design a sophisticated heuristics for a fixing reuse arc strategy.

In [16], we provided a formal method that analyses an initial DAG $G = (V,E,\delta)$ and constructs a new DAG in polynomial time, called the *disjoint value graph* (DVG) of G, noted $DVG(G)$. The DVG represents a partial interval order between lifetime intervals. When an arc (T_i,T_j) exists in the DVG, it means that any acyclic schedule will guarantee that the lifetime interval of T_i is always before T_j. How to construct such DVG for a DAG is a complex question that have been deeply and formally answered in [16]. In order to limit the size of our report, we assume in our current contribution that we have a module computing the DVG of a DAG. The following steps takes benefit of an acyclic DVG in the context of repetitive tasks.

First, how can we use DVG in the context of a cyclic DDG $G = (V,E,\delta,\lambda)$? We start by considering G' the DDG *body* only. G' is a sub-graph of G which consists of G after removing every arc e with $\lambda(e) \neq 0$. That is, $G' = (V,E',\delta)|E' = \{e \in E, \lambda(e) = 0\}$. G' contains V the set of generic tasks and only the arcs between the tasks of the same iteration[3].

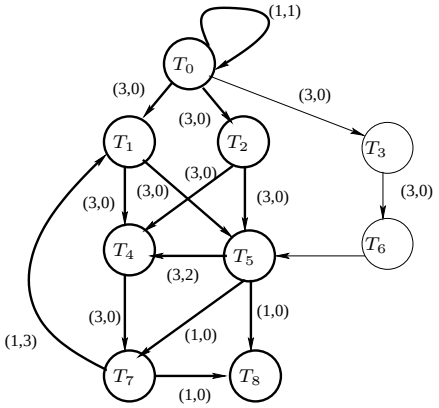
[2] V is the set of nodes, E is the set of arcs, and $\delta(e)$ is the latency of an arc $e \in E$

[3] Such sub-graph G' is necessarily a DAG by construction, commonly called DDG body.

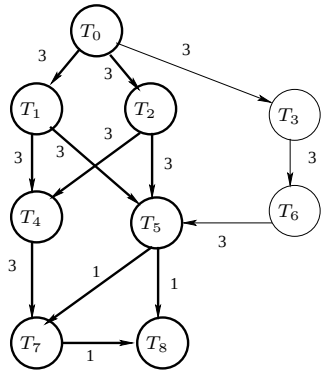
Second, we construct $DVG(G')$, the disjoint value DAG of the DDG body. A polynomial algorithm that constructs $DVG(G')$ is already explained in [16], so we do not report it here. Any arc (T_i, T_j) in G' means that the lifetime of T_i ($\langle T_i, k\rangle$ precisely) ends before the starting of the lifetime interval of T_j ($\langle T_j, k\rangle$ precisely). By computing a Dilworth decomposition of $DVG(G')$, we construct a minimal set C of chains containing non interfering lifetime intervals, for any schedule. This set of chains represent partial reuse decisions: when a chain contains an arc (T_i, T_j), then we fix a reuse arc (T_i, T_j). Recall that when (T_i, T_j) is inside a chain of the DVG, this means that the lifetime intervals of $\langle T_i, k\rangle$ would always end before the starting of the lifetime interval of $\langle T_j, k\rangle$. Consequently, fixing a reuse arc between T_i and T_j introduces a redundant scheduling constraint between K_i and T_j.

Third, the set of chains of the DVG defines only a partial order between lifetime intervals. How do we complete the reuse graph with the missing reuse arcs ? Remember that a reuse graph should consist of a set of elementary and disjoint circuits, as proved by Lemma 1.

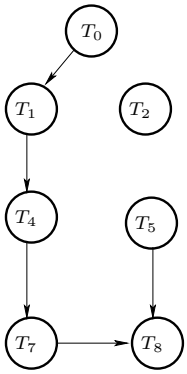
Figure 8 illustrates the three steps explained above. Initially, we have a DDG with possible circuits. In this DDG, tasks requiring storage units are drawn in bold circles. Flow dependences are also drawn in bold arcs. In step 1, we remove all arcs e with $\lambda(e) \neq 0$. This produces G' a DAG (DDG body): the arcs $(T_0, T_0), (T_7, T_1), (T_5, T4)$ have been removed. In step 2, we build the disjoint value DAG according to our method previously published in [16]. In the third step, we compute a minimal chain decomposition on $DVG(G')$ (Dilworth decomposition), producing a set of chains which constitute the partial reuse graph. The next two subsections explains how we complete this partial reuse graph in order to form a well structured one. In order to form well structured reuse graphs, we use two heuristics, depending if rotating storage facilities exist or not.



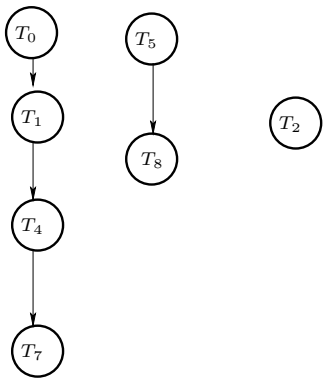
Step 0: initial DDG



Step 1: build G' the DAG (DDG body)



Step 2: build DVG(G')

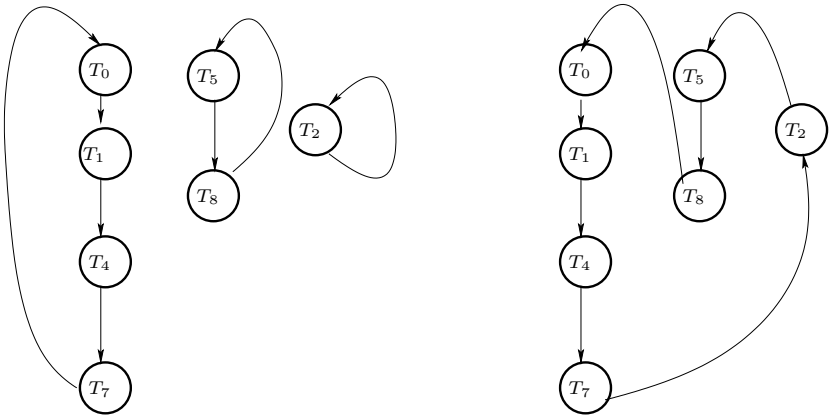


Step 3: Dilworth Decomposition of DVG(G')

Figure 8: Constructing Partial Reuse Graphs

6.3.1 No Rotating Storage Facilities: Pools of Chains

If no rotating storage facilities are present, we decide heuristically to fix a reuse arc to form an elementary circuit based on each chain. A chain $C' \in C$ of $DVG(G')$ consists of a set of nodes $C' = \{T_1, T_2, \cdots, T_p\}$. It defines a set of fixed reuse arcs $\{(T_1, T_2), (T_2, T_3), \cdots, (T_{p-1}, T_p)\}$. In order to form an elementary circuit based on C', we add the fixed reuse arc (T_p, T_1). An example is illustrated in Figure 9(a). The chain (T_0, T_1, T_4, T_7) for instance has been completed by a reuse arc (T_7, T_0) in order to form an elementary circuit. We also added the reuse arcs (T_8, T_5) and (T_2, T_2) producing a well structured reuse graph. Now, the set of all reuse graphs is well defined, producing an instance for the Fixed PSSM problem.



(a) Fixed Reuse Graph: Pools of Chains (b) Fixed Reuse Graph: Rotating Storage

Figure 9: Completing Reuse Graphs

6.3.2 With Rotating Storage Facilities: Hamiltonian Reuse

If rotating storage facilities exist, the reuse graph should contain a Hamiltonian circuit. Having a set C of all chains, we have to connect them in order to form a Hamiltonian circuit. Let be T_1 and T_{C_i} the first and the last node of a chain $C_i \in C$.

1. A Hamiltonian circuit can be constructed arbitrarily by connecting the last node of chain C_i with the first node of chain C_{i+1}, and the last node of the last chain should be connected to the first node of the first chain. An example is illustrated in Figure 9(b): we added the arcs $(T_2, T_5), (T_8, T_0)$ and (T_7, T_2).

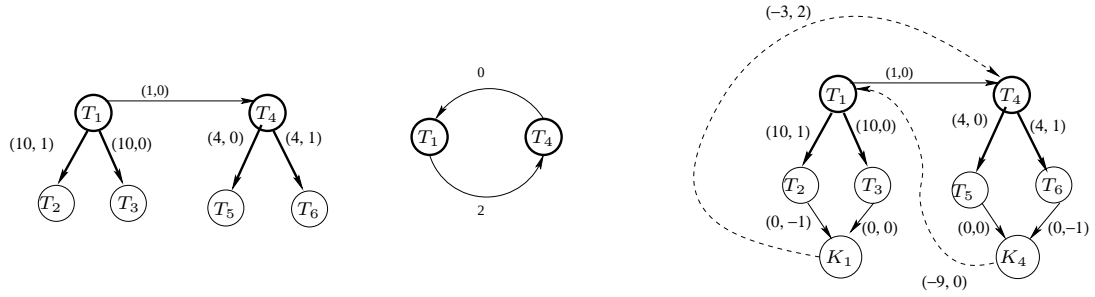
2. Or, we can use a common scheduling heuristics in the community of periodic instruction scheduling (software pipelining [12, 5]). Indeed, one of the list scheduling strategies is to choose a ready node based on the shortest paths information. We use this technique in our context: when we have to connect a node T_i to another node in order to form a Hamiltonian circuit, we choose the closest node T_j. By closest node, we mean the node that is on the shortest path from T_i. The shortest ath is computed on the graph by considering that each arc has a cost equal to $\delta(e) - p \times \lambda(e)$ [4].

As explained before, our model includes writing/reading delays in accessing registers. The non-positive latencies of the introduced storage dependence arcs may cause difficulties if we want to apply a subsequent scheduling step under resources constraints just after storage optimisation step. So, we should solve the problem during the step of storage optimisation. The next section solves this problem.

7 Eliminating Solutions inducing Non-positive Circuits inside the DDG

Our generic machine model allows reading/writing delays in accessing storage units. Consequently, the introduced storage dependences in $G^{\rightarrow r}$ the DDG associated with a reuse graph may have non-positive latencies. The non-

[4] The cyclic cost here reflects the fact that we are considering periodic schedules of repetitive tasks. In the case of an acyclic schedule, computing a shortest path is done by simply considering that $cost(e) = \delta(e)$.



(a) Initial DDG (b) Valid Reuse Graph (c) The DDG Associated with the Reuse Graph

Figure 10: Non-Positive Circuits

positive latencies of the introduced storage dependences allow us to have more opportunities to optimise storage locations. This is because any storage location isn't marked as *reserved* during the whole execution time of task writing in it: in other words, even when a storage location is assigned to hold the data produced by a task T_i, we are still allowed to access that location $\delta_w(T_i)$ time units after the start of T_i execution.

Initially, if the DDG represents the data dependences between the instructions of a sequential loop, then we have necessarily $\lambda(C) > 0$ for any circuit. Afaterwards, solving the PSSM problem modifies the initial DDG and may introduce storage dependence arcs with non-positive latencies which may lead to solutions (DDGs) containing a circuit C with non-positive distance ($\lambda(C) = \sum_{e \in C} \lambda(e) \leq 0$). From the perspective of scheduling theory, and when we assume unbounded resources, circuits with non-positive distances do not prevent a DDG from being scheduled with a positive integral period $p > 0$ [1, 9]. But such circuits impose hard scheduling constraints that may not be satisfiable by resource constraints in the possible subsequent pass of task scheduling. This is because circuits with non-positive distances impose scheduling constraints of type *not later than* that are similar to real time constraints [1, 9]: Given a cyclic DDG, let C^+ be the set of circuits with $\lambda(C) \geq 0$ and C^- be the set of circuits with $\lambda(C) < 0$; Then the following inequality is true[1]:

$$\max_{C \in C^+} \frac{\delta(C)}{\lambda(C)} \leq p \leq \min_{C \in C^-} \frac{\delta(C)}{\lambda(C)}$$

In other words, the existence of circuits inside C^- imposes hard real time constraints on the value of the period p. If the solutions of PSSM generate circuits C^-, we cannot guarantee the existence of at least a feasible schedule when a subsequent pass tries to optimise resource utilisation, because the resource constraints may not allow to have a period lower than $\min_{C \in C^-} \frac{\delta(C)}{\lambda(C)}$, while it is always possible to satisfy the resource constraints with a period greater than $\max_{C \in C^+} \frac{\delta(C)}{\lambda(C)}$.

Example 4 *As an illustration, look at Figure 10(a), where flow dependences are in bold arcs and tasks writing into storage locations are in bold circles. In the initial DDG, there exists a dependence path from T_1 to T_4 with a distance $\lambda = 0$. A reuse decision as shown in Figure 10(b) may assign the same storage location to T_1 and T_4. This creates an storage dependence from K_4 (T_4's killer) to T_1, see Figure 10(c). Since the latency of the reuse arc (K_4, T_1) is negative ($\delta = -9$) and there exists a path $T_1 \rightsquigarrow K_4$ with a latency equal to 5 ($T_1 \rightarrow T_4 \rightarrow T_5 \rightarrow K_4$); The circuit $C = (T_4, T_5, K_4, T_1, T_4)$ in Figure 10(c) has a distance $\lambda(C) = 0$ and a latency $\delta(C) = -4$. This circuit does not prevent the associated DDG from being periodically scheduled (since the precedence constraints can be satisfied assuming unbounded resources), but may do so in the presence of resource constraints.*

In order to avoid the inconvenience of non-positive circuits, we have three possible scenarios:

1. We simply do not insert non-positive arcs inside a DDG (all inserted reuse arcs would have a latency equal to 1). In other words, we do not consider reading/writing delays in the model. This decision simplifies the problem, produces correct/feasible solutions, but loses the optimality of the modelling (the storage requirement becomes sub-optimal).

2. Also, we can include resources constraints in our model. Consequently, PSSM under resources constraints would lead to solutions that satisfy precedences, resources and storage constraints. This is outside the scope of the paper because of the following.

3. In some cases, resources constraints are less important than storage constraints (for instance, the case of periodic register allocation in parallel assembly codes). Consequently, we should proceed in two steps : the first step optimises storage requirement using PSSM methods; Then, a second pass builds a final periodic schedule that satisfies resources constraints. In other words, we should not consider resources constraints when optimising storage requirement.

In this section, we are placed in the latter scenario. That is, we should provide solutions for the PSSM problem with the additional constraint of eliminating solutions with non-positive circuits.

Alain Darte provided as an optimal method for such solutions elimination. We add a quadratic number of linear retiming constraints [10] to avoid non-positive circuits. We define a retiming r_e for each arc $e \in E$. We have then a shift $r_e(T_i)$ for each node $T_i \in V$. We declare then an integer variable $r_{e,i}$ for all $(e, T_i) \in (E \times V)$. Any retiming r_e must satisfy the following constraints:

$$\begin{cases} \forall e' = (T_{i'}, T_{j'}) \neq e, & r_{e,j'} - r_{e,i'} + \lambda(e') \geq 0 \\ \text{for the arc } e = (T_i, T_j), & r_{e,j} - r_{e,i} + \lambda(e) \geq 1 \end{cases} \tag{7}$$

Note that an arc $e = (K_i, j) \in E'$ is a storage dependence, *i.e.*, its distance is $\lambda(e) = \mu_{i,j}$ has to be computed. Since we have $|E|$ distinct retiming functions, we add $|E| \times |V|$ variables and $|E| \times |E|$ constraints. The constraints matrix is totally unimodular, and it does not alter the total unimodularity of System 6. The following lemma proves that satisfying System 7 is a necessary and sufficient condition for building a DDG $G^{\rightarrow r}$ such that every circuit C has necessarily a positive distance $\lambda(C) > 0$.

Lemma 2 *[15] Let $G^{\rightarrow r}$ the solution graph built after solving System 5 or System 6. Then: System 7 is satisfied $\Longleftrightarrow$ any circuit in $G^{\rightarrow r}$ has a positive distance $\lambda(C) > 0$.*

Consequently, any Fixed PSSM solution that satisfies the constraints of System 7 lead necessarily to a DDG that is schedulable under any resource constraints.

8 Experiments

We have developed a complete tool based on the research results presented in this report. It implements all the integer linear programs that solve either the exact optimal problems (PSSM) and the simplified problems (Fixed PSSM, with all heuristics of fixing reuse arcs). Our tool is able to use two distinct LP solvers (CPLEX version 10.1 and LP_SOLVE version 5.5). However, after many experimental studies with both the solvers, CPLEX is definitively the best solver for integer linear programming, and we use it to present the experimental results in this section. We use a PC under linux, equipped with a Pentium IV 3.2 Ghz processor, and 1 Go of memory. We first start by detailing the software setup of our experiments.

8.1 Benchmarks Presentation

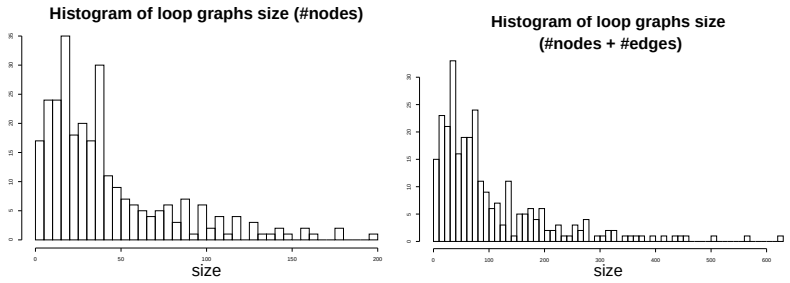
We used an initial set of 28 DDGs extracted from different benchmarks (Spec95, whetstone, livermore, lin-ddot). These DDGs are detailed in [15] and available in [21]. The number of generic tasks in these initial DDGs goes from 2 to 20, and the number of arcs from 2 to 26. In order to obtain larger DDG, we used loop unrolling (increase the size of a DDG till a factor of 10). Figure 11 plots the number of DDGs of a given size: Figure 11(a) defines the DDG size as the number of nodes ($|V|$), while Figure 11(b) defines it as the number of nodes and arcs ($|V| + |E|$). Our benchmarks represent typical loop sizes in programs devoted to periodic register allocation.

In addition to all these DDGs, we experimented all the values of periods, from MEP to L. In all our experiments, we set L (maximal value of a period for a DDG) as the sum of all generic task latencies of the given DDG. This yields to multiple thousands of experiments that last two full months on our workstations.

8.2 Nomenclature

Our storage optimisation tool is named SIRA. It implements all the following storage optimisation methods:

1. OPT_1: minimise storage requirement using the exact method presented in Section 4;
2. OPT_2: minimise storage requirement using the exact method, plus additional constraints for building a Hamiltonian reuse circuit (not present in the current paper, but is explained in [15, 18]);



(a) DDG Sizes (nodes) (b) DDG Sizes (nodes+arcs)

Figure 11: Frequency of Benchmarks Sizes

3. H_3: solve the Fixed PSSM problem (System 5), reuse arcs are fixed according to the buffers method as explained in Section 6.1;
4. H_4: solve the polynomial variant of the Fixed PSSM problem (System 6), reuse arcs are fixed according to the buffers method as explained in Section 6.1;
5. H_5: solve the Fixed PSSM problem (System 5), reuse arcs are fixed arbitrarily to form a Hamiltonian reuse circuit as explained in Section 6.2.
6. H_6: solve the polynomial variant of the Fixed PSSM problem (System 6), reuse arcs are fixed arbitrarily to form a Hamiltonian reuse circuit as explained in Section 6.2.

In addition to the above SIRA methods, we also implemented the strategies described in Section 6.3. In that section, reuse arcs are automatically fixed by building a Dilworth decomposition of a disjoint value DAG. The reuse circuits are then built/fixed thanks to the following heuristics.

1. H_9: solve the Fixed PSSM problem (System 5). Reuse circuits are built with the pools of chains method (each reuse chain is completed to form an elementary reuse circuit);
2. H_{10}: solve the polynomial variant of the Fixed PSSM problem (System 6). Reuse circuits are built with the pools of chains method (each reuse chain is completed to form an elementary reuse circuit);
3. H_{11}: solve the Fixed PSSM problem (System 5). Reuse Chains are completed arbitrarily to form a Hamiltonian reuse circuit;
4. H_{12}: solve the polynomial variant of the Fixed PSSM problem (System 6). Reuse Chains are completed arbitrarily to form a Hamiltonian reuse circuit;
5. H_{13}: solve the Fixed PSSM problem (System 5). Reuse Chains are completed by the short path strategy to form a Hamiltonian reuse circuit, as explained in Section 6.3;
6. H_{14}: solve the polynomial variant of the Fixed PSSM problem (System 6). Reuse Chains are completed by the short path strategy to form a Hamiltonian reuse circuit, as explained in Section 6.3;

SIRA also implements the additional constraints presented in Section 7 that eliminate solutions with non-positive circuits. SIRA implements two variants:

1. Either we consider that all inserted reuse arcs have a latency equal to 1. This variant is tagged by the symbole s. For instance, the SIRA method $H_3 + s$ means that the heuristics H_3 is coupled with an elimination of non-positive circuits using the variant s.
2. Or either we add the constraints of System 7 that eliminate non-positive circuits. This variant is tagged by the symbole n. For instance, the SIRA method $H_4 + n$ means that the constraints of System 7 are added to the constraints of heuristics H_4.

If neither n nor s are written in front of the SIRA method name, it means that non-positive circuits are kept.

Let note by $SR_{H_k}(p)$ the storage requirement computed by a SIRA method H_k for a period p, where $MEP \leq p \leq L$. We note $SR_{best}(p)$ as the minimal storage requirement obtained by all SIRA methods in practice. In theory, $SR_{best}(p) = SR_{OPT_1}(p)$. However, in practice, many DDG instances cannot be solved with optimal methods. In this case, $SR_{best}(p)$ represents the minimal storage requirement obtained in practice.

8.3 Classical vs. Rotating Storage Facilities (OPT_1 vs. OPT_2)

The first set of experiments investigates the solutions computed by OPT_1 versus OPT_2 (optimal solutions when classical storage mechanism versus rotating storage mechanism). We compare the periodic storage requirement of all data dependence graphs versus varying the periods p (this yield to hundreds of experiments). Since both $OPT1$ and $OPT2$ are optimal methods solving NP-complete problems, they cannot be used in practice on large DDG. Our experiments show that in most of cases, classical storage mechanism and rotating storage mechanism need the same number of storage units (using the same fixed period p). However, we found in few cases (within 5% of the experiments) that using a rotating storage mechanism may need the use of one extra storage unit.

8.4 Optimal Results with OPT_1

Figure 12 presents the minimal storage requirement of some small DDGs (less than 20 nodes). All these curves are optimal, *i.e.* they plot $SR_{OPT_1}(p)$), except two DDGs: test-christine (18 nodes and 17 arcs) and liv-loop23 (20 nodes and 26 arcs). These two DDGs are difficult instances, and we stopped the CPLEX optimisation process after two days of computation. So their plotted curves represent $SR_{best}(p)$.

In theory, and as formally proved in [17], we have proved a sufficient condition on L such that any curve $SR_{OPT_1}(p)$ would be monotonic non-increasing. Indeed, it is intuitive that less parallelism (higher values of the period p) corresponds to less storage requirement (lower values of SR). Almost all the curves of Figure 12 are in this situation, except the two non optimal curves (test-christine[5] and liv-loop23) and the case of spec-spice-loop-9. This latter DDG is an interesting example already discussed in [17], where the curve $SR_{OPT_1}(p)$ is increasing because the chosen value of L is not large enough.

There are also situation where the curves $SR_{OPT_1}(p)$ are constants (such that the cases of whet-cycle4). That is, increasing the period (reducing the task parallelism) does not reduce the storage requirement. A careful look on these DDG show that they are composed of a unique elementary circuits C. In this case, the minimal storage requirement is always bounded by $\lambda(C)$, irrespective of p.

In order to have a quantitative measure on the optimal gain of storage requirement versus the optimal gain on task parallelism, we measure the two following metrics:

1. The task parallelism of a DDG is defined as $\frac{|V|}{p}$. So, the relative task parallelism gain (TPG) when the period is reduced from L to MEP is equal to
$$\frac{\frac{|V|}{p} - \frac{|V|}{L}}{\frac{|V|}{L}} = \frac{L}{MEP} - 1$$
2. The storage requirement increase (SRI) obtained when the period reduces from L to MEP is measured as $\frac{SR_{OPT_1}(MEP) - SR_{OPT_1}(L)}{SR_{OPT_1}(L)}$

Table 1 provides these measures for each benchmark (see the last two columns). We classify these results into four classes:

1. Class 1: there is no possible gain in task parallelism (resulted from the inherent data dependences of the DDG), and there are no increase in storage requirement. Spec-spice-loop5 and whet-cycle4-1 for instance belong to this class.
2. Class 2: there is a possible gain in task parallelism (resulted from the inherent data dependences of the DDG), but there is no increase in storage requirement. That is, the SR is constant for any value of p. Spec-spice-loop10 belongs to this class.
3. Class 3 (most usual): there is both a gain in task parallelism and in SR increase. For instance, spec-spice-loop1 can have a maximal TPG equal to 2 with only a SR increase of 50%. Also, the case of test-christine (a complex tree) may benefit in a TPG equal to 229 with a SR increase of 7567%.
4. Class 4: there is a possible gain in task parallelism and an increase in storage requirement. This case is possible in theory, but was not found in our benchmarks.

The next section investigates the efficiency of our various heuristics.

[5] While the curv of test-christine seems monotonic non-increasing, a close look on the values shows that it isn't.

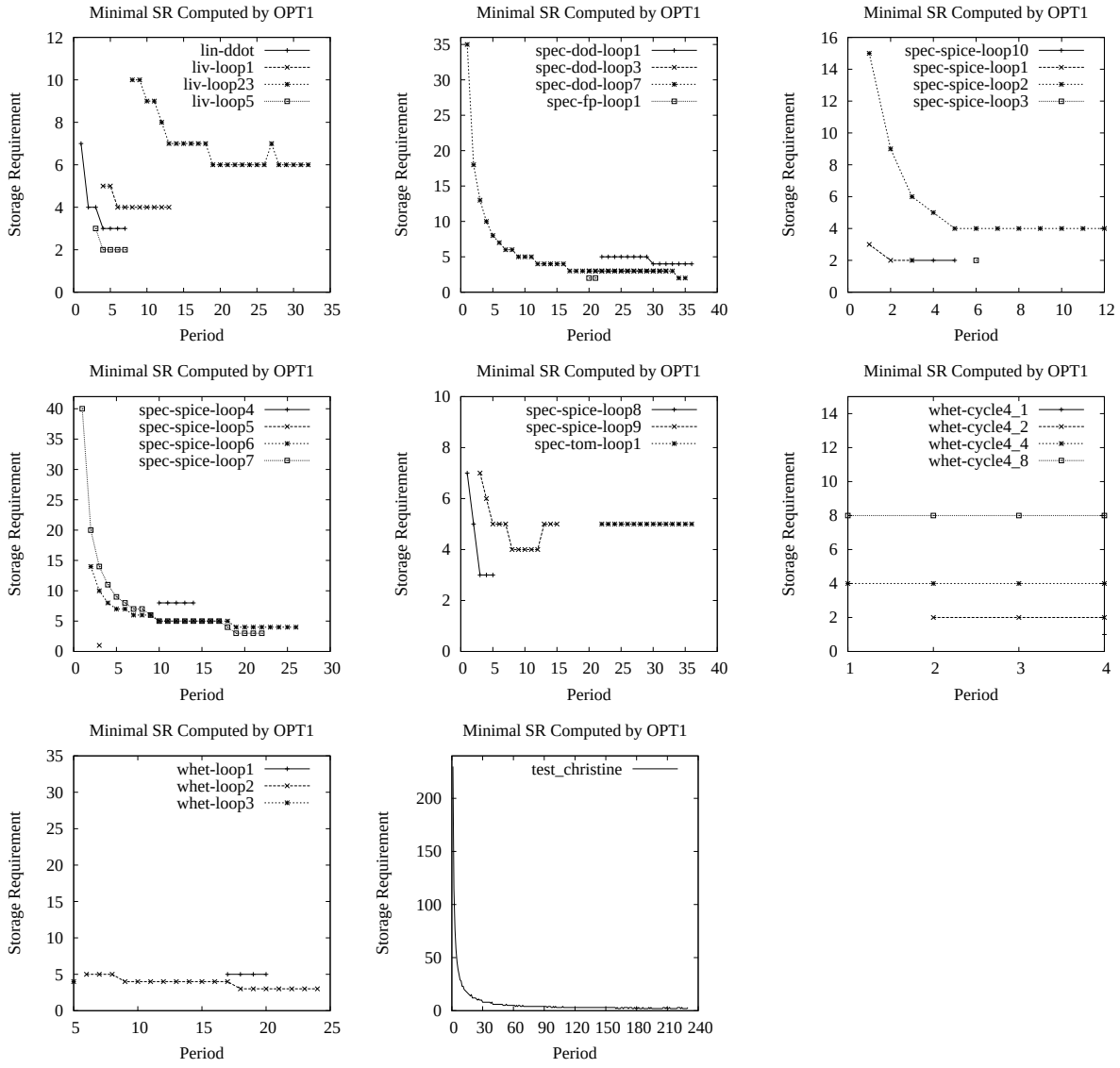


Figure 12: Minimal Storage Requirement in some Benchmarks

DDG	MEP	$SR_{OPT_1}(MEP)$	L	$SR_{OPT_1}(L)$	**SRI**	**TPG**
lin-ddot	1	7	7	3	1,33	6
liv-loop1	4	5	13	4	0,25	2,25
liv-loop23	8	10	32	6	0,67	3
sped-dod-loop1	22	5	36	4	0,25	0,64
sped-dod-loop3	20	3	32	3	0	0,6
sped-dod-loop7	1	35	35	2	16,5	34
spec-spice-loop1	1	3	3	2	0,5	2
spec-spice-loop2	1	15	12	4	2,75	11
spec-spice-loop3	6	2	6	2	0	0
spec-spice-loop4	10	8	14	8	0	0,4
spec-spice-loop5	3	1	3	1	0	0
spec-spice-loop6	2	14	26	4	2,5	12
spec-spice-loop7	1	40	22	3	12,33	21
spec-spice-loop8	1	7	5	3	1,33	4
spec-spice-loop9	3	7	15	5	0,4	4
spec-spice-loop10	3	2	5	2	0	0,67
spec-tom-loop1	22	5	36	5	0	0,64
test-christine	1	230	230	3	75,67	229
whet-cycle4-1	4	1	4	1	0	0
whet-cycle4-2	2	2	4	2	0	1
whet-cycle4-4	1	4	4	4	0	3
whet-cycle4-8	1	8	4	8	0	3
whet-loop1	17	5	20	5	0	0,18
whet-loop2	6	5	24	3	0,67	3
whet-loop3	5	4	5	4	0	0

Table 1: Maximal Task Parallelism Gain versus Optimal Storage Requirement Increase

8.5 Fixed PSSM Heuristics

In order to check the efficiency of our simplified method (Fixed PSSM), we decide to make a quantitative synthesis of $SR_{H_k}(p)$ the storage requirement obtained by the heuristics H_k versus p. For such quantitative study, we compute three metrics/measures:

1. $M_1(H_k)$ quantifies the average difference between $SR_{H_k}(p)$ and $SR_{best}(p)$ for all the DDG:

$$M_1(H_k) = \frac{1}{L - MEP + 1} \times \sum_{MEP \leq p \leq L} (SR_{H_k}(p) - SR_{best}(p))$$

2. $M_2(H_k)$ quantifies the maximal relative difference between $SR_{H_k}(p)$ and $SR_{best}(p)$:

$$M_2(H_k) = \frac{\max_p (SR_{H_k}(p) - SR_{best}(p))}{SR_{best}(p)}$$

3. $M_3(H_k)$ quantifies the minimal relative difference between $SR_{H_k}(p)$ and $SR_{best}(p)$:

$$M_3(H_k) = \frac{\min_p (SR_{H_k}(p) - SR_{best}(p))}{SR_{best}(p)}$$

In this section, we show the efficiency of the following Fixed PSSM Heuristics that resolve System 5: H_3, H_5, H_9, H_{11} and H_{13}. Table 2 provides a synthesis using the statistical measures defined above. The mean difference $M_1(H_k)$ between $SR_{H_k}(p)$ and $SR_{best}(p)$ of our heuristics shows that the sophisticated heuristics H_{11} and H_{13} produce the closest results to the lowest storage requirement. While the buffers (method H_3) exhibit in average the worst storage requirement. Furthermore, we show that $M_2(H_k)$ the maximal relative difference between $SR_{H_k}(p)$ and $SR_{best}(p)$ is less than a factor of 3 when we use the sophisticated heuristics H_{11} and H_{13}. From Table 2, we deduce that the sophisticated heuristics H_{11} and H_{13} that exploit the precedence constraints information of the DDG provide better storage minimisation methods in general.

Method	M_1	M_2	M_3
H_3	2.92	14	0
H_5	2.13	13.5	0
H_9	1.69	3	0
H_{11}	0.19	2	0
H_{13}	0.93	2.75	0

Table 2: Synthesis of Statistical Measures of Fixed PSSM Heuristics

In order to have a more detailed view of the statistical behaviour of our heuristics, we will now show the value of the metrics for each DDG. We use the *boxplot* invented by John Tukey (also known as a box and whisker plot). The boxplot summarises the following five statistical measures:

- the median;
- upper and lower quartiles of the values of the metrics;
- minimum and maximum data values.

In our case, the plotted data are the value of a metrics $M_1(H_k)$, $M_2(H_k)$, $M_1(H_k)$ for each DDG separately. The boxplot is interpreted as follows:

- The box itself contains the middle 50% of the data. The upper edge (hinge) of the box indicates the 75th percentile of the data set, and the lower hinge indicates the 25th percentile. The range of the middle two quartiles is known as the inter-quartile range.
- The line in the box indicates the median value of the data.
- If the median line within the box is not equidistant from the hinges, then the data is skewed
- The ends of the vertical lines or "whiskers" indicate the minimum and maximum data values, unless outliers are present in which case the whiskers extend to a maximum of 1.5 times the inter-quartile range.
- The points outside the ends of the whiskers are outliers or suspected outliers.

Boxplots have the following strengths:

- Graphically display a variable's location and spread at a glance.
- Provide some indication of the data's symmetry and skewness.
- Unlike many other methods of data display, boxplots show outliers.
- By using a boxplot for each categorical variable side-by-side on the same graph, one quickly can compare data sets.

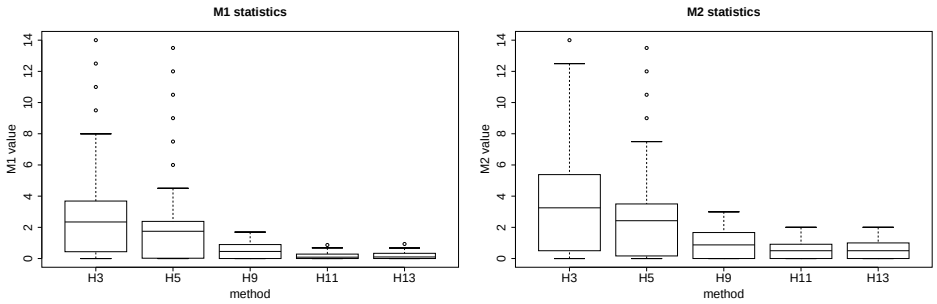
Figure 13 gives the boxplots of M_1, M_2 and M_3. We show clearly that the sophisticated heuristics H_9, H_{11} and H_{13} outperform the other methods in terms of storage requirement.

The next section studies the efficiency in terms of SR of Fixed SIRA method when doing the non bijective variable substitution $\mu'_{i,j} = p \times \mu_{i,j}$.

8.6 Fixed PSSM : System 5 versus System 6

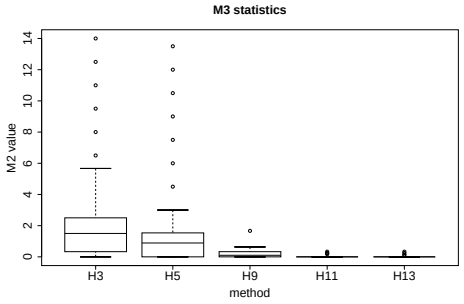
According to our nomenclature defined in Section 8.2, we should compare between the pairs of methods H_3 vs. H_4, H_5 vs. H_6, H_9 vs. H_{10}, H_{11} vs. H_{12}, H_{13} vs. H_{14}. As defined in the previous section, we use boxplots for presenting the values of our quantitative measures M_1 (mean difference between $SR_{H_k}(p)$ and $SR_{best}(p)$), M_2 (max relative difference between $SR_{H_k}(p)$ and $SR_{best}(p)$) and M_3 (min relative difference between $SR_{H_k}(p)$ and $SR_{best}(p)$).

Doing the variable substitution $\mu'_{i,j} = p \times \mu_{i,j}$ makes polynomial the integer linear program of System 5, producing another integer program System 6. However, this variable substitution is not bijective, so it induces additional costs in terms of storage requirement. Figure 14 presents the boxplots. We can deduce that the methods H_3 and H_4 exhibit similar behaviour. It means that the buffer method (H_3) does not suffer from the non bijective variable substitution $\mu'_{i,j} = p \times \mu_{i,j}$. The other heuristics present some storage requirement loss.



(a) M_1: Average Relative Difference

(b) M_2: Maximal Relative Difference



(c) M_3: Minimal Relative Difference

Figure 13: Measures for the Efficiency of Fixed PSSM Heuristics

8.7 Resolution Times

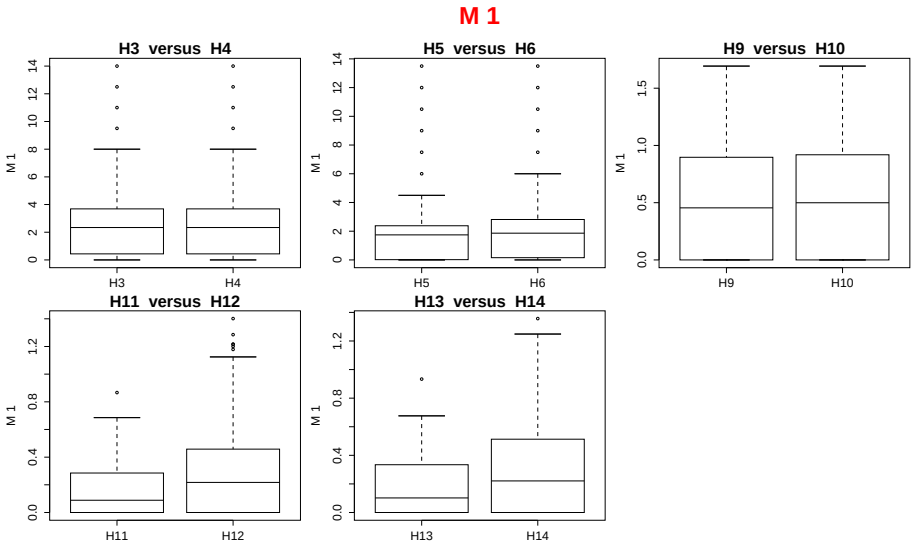
In this section, we present a quantitative synthesis on resolution times (in mili seconds) using CPLEX solver. For each SIRA method, we measure the time needed to pass through all the steps: DDG treatment, fixing reuse arcs strategy and CPLEX resolution. We quantify here the resolution time relatively to the DDG size, *i.e.*, we report the resolution time in milliseconds divided on $|V|$ the DDG size. We use boxplots to present all these time values in Figure 15. Figure 15(a) shows that the relative resolution times for the optimal methods OPT_1 and OPT_2 are very high compared to Fixed PSSM. This is because the PSSM problem is NP-complete and no polynomial instances are known in the literature. An interesting remark is that OPT_2 is faster than OPT_1: including additional constraints in case of rotating storage facilities seems to do not hurt the resolution process. Figure 15(b) provides a zoom on the relative resolution time of the Fixed PSSM heuristics. We can clearly deduce that:

- The polynomial instances of Fixed PSSM (H_3, H_6, H_{10}, H_{12}) are faster than the non polynomial instances. That is, doing the variable substitution $\mu'_{i,j} = p \times \mu_{i,j}$ accelerates the resolution process.
- The buffers method H_3 is the fastest one. However, the difference with the sophisticated heuristics (H_{11} and H_{13})) is less than 2 milliseconds in average per node, and can be neglected in practice.

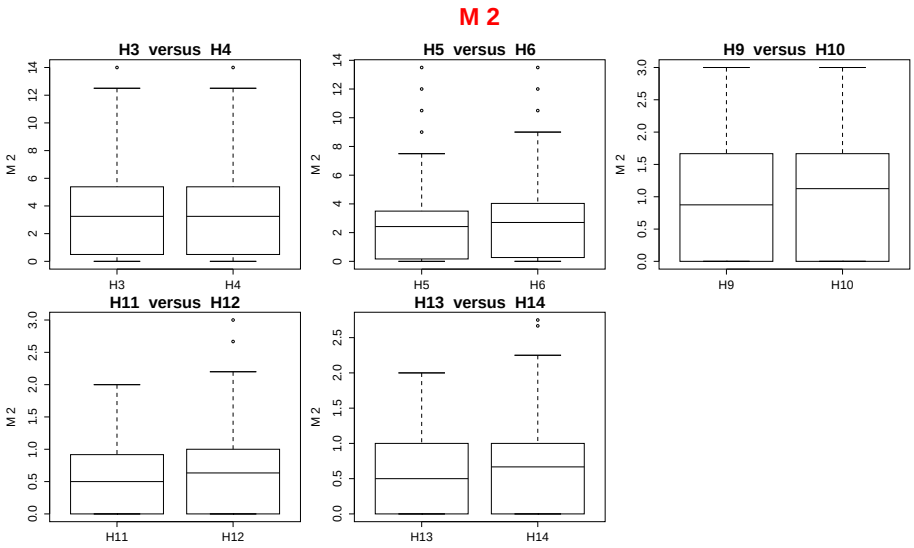
9 Conclusion

This research report presents new heuristics for periodic task scheduling with storage requirement optimisation. These heuristics are designed for specific storage architectures such as rotating storage facilities or buffers. Our task model is more generic than some existing practical ones, since it allows the exact definition of the problem while considering delays in writing and reading into/from storage locations. As a practical application, we use it for solving the problem of periodic register allocation for instruction scheduling on ILP processors, which is a distinct problem and more difficult than the old classical register allocation for sequential processors.

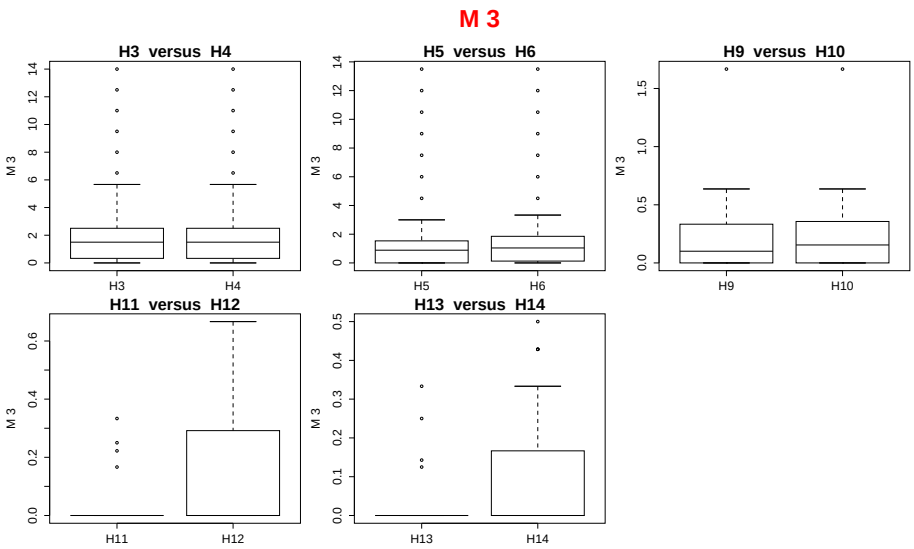
Our exact formulation of the problem has many applications: we can fix a scheduling period while minimising the storage requirement, or we can fix the scheduling period while bounding the storage requirement. The opposite problem of bounding the storage requirement while minimising the scheduling period can be solved by a binary search on p (solving iteratively successive integer problems of PSSM).



(a) M_1: Average Relative Difference

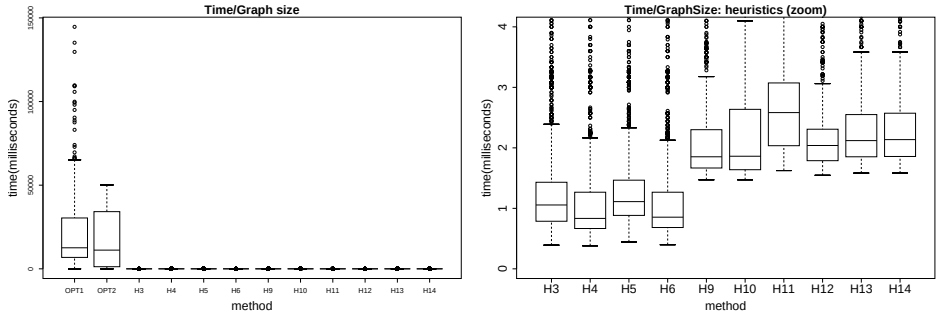


(b) M_2: Maximal Relative Difference



(c) M_3: Minimal Relative Difference

Figure 14: System 5 versus System 6



(a) All Methods (b) Zoom on Heuristics

Figure 15: Relative Resolution Time with CPLEX

Storage allocation is expressed in terms of reuse arcs and reuse distances to model the fact that two tasks use the same storage location. Our integer linear program computes an optimal solution with reduced constraint matrix size, and enables us to make a trade-off between task parallelism and number of storage locations.

Since computing an optimal periodic storage allocation is intractable in large data dependence graphs, we identified an approximate sub-problem by fixing reuse arcs and computing the reuse distances that minimise the overall storage requirement. We can use it in different ways, as setting self-reuse arcs (buffers minimisation) or fixing arbitrary (or with a cleverer algorithm) other reuse circuits. This simplification allows us to consider DDGs with multiple hundreds of nodes, but do not define a polynomial instance of the original problem. A polynomial sub-problem is obtained thanks to a non-bijective variable substitution $\mu'_{i,j} = p \times \mu_{i,j}$. This simplification allows us to consider huge DDGs, but induce an additional cost in terms of storage requirement (sub-optimal results).

Our machine model takes into account delayed read and write from/into storage locations. In graph theory, this is modelled by including arcs with non-positive delays. As a consequence, optimal storage minimisation may lead to solution graphs with non-positive circuits. These sort of circuits create difficulties if we want to apply a post-pass task scheduling under resources constraints, because non-positive circuits are indeed real-time constraints: we cannot guarantee the existence of a feasible schedule when considering limited resources. This report presents an optimal method (using circuit retiming[10]) that eliminates all solution graphs with non-positive circuits when applying storage minimisation, without losing the ability of considering arcs with non-positive latencies.

Our intensive experiments on numerous DDGs show that disabling storage sharing with a self reuse strategy (buffers) as done in [2, 7, 11] isn't a good decision in terms of storage requirement. We demonstrate that how storage elements are shared between different tasks is one of the key issues, and preventing this sharing by using buffers/FIFO consumes much more storage than really needed by other reuse decisions. However, buffers are sometimes mandatory in some hardware designs (networks on chips, data flow embedded systems, etc.).

This report presents many efficient heuristics (H_9, H_{11} and H_{13}) that fix the storage sharing scheme between generic tasks. Our heuristics exploit precedence constraints information and allow to have good storage requirement in practice. Furthermore, their relative resolution time are satisfactory compared to buffers.

References

[1] Alix Munier. A graph-based analysis of the cyclic scheduling problem with time constraints: schedulability and periodicity of the earliest schedule. *Journal of Scheduling*, 2010. To appear (accepted for publication).

[2] Alix Munier Kordon and Jean-Baptiste Note. A Buffer Minimization Problem for the Design of Embedded Systems. *European Journal of Operational Research*, 164(3):669–679, August 2005.

[3] Mounira Bachir, Sid-Ahmed-Ali Touati, and Albert Cohen. Post-pass periodic register allocation to minimise loop unrolling. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, Arizona, 2009. ACM.

[4] Claire Hanen and Alix Munier. A Study of the Cyclic Scheduling Problem on Parallel Processors. *Discrete Applied Mathematics*, 57(2-3):167–192, 1995.

[5] Alain Darte and Guillaume Huard. Loop Shifting for Loop Compaction. *International Journal of Parallel Programming*, 28(5):499–??, 2000.

[6] Karine Deschinkel and Sid-Ahmed-Ali Touati. Efficient method for periodic task scheduling with storage requirement minimization. In *Proceedings of 2nd Annual International Conference on Combinatorial Optimization and Applications (COCOA 2008)*, LNCS, Saint Johns, Newfoundland, Canada, August 2008. Springer.

[7] Jan Korst. *Periodic Multiprocessor Scheduling*. PhD thesis, Eindhoven University of Technology. The Netherlands, 1992.

[8] E. L. Lawler. Optimal Cycles on Graphs and Minimal Cost-to-Time Ratio Problem. In A. Marzojlo, editor, *Periodic Optimization*, volume 1, pages 38–58. Springer-Verlag, 1972.

[9] Tae-Eog Lee, Jeong-Won Seo, and Seong-Ho Park. An extended event graph with negative places and negative tokens for time window constraints. *Discrete Event Systems, International Workshop on*, 0:91, 2002.

[10] Charles E. Leiserson and James B. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, 6:5–35, 1991.

[11] Qi Ning and Guang R. Gao. A Novel Framework of Register Allocation for Software Pipelining. In *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 29–42, Charleston, South Carolina, January 1993. ACM Press.

[12] Antoine Sawaya. *Pipeline Logiciel: D?couplage et Contraintes de Registres*. PhD thesis, Université de Versailles Saint-Quentin-En-Yvelines, April 1997.

[13] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, New York, 1987.

[14] Sebastien Briais and Sid-Ahmed-Ali Touati. Schedule-Sensitive Register Pressure Reduction in Innermost Loops, Basic Blocks and Super-Blocks. Technical Report RR-INRIA-HAL-00436348, University of Versailles Saint-Quentin en Yvelines, November 2009. http://hal.archives-ouvertes.fr/inria-00436348.

[15] Sid-Ahmed-Ali Touati. *Register Pressure in Instruction Level Parallelisme*. PhD thesis, Université de Versailles, France, June 2002. ftp.inria.fr/INRIA/Projects/a3/touati/thesis.

[16] Sid-Ahmed-Ali Touati. Register Saturation in Instruction Level Parallelism. *International Journal of Parallel Programming*, 33(4), August 2005. 57 pages.

[17] Sid-Ahmed-Ali Touati. On Periodic Register Need in Software Pipelining. *IEEE Transactions on Computers*, 56(11), November 2007.

[18] Sid-Ahmed-Ali Touati and Christine Eisenbeis. Early Periodic Register Allocation on ILP Processors. *Parallel Processing Letters*, 14(2), June 2004. World Scientific.

[19] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Beth Simon. Schedule-Independent Storage Mapping for Loops. *ACM SIG-PLAN Notices*, 33(11):24–33, November 1998.

[20] William Thies, Frederic Vivien, Jeffrey Sheldon, and Saman Amarasinghe. A Unified Framework for Schedule and Storage Optimization. *ACM SIGPLAN Notices*, 36(5):232–242, May 2001.

[21] Sid-Ahmed-Ali Touati. DDG : A C++ High Level Data Dependence Graph Library. Academic research tool website, August 2007. http://www.prism.uvsq.fr/˜ touati/sw/DDG/.

UFR des sciences	:	45 avenue des Etats Unis. 78035 Versailles cedex
IUT de Velizy et de Rambouillet	:	10-12 avenue de l'Europe. 78140 Vélizy.
UFR des Sciences Sociales et des Humanité	:	47 boulevard Vauban. 78047 Guyancourt cedex
Faculté de droit et de science politique	:	3, rue de la Division Leclerc. 78280 Guyancourt
IUT de Mantes en Yvelines	:	7 rue Jean Hoët - 78200 Mantes la Jolie
UFR de Médecine Paris-Ile-de-France Ouest	:	9 boulevard d'Alembert Bâtiment François Rabelais. 78280 Guyancourt
Institut des Langues et des Etudes Internationales	:	5-7, boulevard d'Alembert. 78280 Guyancourt
Institut des Sciences et Techniques des Yvelines	:	45 avenue des Etas Unis - 78035 Versailles cedex
Observatoire des Sciences de l'Univers de l'UVSQ	:	11 boulevard d'Alembert. 78280 Guyancourt

Éditeur
UVSQ - Siège : 55 avenue de Paris - 78035 Versailles Cedex (France)
http://www.uvsq.fr