

# Optimal and Approximate Periodic Task Scheduling with Storage Requirement Minimization

Sid-Ahmed-Ali Touati

University of Versailles, France. Sid.Touati@uvsq.fr

---

In this paper, we study an exact formulation of the general problem of one-dimensional periodic task scheduling under storage requirement, irrespective of machine constraints. We present a new theoretical framework that allows an early optimization of periodic storage requirement. This is based on inserting some storage dependence edges (*storage reuse* edges) labeled with *reuse distances* directly on the data dependence graph. In this new graph, we are able to fix the storage requirement measured as the exact number of necessary storage locations. The determination of storage and distance reuse is parametrized by the desired minimal execution period (resp. maximal execution throughput) as well as by the storage requirement constraints - either can be minimized while the other one is bounded, or alternatively, both are bounded.

*Keywords:* Multi-processor Scheduling.

---

## 1 Introduction

This article addresses the problem of storage optimization in cyclic data dependence graphs (DDGs), which is for instance applied in the practical problem of periodic register allocation for innermost loops on modern Instruction Level Parallelism (ILP) processors. The massive introduction of ILP processors since the last decade makes us re-think new ways of optimizing 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 new fine grain parallelism (at the assembly code level) asks to completely revisit the old classical problem of register allocation initially designed for sequential processors. Nowadays, register allocation has not only to minimize the storage requirement, but has also to take care of parallelism and total schedule time. In this research article, we do not assume any resource constraints (except storage requirement); Our aim is to analyze 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 optimization in repetitive tasks.

Existing techniques in this field usually apply a periodic instruction scheduling that is sensitive to register/storage requirement. Therefore a great amount of work tries to schedule the instructions of a loop (under resource and time constraints) such that the resulting code does not use more than  $\mathcal{R}$  values simultaneously alive. Usually they look for a schedule that minimizes the storage requirement under a fixed execution period [3, 4, 6]. In this paper, we satisfy register constraints early before instruction scheduling under resource constraints: we directly handle and modify the DDG in order to fix the storage requirement of any further subsequent periodic scheduling pass while taking care of not altering parallelism exploitation if possible. This idea uses the concept of reuse vector used for multi-dimensional scheduling [12, 13].

Our article is organized as follows. Sect. 2 defines our problem model. Sect. 3 starts the study with a motivating example. The problem of optimal periodic scheduling under storage constraints is described with graph theory and integer linear programming in Sect. 4. Sect. 5 presents simplified sub-problems and polynomial heuristics.

## 2 Tasks Model

We consider a simple innermost loop in a program represented by a set of  $l$  generic tasks (instructions)  $T_0, \dots, T_{l-1}$ . 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  $k^{\text{th}}$  occurrence of task  $T_i$  is noted  $T\langle i, k \rangle$ , which corresponds to task executed at the  $k^{\text{th}}$  iteration of the loop, with  $0 \leq k < n$ .

The tasks (instructions) may be executed in parallel on an ILP processor. 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,  $V = \{T_0, \dots, T_{l-1}\}$ .
- $E$  is the set of edges representing precedence constraints (flow dependences or other serialization constraints). Any edge  $e = (T_i, T_j) \in E$  has a latency  $\delta(e) \in \mathbb{N}$  in terms of processor clock cycles and a distance  $\lambda(e) \in \mathbb{N}$  in terms of number of loop iterations. The distance  $\lambda(e)$  means that the edge  $e = (T_i, T_j)$  is a dependence between the task  $T\langle i, k \rangle$  and  $T\langle j, k + \lambda(e) \rangle$  for any  $k = 0, \dots, n - 1 - \lambda(e)$ .

We make a difference between tasks and precedence constraints depending whether they refer to data to be stored into registers or not

1.  $V_R$  is the set of tasks producing data to be stored into registers.
2.  $E_R$  is the set of flow dependence edges through registers. An edge  $e = (T_i, T_j) \in E_R$  means that the task  $T\langle i, k \rangle$  produces a result stored into a register 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:

$$Cons(T_i) = \{T_j \in V \mid e = (T_i, T_j) \in E_R\}$$

Fig. 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 into registers. Bold edges represent flow dependences (each sink of such edge reads/consumes the data produced by the source and stored in a register). Tasks that are not in bold circles are instructions that do not write into registers (write the data into memory or simply do not produce any data). Non-bold edges are other data or precedence constraints different from flow dependences. Every edge  $e$  in the DDG is labeled by the pair  $(\delta(e), \lambda(e))$ .

In our generic ILP processor model, we assume that the reading and writing from/into registers 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  $\delta_r$  and  $\delta_w$  in which

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

These two delays functions depend on the target processor and model almost all regular ILP hardware architectures (VLIW, EPIC/IA64 and superscalar processors). The next section recalls the definition of periodic task scheduling problem in case of one dimensional schedule.

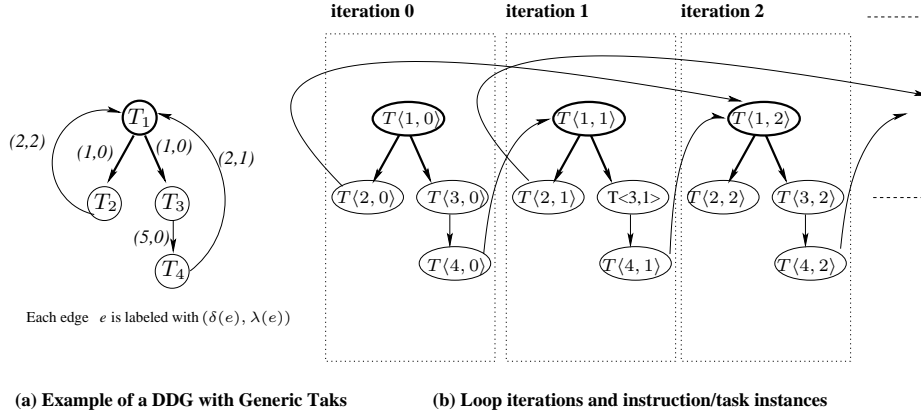


Figure 1: Example of Data Dependence Graphs with Recurrent Tasks

## 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  $\sigma$  which must at least satisfy the precedence constraints defined by the DDG  $G = (V, E, \delta, \lambda)$ :

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

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* schedules since our aim is to generate a final compact code (a loop). A periodic scheduling function  $\sigma$  is associated with a unique integral period  $p$  (to be computed). The execution 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 Equ. 1, we get new periodic scheduling constraints that should be satisfied by  $\sigma$ :

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

Classically, by adding all such inequalities over any cycle  $C$  of the DDG  $G$  we find that  $p$  must be greater than or equal to  $\max_C \left[ \frac{\sum_{e \in C} \delta(e)}{\sum_{e \in C} \lambda(e)} \right]$ , that we will denote in the sequel as the absolute Minimal Execution Period *MEP*. Computing *MEP* of a cyclic graph is a well known polynomial problem [2, 7]. 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 article, we study the problem of periodic scheduling under data dependence and storage constraints.

## 2.2 Storage Requirement

When considering a periodic schedule  $\sigma$  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  $k = 0, \dots, n - 1$ . 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:

$$d_\sigma(T\langle i, k \rangle) = \max_{e=(T_i, T_j) \in E_R} (\sigma_j + \delta_r(T_j) + (\lambda(e) + k) \times p) \quad (3)$$

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:

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

During this 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  $alive_\sigma^t$  be the set of alive data at time  $t \in \mathbb{N}$  when considering a schedule  $\sigma$

$$alive_\sigma^t = \{T\langle i, k \rangle | T_i \in V_R, 0 \leq i < l, 0 \leq k < n, t \in LT_\sigma(T\langle i, k \rangle)\}$$

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

$$SR_\sigma(G) = \max_{t \in \mathbb{N}} |\{alive_\sigma^t\}| \quad (4)$$

When considering unbounded resources and unbounded storage/register facilities, we can easily compute an optimal periodic schedule, *i.e.*, with a minimal execution period  $p = MEP$  [2]. We consider now a register pressure  $R$  (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 analyze here is the minimum  $p$  that can be expected for any periodic schedule using at most  $R$  registers. We will denote this value as  $MEP_R$  and we will try to understand the relationship between  $MEP_R$  and  $R$ . Let us start by an example to fix the ideas.

### 3 Motivating Example

We give in this section more intuitions about the new edges that we add between two tasks in order to restrict the parallelism and hence restrict the whole storage requirement. These edges represent possible register reuse between tasks. This can be viewed as a variant of general storage mapping functions [12, 13].

Let us consider the DDG of Fig. 2(a). The DDG of this loop contains only one flow dependence  $e = (T_1, T_2)$  with distance  $\lambda(e) = 3$ . If we have an unbounded number of registers, all iterations of this loop can be run in parallel since there is no cycle in the DDG. At each iteration  $k = 0, \dots, n - 1$ , the task  $T\langle 1, k \rangle$  writes into a new register. Now, let us assume that we only have  $R = 5$  available registers. The multiple instances of  $T_1$  can use only  $R = 5$  registers to periodically carry their data. In this case, the task  $T\langle 1, k + R \rangle$  writes into the same register previously used by  $T\langle 1, k \rangle$ . This fact creates a *storage* dependence from  $T\langle 2, k + \lambda(e) \rangle$  to  $T\langle 1, k + R \rangle$ ; this is equivalent to inserting a storage dependence in the DDG from  $T_2$  to  $T_1$  with a distance  $R - \lambda(e) = 2$ . Since the writing of  $T_1$  is delayed by  $\delta_w(T_1)$  time units and the reading of  $T_2$  is delayed by  $\delta_r(T_2)$  time units, the latency of the introduced storage dependence is set to  $\delta_r(T_2) - \delta_w(T_1)$ . Consequently the DDG becomes cyclic because of storage limitations (see Fig. 2(b) where the storage dependence is dashed). The introduced dependence, also called *Universal Occupancy*

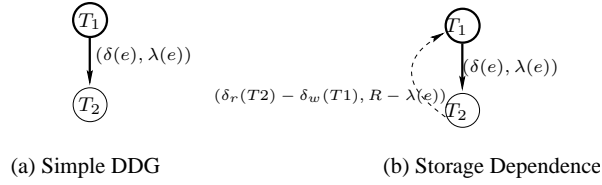


Figure 2: Simple Example

Vector [12], must in turn be counted when computing the new minimum execution period since a new cycle is created:

$$MEP_R \geq \frac{\delta(e) + \delta_r(T_2) - \delta_w(T_1)}{R}$$

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 register is freed. We propose a trick which defines for each task  $T_i$  a fictitious killing task  $K_i$ . We insert an edge 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 Fig. 3). The latency of this edge is set to  $\delta_r(T_j)$  because of the reading delay. We set its distance to  $-\lambda$ , where  $\lambda$  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 [8] without changing the fundamental mathematical problem.

Now a register/storage allocation scheme consists of defining the edges and the distances of reuse. That is, we define for each  $T_i$  the task  $T_j$  and iteration distance  $\mu_{i,j}$  such that  $T\langle j, k + \mu_{i,j} \rangle$  reuses the same destination register as  $T\langle i, k \rangle$ . This reuse creates a new storage edge/dependence from  $K_i$  to  $T_j$  with latency  $-\delta_w(T_j)$ . The distance of this edge is  $\mu_{i,j}$ , to be defined/computed. We will see in a further section that the storage requirement can be expressed in terms of  $\mu_{i,j}$ .

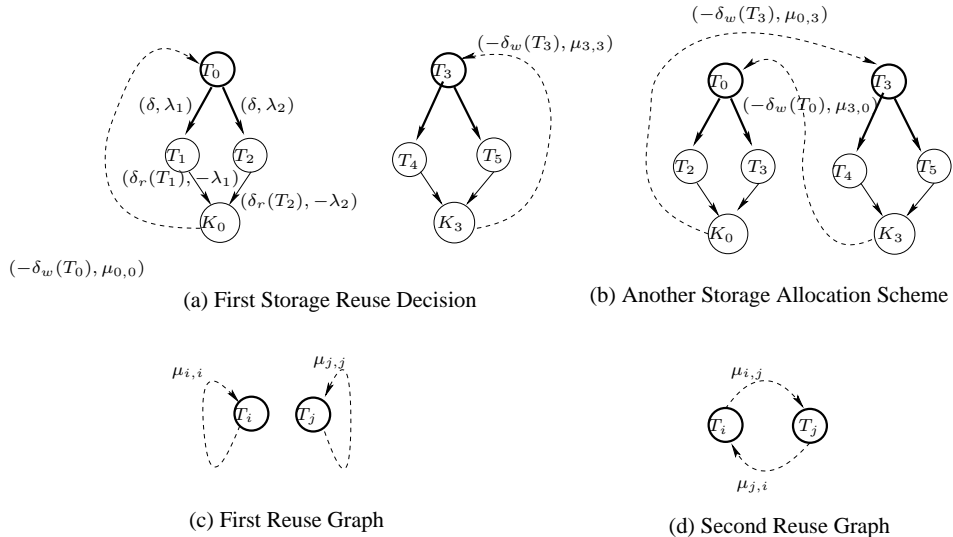


Figure 3: Killing Tasks and Reuse Graphs

Hence controlling register pressure means first determining which instruction should reuse the register killed by another instruction (*where should storage dependences be added?*). Secondly we have to determine data lifetimes or equivalently storage requirement (*how many iterations later ( $\mu$ ) should reuse occur?*)? As defined by the algebraic formulas of  $MEP$ , the lower is the  $\mu$  the lower is the register requirement but also the larger is the  $MEP_R$ .

Fig. 3(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  $K_0$  (resp.  $K_3$ ) to  $T_0$  (resp.  $T_3$ ) with an appropriate distance  $\mu$  as we will see later. Another reuse decision (see Fig. 3(b)) may be that the generic task  $T_0$  (resp.  $T_3$ ) reuses the register freed by  $T_3$  (resp.  $T_0$ ). This is illustrated by adding a storage dependence from  $K_0$  (resp.  $K_3$ ) to  $T_3$  (resp.  $T_0$ ). In both cases, the register pressure is equal to the sum of all  $\mu$  distances, but it is easy to see that the 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 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 number of registers used by any storage allocation scheme must be lower or equal to the number of available registers. Third and last, the critical ratio ( $MEP_R$ ) must be kept as lower as possible in order to save task parallelism. The next section gives a formal definition of the problem and provides an exact formulation.

## 4 Exact Problem Formulation with Graph Theory and Integer Programming

The storage reuse relationship between the generic tasks is described by defining a new graph called a *reuse graph*. Fig. 3(c) shows the first reuse decision where  $T_i$  (resp.  $T_j$ ) reuses the register used by itself  $\mu_{i,i}$  ( $\mu_{j,j}$  resp.) iterations earlier. Fig. 3(d) is the second reuse choice where  $T_i$  (resp.  $T_j$ ) reuses the register used by  $T_j$  (resp.  $T_i$ )  $\mu_{j,i}$  (resp.  $\mu_{i,j}$ ) 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*: Fig. 3(a) is the associated DDG with Fig. 3(c), and Fig. 3(b) is the one associated with Fig. 3(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.

### 4.1 Reuse Graphs

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

**Definition 1 (Reuse Graph)** Let  $G = (V, E, \delta, \lambda)$  be a DDG. The reuse graph  $G^r = (V_R, E_r, \mu)$  is defined by the set of tasks  $V_R$ , the set of edges representing storage reuse relationship, and edge distances  $\mu$ . Two tasks are connected in  $G^r$  by an edge  $e = (T_i, T_j)$  iff  $T\langle j, k + \mu(e) \rangle$  reuses the register freed by  $T\langle i, k \rangle$ .

We call  $E_r$  the set of reuse edges and  $\mu(e)$  a reuse distance. Given  $G^r = (V_R, E_r, \mu)$  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 edge. The distance of this dependence is  $\mu(e)$ . The introduction of these extra edges into  $G$  produces the DDG  $G_{\rightarrow r}$  associated with the reuse graph  $G^r$ .

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** [11] Let  $G^r = (V_R, E_r, \mu)$  be a valid reuse graph for a DDG  $G = (V, E, \delta, \lambda)$ . Then:

- the reuse graph consists of only elementary and disjointed cycles;
- any task  $T_i \in V_R$  belongs to a unique cycle 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  $\mu(G^r)$  registers, in which  $\mu(G^r)$  is the sum of all  $\mu$  distances in the reuse graph  $G^r$ .

**Theorem 1** [11] *Let  $G = (V, E, \delta, \lambda)$  be a DDG and  $G^r = (V_R, E_r, \mu)$  be a valid reuse graph. Then the reuse graph defines a periodic storage allocation with  $\mu(G^r)$  registers. Formally:*

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

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 Minimization (PSSM).

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

This problem is NP-complete [11]. In practice, the problem of register allocation is slightly different. The processor has  $\mathcal{R}$  a finite number of registers and we should find a time-optimal schedule such that the storage requirement is below the limit  $\mathcal{R}$ . In this case, the problem can be re-stated as : Find a valid reuse graph such that  $R = \mu(G^r) \leq \mathcal{R}$  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. The following section gives an integer linear formulation for the PSSM problem.

## 4.2 Exact Formulation for PSSM

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 ( $\implies$ ) by introducing binary variables [11].

### Basic Variables

- a schedule variable  $\sigma_i \geq 0$  for each task  $T_i \in V$ , including  $\sigma_{K_i}$  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  $(T_i, T_j) \in V_R^2$ . It is set to 1 iff  $(T_i, T_j)$  is a reuse edge;
- $\mu_{i,j}$  reuse distances for all  $(T_i, T_j) \in V_R^2$ .

### Linear Constraints

- Data dependences (see Equ. 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 edge:

$$\forall (T_i, T_j) \in V_R^2 : \theta_{i,j} = 1 \implies \sigma_{K_i} - \delta_w(T_j) \leq \sigma_j + p \times \mu_{i,j}$$

- 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.

$$\forall (T_i, T_j) \in V_R^2 : \theta_{i,j} = 0 \implies \mu_{i,j} = 0$$

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 minimize the storage requirement: Minimize  $\sum_{(T_i, T_j) \in V_R^2} \mu_{i,j}$ .

If we want to consider the real problem of periodic register allocation, it is sufficient to find a solution below  $\mathcal{R}$  the number of available registers, *i.e.*, we do not need to minimize the storage requirement at the lowest possible level. In this case, we can avoid defining an objective function, and we can simply add a constraint:  $\sum_{(T_i, T_j) \in V_R^2} \mu_{i,j} \leq \mathcal{R}$

## 5 Problem Simplification : Fixing Reuse Edges

Buffers [1, 5, 9] are FIFO storage facilities that transport data between repetitive 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, since 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. Therefore we consider now the complexity of our storage minimization problem when fixing reuse edges. This generalizes the buffer minimization approach. Formally, the problem can be stated as follows.

**Problem 2 (Fixed PSSM)** Let  $G^r = (V_R, E_r, \mu)$  an incomplete reuse graph with already fixed reuse edges, 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 execution period. Let  $E' \subseteq E$  be the set of already fixed storage dependences (which correspond to the reuse edges of  $G^r$ ). 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 have to be computed). The process of early fixing storage dependences greatly simplifies the integer linear program of Sect. 4.2. Consequently, the Fixed PSSM problem can be solved by the following integer program, assuming a given desired execution period  $p$ . Let first define the integer variables used in our exact formulation:

- a schedule variable  $\sigma_i \geq 0$  for each task  $T_i \in V$ , including  $\sigma_{K_i}$  for each killing node  $K_i$ . We assume a finite upper bound  $L$  for such schedule variables ( $L$  sufficiently large,  $L = \sigma_{e \in E} \delta(e)$ ).
- $\mu_{i,j}$  reuse distances for all fixed storage dependences  $(K_i, T_j) \in E'$ .

The following integer program solves the Fixed PSSM problem:

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

If we want to solve the dual problem, *i.e.*, to compute  $MEP_R$  the minimal execution period given a storage capacity  $R$ , it is sufficient to find a solution below  $R$  the number of available storage elements. In this case, we do not need to minimize 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 the upper-bound  $L$ .

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  $\mu' = p \times \mu$  and System 5 becomes:

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

**Theorem 2** [11] *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 [10]. 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  $\mu$ . If we ceil it by setting  $\mu = \lceil \frac{\mu'}{p} \rceil$ , a sub-optimal solution may result<sup>1</sup>.

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 ( $|V_R|$ ).

## 6 Conclusion

This article presents a new theoretical approach for periodic task scheduling with storage requirement optimization. Our theoretical framework is more generic than the existing ones, since it allows exact definition of the problem while considering delays in writing and reading 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 an execution period while minimizing the storage requirement, or we can fix the execution period while bounding the storage requirement. The dual problem of bounding the storage requirement while minimizing the execution period can be solved by a binary search on  $p$  (solving iteratively successive integer problems of PSSM).

Storage allocation is expressed in terms of reuse edges 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 tradeoff between task parallelism and number of storage locations.

Since computing an optimal periodic storage allocation is intractable in large data dependence graphs, we identified one approximate subproblem by fixing reuse edges and computing the reuse distances that minimize the overall storage requirement. We can use it in different ways, as setting self-reuse edges (buffers

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

minimization) or fixing arbitrary (or with a cleverer algorithm) other reuse cycles. This simplification allow us to consider DDGs with multiple hundreds of nodes, but do not define a polynomial instance of the original problem. A polynomial subproblem is obtained thanks to a non-bijective variable substitution  $\mu' = p \times \mu$ . This simplification allows us to consider huge DDGs, but induce an additional cost in terms of storage requirement (sub-optimal result).

## References

- [1] A. Munier Kordon and J.-B. Note (2005), A Buffer Minimization Problem for the Design of Embedded Systems, *European Journal of Operational Research* **164**(3), 669 – 679.
- [2] C. Hanen and A. Munier (1995), A Study of the Cyclic Scheduling Problem on Parallel Processors, *Discrete Applied Mathematics* **57**(2-3), 167 – 192.
- [3] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham (1996), Minimizing Register Requirements of a Modulo Schedule via Optimum Stage Scheduling, *International Journal of Parallel Programming* **24**(2), 103 – 132.
- [4] D. Fimmel and J. Muller (2001), Optimal Software Pipelining Under Resource Constraints, *International Journal of Foundations of Computer Science (IJFCS)* **12**(6), 697 – 718.
- [5] J. Korst (1992), *Periodic Multiprocessor Scheduling*. PhD thesis, Eindhoven University of Technology. The Netherlands.
- [6] J. Janssen (2001), *Compilers Strategies for Transport Triggered Architectures*, PhD thesis, Delft University, Netherlands.
- [7] E.L. Lawler (1972), Optimal Cycles on Graphs and Minimal Cost-to-Time Ratio Problem. In A. Marzollo, editor, *Periodic Optimization*, volume 1, 38 – 58, Springer-Verlag.
- [8] C.E. Leiserson and J.B. Saxe (1991), Retiming Synchronous Circuitry, *Algorithmica* **6**, 5 – 35.
- [9] Q. Ning and G.R. Gao (1993), A Novel Framework of Register Allocation for Software Pipelining, In *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 29 – 42, Charleston, South Carolina, ACM Press.
- [10] A. Schrijver (1987) *Theory of Linear and Integer Programming*, John Wiley and Sons, New York.
- [11] S.-A.-A. Touati (2002), *Register Pressure in Instruction Level Parallelisme*, PhD thesis, Université de Versailles, France, [ftp.inria.fr/INRIA/Projects/a3/touati/thesis](http://ftp.inria.fr/INRIA/Projects/a3/touati/thesis).
- [12] M. M. Strout, L. Carter, J. Ferrante, and B. Simon (1998), Schedule-Independent Storage Mapping for Loops, *ACM SIG-PLAN Notices* **33**(11), 24 – 33.
- [13] W. Thies, F. Vivien, J. Sheldon, and S. Amarasinghe (2001), A Unified Framework for Schedule and Storage Optimization, *ACM SIGPLAN Notices* **36**(5), 232 – 242.