SIRALINA: efficient two-steps heuristic for storage optimisation in single period task scheduling

Karine Deschinkel, Sid Touati, Sébastien Briais

SIRALINA: Efficient two-steps heuristic for storage optimisation in single period task scheduling

Karine Deschinkel and Sid-Ahmed-Ali Touati and Sébastien Briais
University of Versailles Saint-Quentin-en-Yvelines, France.

Abstract. In this paper, we study the general problem of one-dimensional periodic task scheduling under storage requirement, irrespective of machine constraints. We have already presented in (Touati and Eisenbeis, 2004) a theoretical framework that allows an optimal optimisation of periodic storage requirement in a cyclic schedule. Since our optimization problem is NP-hard (Touati, 2002), solving an exact integer linear programming formulation is too expensive in practice. In this article, we propose an efficient two-steps heuristic using model's properties that allows fast computation times while providing highly satisfactory results. This method includes the solution of an integer linear program with a totally unimodular constraints matrix in first step, then the solution of a linear assignment problem. Our heuristic is implemented for an industrial compiler for embedded VLIW processors.

Keywords: Cyclic Scheduling, Storage Requirement, Repetitive Tasks

1. Introduction

This article addresses the problem of storage optimisation in cyclic data dependence graphs (DDG), which is for instance applied in the practical problem of periodic register allocation for innermost loops on modern Instruction Level Parallelism (ILP) processors(Touati and Eisenbeis, 2004). The massive introduction of ILP processors since the last two decades makes us re-think better 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. Nowadays, 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 article, we do not assume any resource constraints (except storage requirement); Our aim is to optimise storage requirement (registers) with a fixed period value in a cyclic task scheduling problem, or to minimise the period value given a bounded storage requirement. Note that our problem of storage optimisation 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.).

This article is a continuation of our previous work on register allocation (Touati and Eisenbeis, 2004). In that paper, we provided an exact model and preliminary heuristics. Our previous heuristics were based on fixing reuse arcs (to be explained later) then to minimise the storage requirement. This way of approximating the exact model did not provide satisfactory results because: 1) Starting by first fixing reuse arcs before minimising the storage requirement seems an efficienct choice. 2) Although the reuse arcs are fixed, the problem remains combinatorial. So the current paper is an abstraction of our previous results on register optimisation. Furthermore, it extends it with a new efficient polynomial heuristic with full experimental results and analysis.

Our article is organised as follows. Section 2 presents some related work on the topic of register optimisation. Section 3 presents our task model and notations. Section 4 recalls the problem of periodic task scheduling with storage minimisation and writes an exact model with integer linear programming: our detailed experimental results on the optimal solution of this problem have been presented in (Touati and Eisenbeis, 2004; Touati, 2002). Since the exact model is not practical (too expensive in terms of computation time), our current article provides a new look by writing an efficient approximate method in Section 5, that we call *SIRALINA*. Sometimes, for engineering or copyright purposes, it is not allowed to have a linear solver inside a software. Thus, it is necessary to have a full algorithmic method (not based on linear programming). Consequently, we provide in Section 6 a network flow solution for SIRALINA. Before concluding, Section 7 presents the results of our experimental evaluation of SIRALINA, providing practical evidence of its efficiency and effectiveness.

Note that an initial version of this work has previously been published in (Deschinkel and Touati, 2008), the current article is an extended journal version: in addition to (Deschinkel and Touati, 2008), we provide more detailed explanations, a min-cost solution for our linear problem, a publicly released software and full experimental study on more than 9000 graph instances extracted from well known benchmarks (LAO, MEDIABENCH, SPEC2000 and SPEC2006).

2. Related Work

Register allocation in optimising compilers for instruction-level parallelism processors is an old important topic. Existing techniques in this field usually apply a periodic instruction scheduling under resource constraints 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 R values simultaneously alive. Usually they look for a schedule that minimises the stor-

age requirement under a fixed scheduling period while considering resource constraints (Eichenberger et al., 1996; Fimmel and Muller, 2001; Janssen, 2001; de Dinechin, 1996). Some fundamental results that analyse the trade-off between memory (register pressure) and parallelism in one-dimensional cyclic instruction schedules are published in (Touati, 2007).

The problem of considering resource constraints in conjunction with register constraints is the handling of *spilling*. Spilling is the process used when the number of existing storage locations (registers) is not sufficient. Then, some external additional storage (main memory) may be used by introducing new tasks (instructions) to make such data transfers. When we combine register constraints with resource constraints, spilling becomes a problem for which no satisfactory solution exists until now: this is because spilling may dynamically modify the DDG (by inserting new nodes) that is under scheduling process. Consequently, it is not guaranteed that the inserted spilling tasks can be scheduled under resource constraints, yielding to an iterative process of spilling followed by scheduling. Furthermore, in some embedded systems, spilling is not possible simply because no additional memory is present and all program variables should reside inside local registers.

In our approach, based on the theoretical framework presented in (Touati and Eisenbeis, 2004; Touati, 2002), we handle register constraints separately from resource constraints for one main reason: to efficiently control the register pressure in order to avoid spilling. In this paper, we satisfy register/storage constraints before instruction scheduling under resource constraints: we directly handle and modify the DDG in order to limit 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 (Strout et al., 1998; Thies et al., 2001).

Our theoretical framework (Touati and Eisenbeis, 2004) defines what we call *a reuse graph*. It is general enough to allow many variants for register optimisation methods: for instance, a variant may correspond to a particular shape of the reuse graph, or to a specific technique for computing it. It allows to model buffer optimisation problem (Ning and Gao, 1993), as well as rotating register files (Rau et al., 1992). Later, these two variants have been studied in (Touati and Eisenbeis, 2004) and will be compared to our new heuristic.

Compared to the existing work in the field of register optimisation on innermost loops, as far as we know, our research result holds all the following characteristics that are not present conjointly in previous articles:

1. We handle register optimisation independently of one-periodic task schedules.

2. We consider explicit delay access to registers.

3. Proved polynomial heuristics based on both linear programming and min cost network flow.

4. Full experiments on MEDIABENCH, LAO, SPEC2000 and SPEC2006 benchmarks, based on the rigorous performance evaluation methodology recommended in (Jain, 1991).

5. Source codes and data will be delivered publicly in (Briais and Touati, 2009).

Our current contribution adresses the problem of register minimisation in cyclic scheduling. There exists a dual problem called periodic register saturation (Touati and Mathe, 2009). Periodic register saturation tackles the problem of register maximisation instead of minimisation, which defines a distinct mathematical problem. In (Touati and Mathe, 2009), we provided the exact integer linear program that maximises the register requirement, but we did not succeed in defining an efficient polynomial algorithmic heuristic for cyclic register saturation. The current articles deals with an efficient polynomial heuristic for register minimisation, not maximisation.

3. Tasks Model

Our tasks model is similar to (Hanen and Munier, 1995). We consider a set of l generic tasks (instructions inside a program loop) $T_0, \ldots, T_{l-1}$. Each task T_i is 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 i 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 (*i.e.* consumed) by other tasks. The considered loop contains some data dependences represented with a graph $G(V, E)$ such that:

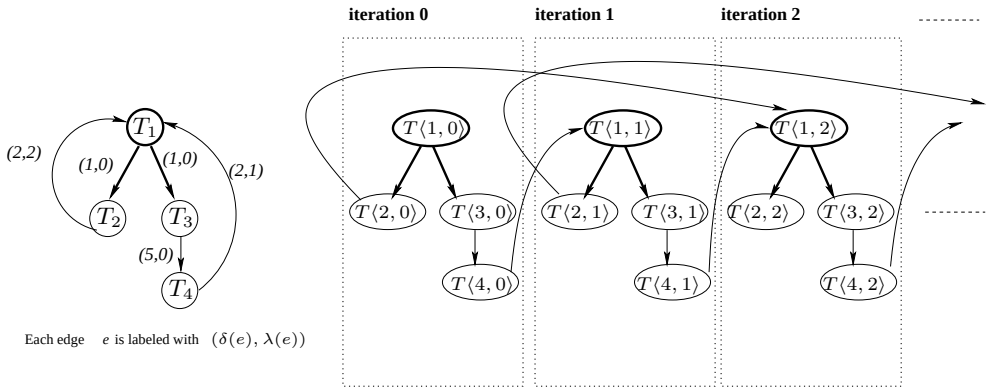
- V is the set of the generic tasks of the loop body, $V = \{T_0, \ldots, T_{l-1}\}$.
- E is the set of arcs representing precedence constraints (flow dependences or other serialisation constraints). Any arc $e = (T_i, T_j) \in E$ has a latency $\delta(e) \in \mathbb{N}$ in terms of processor clock cycles and a distance $\lambda(e) \in \mathbb{N}$ in terms of number of loop iterations. The distance $\lambda(e)$ means that the 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 $k = 0, \ldots, n - 1 - \lambda(e)$.

When dealing with storage constraints, we must make a difference between tasks and precedence constraints depending whether they refer to data to be stored into registers or not:

– $V^R \subseteq V$ is the set of tasks producing data to be stored into registers.

– $E^R \subseteq E$ is the set of flow dependence arcs through registers. An arc $e = (T_i, T_j) \in E^R$ 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:
$Cons(T_i) = \{T_j \in V \mid e = (T_i, T_j) \in E^R\}$

Note that a data dependence graph (DDG) is indeed a multi-graph; this means that more than one arc may exist between two tasks. 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 into registers. Bold arcs represent flow dependences (each sink of such arc 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 to store). Non-bold arcs are other data or precedence constraints different from flow dependences. Every arc e in the DDG is labeled by the pair $(\delta(e), \lambda(e))$.



(a) Example of a DDG with Generic Tasks (b) Loop Iterations and Instructions/Tasks Instances

Figure 1. Example of Data Dependence Graphs with Repetitive Tasks

In our generic 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 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).

In our task model, the number of task occurrences is unknown and unbounded. We could not easily consider any shape of scheduling functions, even if they meet the precedence 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 σ 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 at the expense of an unbounded 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 denote $\sigma_i = \sigma(T\langle i,0\rangle)$. The execution date of any other occurrence $T\langle i,k\rangle$ becomes equal to $\sigma(T\langle i,k\rangle) = \sigma_i + k \times p$. The classical periodic scheduling constraints that should be satisfied by σ are:

$$\forall e = (T_i, T_j) \in E : \sigma_i + \delta(e) \leq \sigma_j + \lambda(e) \times p \tag{1}$$

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. Computing MEP of a cyclic graph is a well known polynomial problem (Hanen and Munier, 1995; Lawler, 1972): since a DDG is computed by a dataflow analysis method, i.e. by the compiler, and since a DDG represents data dependences between the instructions of a program, the value of MEP always exists and is of positive value.

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, explained in the next section.

4. Recalls on the Exact Problem

In (Touati and Eisenbeis, 2004; Touati, 2002), we have introduced a graph theoretical framework that allows us to bound the storage requirement of any cyclic schedule of a DDG. This theoretical framework has multiple fundamental results on register allocation. In this section, we recall the notion of reuse graphs. Then we show the exact integer linear model for our main problem, that will be used later to provide an efficient heuristic.

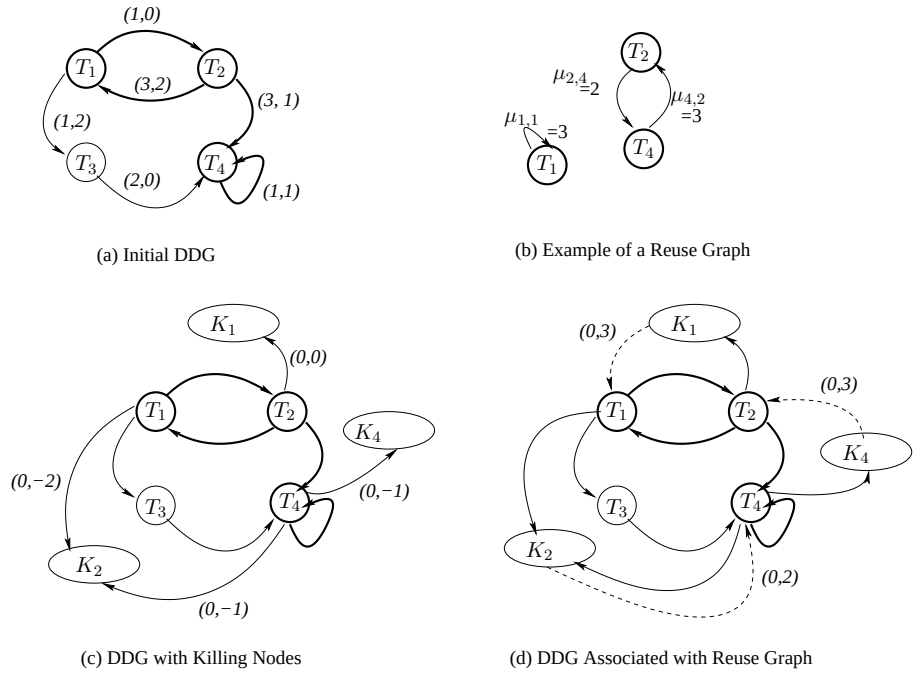
4.1. Reuse Graphs

A simple way to explain and recall the concept of reuse graphs is to provide an example. For formal definitions and results, please refer to (Touati and Eisenbeis, 2004; Touati, 2002). Figure 2(a) provides an initial DDG. The tasks producing results to be stored inside registers are in bold circles. Flow dependence through storage locations are in bold arcs. As an example, $Cons(T_2) = \{T_1, T_4\}$. Each arc e in the DDG is labeled with the pair of values $(\delta(e), \lambda(e))$. In this simple example, we assume that the delay of accessing registers is zero ($\delta_w = \delta_r = 0$). Now, the question is how to optimise the storage requirement for the repetitive tasks in Figure 2(a).

Periodic storage allocation is modeled thanks to *reuse graphs*: our purpose is to decide for storage location sharing, that is, which tasks share which registers. An example of a reuse graph G^{reuse} is given in Figure 2(b). A reuse graph G^{reuse} contains V^R, *i.e.*, only the nodes writing inside registers. These nodes are connected by *reuse arcs*. For instance, in G^{reuse}, there is a reuse arc (T_2, T_4). Each reuse arc (T_i, T_j) is labeled by an integral distance $\mu_{i,j}$. The existence of a reuse arc (T_i, T_j) of distance $\mu_{i,j}$ means that the two task instances $T\langle i, k\rangle$ and $T\langle j, k + \mu_{i,j}\rangle$ *share the same register* as destination for holding their respective results. Hence, reuse graphs allow us to completely define a periodic storage allocation for a given DDG.

In order to be valid, reuse graphs should satisfy two main constraints (Touati and Eisenbeis, 2004; Touati, 2002): 1) They must describe a bijection between the nodes; that is, they must be composed of elementary and disjoint circuits. 2) The *associated DDG* must be schedulable, *i.e.*, it has at least one valid cyclic schedule with a period p.

Now, let us describe what we mean by the DDG *associated with* a reuse graph. Once a reuse graph is fixed, say the reuse graph in Figure 2(b), the periodic storage allocation creates new scheduling constraints between tasks. These new scheduling constraints result from the fact that some tasks share



(a) Initial DDG

(b) Example of a Reuse Graph

(c) DDG with Killing Nodes

(d) DDG Associated with Reuse Graph

Figure 2. Reuse Graphs and DDG Associated to Them

the same storage unit (register). Since each reuse arc (T_i, T_j) in the reuse graph G^{reuse} describes a register sharing between $T\langle i, k\rangle$ and $T\langle j, k + \mu_{i,j}\rangle$, we must guarantee that $T\langle j, k + \mu_{i,j}\rangle$ writes inside the same register after the *killing* of the result of $T\langle i, k\rangle$. We say that the task of a result is killed when all their consumers have already read that data, consequently it is no longer necessary to hold it inside a register. Any last reader of a data is called as its *killer*. If the DDG is already scheduled, then it is easy to compute the killing date of each data (the killing date is the date when a date is killed, when it is read by all the consumers). However, if the DDG is not already scheduled as in our case, then the killing date is not known and hence we must be able to guarantee the validity of the storage allocation for all possible periodic schedules. This is done by creating *the associated DDG* with the reuse graph. This DDG is an extension of the initial one, done in two steps:

1. Firstly, we introduce dummy nodes representing the killing dates of all values (de Dinechin, 1996): the killing node K_i for a task $T_i \in V^R$ represents the virtual last consumer of T_i. The killing node K_i must always be scheduled after all T_i's consumers. Consequently, we add the set of arcs $\{(T_j, K_i)|T_j \in Cons(T_i)\}$. Figure 2(c) illustrates the DDG after adding all the killing nodes. For each added arc $e = (T_j, K_i)$, we set its latency

to $\delta(e) = \delta_r(T_j)$ and its distance to $-\lambda$, where λ is the distance of the arc $(T_i, T_j) \in E^R$. As explained in (Touati and Eisenbeis, 2004; Touati, 2002), this negative distance is a mathematical convention, it simplifies our mathematical formula and does not influence the fundamental results on reuse graphs.

2. Secondly, once all killing nodes are inserted, we are able to introduce new *reuse arcs* resulted by periodic register allocation. For each reuse arc (T_i, T_j) in G^{reuse}, we add a reuse arc $e = (K_i, T_j)$ in the associated DDG. This added arc has a latency equal to $\delta(e) = -\delta_w(T_j)$ and has a distance equal to $\lambda(e) = \mu_{i,j}$. Figure 2(d) illustrates the DDG associated with the reuse graphs of Figure 2(b), where the introduced reuse arcs are in dashed lines. The reader may notice that the critical circuits of the DDG in Figure 2(a) and (d) are identical and equal to $MEP = \frac{4}{2} = 2$ (a critical circuit is (T_1, T_2)).

As explained above, computing a reuse graph implies the creation of new arcs with μ distances. We proved in (Touati and Eisenbeis, 2004; Touati, 2002) that if a reuse graph G^{reuse} is valid, then it describes a periodic register allocation with exactly $\sum \mu_{i,j}$ registers. The reverse is also true: if a cyclic schedule is fixed with a minimal $\sum \mu_{i,j}$, then it has been proved in (de Werra et al., 1999) that a valid periodic register allocation exists with $\sum \mu_{i,j}$ registers. Now the central storage optimisation problem is to compute a valid reuse graph with a minimal $\sum \mu_{i,j}$, with a minimal period $p = MEP$. This is an NP-hard problem (Touati, 2002). Minimising the storage requirement while guaranteeing a minimal period p allows for instance to decide the number of registers to put inside a processor. In some cases, the number R of available registers has already been decided by processor designers. The problem becomes then to find a valid reuse graph such that $\sum \mu_{i,j} \leq R$ with a minimal period. Our method becomes then iterative: we solve the following integer linear problem for a fixed period (starting from $p = MEP$) and we increment it iteratively until we get $\sum \mu_{i,j} \leq R$. If the value of the period hits an upper limit $p = L$ without having $\sum \mu_{i,j} \leq R$, then we say that no solution is found with R the number of available registers. Introducing spilling tasks to use an external memory is outside the scope of our study. Note that a binary search of p (between MEP and L) can also be used instead of an iterative search, but upon the condition that $\sum \mu_{i,j}$ must be minimal (Touati, 2007). This is fortunately the case if we solve the exact integer formulation to optimality, but may not be the case if we use a heuristic or an approximate solution. In this situation, an iterative search on p is recommended.

4.2. EXACT PROBLEM FORMULATION

This section recalls the exact integer linear model for solving the problem of periodic scheduling with storage minimisation. It is built for a fixed desired period $p \in \mathbb{N}$.

4.2.1. *Basic Variables*

- A schedule variable $\sigma_i \in \mathbb{N}$ 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, for instance $L = \sum_{e \in E} \delta(e)$); As our scheduling is periodic, we only consider the integer execution date of the first task occurrence $\sigma_i = \sigma(T\langle i, 0\rangle)$ and the execution date of any other occurrence $T\langle i, k\rangle$ becomes equal to $\sigma(T\langle i, k\rangle) = \sigma_i + k \times p$.
- A binary variable $\theta_{i,j}$ 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;
- A reuse distance $\mu_{i,j} \in \mathbb{N}$ for each pair of tasks $(T_i, T_j) \in V^R \times V^R$;

4.2.2. *Linear Constraints*

- **Data dependences**
  The schedule must at least satisfy the precedence constraints defined by the DDG.

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

- **Flow dependences**
  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 $\sigma_j + \delta_r(T_j) + (\lambda(e) + k) \times p$. Then, we must schedule the killing node K_i of the task T_i after all T_i's consumers. $\forall T_i \in V^R$, $\forall T_j \in Cons(T_i) | e = (T_i, T_j) \in E^R$:

$$\sigma_{K_i} \geq \sigma_j + \delta_r(T_j) + p \times \lambda(e) \tag{3}$$

- **Storage dependences**
  There is a storage dependence between K_i and T_j if (T_i, T_j) is a reuse arc. $\forall (T_i, T_j) \in V^R \times V^R$:

$$\theta_{i,j} = 1 \Longrightarrow \sigma_{K_i} - \delta_w(T_j) \leq \sigma_j + p \times \mu_{i,j}$$

  This involvement can result in the following inequality: $\forall (T_i, T_j) \in V^R \times V^R$,

$$\sigma_j - \sigma_{K_i} + p \times \mu_{i,j} + M_1 \times (1 - \theta_{i,j}) \geq -\delta_w(T_j) \tag{4}$$

where M_1 is an arbitrarily large constant.
If there is no register reuse between two tasks T_i and T_j, then $\theta_{i,j} = 0$ and the storage dependence distance $\mu_{i,j}$ must be set to 0 in order to not be accumulated in the objective function.

$$\forall (T_i, T_j) \in V^R \times V^R : \ \mu_{i,j} \leq M_2 \times \theta_{i,j} \tag{5}$$

where M_2 is an arbitrarily large constant.

– **Reuse Relations**
The reuse relation must be a bijection from V^R to V^R. A register can be reused by one task and a task can reuse one released register:

$$\forall T_i \in V^R : \sum_{T_j \in V^R} \theta_{i,j} = 1 \tag{6}$$

$$\forall T_j \in V^R : \sum_{T_i \in V^R} \theta_{i,j} = 1 \tag{7}$$

4.2.3. *Objective Function*
As proved in (Touati and Eisenbeis, 2004), the storage requirement has a reachable upper limit equal to $\sum \mu_{i,j}$. In our periodic scheduling problem, we want to minimise the storage requirement:

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

Using the above integer linear program to solve our NP-hard problem is not efficient in practice. With a classical Branch and Bound method, we are only able to solve small instances (DDG sizes), in practice around 12 nodes. For this reason, we suggest to make use of the problem structure to propose an efficient heuristic as follows.

5. SIRALINA : A Two Steps Heuristic

Our solution strategy is based on the analysis of the model constraints. As the problem involves scheduling constraints and assignment constraints, and the reuse distances are the link between these two sets of constraints, we attempt to decompose the problem into two sub-problems :

– A schedule problem : to find a scheduling for which the potential reuse distances are as small as possible.

– An assignment problem : to select which pairs of tasks do share the same register.

5.1. PRELIMINARIES

If a pair of tasks $(T_i, T_j) \in V^R \times V^R$ represents a reuse arc, its reuse distance must satisfy the inequality given by (4), where $\theta_{i,j} = 1$. This inequality gives a lower bound for each reuse distance. If $(T_i, T_j) \in V^R \times V^R$ is a reuse arc (E^{reuse} denotes the set of reuse arcs) then :

$$\forall (T_i, T_j) \in E^{reuse}, \mu_{i,j} \geq \frac{1}{p}(\sigma_{K_i} - \delta_w(T_j) - \sigma_j) \tag{8}$$

If $(T_i, T_j) \in V^R \times V^R$ is not a reuse arc then $\mu_{i,j} = 0$ according to the inequality (5).

$$\forall (T_i, T_j) \notin E^{reuse} : \ \mu_{i,j} = 0$$

The aggregation of constraints (8) for each reuse arc provides a lower bound of the objective function value

$$z = \sum_{(T_i,T_j)\in V^R\times V^R} \mu_{i,j} \geq \frac{1}{p}(\sum_{(T_i,T_j)\in E^{reuse}} \sigma_{K_i} - \delta_w(T_j) - \sigma_j)$$

As the reuse relation is a bijection from V^R to V^R, the sum in the right hand side of the above inequality can be separated into two parts as follows:

$$\sum_{(T_i,T_j)\in E^{reuse}} \sigma_{K_i} - \delta_w(T_j) - \sigma_j$$

$$= \sum_{i\in V^R} \sigma_{K_i} - \sum_{j\in V^R} (\delta_w(T_j) + \sigma_j)$$

$$= \left(\sum_{i\in V^R} \sigma_{K_i} - \sum_{j\in V^R} \sigma_j \right) - \sum_{j\in V^R} \delta_w(T_j)$$

We deduce from this inequality a lower bound on the number of required registers. In this context, it may be useful to find an appropriate schedule for which this value is minimum. As $\sum_{j\in V^R} \delta_w(T_j)$ is a constant for the problem, we can ignore it in the following optimisation problem. We consider *the schedule problem (P)*:

$$\begin{cases} \min \sum_{i\in V^R} \sigma_{K_i} - \sum_{j\in V^R} \sigma_j \\ \text{subject to :} \\ \sigma_j - \sigma_i \geq \delta(e) - p \times \lambda(e), & \forall e = (T_i, T_j) \in E \\ \sigma_{K_i} - \sigma_j \geq \delta_r(T_j) + p \times \lambda(e), & \forall e = (T_i, T_j) \in E^R \end{cases} \tag{9}$$

The constraints matrix of System (9) is clearly an incidence matrix (Schrijver, 1987) of the DDG augmented with killing nodes (see Figure 2(c)). The number of integer variables is $O(|V|)$ and the number of linear constraints is $O(|E|)$. Since it is an incidence matrix, System (9) 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 (Schrijver, 1987). For instance, the ellipsoid method can compute the optimal feasible integral solution in $O(|V|^3(|V|^2 + |E|))$. However, in practice, the simplex method can also be applied for totally unimodular constraints matrix, since it computes optimal values σ_i^* for each task $T_i \in V^R$ and the optimal values $\sigma_{K_i}^*$ for each killing node K_i. While it is a pseudo-polynomial method in theory, it is really fast in practice (a well known observation). We will confirm this fact later in our experimental section, and we will show that it is even faster than a polynomial network-flow implementation.

Once the scheduling variables have been fixed, the minimal value of each potential reuse distance would be equal to $\overline{\mu_{i,j}} = \lceil \frac{\sigma_{K_i}^* - \delta_w(T_j) - \sigma_j^*}{p} \rceil$ according to inequality (8). Knowing the reuse distance values $\overline{\mu_{i,j}}$ if T_j reuses the register freed by T_i, the storage allocation which consists of choosing which task reuses which released register can be modeled as a linear assignment problem. We consider *the linear assignment problem (A)*:

$$\begin{cases} \min \quad \sum_{(T_i,T_j)\in V^R\times V^R} \overline{\mu_{i,j}} \times \theta_{i,j} & \\ \text{Subject to} & \\ \sum_{T_j\in V^R} \theta_{i,j} = 1, & \forall T_i \in V^R \\ \sum_{T_i\in V^R} \theta_{i,j} = 1, & \forall T_j \in V^R \\ \theta_{i,j} \in \{0,1\} & \end{cases} \tag{10}$$

where $\overline{\mu_{i,j}}$ is a fixed value for each arc $e = (T_i, T_j) \in V^R \times V^R$.

5.2. SUMMARY OF SIRALINA HEURISTIC

Given a fixed period p, we suggest to solve the storage minimisation problem with the following heuristic:

- Solve the problem (P) to deduce the optimal values σ_i^* for each task $T_i \in V^R$ and the optimal values $\sigma_{K_i}^*$ for each killing node K_i.
- Compute the cost $\overline{\mu_{i,j}} = \left\lceil \frac{\sigma_{K_i}^* - \delta_w(T_j) - \sigma_j^*}{p} \right\rceil$ for each pair of tasks $(T_i, T_j) \in V^R \times V^R$.
- Solve the linear assignment problem (A) with the Hungarian algorithm (Kuhn, 1955) which solves assignment problems in polynomial time ($O(|V^R|^3)$) to deduce the optimal values $\theta_{i,j}^*$.

– If $\theta^*_{i,j} = 1$ for the pair $(T_i, T_j) \in V^R \times V^R$, then (T_i, T_j) becomes a reuse arc with a reuse distance equal to $\mu_{i,j} = \overline{\mu_{i,j}}$.

This section presented a two steps heuristic for the storage minimisation problem with a fixed period p. The first step requires to solve a linear program. However, in practice, some compilers are not allowed to embed a linear solver for engineering or copyright reasons. So, it is important to provide appropriate algorithms that we can implement independently of a linear solver. The next section shows how to write our schedule problem as a network flow problem in order to use an algorithmic method.

6. Network Flow Formulation of SIRALINA

The first step of SIRALINA solves the schedule problem (P), a problem of linear programming to get the values for the σ variables and potential values for the μ variables. The second step executes the Hungarian algorithm to get the assignment of reuse arcs. The second method is already an algorithmic one, we focus in this section on writing an algorithmic method for the first step only. For the purpose of this section, we note by E^k the set of all arcs going from consumers to killing nodes:

$$E^k = \{e = (T_j, K_i) | T_j \in Cons(T_i)\}$$

Let us consider the schedule problem (P) formally written by System (9). In order to write the dual problem of (P), let name the right hand sides of the inequalities:

$$\begin{array}{ll} a(e) = \delta(e) - p \times \lambda(e), & \forall e = (T_i, T_j) \in E \\ b(e) = \delta_r(T_j) + p \times \lambda(e), & \forall e = (T_j, K_i) \in E^k \end{array}$$

For the purpose of writing the dual problem, we assume that σ variables are of arbitrary sign, that is $\sigma \in \mathbb{Z}$. Since σ defines a periodic scheduling function, its sign is not important since we can always shift the scheduling date with a constant factor. This allows us to write the schedule problem as:

$$\left\{ \begin{array}{ll} \min & (\sum_{i \in V^R} 1 \times \sigma_{K_i} \\ & - \sum_{j \in V^R} 1 \times \sigma_j \\ & - \sum_{j \in V \setminus V^R} 0 \times \sigma_j) \\ \text{subject to:} & \\ \sigma_j - \sigma_i \geq a(e), & \forall e = (T_i, T_j) \in E \\ \sigma_{K_i} - \sigma_j \geq b(e), & \forall e = (T_j, K_i) \in E^k \\ \sigma \in \mathbb{Z} & \end{array} \right. \quad (11)$$

This system defines the classical periodic scheduling problem with only precedence constraints: it is feasible if and only if $p \geq MEP$ (Hanen and Munier,

1995). Assuming that $\sigma \in \mathbb{Z}$ simplifies the writing of the dual problem of P since we can now have a dual problem with equalities instead of inequalities. We declare an integral dual variable $f(e) \in \mathbb{N}$ for each linear constraint on $e \in E \cup E^k$ in the primal problem. The coefficients of the primal objective function become the right hand sides of the equalities in the dual problem. The right hand sides of the primal problem become the coefficients of the dual objective function. And for each primal variable $\sigma \in \mathbb{Z}$, we have a constraint in the dual problem as follows[1] :

– Dual constraints for primal variables σ_{K_i} ($T_i \in V^R$):

$$\forall T_i \in V^R, \sum_{?\overset{e\in E^k}{\longrightarrow} K_i} f(e) = 1$$

– Dual constraints for primal variables σ_j ($T_j \in V^R$):

$$\forall T_j \in V^R, \sum_{?\overset{e\in E}{\longrightarrow} j} f(e) - \sum_{j\overset{e\in E}{\longrightarrow} ?} f(e) - \sum_{j\overset{e\in E^k}{\longrightarrow} K_?} f(e) = -1$$

– Dual constraints for primal variables σ_j ($T_j \in V \setminus V^R$):

$$\forall T_j \in V \setminus V^R, \sum_{?\overset{e\in E}{\longrightarrow} j} f(e) - \sum_{j\overset{e\in E}{\longrightarrow} ?} f(e) - \sum_{j\overset{e\in E^k}{\longrightarrow} K_?} f(e) = 0$$

– The dual objective function is:

$$\max \sum_{e\in E} a(e) \times f(e) + \sum_{e\in E^k} b(e) \times f(e)$$

In the following section, we prove that this dual problem can be solved as a min-cost-flow problem.

6.1. Minimal Cost Network Flow Solution for the Dual Problem

In this section, we prove that the dual problem defined above represents a network flow problem. The network is simply the initial data dependence graph (DDG) augmented with the killing nodes K_i and the killing arcs E^k. Note that in the initial DDG, there is no arc exiting from the killing nodes K_i. Now, let us examine each constraint of the dual problem to check whether it represents flow constraints:

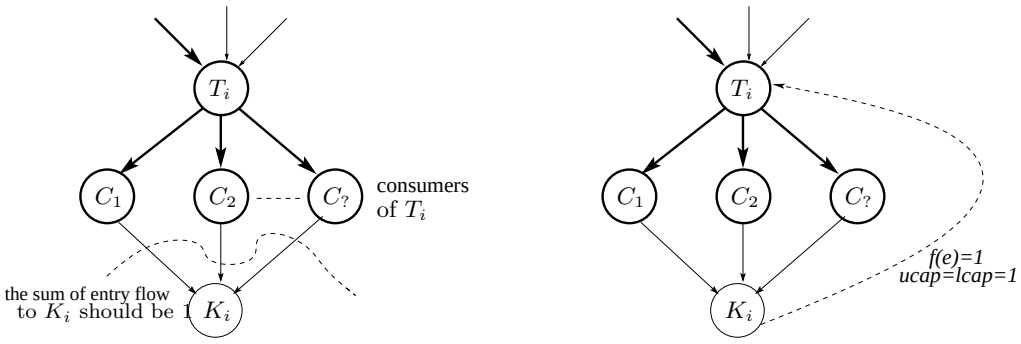
[1] In these constraints, we use the notation of '?' used in graph theory to note an arbitrary node or value: for instance $? \xrightarrow{e} j$ means any arc named e finishing at node j.

6.1.1. *Dual constraints for primal variables σ_{K_i} ($T_i \in V^R$)*

$$\forall T_i \in V^R, \sum_{?\xrightarrow{e\in E^k} K_i} f(e) = 1$$

This set of constraints imposes that the input flow to any node K_i must be equal to 1. Figure 3(a) illustrates this fact in the network. In order to satisfy this constraint, we add a fictitious arc (that we remove afterwards) between K_i and $T_i \in V^R$. Then, we can put a minimal and a maximal flow capacity to this arc as equal to 1, see Figure 3(b). Note that this added arc (K_i, T_i) becomes the unique arc exiting from K_i in the network. The cost of this arc is set to zero. Thanks to the set of new arcs, that we note by E^f, the constraints for primal variables σ_{K_i} represent network flow constraints by writing:

$$\forall T_i \in V^R, \sum_{?\xrightarrow{e\in E^k} K_i} f(e) - \sum_{K_i\xrightarrow{e\in E^f} ?} f(e) = 0$$



(a) Constraints on entry flow to K_i (b) Adding fictitious arcs to impose the flow to be equal to 1

Figure 3. Flow Constraints on Killing Nodes

6.1.2. *Dual constraints for primal variables σ_j ($T_j \in V^R$)*

$$\forall T_j \in V^R, \sum_{?\xrightarrow{e\in E} j} f(e) - \sum_{j\xrightarrow{e\in E} ?} f(e) - \sum_{j\xrightarrow{e\in E^k} K_?} f(e) = -1$$

$$\Longrightarrow \sum_{?\xrightarrow{e\in E} j} f(e) + 1 - \sum_{j\xrightarrow{e\in E} ?} f(e) - \sum_{j\xrightarrow{e\in E^k} K_?} f(e) = 0$$

For the node $T_j \in V^R$, we know that the flow between K_j and T_j is equal to one (see the previous paragraph, Figure 3(b)). Thus, we can write the following flow constraints:

$$\forall T_j \in V^R, \sum_{?\overset{e\in E\cup E^f\cup E^k}{\longrightarrow} j} f(e) - \sum_{j\overset{e\in E\cup E^f\cup E^k}{\longrightarrow} ?} f(e) = 0$$

6.1.3. *Dual constraints for primal variables σ_j ($T_j \in V \setminus V^R$)*

$$\forall T_j \in V \setminus V^R, \sum_{?\overset{e\in E}{\longrightarrow} j} f(e) - \sum_{j\overset{e\in E}{\longrightarrow} ?} f(e) - \sum_{j\overset{e\in E^k}{\longrightarrow} K_?} f(e) = 0$$

Since no arcs belonging to E^f is entering to T_j, we can write as the following flow constraints:

$$\forall T_j \in V \setminus V^R, \sum_{?\overset{e\in E\cup E^f\cup E^k}{\longrightarrow} j} f(e) - \sum_{j\overset{e\in E\cup E^f\cup E^k}{\longrightarrow} ?} f(e) = 0$$

6.1.4. *Objective function*

Since we added a set $E^f = \{e = (K_i, T_i) | T_i \in V^R\}$ of fictitious arcs, the objective function becomes

$$\max \sum_{e\in E} a(e) \times f(e) + \sum_{e\in E^k} b(e) \times f(e) + \sum_{e\in E^f} 0 \times f(e)$$

In order to have a min-cost-flow problem, it is sufficient to invert the signs of the costs as:

$$\min \sum_{e\in E} (-a(e)) \times f(e) + \sum_{e\in E^k} (-b(e)) \times f(e) + \sum_{e\in E^f} 0 \times f(e)$$

From above, we deduce that our dual problem is a min-cost-flow problem. We summarise it as follows. The network is the initial DDG augmented with killing nodes K_i, killing arcs $E^k = \{e = (T_j, K_i) | T_j \in Cons(T_i)\}$ and fictitious arcs $E^f = \{e = (K_i, T_i) | T_i \in V^R\}$. We declare an integral flow $f(e)$ on each arc $e \in E \cup E^k \cup E^f$. The lower flow capacities of all arcs are equal to zero, except those of the arcs in E^f. The upper flow capacities of all arcs are unbounded, except for the arcs in E^f. For the set E^f, lower and upper flow capacities are set to 1. This min-cost network flow problem can be solved by many polynomial algorithms (Ahuja and ans James B. Orlin, 1993).

Once the flow with minimal cost computed, we should be able to compute the values of the primal variable, as explained in the next section.

6.2. Back Substitution to the Initial Scheduling Problem (P)

Applying a min-cost flow algorithm computes optimal flow values $f^*(e), \forall e \in E \cup E^k$ (the arcs in E^f can be removed now). In order to retrieve the values of the σ variables in the primal scheduling problem, we use the primal-dual relationship as follows:

- If $f^*(e) > 0$ then the corresponding constraint in the primal scheduling problem becomes an equality constraint.
- If $f^*(e) = 0$ then the corresponding constraint in the primal scheduling problem becomes an inequality constraint.

Formally, it yields the following solutions:

- If $e = (T_j, T_j) \in E$ then:
  - If $f^*(e) > 0$ then $\sigma_j - \sigma_i = \delta(e) - p \times \lambda(e) \Longrightarrow \sigma_j = \sigma_i +$ a(e).
  - If $f^*(e) = 0$ then $\sigma_j - \sigma_i \geq \delta(e) - p \times \lambda(e)$.
- If $e = (K_i, T_j) \in E^k$ then:
  - If $f^*(e) > 0$ then $\sigma_{K_i} - \sigma_j = \delta_r(T_j) + p \times \lambda(e) \Longrightarrow \sigma_{K_i} = \sigma_j +$ b(e).
  - If $f^*(e) = 0$ then $\sigma_{K_i} - \sigma_j \geq \delta_r(T_j) + p \times \lambda(e)$.

The above set of equalities describes a simple potential problem on the DDG (page 316 of (Ahuja and ans James B. Orlin, 1993)). Here, the potential function is perfectly defined by the function σ. Recall that the potential of a graph is equivalent to computing the longest path between a single source to the remaining nodes. It can be solved by Bellman-Ford algorithm with a complexity equal to $O(|E| \times |V|)$ (Cormen et al., 1990).

7. Experimental Evaluation

In this section, we present the results of extensive computational experiments conducted on thousands of DDG extracted from many well known benchmarks. Our benchmarks are a selection of five applications families: two families of embedded applications (MEDIABENCH and LAO) and three families of general purpose and intensive computations(SPEC2000-CINT, SPEC2000-CFP and SPEC2006). These applications consist of many program files written in C and C++ languages to be optimised by compilation. The SIRALINA method has been incorporated within an industrial compilation framework from STmicroelectroncis. We extract all the loop (DDG) of

the benchmarks to optimise their storage requirement with SIRALINA. This section describes our experimental environment, then presents the evaluation results.

7.1. FRAMEWORK

In order to evaluate SIRALINA, we compare it with two existing well known heuristics (named as Method 3 and Method 5), already studied in (Touati and Eisenbeis, 2004):

- Method 3 is a variant of buffer optimisation as studied in (Ning and Gao, 1993). It starts by first pre-fixing reuse arcs (K_i, T_i), $\forall T_i \in V^R$, then it minimises $\sum \mu_{i,i}$. That is, this method sets that each task reuses the register freed by itself (no register sharing between distinct tasks). The date dependence constraints of the DDG always guarantees that $\mu_{i,i} \geq 1$, $\forall T_i \in V^R$, then $\sum \mu_{i,i} \geq |V^R|$. We clearly see that the minimal storage requirement of method 3 is always limited by the size of V^R, but method 3 executes faster than method 5 explained below.

- Method 5 is a variant of (Rau et al., 1992). It starts by first pre-fixing an arbitrary Hamiltonian reuse circuit, then it minimises $\sum \mu_{i,j}$. This method allows register sharing between the tasks, yielding a lower register requirement than that found by method 3, but requires higher computation times in practice.

We also compare SIRALINA results against the ones obtained by the solution of the exact integer linear model with a branch and bound method. Since computing an optimal periodic storage allocation is intractable for large DDG (larger than 12 nodes), we have limited the computation time of the exact method by 10 seconds: this time can be chosen by the user of the compiler; in our case, we select 10 seconds because it is a maximal time allowed for compiling a single loop as usually done in embedded systems area in an offline compilation process. The value of the solution is then the best solution of the objective function found within 10 seconds: this defines a naive heuristic based on the optimal integer linear model. This naive heuristic, that we call Method 1, is the least competitor that any new *clever* heuristic should beat. Note that our SIRALINA heuristic always succeed in finishing less than 1 second without fixing any timeout.

We have done extensive set of experiments on both high performance and embedded benchmarks. The total number of experimented DDG is 6748. The sizes of the DDG (counted as the number of nodes and arcs) are illustrated by boxplots in Figure 4. A boxplot (Crawley, 2007) is a convenient way to graphically depict group of numerical data through their five-number summaries (the smallest data, lower quartile($Q1 = 25\%$), median($Q2 = 25\%$), upper

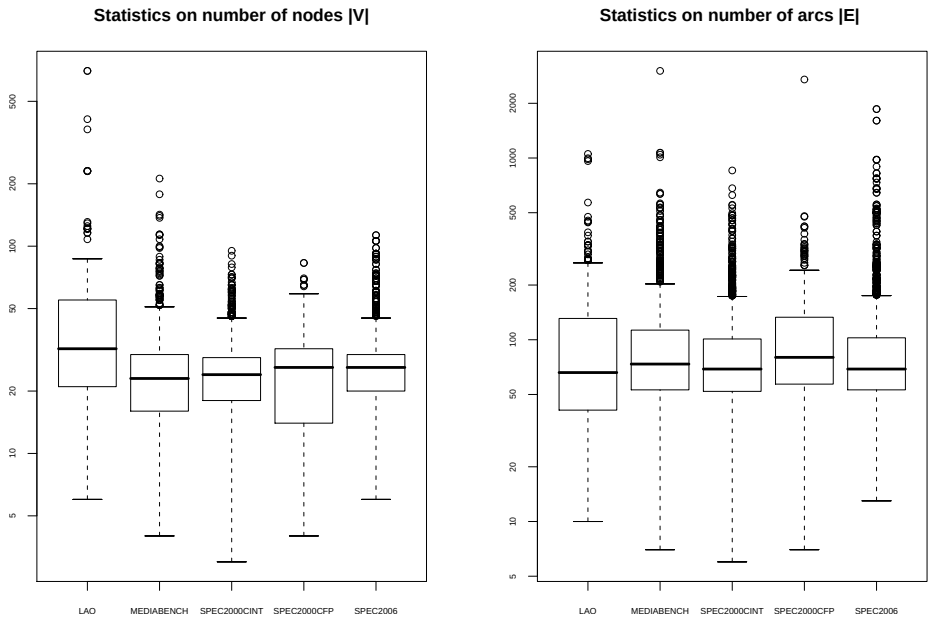


Figure 4. DDG Sizes

quartile($Q3 = 75\%$), and largest data). The left part of Figure 4 represents the numbers of nodes per benchmark family, whereas the right part represents the numbers of arcs. In the x-axis, we plot each benchmark family (LAO, MEDIABENCH, SPEC, etc.); the y-axis presents the data where: 1) the first small horizontal line is the minimal value, 2) the rectangle represents the first and the third quartiles, 3) the small horizontal line cutting the rectangle represents the median, 4) the last upper point represents the maximal value. We can remark for instance that 50% of the experimented DDG have between 15 and 30 nodes, and between 50 and 100 arcs. We also remark that the size of extreme DDG is greater than 500 nodes and 2000 arcs.

Our experiments are conducted to optimise registers (storage locations) inside a VLIW embedded processor (ST231 processor family). This family of processors have two types of registers BR and GR. Registers of type BR hold branch results (boolean data of the program) and registers of type GR hold other numerical data (integers). SIRALINA is able to optimise each register type separately. For instance, when considering GR, the set of tasks V^R of the DDG becomes the set of tasks producing results of type GR. For BR, we consider then that V^R is the set of tasks writing inside registers of type BR. Statistical distribution of number of nodes and arcs for each type (BR or GR) are displayed by the different boxplots in Figure 5. The average number of tasks producing results of type BR is around 4, and the average number of

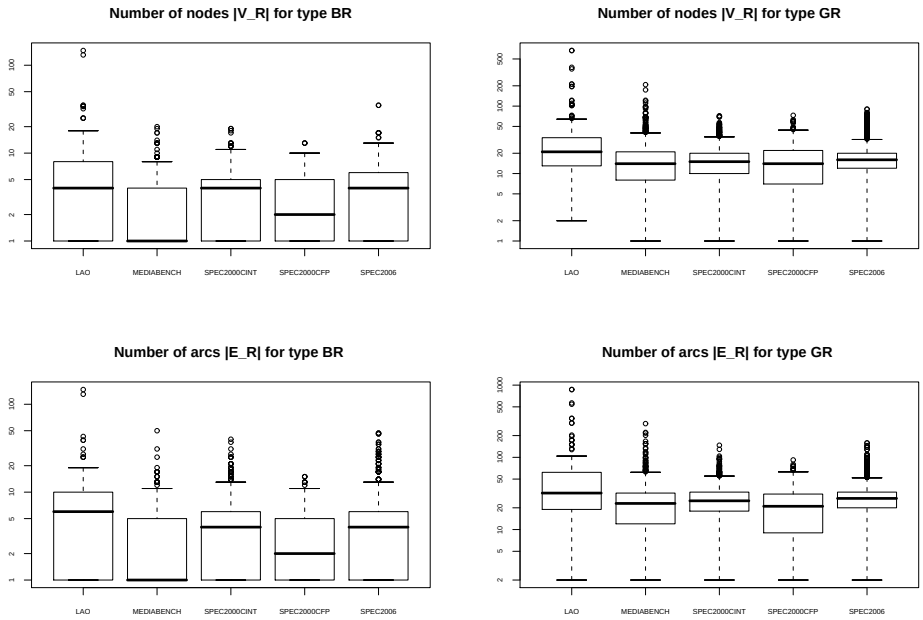


Figure 5. $|V^R|$ and $|E^R|$ Sizes for Register Types BR and GR

flow dependence arcs through registers of type BR is 5, whereas these values are multiplied by more than 5 for registers of type GR (around 20 nodes and 30 arcs).

We use the ILOG CPLEX 10.2 to solve the integer linear programs. The experiments were running on a linux workstation, equipped with a Pentium Xeon 2.33 Ghz processor, and 4 Giga bytes of main memory.

7.2. Comparative Analysis Against Previous Heuristics

In this section we first present a summary for an experimental comparison of the three methods (Method 1, Method 3 and Method 5) with our SIRALINA heuristic. Method 3 and method 5 are previous heuristics (Touati and Eisenbeis, 2004) based on simplified integer linear programs. These previous heuristics are not polynomial but allowed us to solve medium size instances (which was impossible with the exact method). In our experiments, we limit the computation time of these heuristics to one second for a given period value, otherwise the computation time becomes high (few minutes or hours): for interactive compilation, we are asked to not exceed one second for optimising a given loop (DDG) with a given period[2]. SIRALINA is a polynomial

[2] One second is the time limit for interactive compilation (devoted to methods 3 and 5), while 10 seconds is a time limit for offline compilation (devoted to method 1).

heuristic, we are able to solve all problems within one second for a given period without limiting the computation time.

Table I (respectively III) gives the number of problems (in percentage) for which the number of registers of type BR (resp. GR) computed by SIRALINA for the minimal period is strictly lower than the number of registers computed by the other three methods. Table II (respectively IV) gives the percentage of cases where SIRALINA provides the same result as the other method. As can be seen, in most cases, SIRALINA computes at least the same solution as the methods 1, 3 and 5. SIRALINA always outperforms the heuristic 3. For registers of type BR there are few instances where SIRALINA is strictly better than method 1. When the number of nodes of V^R is higher, namely for registers of type GR, SIRALINA is superior. We expected this observation: instances with registers of type BR can be solved to optimality with method 1 because the number of nodes V^R and arcs E^R is relatively weak. But for large instances, with registers of type GR, SIRALINA produces substantially better results.

Table I. SIRALINA vs. Other Methods (Register Type BR) - % of instances where SIRALINA is strictly better

Family of Benchmarks	#benchmarks	M1	M5	M3
LAO	286	2%	27%	56%
MEDIABENCH	1168	0%	19%	41%
SPEC2000-CINT	2297	1%	24%	63%
SPEC2000-CFP	293	0%	15%	48%
SPEC2006	2704	0%	29%	67%

Table II. SIRALINA vs. Other Methods (Register Type BR) - % of instances where SIRALINA provides the same result

Family of Benchmarks	#benchmarks	M1	M5	M3
LAO	286	63%	53%	44%
MEDIABENCH	1168	81%	69%	59%
SPEC2000-CINT	2297	69%	55%	37%
SPEC2000-CFP	293	78%	71%	52%
SPEC2006	2704	68%	49%	33%

Table III. SIRALINA vs. Other Methods (Register GR)- % of instances where SIRALINA is strictly better

Family of Benchmarks	#benchmarks	M1	M5	M3
LAO	286	49%	68%	86%
MEDIABENCH	1168	34%	67%	87%
SPEC2000-CINT	2297	37%	74%	90%
SPEC2000-CFP	293	31%	55%	82%
SPEC2006	2704	36%	80%	92%

Table IV. SIRALINA vs. Other Methods (Register GR)-% of instances where SIRALINA provides the same result

Family of Benchmarks	#benchmarks	M1	M5	M3
LAO	286	33%	19%	14%
MEDIABENCH	1168	54%	28%	13%
SPEC2000-CINT	2297	50%	22%	10%
SPEC2000-CFP	293	54%	35%	18%
SPEC2006	2704	52%	16%	8%

For each group of instances we examine the mean percentage deviation in terms of number of required registers between SIRALINA and another method x ($x = 1, 3$ or 5). The mean percentage deviation is calculated as:

$$\frac{\sum_{\text{DDG}}(SR(x) - SR(SIRALINA))}{\sum_{\text{DDG}} SR(x)} \times 100\%$$

where $SR(x) = \sum \mu_{i,j}$ designs the storage requirement (*i.e.* number of required registers) for the minimal period ($p = MEP$) computed by the method x. This percentage allows us to quantify the overall improvement in terms of storage requirement computed by SIRALINA. The results of the mean percentage deviation are summarised in Table V and Table VI for registers of type BR and registers of type GR. The following observations are made:

– For the register type GR, SIRALINA clearly outperforms all the methods. The method 1 that solves the exact integer model (limited to 10 seconds) fails to provide better solutions than SIRALINA. This is be-

cause the mean number of tasks of type GR is more than 12, which is significant for our exact model (see Figure 4).

– For the register type BR, SIRALINA outperforms methods 3 and 5, but not method 1. This is because the mean number of tasks of type BR is less than 5, which allows to solve the exact model to optimality in most cases within the limit of 10 seconds, while SIRALINA executes in less than one second.

Table V. Mean Storage Requirement Improvement of SIRALINA Compared to Combinatorial Methods (Register Type BR)

Family of Benchmarks	M1	M5	M3
LAO	-25%	+8%	+65%
MEDIABENCH	-17%	+23%	+44%
SPEC2000-CINT	-25%	+23%	+49%
SPEC2000-CFP	-20%	+24%	+47%
SPEC2006	-27%	+26%	+52%

Table VI. Mean Storage Requirement Improvement of SIRALINA Compared to Combinatorial Methods (Register Type GR)

Family of Benchmarks	M1	M5	M3
LAO	+27%	+47%	+75%
MEDIABENCH	+13%	+35%	+65%
SPEC2000-CINT	+7%	+33%	+62%
SPEC2000-CFP	+7%	+29%	+61%
SPEC2006	+8%	+34%	+63%

All the previous statistics use a sample of 6748 representative DDG extracted from many benchmarks. Now, we want to compute the confidence level of our statistics. According to (Jain, 1991), having a statistical guarantee of comparison between two methods asks us to compute the confidence intervals of the paired observations $SR(x)$ v.s.
$SR(SIRALINA)$. This is done by performing a t-test on the set of all

differences $(SR(x) - SR(SIRALINA))$ and to check if the value zero is inside the confidence interval. If zero is inside the confidence interval, we cannot have a statistical guarantee that SIRALINA would always outperform the other method if we consider another set of benchmarks. In our study, we choose a high confidence level equal to 99%, *i.e.*, our statistics have 1% of chance to be wrong if we take another set of benchmarks. Tables VII and VIII report the mean and the confidence intervals of the differences $(SR(x) - SR(SIRALINA))$ for all DDG and methods. As we can see, the value zero is outside all confidence intervals. Consequently, we assert with a confidence level of 99% that:

– SIRALINA outperforms in average all the other methods for any register type except the method 1 for register type BR.

– Method 1 for register type BR always outperforms SIRALINA in average for the reasons explained before.

Table VII. Register Type GR: Confidence Intervals (t-test) with 99% Confidence Level

Method	Mean	Confidence Interval
M1	0.66	[0.56; 0.76]
M5	3.52	[3.19; 3.50]
M3	11.20	[10.72; 11.69]

Table VIII. Register Type BR: Confidence Intervals (t-test) with 99% Confidence Level

Method	Mean	Confidence Interval
M1	-0.36	[-0.39; -0.32]
M5	0.57	[0.51; 0.63]
M3	1.89	[1.79; 1.99]

In the current section, we studied the problem of minimising the storage requirement for a fixed period $p = MEP$. In the next section, we consider a

fixed available number of storage locations (registers) and we study the period increase.

7.3. Study of the Loss of Task Parallelism

Our storage optimisation methodology tries to minimise $\sum \mu_{i,j}$ for a fixed period p. When we consider R a fixed number of available registers, the problem becomes to find a solution $\sum \mu_{i,j} \leq R$ with a minimal period p. If $\sum \mu_{i,j} > R$, then we should try with another higher period. Increasing the period for getting $\sum \mu_{i,j} \leq R$ leads to a task parallelism loss that we study in this section. We distinguish three cases:

- Case 1 ($\sum \mu_{i,j} > R$): the storage requirement is strictly greater than the number of available registers regardless of the period p ($MEP \leq p \leq L$). We say that the storage allocation problem has no solution in this case.
- Case 2 ($\sum \mu_{i,j} \leq R$): the storage requirement is less or equal than the number of available registers for the minimal period ($p = MEP$).
- Case 3: the storage requirement is greater than the number of available registers for the minimal period ($p = MEP$), but there exists a period p' (the smallest p) for which the storage requirement is less than the the number of available registers. In this case, we express the loss of parallelism by the deviation: $\frac{p'-MEP}{MEP} \times 100\%$.

As a concrete example, we consider the number of available registers in the ST231 VLIW processor from STmicroelectroncis. The numbers of available registers of type GR and BR are 61 and 8 respectively.

Tables IX and X indicate the number of benchmarks in each benchmark family which belongs to the Case 1 for register types BR and GR. We remark that the number of SIRALINA cases without solution is rare, consequently 61 GR registers and 8 BR registers seem to be a sufficient architectural choice to avoid using the main memory as external storage location. Hence, in most cases, increasing the period value allows to use the available registers as storage locations.

Now, we study the loss of parallelism due to the increase of the period. We observed a loss of parallelism with SIRALINA for only five instances with type BR (the parallelism loss is equal to 5.6%, 544%, 526%, 255%, 255%). For these five instances, methods 1 and 5 compute a number of required registers below the number of available registers for the minimal period. The reported loss of parallelism is considerable but we should be aware that: 1) SIRALINA is a polynomial heuristic while method 1 and 5 are not. Since the number of tasks of type BR is reduced, the aggressiveness of methods 1 and 5 outperforms SIRALINA in some cases. 2) SIRALINA leads to a parallelism

Table IX. Number of Problems with No Solution (8 Available Registers of Type BR)

Family of Benchmarks	SIRALINA	M1	M5	M3
LAO	2	5	11	24
MEDIABENCH	0	1	2	29
SPEC2000-CINT	0	0	0	4
SPEC2000-CFP	0	0	0	2
SPEC2006	0	0	0	25

Table X. Number of Problems with No Solution (61 Available Registers of Type GR)

Family of Benchmarks	SIRALINA	M1	M5	M3
LAO	4	2	3	69
MEDIABENCH	0	0	3	34
SPEC2000-CINT	1	0	3	62
SPEC2000-CFP	0	0	0	8
SPEC2006	0	0	10	95

loss for register type BR in only 5 DDG within 6748 instances. That is, in 6743 instances, we did not iterate on p to reduce the storage requirement, since a solution is found with $p = MEP$.

For registers of type GR, we identify only three instances with a loss of parallelism within 6748; That is, 6745 instances does not require to iterate on p, since a solution is found with $p = MEP$. These three instances are interesting to summarise in Table XI. The parallelism loss is given between parentheses. The table shows that when SIRALINA computes a solution in Case 3 (with a parallelism loss), other methods do not provide a solution (Case 1). Even in the last instance, SIRALINA finds a solution without parallelism loss (Case 2) whereas method 1 leads to a parallelism loss and method 3 and 5 lead to no solution.

7.4. About the Min-cost Network Flow Implementation

We have made a min-cost network flow implementation based on Section 6. While many interesting polynomial algorithms exist for min-cost flow with

Table XI. Parallelism Loss (61 Available Registers of Type GR)

Instance Name	SIRALINA	M1	M5	M3
LAO-polysyn	Case 3 (642%)	No sol	No sol	No sol
MEDIABENCH-gsm-long-term	Case 3 (162%)	No sol	No sol	No sol
MEDIABENCH-ghostscript	Case 2	Case 3 (5.6%)	No sol	No sol

different complexity formulas (Ahuja and ans James B. Orlin, 1993), it is not clear which algorithm will always be the faster one in practice1. Implementing all the existing algorithms is fastidious. Consequently, we have chosen the cost scaling algorithm of Goldberg and Tarjan (Goldberg and Tarjan, 1990) for its experimental evidence of efficiency (Goldberg, 1992). It runs in $O(n^3 \log(n.C))$, where n is the size of the network and C is the maximal cost. In addition to the experimental conclusions made by the authors, we think that implementing the algorithm of Goldberg and Tarjan is easier than other existing algorithms; compared to the double scaling algorithm for instance, this later requires to build another network flow while the cost scaling works on the initial network (DDG).

In order to compare the execution times of the two implementations of SIRALINA (linear programming vs. min-cost network flow), we conducted 30 runs for each problem instance, all executed on the same machine: these 30 runs are recommended by the test of student (Jain, 1991) to handle the variability of execution times because when we run the same program with the same input data multiple times, we do not always get the same execution times. Based on the 30 runs of each instance, we have made a test of student on all the numerical data. We are then able to assert with confidence level of 90% that the speed of SIRALINA using simplex method is better in average than SIRALINA using network flow implementation. In average, we found that SIRALINA based on simplex method is 70% faster than SIRALINA based on min-cost network flow algorithm.

These results were unexpected because the simplex algorithm has a pseudo-polynomial complexity whereas cost scaling algorithm is a polynomial algorithm. However, it may be possible that another implementation of the min cost flow algorithm gives better results. Indeed, initially we used the minimum mean cycle cancelling algorithm to solve the min cost flow problem and the results were at least ten times worse than the actual ones!

8. Conclusion

This article proposes an efficient heuristic for the periodic task scheduling problem under storage constraints. Our model is based on the theoretical approach of reuse graphs studied in (Touati and Eisenbeis, 2004; Touati, 2002). Storage allocation is expressed in terms of reuse arcs and reuse distances to model the fact that two tasks use the same storage location. This problem is used in practice to optimise the number of registers to put inside a processor given a set of benchmarks, or to optimise the register requirement inside programs (loops) if the number of available registers is fixed.

Computing an optimal periodic storage allocation is intractable in practice (NP-hard), we identified a two steps heuristic that we name SIRALINA. A first step computes scheduling variables and allows us to compute the potential reuse distances if the corresponding reuse arc is added. Then a second step solves a linear assignment problem using the Hungarian method in order to select the appropriate reuse arcs. This two steps heuristic greatly improve our ability to solve the problem for large instances in faster times. The improvement comes basically from the fact that SIRALINA starts by first computing minimal μ values before fixing reuse arcs, while the previous methods did the contrary.

SIRALINA has been implemented inside a framework based on the industrial compiler of STmicroelectronics for embedded code generation and optimisation. Practical experiments on well known benchmarks (SPEC2000, MEDIABENCH, LAO, SPEC2006) show that SIRALINA provides satisfactory solutions with fast computation times (less than one second). In almost all cases, SIRALINA succeeds in limiting the register requirement of all innermost loop programs under the number of available registers in ST231 embedded processor. And this without any parallelism loss, *i.e.*, the computed period is in almost all DDG is equal to $p = MEP$. In some critical cases, SIRALINA leads to a parallelism loss while previous methods do not provide solutions.

Finally our future work will concentrate on two main research subjects. Firstly, we aim to study the particular structure of the exact model constraints to consider the application of Lagrangian relaxation. Secondly, we are willing to consider special shapes for the reuse graph because some processor characteristics impose particular reuse graph structures; For instance, the use of a rotating register file (present inside some processors) implies the presence of a unique Hamiltonian reuse circuit in the reuse graph.

Acknowledgement

This work has been partially supported by the ANR MOPUCE project (ANR number 05-JCJC-0039). We would like to thank Frederic BRAULT and Mounira BACHIR (from INRIA) and Benoit Dupont-de-Dinechin (from STmicroelectronics) for their valuable help.

References

Ahuja, R. K. and ans James B. Orlin, T. L. M. (1993). *Network Flows*. Prentice Hall. ISBN=0-13-61-617549-X.

Briais, S. and Touati, S.-A.-A. (2009). Schedule-Sensitive Register Pressure Reduction in Innermost Loops, Basic Blocks and Super-Blocks. Technical Report HAL-INRIA-00436348, University of Versailles Saint-Quentin en Yvelines. Research report. http://hal.archives-ouvertes.fr/inria-00436348.

Cormen, T., Leiserson, C. E., and Rivest, R. (1990). *Introduction to Algorithms*. MIT Press, McGraw-Hill, Cambridge, Massachusetts.

Crawley, M. J. (2007). *The R Book*. Wiley. ISBN-13: 978-0-470-51024-7.

de Dinechin, B. D. (1996). Parametric Computation of Margins and of Minimum Cumulative Register Lifetime Dates. In David C. Sehr and Utpal Banerjee and David Gelernter and Alexandru Nicolau and David A. Padua, editor, *LCPC*, volume 1239 of *Lecture Notes in Computer Science*, pages 231–245. Springer.

de Werra, D., Eisenbeis, C., Lelait, S., and Marmol, B. (1999). On a Graph-Theoretical Model for Cyclic Register Allocation. *Discrete Applied Mathematics*, 93(2-3):191–203.

Deschinkel, K. and Touati, S.-A.-A. (2008). Efficient Method for Periodic Task Scheduling with Storage Requirement Minimization. In *Proceedings of 2nd Annual International Conference on Combinatorial Optimization and Applications (COCOA 2008)*. Springer.

Eichenberger, A. E., Davidson, E. S., and Abraham, S. G. (1996). Minimizing Register Requirements of a Modulo Schedule via Optimum Stage Scheduling. *International Journal of Parallel Programming*, 24(2):103–132.

Fimmel, D. and Muller, J. (2001). Optimal Software Pipelining Under Resource Constraints. *International Journal of Foundations of Computer Science (IJFCS)*, 12(6):697–718.

Goldberg, A. V. (1992). An Efficient Implementation Of A Scaling Minimum-Cost Flow Algorithm. *Journal of Algorithms*, 22:1–29.

Goldberg, A. V. and Tarjan, R. E. (1990). Finding minimum-cost circulations by successive approximation. *Math. Oper. Res.*, 15(3):430–466.

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

Jain, R. (1991). *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley and Sons, Inc., New York.

Janssen, J. (2001). *Compilers Strategies for Transport Triggered Architectures*. PhD thesis, Delft University, Netherlands.

Kuhn, H. W. (1955). The Hungarian Method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97.

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

Ning, Q. and Gao, G. R. (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, pages 29–42, Charleston, South Carolina. ACM Press.

Rau, B. R., Lee, M., Tirumalai, P. P., and Schlansker, M. S. (1992). Register Allocation for Software Pipelined Loops. *SIGPLAN Notices*, 27(7):283–299. Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.

Schrijver, A. (1987). *Theory of Linear and Integer Programming*. John Wiley and Sons, New York.

Strout, M. M., Carter, L., Ferrante, J., and Simon, B. (1998). Schedule-Independent Storage Mapping for Loops. *ACM SIG-PLAN Notices*, 33(11):24–33.

Thies, W., Vivien, F., Sheldon, J., and Amarasinghe, S. (2001). A Unified Framework for Schedule and Storage Optimization. *ACM SIGPLAN Notices*, 36(5):232–242.

Touati, S.-A.-A. (2002). *Register Pressure in Instruction Level Parallelisme*. PhD thesis, Université de Versailles, France. ftp.inria.fr/INRIA/Projects/a3/touati/thesis.

Touati, S.-A.-A. (2007). On the Periodic Register Need in Software Pipelining. *IEEE Transactions on Computers*, 56(11).

Touati, S.-A.-A. and Eisenbeis, C. (2004). Early Periodic Register Allocation on ILP Processors. *Parallel Processing Letters*, 14(2). World Scientific.

Touati, S.-A.-A. and Mathe, Z. (2009). Periodic Register Saturation in Innermost Loops. *Parallel Computing*, 3:239–254.