RAIRO-Oper. Res. 47 (2013) 223–249
DOI: 10.1051/ro/2013036

RAIRO Operations Research
www.rairo-ro.org

HOW TO ELIMINATE NON-POSITIVE CIRCUITS IN PERIODIC SCHEDULING: A PROACTIVE STRATEGY BASED ON SHORTEST PATH EQUATIONS

Sid-Ahmed-Ali Touati[1], Sébastien Briais[2] and Karine Deschinkel[3]

Abstract. Usual periodic scheduling problems deal with precedence constraints having non-negative latencies. This seems a natural way for modelling scheduling problems, since task delays are generally non-negative quantities. However, in some cases, we need to consider edges latencies that do not only model task latencies, but model other precedence constraints. For instance in register optimisation problems devoted to optimising compilation, a generic machine or processor model can allow considering access delays into/from registers. Edge latencies may be then non-positive leading to a difficult scheduling problem in presence of resources constraints. This research result is related to the problem of periodic scheduling with storage requirement optimisation; its aims is to solve the practical problem of register optimisation in optimising compilation. We show that pre-conditioning a data dependence graph (DDG) to satisfy register constraints before periodic scheduling under resources constraints may create circuits with non-positive distances, resulted from the acceptance of non-positive edge latencies. As a compiler construction strategy, it is forbidden to allow the creation of circuits with non-positive distances during the compilation flow, because such DDG circuits do not guarantee the existence of a valid instruction schedule under resource constraints. We study two solutions to avoid the creation of these problematic circuits. A first solution is reactive, it tolerates the creation of non-positive circuit in a first step, and if detected in a further check step, makes a backtrack to eliminate them. A second solution is proactive, it prevents the creation of

Received January 24, 2011. Accepted May 14, 2013.

[1] Université de Nice Sophia-Antipolis, France. Sid.Touati@unice.fr; Sid.Touati@inria.fr

[2] Université de Versailles Saint-Quentin-en-Yvelines, France

[3] Université de Franche-Comté, France

Article published by EDP Sciences

non-positive circuits in the DDG during the register optimisation process. It is based on shortest path equations which define a necessary and sufficient condition to free any DDG from these problematic circuits. Then we deduce a linear program accordingly. We have implemented our solutions and we present successful experimental results.

Keywords. Periodic scheduling, linear programming, storage constraints, register constraints, code optimisation.

Mathematics Subject Classification. 68 computer science, 90 operational research.

1. Introduction

In an optimising compilation process for instruction level parallelism, we may be faced to the opportunity of bounding the register pressure before instruction scheduling. A typical problem, that we solve in this article, arises for all strategies handling registers before instruction scheduling. Indeed, when we have a target processor with architecturally visible delays to access registers (such as in VLIW, DSP, EPIC and transport triggered architectures), the model of register requirement offers more opportunities to reduce the register pressure than in a regular sequential/superscalar processor. Such architectures are also called NUAL (non-unit-assumed-latencies).

Unfortunately, the opportunities offered by NUAL architectures are not fully or optimally exploited in existing register allocators. Two main reasons:

1. Periodic instruction (task) scheduling uses a model based on data dependence graph (DDG), that represents the periodic precedence constraints between the instructions of a loop (to be defined later). The exploitation of register access delay means the usage of non-positive edges latencies in a DDG. As far as we know, current instruction schedulers do not exploit yet these sort of edges, and consider them as positive edges latencies.
2. If the register constraints are handled before instruction scheduling, an open problem arises regarding the possible creation of circuits with non-positive distances.

This article studies the latter point. That is, we show that minimising the number of registers (storage) in a loop program may prohibit the compiler from generating a code. That is, in theory, if registers are optimised before instructions scheduling as done in some optimising compilers, we may be faced to the problem of impossible instructions scheduling. This situation is not acceptable in optimising compilation, because when a programer writes a correct code, the compiler must be able to generate a executable low-level code.

We follow a formal methodology to deal with the above problem. We use a graph theoretical framework to define the exact problem and to prove a necessary

and sufficient condition for its existence. We provide solutions based on graphs and linear programming. This article demonstrates the following points:

- The DDG circuits with non-positive distances may prohibit periodic instruction scheduling under resource constraints from finding a solution, making the compilation process to fail.
- DDG circuits with non-positive distances are not rare in practice, so the problem is not marginal.
- We show how to avoid the creation of non-positive circuits with two strategies: a reactive strategy (tolerate the problem then fix it if detected) and a proactive strategy (prevent the problem).

While the problem of non-positive circuits may arise in theory for any register optimisation method performing before periodic scheduling, this article shows how to avoid the problem in the SIRA framework [22]. As far as we know, the SIRA framework is the only existing formal method that handles registers constraints before periodic instruction scheduling with multiple register types and delayed access to registers. The upcoming related work section will give more details on the litterature.

Our article is organised as follows. We start with describing some related work in Section 2. Section 3 recalls our notations for data dependence graphs (DDG) and periodic instruction scheduling. This section also recalls our previous results on bounding the register need in DDG. If the reader is familiar with our previous articles on the topic, he may skip reading Section 3. Section 4 defines the problem that is resulted if we insert non-positive edges inside DDG (which in turns create non-positive circuits). Section 5 studies a sufficient and necessary condition that defines a DDG without non-positive circuits. This mathematical characteristic is used to propose a linear program in Section 6.1, which is the core of our proactive method. Section 7 summarises our experimental results, and compare between the efficiency of the reactive and the proactive methods.

The article size limitation imposes us to put the formal proof in an external research report, publicly accessible in [4]. Concerning the reproducibility of the experimental part by any third party, our implemented software and experimental data are released as an open source code in [4].

2. Related work

Register optimisation is a broad research topic in optimising compilation since long time. We must distinguish three possible phase orderings between register optimisation and instruction scheduling:

1. Register constraints are handled after instruction scheduling. This topic is called register allocation, and is well studied mathematically in [1–3] for acyclic

scheduling, and in [7] for periodic scheduling. Our article does not treat a register allocation problem, since we apply register optimisation before instruction scheduling and not after.

2. Register constraints are handled in combination with periodic instruction scheduling . This is the most studied topic. Exact and formal methods are published using integer linear programming [9,10,12,17,18,20], not all are implemented in real compilers because they do not scale to large loops. Almost all these articles treat UAL code semantics, except in [12]. In [12], the target processor architecture is special, called TTA (transport triggered architectures). The problem of non-positive circuits have not been addressed in [12], and the code generation is allowed to fail in theory for TTA processor. Many *ad-hoc* heuristics that work well in practice for superscalar and VLIW processors are already present and presented in some computer engineering conferences, that we do not discuss here because they are not always formally defined with clear algorithmic complexity.
   Our article does not consider resource constraints, since we are faced to a compilation strategy that asks to firstly satisfy the register constraints (to guarantee the absence of spilling) before satisfying resource constraints. The separation between register and resource constraints is an old debate in the community, recent arguments are presented in [21]. Furthermore, a recent efficient heuristic on this topic has been published in [8]. In this recent article the problem of non-positive circuits were not solved. We dedicate the current article to clearly explain the problem, prove a necessary an sufficient condition for it, and propose solutions with experimental evaluation.
3. Register constraints are handled before instruction scheduling. In this situation, most of the published literature is devoted to acyclic scheduling in UAL code semantics [11] with a single register type. In this context, the problem of non-positive circuits is not present. As far as we know, the unique study in computer science devoted to register optimisation before SWP, in the context of periodic scheduling with multiple register types and NUAL code semantics, is the SIRA framework that we discussed in Section 3.3.

The problem of non-positive circuits that we are solving in this article have already been highlighted in [22]. In that time, we provided a necessary and sufficient condition to eliminate non-positive circuits based on circuit retiming [15]. Unfortunately, that preliminary solution is not satisfactory because it imposes the usage of integer linear programming: in practice, with multiple register types, we are not able to solve real life instances using circuit retiming. A full comparison between the presented solution in this article and the circuit retiming method can ben found in [4]. The current article is a major improvement compared to [22] in two ways: 1) for the formal study, we provide a new necessary and sufficient condition to avoid non-positive circuits, that may be implemented with linear programming; 2) we show how it can be integrated to SIRALINA in an iterative process; 3) in practice, the proactive method is implemented and tested on a large set of representative benchmarks, even on very large DDG (multiple thousands of nodes and edges).

3. Data dependence graphs and periodic register constraints

We consider an innermost program loop devoted to code optimisation. Such loop is analysed by the compiler to deduce the data dependences between the instructions (producer/consumer relationship between loop statements). A *data dependence graph* (DDG) is a directed multi-graph $G = (V, E)$ where V is a set of vertices (also called instructions, statements, nodes, operations), E is a set of edges (data dependencies and serial constraints). A DDG is a multi-graph because it is possible to have multiple edges between two vertices.

The modelled processor may have several register types: we note $\mathcal{T}$ the set of available register *types*. For instance, $\mathcal{T} = \{BR, GR, FP\}$ for branch, general purpose, and floating point registers respectively. The number of available registers of type t is noted $\mathcal{R}^t$: $\mathcal{R}^t$ may be the full number of architectural registers of type t, or may be a subset of it if some architectural registers are reserved for other purposes.

We make a difference between tasks and precedence constraints depending whether they refer to data to be stored into registers or not. For a given register type $t \in \mathcal{T}$, we note $V^{R,t} \subseteq V$ the set of statements $u \in V$ that produce values to be stored inside registers of type t. The superscript R means *R*egisters. We write u^t the value of type t created by the instruction $u \in V^{R,t}$. Our theoretical model assumes that a statement u can produce multiple values of distinct types (but a single value per type), which is sufficient to model numerous processor architectures.

Concerning the set of edges E, we distinguish between *flow* edges of type t – noted $E^{R,t}$– from the remaining edges. A flow edge $e = (u, v)$ of type t represents the producer-consumer relationship between the two statements u and v: u creates a value u^t read by the statement v. The set of *consumers* of a value $u \in V^{R,t}$ is defined as

$$\mathrm{Cons}(u^t) = \{\mathrm{tgt}(e) \mid e \in E^{R,t} \wedge \mathrm{src}(e) = u\}$$

where $\mathrm{src}(e)$ and $\mathrm{tgt}(e)$ are the notations used for the source and target of the edge e. When we consider a register type t, the set $E - E^{R,t}$ of remaining edges are simply called *serial* edges: $E - E^{R,t}$ contains non-flow edges and flow edges of other register types $t' \neq t$.

If a value $u^t \in V^{R,t}$ is not read inside the considered code scope ($Cons(u^t) = \emptyset$), it means that either u can be eliminated from the DDG as a dead code, or can be kept by introducing a dummy node reading it.

When we consider an innermost loop, the DDG $G = (V, E)$ may contain circuits. Each edge $e \in E$ is labelled by a pair of values $(\delta(e), \lambda(e))$. $\delta : E \to \mathbb{Z}$ defines the latency of edges and $\lambda : E \to \mathbb{Z}$ defines the distance in terms of number of iterations. The latency $\delta(e)$ of an edge $e = (u, v)$ expresses the minimal time (processor clock cycle, steps) that must separate the execution dates of the two instructions u and v. The distance $\lambda(e)$ defines which instruction instances are connected with the arc e, detailed below.

Since we are dealing with a loop, there may be multiple instances of each instruction, since the instruction is executed at each loop iteration. In order to distinguish between the multiple instances of the same instruction, we speak about operations of the same instruction. If the loop iteration count is n, it means that every instruction is executed n times. In this case, every instruction $u \in V$ has n copies that we number from $u(0)$ to $u(n-1)$. The distance $\lambda(e)$, with $e = (u, v)$, defines the fact that the edge e connects between the operation $u(i)$ and $v(i+\lambda(e)$, $0 < i < n - \lambda(e)$. The latency $\delta(e)$ models the fact that the execution dates of $u(i)$ and $v(i + \lambda(e)$ must be separated by at least $\delta(e)$ steps or processor clock cycles.

3.1. NUAL and UAL code semantics

Processor architectures can be decomposed into many families. One of the used classifications is related to the instruction set semantics [19]:

UAL code semantic : These processors have Unit-Assumed-Latencies at the architectural level. Sequential and superscalar processors belong to this family. In UAL, the assembly code has a sequential semantic, even if the micro-architectural implementation executes instructions of longer latencies, in parallel, out of order or with speculation. The compiler instruction scheduler can always generate a valid code if it considers that all operations have a unit latency.

NUAL code semantic : These processors have Non-Unit-Assumed-Latencies at the architectural level. VLIW, EPIC and some DSP processors belong to this family. In NUAL, the hardware pipeline steps (latencies, structural hazards, resource conflicts) may be visible at the architectural level. Consequently, the compiler has to know about the instructions latencies, and sometimes with the underlying micro-architecture. The compiler instruction scheduler has to take care of these latencies to generate a correct code that does not violate data dependences.

Our processor model considers both UAL and NUAL semantics. Given a register type $t \in \mathcal{T}$, we model possible delays when reading from or writing into registers of type t. We define two delay functions $\delta_{r,t} : V \mapsto \mathbb{N}$ and $\delta_{w,t} : V^{R,t} \mapsto \mathbb{N}$. These delay functions model NUAL semantics. Thus, the statement u reads from a register $\delta_{r,t}(u)$ steps after the schedule date of u. Also, u writes into a register $\delta_{w,t}(u)$ steps after the schedule date of u. In UAL code semantic, these delays are not visible to the compiler, so we have $\delta_{w,t} = \delta_{r,t} = 0$.

3.2. Periodic instruction scheduling

In order to exploit the parallelism between the instructions belonging to different loop iterations, we rely on periodic scheduling.

One of the most used periodic scheduling methods is called *software pipelining* (SWP). SWP is defined by a periodic schedule function $\sigma : V \to \mathbb{Z}$ and an *initiation interval* II. The operation u of the i^{th} loop iteration is noted $u(i)$, it is scheduled

at time $\sigma(u)+i\times II$. Here, the schedule time $\sigma(u)$ represents the execution date of $u(0)$ ($i=0$ is the first iteration). A good schedule is the one that has minimal II.

The schedule function σ is valid *iff* it satisfies the usual precedence constraints in periodic scheduling

$$\forall e=(u,v)\in E:\sigma(u)+\delta(e)\leq\sigma(v)+\lambda(e)\times II \tag{1}$$

If G contains a circuit, a well known necessary condition for a valid SWP schedule to exist is that

$$II\geq\max_{c\text{ is a circuit}}\frac{\sum_{e\in c}\delta(e)}{\sum_{e\in c}\lambda(e)}=MII$$

MII is called the *minimum initiation interval* defined by data dependences. Any circuit c such that $\frac{\sum_{e\in c}\delta(e)}{\sum_{e\in c}\lambda(e)}=MII$ is called a *critical circuit*. If G does not contain a circuit, we define $MII=1$ and not $MII=0$. This is because no code generation is possible in compilation with $MII=0$: the reason is that $MII=0$ models infinite instruction level parallelism, not possible to implement with a loop.

In our paper, we may use the term *periodic* scheduling instead of software pipelining.

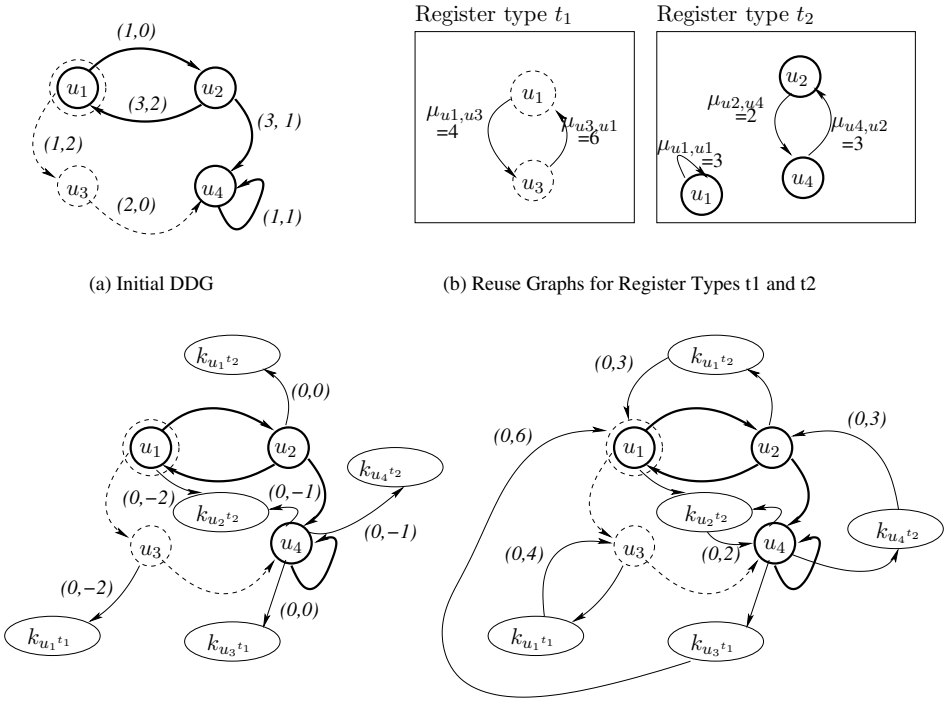
3.3. Periodic register constraints

Depending on the compilation strategy, periodic instruction scheduling may be constrained by the number of registers available in the processors. The instruction schedule may be asked to either minimise or to bound the register requirement. The current theoretical framework that models precisely the periodic register constraints is called SIRA [22], this section gives a synthetic recall.

A simple way to explain and recall the concept of SIRA is to provide an example. All the theory has already been presented in [22]. Figure 1a provides an initial DDG with two register types t_1 and t_2. Statements producing results of type t_1 are in dashed circles, and those of type t_2 are in bold circles. Statement u_1 writes two results of distinct types. Flow dependence through registers of type t_1 are in dashed edges, and those of type t_2 are in bold edges.

As an example, $Cons(u_2^{t_2})=\{u_1,u_4\}$ and $Cons(u_3^{t_1})=\{u_4\}$. Each edge e in the DDG is labelled with the pair of values $(\delta(e),\lambda(e))$. In this simple example, we assume that the delay of accessing registers is zero ($\delta_{w,t}=\delta_{r,t}=0$). Now, the question is how to bound the register need for the loop in Figure 1a without increasing the initiation interval II if possible.

As formally studied in [22], periodic register constraints are modelled thanks to *reuse graphs*. We associate a reuse graph $G^{\text{reuse},t}$ to each register type t, see Figure 1b. The reuse graph has to be computed by the SIRA framework, Figure 1b is one of the examples that SIRA may produce. Note that the reuse graph is not unique, other valid reuse graphs may exist.



(a) Initial DDG

(b) Reuse Graphs for Register Types t1 and t2

(c) DDG with Killing Nodes

(d) Preconditioned DDG after Applying SIRA

FIGURE 1. Example for SIRA and Reuse Graphs.

A reuse graph $G^{\text{reuse},t} = (V^{R,t}, E^{\text{reuse},t})$ contains $V^{R,t}$, *i.e.*, only the nodes writing inside registers of type t. These nodes are connected by *reuse edges*. For instance, in G^{reuse,t_2} of Figure 1b, the set of reuse edges is $E^{\text{reuse},t_2} = \{(u_2, u_4), (u_4, u_2), (u_1, u_1)\}$. Also, $E^{\text{reuse},t_1} = \{(u_1, u_3), (u_3, u_1)\}$. Each reuse edge $e_r = (u, v)$ is labelled by an integral distance $\mu^t(e_r)$. The existence of a reuse edge $e_r = (u, v)$ of distance $\mu^t(e_r)$ means that the two operations $u(i)$ and $v(i + \mu^t(e_r))$ *share the same destination register* of type t. In the example of Figure 1b, we have in G^{reuse,t_2} $\mu^{t_2}((u_2, u_4)) = 2$ and $\mu^{t_2}((u_4, u_2)) = 3$.

In order to be valid, reuse graphs should satisfy two main constraints [22]: 1) They should describe a bijection between the nodes; that is, they must be composed of elementary and disjoint circuits. 2) The *associated DDG* should be schedulable, *i.e.*, it has at least one valid SWP.

Now, let us describe what we mean by the DDG *associated with* a reuse graph. Once a reuse graph is fixed before SWP, say the reuse graphs of types t_1 and t_2 in Figure 1b, the register constraints create new periodic scheduling constraints between loop statements. These scheduling constraints result from the

anti-dependencies created by register reuse. Since each reuse arc (u, v) in the reuse graph $G^{\text{reuse},t}$ describes a register sharing between $u(i)$ and $v(i + \mu^t((u, v)))$, we must guarantee that $v(i+\mu^t((u, v))$ writes inside the same register after the execution of all the consumers of $u^t(i)$. That is, we should guarantee that $v(i+\mu^t((u, v)))$ writes its result after the killing date of $u^t(i)$. If the loop is already scheduled, the killing date is known. However, if the loop is not already scheduled, then the killing date is not known and hence we should be able to guarantee the validity for all possible SWP schedules.

Guaranteeing precedence relationship between lifetime intervals for any SWP is done by creating *the associated DDG* with the reuse graph. This DDG is an extension of the initial one in two steps:

1. **Killing nodes:** First, we introduce dummy nodes representing the killing dates of all values[6]. For each value $u \in V^{R,t}$, we introduce a node k_{u^t} which represents the killing date of u^t. The killing node k_{u^t} must always be scheduled after all u^t's consumers, so we add edges of the form $e = (v, k_{u^t})$ where $v \in Cons^t(u)$. If a value u^t has no consumer (not read inside the loop), it means that the node can be killed just after the creation of its result. Figure 1c illustrates the DDG after adding all the killing nodes for all register types. For each added edge $e = (v, k_{u^t})$, we set its latency to $\delta(e) = \delta_{r,t}(v)$ and its distance to $-\lambda$, where λ is the distance of the flow dependence edge $(u, v) \in E^{R,t}$. As explained in [22], this negative distance is a mathematical convention, it simplifies our mathematical formula and does not influence the fundamental results of reuse graphs. Formally, if $u \in V^{R,t}$ is a node writing a value of type $t \in \mathcal{T}$, then we note k_{u^t} the killer node of type t of the value u^t. The set of killing nodes of type t is noted $V^{k,t}$. For each type $t \in \mathcal{T}$, we note $E^{k,t}$ the set of edges defining the precedence constraints between $V^{R,t}$ nodes and the killer nodes:

$$E^{k,t} = \{e = (v, k_{u^t}) \mid u \in V^{R,t} \wedge v \in Cons^t(u) \wedge \delta(e) = \delta_{r,t}(v)\}$$
$$\cup \{(u, k_{u^t}) \mid u \in V^{R,t} \wedge Cons^t(u) = \emptyset \wedge \delta(e) = 1\}$$

   For instance, in Figure 1b, we have $V^{k,t_2} = \{k_{u_1{}^{t_2}}, k_{u_2{}^{t_2}}, k_{u_4{}^{t_2}}\}$, and we have $E^{k,t_2} = \{(u_2, k_{u_1{}^{t_2}}),$
   $(u_1, k_{u_2{}^{t_2}}), (u_4, k_{u_2{}^{t_2}}), (u_4, k_{u_4{}^{t_2}})\}$.
   If we note $K = \bigcup_{t\in\mathcal{T}} V^{k,t}$ and $E^k = \bigcup_{t\in\mathcal{T}} E^{k,t}$, then the DDG with killing nodes is defined by $(V \cup K, E \cup E^k)$.

2. **Anti-dependence edges:** Second, we introduce new anti-dependence edges implied by periodic register constraints. For each reuse edge $e_r = (u, v)$ in $G^{\text{reuse},t}$, we add an edge $e'_r = (k_{u^t}, v)$ representing an *anti-dependence* in the associated DDG. We say that the anti-dependence $e'_r = (k_{u^t}, v)$ in the DDG G is associated with the reuse edge $e_r = (u, v)$ in $G^{\text{reuse},t}$. We write $\Phi(e_r) = e'_r$ and $\Phi^{-1}(e'_r) = e_r$.

The added anti-dependence edge $e'_r = (k_{u^t}, v)$ has a distance equal to the reuse distance $\lambda(e'_r) = \mu^t(e_r)$, and a latency equal to:

- $\delta(e'_r) = -\delta_{w,t}(v)$ if the processor has NUAL semantics.
- $\delta(e'_r) = 1$ if the processor has UAL semantics. Note that we can still assume a latency $\delta(e'_r) = \delta_{w,t} - \delta_{r,t} = 0$, since the instruction scheduler will generate a sequential code, so this zero edge imposes to schedule k_{u^t} before v.

Figure 1d illustrates the DDG associated to the two reuse graphs of Figure 1(b). Periodic register constraints with multiple register types are satisfied conjointly on the same DDG even if each register type has its own reuse graph. The reader may notice that the critical circuit of the DDG in Figure 1a and c are the same and equal to $MII = \frac{4}{2} = 2$ (a critical circuit is (u_1, u_2)). The set of added anti-dependence edges of type t is noted $E^{\mu,t}$

$$E^{\mu,t} = \{e = (k_{u^t}, v) \mid e_r = (u, v) \in E^{\text{reuse},t} \wedge \Phi(e_r) = e\}$$

In Figure 1d, $E^{\mu,t_1} = \{(k_{u_1{}^{t_1}}, u_3), (k_{u_3{}^{t_1}}, u_1)\}$ and $E^{\mu,t_2} = \{(k_{u_1{}^{t_2}}, u_1), (k_{u_2{}^{t_2}}, u_4), (k_{u_4{}^{t_2}}, u_2)\}$. If we note $E^{\mu} = \bigcup_{t \in \mathcal{T}} E^{\mu,t}$, then the DDG G' (with killing nodes) associated with the reuse graphs $\left(V^{R,t}, E^{\text{reuse},t}\right)_{t \in \mathcal{T}}$ is defined by $G' = (\mathcal{V} = V \cup K,\ \mathcal{E} = E \cup E^k \cup E^{\mu})$.

As can be seen, computing a reuse graph of a register type t implies the creation of new edges with μ^t distances. We proved in [22] that if a reuse graph $G^{\text{reuse},t}$ is valid, then any valid SWP cannot require more than $\sum_{e_r \in E^{\text{reuse},t}} \mu^t(e_r)$ registers of type t, and this upper-bound is reachable. We call $\sum_{e_r \in E^{\text{reuse},t}} \mu^t(e_r)$ the *register requirement* of type t.

Now the problem that must be solved by SIRA is to compute a valid reuse graph with minimal $\sum_{e_r \in E^{\text{reuse},t}} \mu^t(e_r)$, without increasing the initiation interval II if possible. Also, instead of minimising the register requirement, SIRA may simply look for a solution such that $\sum_{e_r \in E^{\text{reuse},t}} \mu^t(e_r) \leq \mathcal{R}^t$, where $\mathcal{R}^t$ is the number of available registers of type t. We may propose many exact method models (the problem has been proved NP-complete in [22]) or heuristics based on the SIRA framework [8, 21].

As explained before, the latency of an added anti-dependence edge $e'_r = (k_{u^t}, v)$ is equal to $\delta(e'_r) = -\delta_{w,t}(v)$ if the target processor has NUAL semantics. The next section explains the problem that may arise when we use edges with negative latencies.

4. Problem description of non-positive circuits

A circuit C is said lexicographic-positive iff $\lambda(C) > 0$, while $\lambda(C)$ is a notation for $\sum_{e \in C} \lambda(e)$. A data dependence graph (DDG) is said lexicographic-positive iff all its circuits are lexicographic-positive too. A DDG is said schedulable iff there exists a valid SWP, *i.e.*, a SWP satisfying all its periodic precedence constraints, not necessarily satisfying other constraints such as resources or registers. A data

dependence graph computed from a sequential program is always lexicographic positive, it is an inherent characteristic of imperative sequential languages. When a DDG is lexicographic-positive, there is a guarantee that a schedule exists for it (at least the initial sequential schedule).

When a DDG is modified by introducing non-positive edges as explained in Section 3.3, it modifies the DDG under the condition that it remains schedulable. If the target architecture has a UAL code semantics (sequential code), then the introduced edges by has unit-assumed latencies, and the DDG remains lexicographic positive. If the target architecture has explicit architectural delays in accessing registers (NUAL code semantics), then the introduced edges are of the form $e' = (k_{u^t}, v)$ with latencies $\delta(e') = -\delta_{w,t}(v)$. Such latencies are non-positive.

If an edge latency is non-positive, this does not create specific problem for periodic scheduling in theory, unless if the latency of a circuit is negative too. The following lemma proves that if $\delta(C) < 0$, then the DDG may not be lexicographic positive.

Lemma 4.1. *Let G be a schedulable loop DDG, i.e., a SWP exists. Let C be an arbitrary circuit in G. Then the following implications are true:*

1. $\delta(C) \geq 0 \Longrightarrow \lambda(C) \geq 0$.
2. $\delta(C) \leq 0 \Longrightarrow \lambda(C)$ *may be non-positive (either* $\lambda(C) \leq 0$ *or* $\lambda(C) \geq 0$*).*

Proof. Since the DDG is schedulable, then there exists a valid SWP with $II > 0$. It is well known that $\forall C$ a cycle $: II \times \lambda(C) \geq \delta(C)$, hence $\lambda(C) \geq \frac{\delta(C)}{II}$.

1. $II > 0 \wedge \delta(C) \geq 0 \Longrightarrow \lambda(C) \geq 0$.
2. $II > 0 \wedge \delta(C) \leq 0 \Longrightarrow \lambda(C) \geq x$, where $x = \frac{\delta(C)}{II} \leq 0$. □

Recall that SIRA looks for a SWP for a given DDG and inserts edges inside the DDG based on the SWP. The previous lemma proves such SIRA mechanism may insert negative edges inside a DDG can generate circuits with $\lambda(C) \leq 0$ (observed in practice, as we will show later).Such circuits with $\lambda(C) \leq 0$ cause hard scheduling problems. Indeed, an explanation may be clear following the periodic scheduling theory: Given a DDG (with circuits), let C^+ be the set of circuits with $\lambda(C) > 0$, le C^- be the set of circuits with $\lambda(C) < 0$, and let C^0 be the set of circuits with $\lambda(C) = 0$. Then the following inequality is true [14,16]:

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

In other words, the existence of circuit inside C^- imposes hard real time constraints on the value of II. Such constraints can be satisfied with periodic scheduling if we consider only precedence constraints [14,16]. However, if we add resource constraints (as will be carried out during the subsequent instruction scheduling pass), then the DDG may not be schedulable. Simply it may be possible that the conflicts on the resources may not allow to have an II lower than $\min_{C \in C^-} \frac{\delta(C)}{\lambda(C)}$.

When a circuit $C \in C^0$ exists, this means that there is a precedence relationship between the statements belonging to the same iteration: that is, the loop body a graph without circuits.

By abuse of language, we say also that a circuit in $C^0 \cup C^-$ is a *non-positive circuit*. Some concrete examples demonstrating the possible existence of non-positive circuits are drawn in [4]. According to our experiments that will be addressed later, inserting non-positive edges inside a large sample of representative DDG produces non-positive circuits in 30.77% of loops in SPEC2000 C applications (resp. 28.16%, 41.90% and 92.21% for SPEC 2006, MEDIABENCH and FFMPEG loops). Note that this problem of non-positive circuits is not related exclusively to SIRA, but it is related to any pass of register optimisation performing on the DDG level before SWP.

As a compiler construction strategy, we must guarantee that the schedulable DDG produced after constructing any reuse graph by SIRA is always lexicographic positive. Otherwise, there is no guarantee that the subsequent SWP pass would find a solution under resource constraints, and the code generation may fail. We start by studying a necessary and sufficient condition to detect and prevent the problem in the next section.

5. Necessary and sufficient condition to avoid non-positive circuits

As mentioned previously, we need to ensure that the associated DDG computed by any SIRA method is lexicographic positive. We have also noted that if the processor has a UAL semantics then it is guaranteed that any associated DDG found by SIRA is lexicographic positive. This is because the UAL semantic is used to model sequential processors, all inserted anti-dependences edges have latency equal to 1. Since all the edges in the associated DDG have positive latencies, and since the associated DDG is schedulable by SWP (guaranteed by SIRA), then the DDG is necessarily lexicographic positive.

Hence, a naive strategy is to always consider UAL semantics, which defines a first sufficient condition to eliminate non-positive circuits. That is, we do not exploit the access delays to registers. This solution works in practice but the register requirement model is not optimal, since it does not exploit NUAL code semantics. Consequently, the computed register requirement is not well optimised.

A more clever, yet naive, way to ensure that any associated DDG computed by SIRA is lexicographic positive is to have a *reactive* strategy. It tolerates the problem as follows:

1. Consider SIRA with NUAL semantics.
2. Check whether the associated DDG is lexicographic positive[4] and
   - if it is, then return the computed solution.
   - if it is not, then apply SIRA considering UAL semantics.

[4]Thanks to Corollary 5.3, to be defined later.

Considering a UAL semantic for SIRA on a processor that has a NUAL semantics cannot hurt: it just possibly implies a loss of optimality in either II or in the register requirement. The above method is optimistic (reactive) in the sense that it considers that non lexicographic DDG are rare in practice. This is not true in theory of course, but maybe the practice would highlight that the proportion of the problems producing DDG that must be *corrected* is low. In this case, it is in practice better to do not try to restrict SIRA, but to correct the solution afterwards if we detect the problem.

The question that thus arises naturally is the following: is it possible to devise a better method to ensure *a priori* that the associated DDG computed by SIRA are lexicographic positive while exploiting the benefit of NUAL semantics ? That is, we are willing to study a *proactive* strategy that prevents the problem. The following simple lemmas, deduced from [5], are a basis for a necessary and sufficient condition to eliminate non-positive circuits. Their proof is easy thanks to shortest path equations.

Lemma 5.1. *Let $G = (V, E)$ a directed graph and $w : E \to \mathbb{Z}$ a cost function. Then G has a circuit C of non-positive cost with respect to cost w* if and only if *G has a circuit of negative cost with respect to cost w' defined by $w'(e) = \|V\| \cdot w(e) - 1$:*

$$\sum_{e \in C} w(e) \leq 0 \iff \sum_{e \in C} w'(e) < 0$$

Proof. Clearly any cycle of G of negative cost w.r.t. w is also a cycle of strictly negative cost w.r.t. w'.

Let $c = (e_1, \ldots, e_p)$ be an elementary cycle.

Then

$$\begin{aligned} \sum_{1 \leq i \leq p} w'(e_i) < 0 &\iff \|V\| \cdot \sum_{1 \leq i \leq p} w(e_i) - p < 0 \\ &\iff \|V\| \cdot \sum_{1 \leq i \leq p} w(e_i) < p \\ &\iff \sum_{1 \leq i \leq p} w(e_i) < \frac{p}{\|V\|} \end{aligned}$$

Hence if c is an elementary cycle of strictly negative cost w.r.t. w', *i.e.* if

$$\sum_{1 \leq i \leq p} w'(e_i) < 0$$

Then we have

$$\sum_{1 \leq i \leq p} w(e_i) < \frac{p}{\|V\|}$$

Since c is elementary, we have $p \leq \|V\|$ and thus

$$\sum_{1 \leq i \leq p} w(e_i) < 1$$

And since the left hand term is an integer, we have

$$\sum_{1\leq i\leq p} w(e_i) \leq 0$$

This shows that c is a cycle of negative cost w.r.t. w. □

Lemma 5.2. *Let $G = (V, E)$ a directed graph and $w : E \to \mathbb{R}$ a cost function.*

Then G has a circuit C of negative cost (i.e. $\sum_{e\in C} w(e) < 0$) if and only if *the constraints system $\mathcal{S}_{G,w}$ defined below is infeasible.*

$$(\mathcal{S}_{G,w}) \begin{cases} \forall e \in E :, x_{\mathrm{tgt}(e)} - x_{\mathrm{src}(e)} \leq w(e) \\ \forall v \in V, x_v \in \mathbb{R} \end{cases}$$

Proof. For the proof, see also [5].

- Assume that G has a cycle of negative cost.
  Let $e_1, \ldots, e_p$ be a cycle with $\mathrm{tgt}(e_i) = \mathrm{src}(e_{i+1})$ for $1 \leq i < p$, $\mathrm{tgt}(e_p) = \mathrm{src}(e_1)$ and $\sum_{1\leq i\leq p} w(e_i) < 0$
  Assume that $\mathcal{S}_{G,w}$ has a solution.
  This solution should satisfy:

$$\begin{cases} x_{\mathrm{tgt}(e_1)} - x_{\mathrm{src}(e_1)} \leq w(e_1) \\ x_{\mathrm{tgt}(e_2)} - x_{\mathrm{src}(e_2)} \leq w(e_2) \\ \qquad\qquad\qquad\vdots \\ x_{\mathrm{tgt}(e_p)} - x_{\mathrm{src}(e_p)} \leq w(e_p) \end{cases}$$

  Thus, by summing all these inequalities, we get

$$\sum_{1\leq i\leq p} (x_{\mathrm{tgt}(e_i)} - x_{\mathrm{src}(e_i)}) \leq \sum_{1\leq i\leq p} w(e_i)$$

  Thus

$$\sum_{1\leq i\leq p} (x_{\mathrm{tgt}(e_i)} - x_{\mathrm{src}(e_i)}) < 0$$

But we have

$$\begin{aligned}\sum_{1\le i\le p}(x_{\mathrm{tgt}(e_i)}-x_{\mathrm{src}(e_i)}) &= \sum_{1\le i\le p} x_{\mathrm{tgt}(e_i)} - \sum_{1\le i\le p} x_{\mathrm{src}(e_i)}\\ &= \left(x_{\mathrm{tgt}(e_p)} + \sum_{1\le i<p} x_{\mathrm{tgt}(e_i)}\right) - \left(x_{\mathrm{src}(e_1)} + \sum_{2\le i\le p} x_{\mathrm{src}(e_i)}\right)\\ &= \sum_{1\le i<p} x_{\mathrm{tgt}(e_i)} - \sum_{2\le i\le p} x_{\mathrm{src}(e_i)}\\ &= \sum_{1\le i<p} x_{\mathrm{src}(e_{i+1})} - \sum_{2\le i\le p} x_{\mathrm{src}(e_i)}\\ &= \sum_{2\le i\le p} x_{\mathrm{src}(e_i)} - \sum_{2\le i\le p} x_{\mathrm{src}(e_i)}\\ &= 0\end{aligned}$$

Thus $0 < 0$, which is a contradiction.
Hence $\mathcal{S}_{G,w}$ has no solution and is thus infeasible.

- Assume that $\mathcal{S}_{G,w}$ is infeasible.
  If G has no cycle of strictly cost, then G can be modified as follows.
  Add a source $\top$ to G connected to each vertex $v \in V$ and extend w so that $w(\top, v) = 0$ for any $v \in V$.
  Then the distance from $\top$ to any other vertex v (ie the length of the shortest path from $\top$ to v) is well defined, since this extension of G has no cycle of negative cost (observe that $\top$ cannot belong to any cycle since it has no incoming edges). For instance, Bellman Ford algorithm can successfully compute this distance function.
  Let $d(\top, v)$ denote the distance from $\top$ to v for any vertex $v \in V$.
  Let $e \in E$. Then we have $d(\top, \mathrm{tgt}(e)) \le d(\top, \mathrm{src}(e)) + w(e)$ by property of a distance function.
  Thus if we pose for any $v \in V$, $x_v = d(\top, v)$ then we have just found a solution of $\mathcal{S}_{G,w}$.
  This is a contradiction.
  So G has a cycle of strictly negative cost. □

From Lemmas 5.1 and 5.2, we deduce the following corollary, which defines a necessary and sufficient condition to eliminate non-positive circuits.

Corollary 5.3. *Let $G = (V, E)$ a directed graph and $w : E \to \mathbb{Z}$ a cost function.*
Then G has a circuit C of non-positive cost with respect to cost w (i.e. $\sum_{e\in C} w(e) \le 0$) if and only if the system composed of the following constraints is infeasible.

$$\forall e \in E, x_{\mathrm{tgt}(e)} - x_{\mathrm{src}(e)} \le \|V\| \cdot w(e) - 1$$

where $\forall v \in V, x_v \in \mathbb{R}$.

Proof. A direct consequence of Lemma 5.1 and 5.2. □

Now, any produced DDG by any register optimisation method (hence any SIRA method), must guarantee the above condition to eliminate non-positive circuits. In our article, we show how to combine the above condition with an efficient heuristic of SIRA that constructs a reuse graph (and hence the associated DDG accordingly), named SIRALINA. The next section studies this question.

6. Application to the SIRA framwork

The problem that must be solved by SIRA is to construct a reuse graph. There are many heuristics and methods that may be used. SIRALINA [8, 21] is our most powerful method, we have already demonstrated that it a really efficient heuristic for SIRA: it considers an initial DDG with multiple register types, and produces an associated DDG to bound or to minimise the register requirement before SWP. SIRALINA is a two steps heuristic, with an algorithmic complexity equal to $O(\|V\|^3 \log \|V\|)$, where $\|V\|$ is a notation for the cardinality of a set. It has been shown that SIRALINA, applied on a large set of benchmarks [8,21], is fast and efficient in practice. So it has been connected to an industry quality compiler for embedded systems targeting VLIW ST231 processors. The next section briefly recalls SIRALINA, if the reader is familiar with our previous publication on the topic, he may skip the section. The section after shows how to eliminate non-positive circuits in the context of SIRALINA.

6.1. Recall on SIRALINA heuristic

Computing a valid reuse graph for a fixed period II that minimises $\sum_{e_r \in E^{\text{reuse},t}} \mu^t(e_r)$ is NP-complete [22]. SIRALINA heuristic [8, 21] computes an approximate solution to this problem for all register types conjointly. In order to balance between the importance of each involved register type, we assume to have a weight $\alpha_t \in \mathbb{R}$ attributed to each type $t \in \mathcal{T}$. This weight may be set to 1 if all register types have the same importance.

SIRALINA is composed of two polynomial steps summarised as follows (here, the period II is fixed):

1. Step 1 (scheduling problem): Determine minimal reuse distances for all pairs of values (*i.e.* compute, for each type t, a function $\widehat{\mu}^t : V^{R,t} \times V^{R,t} \to \mathbb{Z}$);
2. Step 2 (linear assignment problem): Determine a bijection $E^{\text{reuse},t} : V^{R,t} \to V^{R,t}$ that minimises $\sum_{(u,v) \in V^{R,t} \times V^{R,t}} \widehat{\mu}^t(u,v)$ for each t.

These two steps allows the construction of a reuse graph for a period II. Then $G' = (\mathcal{V}, \mathcal{E})$ the associated DDG is constructed as explained previously: $\mathcal{V} = V \cup K$ and $\mathcal{E} = E \cup E^k \cup E^\mu$. The two following sections details each of the two above steps.

6.2. Step 1: the scheduling problem for a fixed II

The scheduling problem [8,21] is to guarantee the existence of a SWP schedule for the associated DDG. The problem is formulated as an integer linear problem with totally unimodular constraints matrix. In addition, it aims at determining minimal reuse distances for all pairs of values. The two next paragrahs define the integer linear program of the scheduling problem.

Integer variables of the linear problem
For any $u \in \mathcal{V}$, define a variable $\sigma_u \in \mathbb{Z}$ representing a scheduling date.

Linear program formulation
The scheduling problem is expressed according to equation 1:

$$\begin{cases} \text{minimise} & \displaystyle\sum_{t\in\mathcal{T}} \alpha_t \left(\sum_{u\in V^{R,t}} \sigma_{k_u^t} - \sum_{u\in V^{R,t}} \sigma_u \right) \\ \text{subject to} & \\ & \forall e \in E \cup E^k, \sigma_{\mathrm{tgt}(e)} - \sigma_{\mathrm{src}(e)} \geq \delta(e) - II \times \lambda(e) \end{cases}$$

The constraints matrix of this integer linear program is an incidence matrix of the DDG G (with killing nodes), consequently it is totally unimodular. Hence it can be solved with a polynomial algorithm.

Let ${\sigma_u}^*$ and ${\sigma_{k_u^t}}^*$ be the values of the variables of the optimal solution of the above scheduling problem. The minimal reuse distance function, noted $\widehat{\mu^*}^t$ is then defined as follows for all pairs of values (u, v):

$$\widehat{\mu^*}^t(u,v) = \left\lceil \frac{{\sigma_{k_u^t}}^* - \delta_{w,t}(v) - {\sigma_v}^*}{II} \right\rceil$$

This minimal reuse distance constitutes the lower bound of the optimal values of the optimisation problem solved by SIRALINA.

6.3. Step 2: the linear assignment problem

The linear assignment problem for a register type t is to find a bijection $\theta^t : V^{R,t} \to V^{R,t}$ such that $\sum_{u\in V^{R,t}} \widehat{\mu^*}^t(u, \theta^t(u))$ is minimal. It can be solved in polynomial time complexity with the so-called Hungarian algorithm [13]. Such an optimal bijection θ^t defines a set of reuse edges $E^{\mathrm{reuse},t}$ as follows.

$$E^{\mathrm{reuse},t} = \{e_r = (u, \theta^t(u)) \mid u \in V^{R,t} \wedge \mu^t(e_r) = \widehat{\mu^*}^t(u, \theta^t(u))\}$$

6.4. Eliminating non-positive circuits in SIRALINA

Our idea is thus the following. Once an initial reuse graph has been computed by SIRALINA, the DDG associated to it may contain non-positive circuits. So we have to eliminate these circuits. All the edge distances are fixed, except that we can increase the anti-dependence edges distances. That is, we can modify the values of $\widehat{\mu}^t$ to eliminate non-positive circuits. Modifying these reuse distances is a valid transformation as soon as it does not violate the scheduling constraints. However, this transformations may ask to use more registers.

Indeed, observe that in the associated DDG, the added edges $e'_r = (k^t_u, v) \in E^{\mu,t}$, where $e_r = (u, v)$ is a reuse edge of type t, have a distance equal to $\lambda(e) = \widehat{\mu}^t(u, v)$, and that the distances of the other edges are entirely determined by the initial DDG and are not subject to changes. By modifying $\widehat{\mu}^t(u, v)$, the optimal solution to the linear assignment problem may be affected (step 2 of SIRALINA). In this case, we may choose to recompute the linear assignment. This defines an iterative process. We start by explaining this iterative process, then we describe how we modify the reuse distances.

Our iterative process is thus given by Algorithm 1. At each iteration i of the algorithm, it computes new reuse distances $\widehat{\mu}^t_{(i)}$ and new reuse edges $E^{\text{reuse},t}_{(i)}$, based on the previous reuse distances $\widehat{\mu}^t_{(i-1)}$ and previous reuse edges $E^{\text{reuse},t}_{(i-1)}$. This algorithm is parametrised by two functions:

- $LinearAssignment(G, \widehat{\mu}^t)$ computes a bijection $\theta^t : V^{R,t} \times V^{R,t}$ that minimises $\sum_{(u,v)\in V^{R,t}\times V^{R,t}} \widehat{\mu}^t(u, v)$. In other words, it solves the linear assignment problem and is typically implemented by the Hungarian algorithm, as done in the second step of SIRALINA. The result of this function is a new set of reuse edges $E^{\text{reuse},t}_{(i)}$.
- $UpdateReuseDistances(G, (\widehat{\mu}^t_{(i-1)})_{t\in\mathcal{T}}, (E^{\text{reuse},t}_{(i-1)})_{t\in\mathcal{T}})$ uses Corollary 5.3 to compute new distance functions $(\widehat{\mu}^t_{(i)})_{t\in\mathcal{T}}$ such that the associated DDG w.r.t.$(\widehat{\mu}^t_{(i)})_{t\in\mathcal{T}}$ and $(E^{\text{reuse},t}_{(i)})_{t\in\mathcal{T}}$ is lexicographic positive.

Our process stops after a certain number of iterations according to the time budget allowed for this optimisation process. The body of the repeat-until loop is executed with a finite number of iterations, noted n. The loop may be interrupted before reaching n iterations when a fix-point is reached, *i.e.* when the set of reuse edges stabilises from one iteration to another ($E^{\text{reuse},t}_{(i)} = E^{\text{reuse},t}_{(i-1)}$). Since the body of algorithm loop is executed at least once, it is guaranteed that the associated DDG will be lexicographic positive.

The following section explains our implementation of the function *UpdateReuseDistances*.

6.5. Updating reuse distances

Our proactive method, named *SPE* (Shortest Path Equations), is based on Corollary 5.3. We deduce from it that the associated DDG is lexicographic positive

Algorithm 1 The Algorithm *IterativeSIRALINA*

Require: G a loop DDG
Require: n maximal number of iterations
  $(\widehat{\mu}^t_{(0)})_{t\in\mathcal{T}} \leftarrow (\widehat{\mu^*}^t)_{t\in\mathcal{T}}$ {Compute initial distance functions by solving the scheduling problem}
  for $t \in \mathcal{T}$ **do**
    $E^{\text{reuse},t}_{(0)} \leftarrow LinearAssignment(G, \widehat{\mu}^t_{(0)})$ {Compute intial reuse edges}
  end for
  $i \leftarrow 0$
  repeat
    $i \leftarrow i + 1$
    $(\widehat{\mu}^t_{(i)})_{t\in\mathcal{T}} \leftarrow UpdateReuseDistances(G, (\widehat{\mu}^t_{(i-1)})_{t\in\mathcal{T}}, (E^{\text{reuse},t}_{(i-1)})_{t\in\mathcal{T}})$
    for $t \in \mathcal{T}$ **do**
      $E^{\text{reuse},t}_{(i)} \leftarrow LinearAssignment(G, \widehat{\mu}^t_{(i)})$
    end for
    if $E^{\text{reuse},t}_{(i)} = E^{\text{reuse},t}_{(i-1)}$ for every $t \in \mathcal{T}$ **then**
      break {A fix-point has been reached}
    end if
  until $i > n$
  return $(\widehat{\mu}^t_{(i)})_{t\in\mathcal{T}}$ and $(E^{\text{reuse},t}_{(i)})_{t\in\mathcal{T}}$

if and only if there exists $|\mathcal{V}|$ variables $x_v \in \mathbb{R}$ for $v \in \mathcal{V}$ such that

$$\forall e \in \mathcal{E} : x_{\text{tgt}(e)} - x_{\text{src}(e)} \leq \|\mathcal{V}\| \cdot \lambda(e) - 1$$

Recall that $\mathcal{V} = V \cup K$ where V is the set of vertices of the initial DDG and K is the set of all killing nodes. We are willing to modify each reuse distance by adding to it an integral increment γ^t. Our objective is still to minimise the register requirement, which means that we need to minimise the sum of γ^t. We thus define a linear problem as follows, which is the heart of our contribution in this article.

For each vertex $v \in \mathcal{V}$, we define a *continuous* variable x_v. For each anti-dependence edge $e = (k^t_u, v)$ corresponding to the reuse edge $e_r = (u, v)$, we define a variable $\gamma^t(u, v)$, so that the distance of e is $\lambda(e) = \widehat{\mu}^t_{(i-1)}(u, v) + \gamma^t(u, v)$.

We seek to minimise the register requirement, which means to minimise $\sum_{t\in\mathcal{T}} \alpha_t \sum_{(u,v)\in E^{\text{reuse},t}} \gamma^t(u, v)$, where α_t is a weight given to a register type, as defined in Section 6.1. In order to guarantee that modifying the reuse distances is a valid transformation, we must ensure that the scheduling constraints are not violated. This means that the modified reuse distances must be greater than or equal to their minimal values: $\widehat{\mu}^t_{(i-1)}(u, v) + \gamma^t(u, v) \geq \widehat{\mu^*}^t(u, v)$ for any $(u, v) \in E^{\text{reuse},t}$, where $\widehat{\mu^*}^t(u, v)$ is the solution of the scheduling problem (first step of SIRALINA), which are indeed the minimal valid values for the reuse distances. Since γ^t is integral values, we should write a mixed integer linear program. But such solution is computationally expensive. So we decide to write a relaxed linear program in Figure 2, where γ^t variables are declared as continuous. Afterwards, we safely ceil these variables to obtain integer values. The linear program of Figure 2 contains

$$
\left\{
\begin{array}{ll}
\text{minimise} & \displaystyle\sum_{t\in\mathcal{T}} \alpha_t \left(\sum_{(u,v)\in E^{\text{reuse},t}} \gamma^t(u,v) \right) \\
\text{Subject to:} & \\
\forall e \in E \cup E^k, & x_{\text{tgt}(e)} - x_{\text{src}(e)} \leq \|\mathcal{V}\| \cdot \lambda(e) - 1 \\
\forall t \in \mathcal{T}, \forall e = (k_u^t, v) \in E^{\mu,t}, & x_{\text{tgt}(e)} - x_{\text{src}(e)} - \|\mathcal{V}\| \cdot \gamma^t(u,v) \leq \|\mathcal{V}\| \cdot \widehat{\mu}^t_{(i-1)}(u,v) - 1 \\
\forall t \in \mathcal{T}, \forall (u,v) \in E^{\text{reuse},t}_{(i-1)}, & \gamma^t(u,v) \geq \widehat{\mu^*}^t(u,v) - \widehat{\mu}^t_{(i-1)}(u,v) \\
\forall u \in \mathcal{V}, & x_u \in \mathbb{R} \\
\forall t \in \mathcal{T}, \forall (u,v) \in E^{\text{reuse},t}_{(i-1)}, & \gamma^t(u,v) \in \mathbb{R} \\
\text{where:} & \\
\forall t \in \mathcal{T}, & E^{\mu,t} \stackrel{\text{def}}{=} \{\Phi(e_r) \mid e_r \in E^{\text{reuse},t}_{(i-1)}\}
\end{array}
\right.
$$

FIGURE 2. Linear program based on shortest paths equations (SPE).

$O(|\mathcal{V}|+|\mathcal{E}|)$ variables and $O(|\mathcal{E}|)$ linear equations. Once a solution is found for the linear program of Figure 2, we set the new distance of $e = (k_{u^t}, v) \in E^{\mu,t}$ as equal to $\lambda(e) = \widehat{\mu}^t_{(i-1)}(u,v) + \lceil \gamma^t(u,v) \rceil$.

Hence our implementation of $UpdateReuseDistances(G, (\widehat{\mu}^t)_{t\in\mathcal{T}}, (E^{\text{reuse},t})_{t\in\mathcal{T}})$ is given by Algorithm 2.

Algorithm 2 The Function *UpdateReuseDistances*

Require: $(\widehat{\mu}^t_{(i-1)})_{t\in\mathcal{T}}$ previously computed reuse distances for all register types
Require: $(E^{\text{reuse},t}_{(i-1)})_{t\in\mathcal{T}}$ previously computed reuse edges for all register types
Solve the linear program of Figure 2 to compute $(\gamma^t(u,v))$
return $(\widehat{\mu}^t_{(i)})_{t\in\mathcal{T}}$ where $\widehat{\mu}^t_{(i)}(u,v) \stackrel{\text{def}}{=} \begin{cases} \widehat{\mu}^t_{(i-1)}(u,v) + \lceil \gamma^t(u,v) \rceil & \text{if } (u,v) \in E^{\text{reuse},t}_{(i-1)} \\ \widehat{\mu}^t_{(i-1)}(u,v) & \text{otherwise} \end{cases}$

7. Experimental study

The full experimental study is is available in [4]: source code and experimental data are also delivered. This section is a synthesis, it presents our conclusions.

These experiments have been conducted on approximately 9000 representative DDG, extracted from well known benchmarks (FFMPEG, MEDIABENCH, SPEC2000 and SPEC2006). We assume three register types $\mathcal{T} = \{GR, BR, FP\}$. We used a regular Linux workstation (Intel Xeon, 2.33 GHZ, 9 Gigabytes of memory).

7.1. Heuristics nomenclature

Our methods to avoid the creation of non-positive circuits are of three sorts:

1. UAL is the (pessimistic) naive heuristic which consists in applying SIRALINA with an UAL semantics only. That is, we do not consider NUAL code semantics from the beginning.
2. CHECK is the reactive strategy which consists in firstly applying SIRALINA with NUAL semantics. If a non-positive circuit is detected, we apply a second pass, which apply SIRALINA but with a UAL semantics.
3. SPE is the proactive strategy, based on shortest paths equations (SPE). If $n \geq 1$ is the maximal allowed number of iterations used, we write SPE_n.

7.2. Empirical efficiency measures

For each heuristic of non-positive circuit elimination, for each DDG, for each initiation interval II between MII and L (L is a fixed upper bound on the admissible values for II), we measured the execution time taken by each heuristic (listed above) to minimise the register requirement; we recorded also the number of registers computed by the three methods (UAL,CHECK and SPE). We are going to examine these results in the next sections.

We have also considered three possible target processor architectures (small, medium and large) as described in [4]: the difference between these architecture is the number of available registers per register type. When the number of available registers is fixed in the architecture, we may need to iterate on multiple values for II in order to get a solution below the processor capacity; that is, since register minimisation is applied for a fixed II, we may need to iterate on multiple values of II if the minimised register requirement is still above the number of available registers.

The strategy for iterating over II for one of our heuristic (Here, any of the three methods previously described can be used: UAL, CHECK, SPE) is the following:

- Check whether the heuristic produces a solution that satisfies the register constraints for II = MII.
  - if yes, stop and return the solution.
  - if no, check whether the method gives a solution that satisfies the constraints for II = L (maximal allowed value for II).
    - if yes, search linearly the smallest II > MII such that the heuristic computes a solution that satisfies the register constraints.
    - if no, then fail (no solution found, spilling is required).

For each architecture and for each DDG G, we determined whether the heuritic (UAL, CHECK or SPE) is able to find a solution that satisfies the architecture constraints. We thus measured:

- the elapsed time needed to determine whether a solution exists;
- the smallest II for which a solution exists (when applicable).

Regarding the iterative heuristic of non-positive circuits elimination (SPE), we arbitrary fixed the maximal number of iterations to 3 and 5. In order to get an idea of how many iterations the iterative methods could take in the worst case before

TABLE 1. Execution times of each heuristic.

Strategy	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
UAL	0.001	0.002	0.03	0.03
CHECK	0.002	0.002	0.003	0.07
SPE_5	0.012	0.015	0.015	0.27

reaching a fixed point (convergence), we also did the experiments by settings a maximal allowed number of iterations to 1000 and recorded the reached number of iterations. Remember that if a fixed point (convergence) is detected, the iterative algorithm stops before reaching 1000 iterations.

7.3. Comparison of the heuristics execution times

In this section, we compare and comment the execution times of the heuristics of non-positive circuit elimination.

7.3.1. Time to minimise register requirement for a fixed II

In this section, we apply the three methods with all values of II. Table 1 shows the median execution time for each heuristic and for each benchmark family.

From our results, we see as expected that UAL is the fastest heuristic. CHECK is between one and three times slower than UAL, which was also expected because it consists in running SIRALINA, performing a check and in the worst case running SIRALINA a second time.

Regarding our proactive heuristic, SPE heuristic seems to have a quite reasonable running time, but is yet sensibly more expensive than UAL or CHECK (about 10 times slower).

7.3.2. Convergence of the proactive heuristic (SPE)

We study in this section the speed of convergence (in terms of number of iterations) of SPE heuristic. Recall that SPE is said to *converge* when it reaches a fixed point, *i.e.* when the set of reuse edges does not change between two consecutive iterations of Algorithm 1. All the values of II are tested, so the experiments we consider in this section are for all DDG and for all II values.

Table 2 shows the distribution of the number of iterations of SPE heuristic (truncated at 1000): the table reports the values of the minimum (FST), the first quartile (FST), the median, the third quartile (THD) and the maximum (MAX). We observe that on a few number of DDG, the upper bound of 1000 iterations has been reached by SPE heuristic. It is indeed well possible that the iterative process does not terminate in the general case. Note finally that this information may be used to set in an industrial compiler the upper bound on the maximal number of iterations: 5 iterations seems to be a satisfactory pratical choice since it allows the convergence in 75% of the cases for SPEC2000, SPEC2006 and MEDIABENCH benchmarks.

TABLE 2. Maximum observed number of iterations for SPE.

	SPEC2000	SPEC2006	MEDIABENCH	FFMPEG
MIN	1	1	1	1
FST	3	3	3	6
MEDIAN	3	3	4	11
THD	5	5	5	21
MAX	1000	1000	66	1000

7.4. Qualitative analysis of the heuristics

In this section, we study the quality of the solution produced by the heuristics. The qualitative aspects include the number of registers needed to schedule the DDG and the loss of parallelism due to an increase of the MII resulted from UAL, CHECK and SPE.

7.4.1. Number of saved registers

In this section, we analyse the number of registers each heuristic manage to optimise. Our tests are for all DDG, for all II values. We compare graphically the heuristics: for each set of benchmarks, and each register types, we construct a partial order (lattice) as follows:

- the vertices are labelled with the name of the heuristic
- a directed edge links an heuristic A to an heuristic B iff the number of registers (of considered type) computed by heuristic B is statistically greater (worse) than the number of registers (of the same type) computed by heuristic A: by statistically greater, we mean that we applied a one-sided Student's t-test between the alternatives A and B, and we report the risk level of this statistical test (between brakets in the edges). The edge is also labelled with the ratio $\frac{\sum_{G,II} R_B}{\sum_{G,II} R_A}$ where R_B is the number of registers (of considered type) computed by heuristic B and R_A is the number of registers computed by heuristic A.

The lattices are given in [4]. Firstly, from our results, we observe that the ordering of the heuristics depends on the register type. Indeed, since the heuristics try to reduce register pressure of all types simultaneously, it happens that some performs better on one type that on the others.

Secondly, we see that UAL is the worst heuristics regarding register requirement. This is not surprising since this is the most naive way to eliminate non-positive circuits.

Finally, we observe that CHECK is sometimes the best heuristic (in particular for type GR and FP on all benchmarks except FFMPEG). We can explain this by the fact that the proportion of DDG with non-positive circuits on SPEC2000, SPEC2006 and MEDIABENCH is low (less than 40%). Consequently, the reactive strategy (CHECK) is appropriate, since more than 60% of the DDG did not get a non-positive circuit from the beginning (so they did not require a correction step).

7.4.2. Proportion of success when looking for a solution that satisfies the register constraints

In this section, we do not analyse the amount of registers needed as in the previous section. We assume an architecture with a fixed number of available registers, and we count the number of solutions that have a register requirement below the processor capacity. We decompose the solutions into three families: the DDG that have been solved without MII increase, the DDG that have been solved with MII increase, and the DDG that was not solved with *the heuristic *(spilled). All the results are present in [4].

We find that our heuristics found most of the time a solution that satisfies the register constraints. Of course, the percentage of success increased while the architecture constraints were relaxed. Apart from the FFMPEG benchmarks under the small architecture constraints (where the number of available registers is very small, so the constraints on register pressure are harder to satisfy), the percentage of success is above 95%. In these cases, all the heuristics give comparable results.

For the FFMPEG benchmarks, we see that SPE_5 and SPE_{1000} give slightly better results than the naive UALheuristic (1 to 3% better). We Observe that in most of the cases of success, the MII has not been increased at all.

7.4.3. Increase of the MII when looking for a solution that satisfies the register constraints

We count the MII increase by the formula $\frac{\sum MII_h(G)}{\sum MII(G)} - 1$, where $MII_h(G)$ is the MII of the associated DDG computed by heuristic h. In other words MII_h is the smallest period II that satisfies the register constraints when we use heuristic h ($h \in$ {UAL, CHECK, SPE_n}).

These results show that the increase of the MII is very low (less than 6% in the worst case). It is clearly negligible on SPEC2000, SPEC2006 and MEDIABENCH benchmarks. On FFMPEG benchmarks, we see that when dealing with small architecture, SPE heuristics tends to increase the MII more than UAL or CHECK heuristics, whereas for bigger architecture, SPE_5 and SPE_{1000} gives slightly better results than UAL or CHECK.

The conclusions we can take from this extensive experimental study are contrasted. On one hand, the results show that the proactive heuristic SPE allows to save a bit more of registers than the two naive heuristics UAL and CHECK. On the other hand, these results also show that our proactive heuristic is more expensive regarding the execution times than the reactive one.

We thus advise the following policy. If the target architectures are embedded systems, where compilation time does not need to be interactive and where register constraints are strong, we advise to use SPE proactive heuristic. As we have seen, it optimises registers better than the reactive heuristic while being still quite cheap. On the contrary, if the target architecture is a general purpose computer (workstation, desktop, supercomputer), where register constraints are not too strong, it is probably sufficient to use the reactive heuristic CHECK as it already gives good

results in practice and it is only between one and three times slower than UAL heuristic.

8. Conclusion

Pre-conditioning a data dependence graph before SWP is a beneficial approach for reducing spill code and improving the performance of loops. Until now, schedule-sensitive register optimisation was studied only for sequential and superscalar codes, with UAL code semantics.

When considering NUAL code semantics, the access to registers may be architecturally delayed. These delay accesses provide interesting compilation opportunities to save registers. These opportunities are exploited by the insertion of edges with non-positive latencies inside DDG.

Inserting edges with non-positive latencies inside DDG highlight two open questions. First, existing software pipelining (SWP) and periodic scheduling methods do not handle yet these non-positive latencies. Second, a pre-conditioning step that optimises registers before SWP may create circuits with non-positive distances.

DDG with non-positive circuits have the drawback of not being lexicographic positive. This means that, when resource constraints are considered, the existence of a valid SWP is no longer guaranteed. This may cause the failure of the compilation process (no code is generated while the program is correct). Our experiments show that, if no care is taken, 30.77% of loops in SPEC2000 applications induce non-lexicographic positive circuits (respectively 28.16%, 41.90% and 92.21 for SPEC 2006, MEDIABENCH and FFMPEG loops).

In order to avoid the situation of creating non lexicographic positive DDG, we studied two strategies. First, we studied a reactive strategy that tolerates the problem: we start by optimising the register pressure at the DDG level without special care; if a non-positive circuit is detected, then backtrack and consider a UAL code semantics instead of NUAL; this means that we degrade the model of the processor architecture by not exploiting the opportunities offered by delayed accesses to registers. Second, we designed a proactive strategy that prevents the problem. The proactive strategy is based on a necessary and sufficient condition that we prove. It is implemented as an iterative process that increases the reuse distances until a fixed point is observed (or until we reach a limit in terms of iterations).

Concerning the efficiency of our strategies, the reactive strategy seems to perform well in practice in a regular compilation process: when the number of architectural registers is fixed, register minimisation is not necessary (just be sure to be below the architectural capacity). In this context, it is advised to not to try to prevent the problem of non-positive circuits, but to tolerate it in order to save compilation time. In other contexts of compilation, the number of architectural registers is not fixed. This is the case of reconfigurable circuits where the number of registers needed may be decided after code optimisation and generation. It is also the case of architectures with *frame* registers such as EPIC IA64, where a

minimal register requirement reduces the cost of function calls. Also, this may be used to keep free as many registers as possible in order to be used for other code optimisation methods. In such situations, our proactive strategy is efficient in practice: the iterative register minimisation saves better registers than in the reactive strategy, while the compilation time stays reasonable (though greater than the reactive strategy).

References

[1] F. Bouchez, A. Darte, C. Guillon and F. Rastello, Register Allocation: What does the NP-Completeness Proof of Chaitin et al. Really Prove?, in *International Workshop on Languages and Compilers for Parallel Computing (LCPC'06)*, Springer *Lect. Notes Comput. Sci.* (2006) 283–298.

[2] F. Bouchez, A. Darte and F. Rastello, On the Complexity of Register Coalescing, in *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE Computer Society Press (2007) 102–114.

[3] F. Bouchez, A. Darte and F. Rastello, On the complexity of spill everywhere under SSA form, in *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*. ACM Press (2007) 103–112.

[4] S. Briais, S.-A.-A. Touati and K. Deschinkel, Ensuring Lexicographic-Positive Data Dependence Graphs in the SIRA Framework. Technical Report HAL-INRIA-00452695, University of Versailles Saint-Quentin en Yvelines (2010). Research report. http://hal.archives-ouvertes.fr/inria-00452695.

[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company (2001).

[6] B. Dupont de Dinechin, Parametric Computation of Margins and of Minimum Cumulative Register Lifetime Dates, in *LCPC '96: Proceedings of the 9th International Workshop on Languages and Compilers for Paral. Comput.*, London, UK. Springer-Verlag (1997) 231–245.

[7] D. de Werra, C. Eisenbeis, S. Lelait and B. Marmol, On a graph-theoretical model for cyclic register allocation. *Discrete Appl. Math.* **93** (1999) 191–203.

[8] K. Deschinkel, S.-A.-Ali Touati and S. Briais, SIRALINA: efficient two-steps heuristic for storage optimisation in single period task scheduling. *J. Combin. Optim.* **22** (2011) 819–844.

[9] A.E. Eichenberger and E.S. Davidson, Efficient formulation for optimal modulo schedulers. *SIGPLAN Notice* **32** (1997) 194–205.

[10] D. Fimmel and J. Muller, Optimal Software Pipelining Under Resource Constraints. *Int. J. Found. Comput. Sci. (IJFCS)* **12** (2001) 697–718.

[11] R. Govindarajan, H. Yang, J.N. Amaral, C. Zhang and G.R. Gao, Minimum Register Instruction Sequencing to Reduce Register Spills in Out-of-Order Issue Superscalar Architecture. *IEEE Trans. Comput.* (2003) 4–20.

[12] J. Janssen, *Compilers Strategies for Transport Triggered Architectures*. Ph.D. thesis, Delft University, Netherlands (2001).

[13] H.W. Kuhn, The Hungarian Method for the assignment problem. *Nav. Res. Logist. Q.* **2** (1955) 83–97.

[14] T.-Eog Lee and S.-Ho Park, An extended event graph with negative places and tokens for time window constraints. *IEEE Trans. Autom. Sci. Eng.* **2** (2005) 319–332.

[15] C.E. Leiserson and J.B. Saxe, Retiming Synchronous Circuitry. *Algorithmica* **6** (1991) 5–35.

[16] A. Munier, A graph-based analysis of the cyclic scheduling problem with time constraints: schedulability and periodicity of the earliest schedule. *J. Scheduling* **14** (2011) 103–117.

[17] S.G. Nagarakatte and R. Govindarajan, Register Allocation and Optimal Spill Code Scheduling in Software Pipelined Loops Using 0-1 Integer Linear Programming Formulation, in *Compiler Construction (CC)*, vol. 4420, *Lecture Notes in Computer Science*, Braga, Portugal (2007) 126–140. Springer.

[18] J. Ruttenberg, G.R. Gao, A. Stoutchinin and W. Lichtenstein, Software Pipelining Showdown : Optimal vs. Heuristic Methods in a Production Compiler, in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implemantation*, New York. ACM Press (1996) 1–11.

[19] M. Schlansker, B. Rau and S. Mahlke, Achieving High Levels of instruction-Level Parallelism with Reduced Hardware Complexity. Technical Report HPL-96-120, Hewlet Packard (1994).

[20] P. Sucha and Z. Hanzálek, Scheduling of Tasks with Precedence Delays and Relative Deadlines - Framework for Time-optimal Dynamic Reconfiguration of FPGAs, in *IPDPS, IEEE* (2006) 1–8.

[21] S.-A.-A. Touati, F. Brault, K. Deschinkel and B.D. de Dinechin, Efficient Spilling Reduction for Software Pipelined Loops in Presence of Multiple Register Types in Embedded VLIW Processors. *ACM Trans. Embedded Comput. Syst.* **10** (2011) 1–47.

[22] S.-A.-A. Touati and C. Eisenbeis, Early Periodic Register Allocation on ILP Processors. *Paral. Proc. Lett.* **14** (2004) 287–313.