On the Periodic Register Need in Software Pipelining

Sid-Ahmed-Ali TOUATI

***Abstract*— This article presents several theoretical and fundamental results on register need in periodic schedules, also known as MAXLIVE. Our first contribution is a novel formula for computing the exact number of registers needed by a scheduled loop. This formula has two advantages: its computation can be done using a polynomial algorithm with $\mathcal{O}(n \lg n)$ complexity (n is the number of instructions in the loop), and it allows the generalization of a previous result [13]. Second, during software pipelining, we show that the minimal number of registers needed may increase when incrementing the initiation interval (II), contrary to intuition. For the case of zero architectural delays in accessing registers, we provide a sufficient condition for keeping the minimal number of registers from increasing when incrementing the II. Third, we prove an interesting property that enables to optimally compute the minimal periodic register sufficiency of a loop for all its valid periodic schedules, irrespective of II. Fourth and last, we prove that the problem of optimal stage scheduling under register constraints is polynomially solvable for a subclass of data dependence graphs, while this problem is known to be NP-complete for arbitrary dependence graphs [7]. Our latter result generalizes a previous achievement [13] which addressed data dependence trees and forest of trees. In this study we consider cyclic data dependence graphs without taking into account any resource constraints. The aim of our theoretical results on periodic register need is to help current and future software pipeliners achieve significant performance improvements by making better (if not best) use of the available resources.**

***Index Terms*— Periodic Register Requirement, MAXLIVE, Periodic Register Sufficiency, Software Pipelining, Stage Scheduling, Instruction Level Parallelism.**

I. INTRODUCTION

SOFTWARE pipelining (SWP) is a common way to schedule innermost loops in order to extract a large amount of instruction level parallelism (ILP). In addition to the inherent data dependence constraints, computing a periodic schedule of a loop must obey two main families of constraints. The first one is related to resource constraints that must be satisfied in order to avoid oversaturating the functional units of the underlying processor. The second family consists of register constraints: the computed periodic schedule must not require more registers than the ones available. The software pipelining must not only obey these two families of constraints, but it must also maximize the execution rate (minimize the initiation interval) of the loop. In this article, we focus only on the register constraints and we do not consider any functional unit limitation nor any resource model.

Ideally, one should prefer to bring effective methods to minimize the initiation interval (II) under a fixed number of available registers. Unfortunately, the literature focuses on the dual method: given a fixed integral II, how to minimize the register need ? This is because, as far as we know until now, the register need is hard to minimize if an integral II is not fixed. So, many people provide heuristics to reduce the register need for a fixed integral II [9], [10], [15], [17], [24]. If II is assumed as a rational period, the article [14] provides a method for minimizing II with a limited number of registers. Assuming rational periods is another method of periodic scheduling, distinct from common software pipelining. In this paper we consider integral periods.

This article investigates several fundamental aspects in the field of minimizing periodic register need in software pipelining. The ancestor problem of minimal register need in the case of basic blocks (acyclic schedules) profits from plenty of studies, resulting in a rich theoretical literature. Unfortunately, the periodic (cyclic) problem suffers somehow from fewer fundamental results. Our present fundamental results in this topic allow to better understand the register constraints in periodic instruction scheduling, and hence help the community to provide better SWP heuristics and techniques in the future.

In order to be "optimal", SWP techniques should schedule the instructions of a loop in harmony with many constraints, such as data dependence constraints and target processor constraints. The usual processor limitations commonly taken into account are registers, functional units, instruction selection and coding constraints. However, there are other hardware characteristics that are not currently considered in optimal SWP techniques (a far as we know): cache effects (variable loads latencies), memory disambiguation mechanisms, memory banking and interleaving, load-store queues, dynamic speculation, dynamic register renaming, etc. We think that if all these hardware constraints are modeled inside the same complex SWP, it will be hard to come up with a mathematical intelligibility of the SWP problem. So, studying SWP under data dependences and registers constraints separately from the other resource constraints serves the purpose of separating complex problems to deduce some mathematical characteristics useful for writing better general SWP heuristics.

Our contribution is organized as follows. Section II recalls formal notations and definitions about software pipelining and periodic register need. Section III finds a new formula for computing the register need of a scheduled loop which can be computed by a polynomial algorithm. Section IV provides a sufficient condition so that the minimal register need does not increase when incrementing II: however, such condition is proved in the case of architectures with zero delays in accessing registers, which reflect the most common architectures. This condition is used to show how to compute the periodic register sufficiency of

University of Versailles, France. Email:`Sid.Touati@uvsq.fr`

a loop independently of any periodic schedule. Before conclusion, Section V examines the problem of stage scheduling with register minimization in the special case of expression trees, and in more general cases which are data dependence graphs that assign a unique killer per variable.

II. Background

We consider a simple innermost loop (without branches, with possible recurrences). It is represented by a data dependence graph (DDG) $G = (V, E, \delta, \lambda)$, such that:

- V is the set of the statements in the loop. The instance of statement u (an operation) of iteration i is denoted by $u(i)$, and when referring to an arbitrary iteration of a statement u, we simply write u;
- E is the set of precedence constraints (flow dependences, or other serial constraints), any edge e has the form $e = (u, v)$, where $\delta(e)$ is the latency of the edge e in terms of processor clock cycles and $\lambda(e)$ is the distance of the edge e in terms of number of iterations.
- A valid schedule σ must satisfy:

$$\forall i, \forall e = (u, v) \in E: \ \sigma\big(u(i)\big) + \delta(e) \leq \sigma\big(v(i + \lambda(e))\big)$$

We consider a target RISC-style architecture and we distinguish between statements and precedence constraints, depending upon whether they refer to values to be stored in registers or not:

1) $V_R \subseteq V$ is the set of statements that produce values to be stored in registers.
2) $E_R \subseteq E$ is the set of flow dependence edges through a register. The set of consumers (readers) of a value $u \in V_R$ is therefore the set:

$$Cons(u) = \{v \in V \mid (u, v) \in E_R\}$$

In order to consider static issue VLIW processors in which the hardware pipeline steps are visible to compilers (we consider superscalar processors too), we assume that reading from and writing into a register may be delayed from the beginning of the schedule time, and these delays are visible to the compiler (architecturally visible). We define two delay (offset) functions δ_r and δ_w in which:

$$\begin{aligned} \delta_w : \quad & V_R \to \mathbb{N} \\ & u \mapsto \delta_w(u) |\ 0 \leq \delta_w(u) \\ & \text{ the write cycle of } u \text{ into a register is} \\ & \sigma(u) + \delta_w(u) \end{aligned}$$

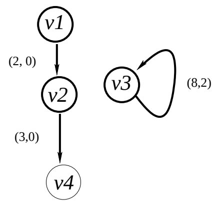
$$\begin{aligned} \delta_r : \quad & V \to \mathbb{N} \\ & u \mapsto \delta_r(u) |\ 0 \leq \delta_r(u) \\ & \text{ the read cycle of } u \text{ from a register is} \\ & \sigma(u) + \delta_r(u) \end{aligned}$$

According to the semantics of superscalar processors (sequential semantics) and EPIC/IA64, δ_r and δ_w are equal to 0. Also, most of the VLIW processors has zero reading/writing delays. But few VLIW processors such as Trimedia have non-zero reading/writing delays.

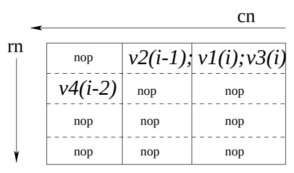
The next section recalls basic notations and definitions in software pipelining.

A. Software Pipelining

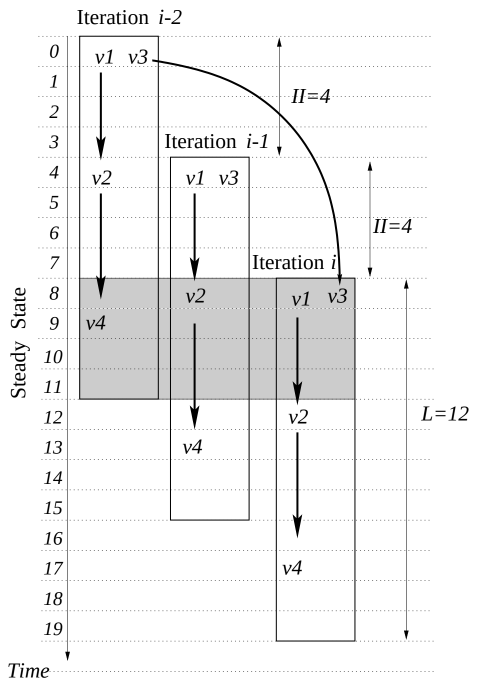
Software pipelining (SWP) is basically a scheduling method. It can be modeled by a function σ that assigns to each statement u a scheduling date (in terms of clock cycle) that satisfies the precedence constraints. The most common form, modulo scheduling, is defined by an initiation interval, denoted II, and the scheduling date $\sigma_u \in \mathbb{N}$ for each operation $u(0)$ of the first iteration. Operation u of iteration i is scheduled at time $\sigma_u + i \times II$. The total schedule time of one iteration of the original loop body is noted L with $L \geq \max_{u \in V} \sigma_u$, and $II \leq L$ is the total schedule time of the new loop kernel. We call L the *duration* (sometimes called iteration length or time horizon). Figure 1(b) is an example of a software pipelined schedule of the DDG shown in Figure 1(a), in which the values and flow edges are drawn with bold lines. A pair of the labels $(\delta(e), \lambda(e))$ is associated with each edge e.



(a) the DDG



(c) Software Pipelining Kernel



(b) Overlapping the Successive Iterations

Fig. 1. Software Pipelining

Any valid periodic schedule must satisfy:

$$\forall e = (u, v) \in E, \ \sigma_u + \delta(e) \leq \sigma_v + \lambda(e) \times II$$

Classically, by adding all such inequalities (precedence constraints) along any cycle C of G, we find that II 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 MII (minimal initiation interval). In this paper, we ignore $ResMII$ since we do not assume any resource model.

So, MII in our text is equivalent to $RecMII$.

Wang *et al.* [23] modeled the kernel of a software pipelined (SWP) schedule as a two dimensional matrix by defining a column number cn and row number rn for each statement, see Figure 1(c). This brings a new definition for SWP, which becomes a triple (rn, cn, II). The row number rn of a statement u is its issue date inside the kernel. The column number cn of a statement u inside the kernel, sometimes called *kernel cycle*, is its stage number. The last parameter II is the kernel length (initiation interval). This triple formally defines the SWP schedule σ as:

$$\forall u \in V,\ \forall i \in \mathbb{N}: \qquad \sigma(u(i)) = rn(u) + II \times (cn(u) + i)$$

where $cn(u) = \lfloor \frac{\sigma_u}{II} \rfloor$ and $rn(u) = \sigma_u \mod II$. For the rest of the article, we will write $\sigma = (rn, cn, II)$ to reflect the equivalence (equality) between the SWP scheduling function σ, defined from the set of statements to clock cycles, and the SWP scheduling function defined by the triple (rn, cn, II).

Let $\Sigma(G)$ be the set of all valid software pipelined schedules of a loop G. We denote by $\Sigma_L(G)$, the set of all valid software pipelined schedules whose durations (total schedule time of one original iteration) do not exceed L:

$$\forall \sigma \in \Sigma_L(G),\ \forall u \in V: \qquad \sigma_u \le L$$

$\Sigma(G)$ is an infinite set of schedules, while $\Sigma_L(G) \subset \Sigma(G)$ is finite. Bounding the duration L in SWP scheduling allows for instance to look for periodic schedules with finite prologue/epilogue codes, since the size of the prologue/epilogue codes is $L - II$ and $0 \le II \le L$.

The next section recalls the notion of register need in periodic schedules.

B. Periodic Register Need

The value produced by the operation $u(0)$ is written into a register at $\sigma_u + \delta_w(u)$ clock cycles starting from the execution date of the whole loop, defining its *birth date*. The killers of this value are all the last scheduled consumers (readers). The value $u(0)$ is dead after its last use(s), at a cycle we denote:

$$d_\sigma(u) = \max_{e=(u,v)\in E_R} \left(\sigma_v + \delta_r(v) + \lambda(e) \times II\right)$$

To generalize, the value u of the i^{th} iteration ($u(i)$) is defined at the absolute time $\sigma_u + \delta_w(u) + i \times II$ (starting from the execution date of the whole loop) and killed at the absolute time $d_\sigma(u) + i \times II$. Thus, the endpoints of the lifetime intervals of the distinct operations of any statement u are all separated by a constant time equal to II. Given such fixed II, we can model the periodic lifetime intervals during the steady state by considering the lifetime interval of only one instance $u(i)$ per statement, say $u(0)$, that we will simply abbreviate by u.

In our model, we assume that a value written at instant c is alive one step later. This is not a limitation of the model, but a choice. It does not alter the mathematical results of our study.

The *acyclic lifetime interval* (range) of the value $u \in V_R$ is then equal to:

$$LT_\sigma(u) =]\sigma_u + \delta_w(u), d_\sigma(u)]$$

As can be seen, this interval is left open because we assume that the value is alive one step after its writing. For instance, the acyclic lifetime intervals of $v1$, $v2$ and $v3$ in Figure 2 are (resp.) $]1, 3]$, $]6, 9]$ and $]2, 8]$.

The *lifetime* of a value $u \in V_R$ is the total number of clock cycles during which this value is alive according to the schedule σ. It is the difference between the death and the birth date, and given as:

$$lifetime_\sigma(u) = d_\sigma(u) - \sigma_u - \delta_w(u)$$

For instance, the lifetimes of $v1$, $v2$ and $v3$ in Figure 2 are (resp.) 2, 3 and 6 clock cycles.

The periodic register need (also known in the literature as register requirement or MAXLIVE) is the maximal number of values which are simultaneously alive in the SWP kernel. In order to clarify potential confusion, the reader must distinguish between two concepts about register need:

1) As we define in this article, the register need is the exact maximal number of values simultaneously alive (called also MAXLIVE).
2) Sometimes, the register need refers to the number of registers used for the final SWP schedule (at register allocation and code generation step). For this case, MAXLIVE constitutes a lower bound of register need. However, this lower bound for the final register need can always be reachable and equal to MAXLIVE, as proved in [2], if we unroll the loop sufficiently or if we insert *move* instructions. If neither loop unrolling nor inserting *move* instructions are allowed, we may require MAXLIVE+1 registers to generate the code (experimental evidence established in [16], proved later in [12]).

In the case of a periodic schedule, some values may be alive during several consecutive kernel iterations and different instances of the same variable may interfere. Figure 2 illustrates another schedule of the DDG previously shown in Figure 1(a): the value v_3 for instance interferes with itself.

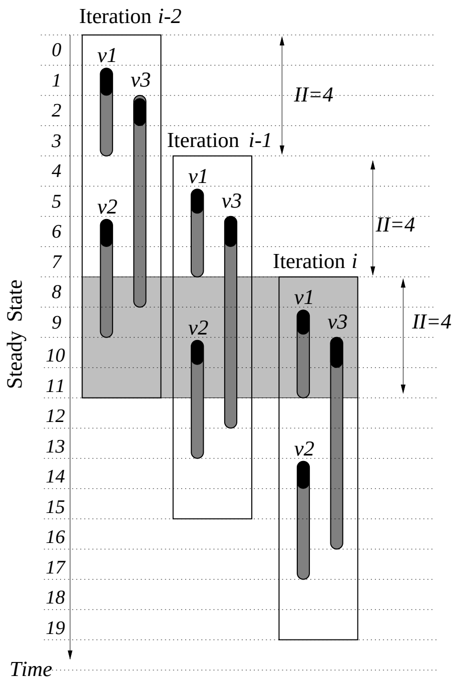
Previous important results published by Laurie Hendren [8] show that the lifetime intervals during the steady state describe a circular lifetime interval graph around the kernel: we “wrap” (roll up) the acyclic lifetime intervals of the values around a circle of circumference II, and therefore the lifetime intervals become cyclic. We give here a formal definition of such circular intervals.

Definition 1 (Circular Lifetime Interval): A circular lifetime interval produced by wrapping a circle of circumference II by an acyclic interval $I =]a, b]$ is defined by a triplet of integers (l, r, p), such that:

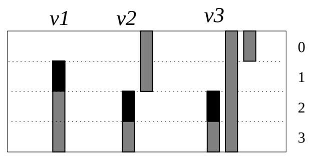
- $l = a \mod II$ is called the **l**eft end of the cyclic interval;
- $r = b \mod II$ is called the **r**ight end of the cyclic interval;
- $p = \left\lfloor \frac{b-a}{II} \right\rfloor$ is the number of complete **p**eriods (turns) around the circle.

Let us consider the examples of the circular lifetime intervals of v_1, v_2 and v_3 in Figure 2(b). These intervals are drawn in a circular way inside the SWP kernel. Their corresponding acyclic intervals are drawn in Part (a) of the same figure. The left ends of the cyclic intervals are simply the dates when the lifetime intervals begin inside the SWP kernel. So, the left ends of the intervals of v_1, v_2 and v_3 are 1, 2, 2 respectively (according to

Definition 1). The right ends of the cyclic intervals are simply the dates when the intervals finish inside the SWP kernel. So the corresponding right ends of v_1, v_2 and v_3 are 3, 1, 0 respectively. Concerning the number of periods of a circular lifetime interval, it is the number of complete kernels (II fractions) spanned by the considered interval. For instance, the intervals v_1 and v_2 do not cross any complete SWP kernel; their number of complete periods is then equal to zero. The interval v_3 crosses one complete SWP kernel, so its number of complete period is equal to one. Finally, the definition of a circular lifetime interval groups its left end, right end and number of complete periods inside a triple. The circular interval of v_1, v_2 and v_3 are then denoted as $(1, 3, 0)$, $(2, 1, 0)$ and $(2, 0, 1)$ respectively.



(a) Software Pipelining

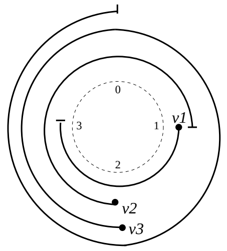


■ indicates the definition of the value

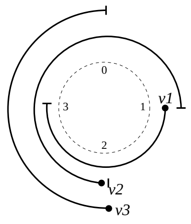
(b) Lifetime Intervals inside the Kernel

Fig. 2. Periodic Register Need in Software Pipelining Schedules

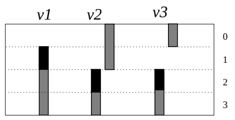
The set of all the circular lifetime intervals around the kernel defines a circular interval graph which we denote by $C(G)$. In this article, we use the short term of circular interval to indicate a circular lifetime interval, and the term of circular graph for indicating a circular lifetime intervals graph. Figure 3(a) gives an example of a circular graph. The maximal number of simultaneously alive values is the width of this circular graph, i.e., the maximal number of circular intervals which interfere at a certain point of the circle. For instance, the width of the circular graph of Figure 3(a) is 4. Figure 2(b) is another representation of the circular graph. We denote by $RN_{\sigma}(G)$ the periodic register need of the DDG G according to the schedule σ, which is equal to the width of the circular graph.



(a) Circular Graph



(b) Fractional Circular Graph



(c) Fractional Intervals

Fig. 3. Circular Lifetime Intervals

III. Computing Periodic Register Need

Computing the width of a circular graph is straightforward. We can compute the number of simultaneously alive values at each clock cycle in the SWP kernel. This method is commonly used in the literature [9], [10], [15], [17], [24]. Unfortunately, it leads to a method whose complexity depends on the initiation interval II. This factor is pseudo-polynomial because it does not strictly depend on the size of the input DDG, but rather depends on the specified latencies in the DDG, and on its structure (critical cycle). We want to provide a better method whose complexity depends only on the size of the DDG, i.e., depends only on n, the number of statements (number of DDG vertices). For this purpose we find a relationship between the width of a circular interval graph and the size of a maximal clique in the interference graph [1]. We are aware that other good polynomial methods for computing MAXLIVE may exist. However, our method brings a new formula that has two main advantages: First, it is formal, provably correct. Second, it is important for improving and generalizing previous results [3], [13] in Section V.

In general, the width of a circular interval graph is not equal to the size of a maximal clique in the interference graph [22].

[1]Remember that the interference graph is an undirected graph that models interference relations between lifetime intervals: two statements u and v are connected iff their (circular) lifetime intervals share a unit of time.

This is contrary to the case of acyclic intervals graphs where the size of a maximal clique in the interference graph is equal to the width of the intervals graph. In order to effectively compute this width (which is equal to the register need), we decompose the circular graph $C(G)$ into two parts.

1) The first part is the integral part. It corresponds to the number of complete turns around the circle, i.e., the total number of values instances simultaneously alive during the whole steady state of the SWP schedule: $\sum_{(l,r,p) \text{ a circular interval}} p$.
2) The second part is the fractional (residual) part. It is composed of the remainder of the lifetime intervals after removing all the complete turns (see Figures 3(b) and (c)). The size of each remaining interval is strictly less than II, the size of the SWP kernel. Note that if the left end of a circular interval is equal to its right end ($l = r$), then the remaining interval after ignoring the complete turns around the circle is empty ($]l, r] =]l, l] = \emptyset$). These empty intervals are then ignored from this second part. Two classes of intervals which remain are as follows:
   a) Intervals that do not cross the kernel barrier, i.e., when the left end is less than the right end ($l < r$). In Figures 3(b) and (c), v_1 belongs to this class.
   b) Intervals that cross the kernel barrier, i.e., when the left end is greater than the right end ($l > r$). In Figures 3(b) and (c), v_2 and v_3 belong to this class. These intervals can be seen as two fractional intervals ($]l, II]$ and $]0, r]$) which represent the left and the right parts of the lifetime intervals. If we merge these two acyclic fractional intervals of two successive SWP kernels, we create a new contiguous circular interval.

   These two classes of intervals define a new circular graph. We call it a *fractional* [1] circular graph because the size of its lifetime intervals is less than II. This circular graph contains the circular intervals of the first class, and those of the second class after merging the left part of each interval with its right part, see Figure 3(b).

Definition 2 (Fractional Circular Graph): Let $C(G)$ be a circular graph of a DDG $G = (V, E, \delta, \lambda)$. The fractional circular graph, denoted by $\underline{C}(G)$, is the circular graph after ignoring the complete turns around the circle:

$$\underline{C}(G) = \{(l, r) \mid \exists (l, r, p) \in C(G) \ \wedge \ r \neq l\}$$

We call the circular interval (l, r) a *circular fractional interval*. The length of each fractional interval $(l, r) \in \underline{C}(G)$ is less than II clock cycles. Therefore, the periodic register need becomes equal to:

$$RN_\sigma(G) = \Big(\sum_{(l,r,p)\in C(G)} p\Big) + w\Big(\underline{C}(G)\Big) \qquad (1)$$

where w denotes the width of the fractional circular graph (the maximal number of values simultaneously alive). Computing the first term of this formula (complete turns around the circle) is easy and can be computed in linear time (provided lifetime intervals) by iterating over the n lifetime intervals and adding the integral part of $\left\lfloor \frac{lifetime(u)}{II} \right\rfloor$.

However, the second term of the formula is more difficult to compute in polynomial time. This is because, as stated before, the size of a maximal clique (in the case of an arbitrary circular graph) in the interference graph is not equal to the width of the circular interval graph [22]. In order to find an effective algorithmic solution, we use the fact that the fractional circular graph $\underline{C}(G)$ has circular intervals which do not make complete turns around the circle. Then, if we unroll the kernel exactly once to consider the values produced during two successive kernel iterations, some circular interference patterns become visible inside the unrolled kernel. For instance, the circular graph of Figure 4(a) has a width equal to 2. Its interference graph in Figure 4(b) has a maximal clique of size 3. Since the size of these intervals does not exceed the period II, we unroll the circular graph once as shown in Figure 4(c). The interference graph of the circular intervals in Figure 4(d) has a size of a maximal clique equal to the width, which is 2: note that $v2$ does not interfere with $v3'$ because, as said before, we assume that all lifetime intervals are left open.

When unrolling the kernel once, each fractional interval $(l, r) \in C(G)$ becomes associated with two acyclic intervals I and I' constructed by merging the left and the right parts of the fractional interval of two successive kernels. I and I' are then defined as follows:

- If $r \geq l$, then $I =]l, r]$ and $I' =]l + II, r + II]$.
- If $r < l$, then $I =]l, r + II]$ and $I' =]l + II, r + 2 \times II]$.

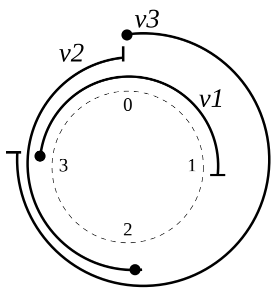
Theorem 1: Let $\underline{C}(G)$ be a circular fractional graph (no complete turns around the circle exists). For each circular fractional interval $(l, r) \in C(G)$, we associate the two corresponding acyclic intervals I and I'. The cardinality of any maximal clique in the interference graph of all these acyclic intervals is equal to the width of $\underline{C}(G)$.

Proof: Please consult Appendix I. ■

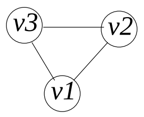
Theorem1 proves an important property that allows us to compute Equation 1 in polynomial time. The second term of that formula, which is the width of the circular graph, can now be computed after unrolling the kernel once (linear time complexity) and then by computing the width of the acyclic fractional intervals graph. This can be done with a complexity of $\mathcal{O}(2 \times n \lg n) = \mathcal{O}(n \lg n)$ [6]. The first part of the formula, as stated before, can be computed in a linear time complexity (assuming circular intervals are provided).

The result presented in this section shows an interesting formula (Equation 1) that allows us to compute the exact register need of a scheduled loop using a polynomial algorithm. This is a new aspect about software pipelining where the usual methods of computing RN are pseudo-polynomial. Also, this new mathod of RN computation will be used in Section V to generalize previous results [3], [13]. Someone could argue that computing the periodic register need by traversing the II, even if it is pseudo-polynomial, in practice is very fast and simple. Such argument is valid from the computer engineering point of view. However, this is not an acceptable claim from the computer science point of view because of two main reasons:

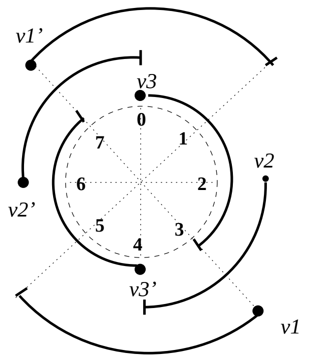
1) If a method computing the periodic register need by traversing the II is fast in practice, we should (try) to prove it formally that it would be fast for *any* input DDG. Usually, experiments are done on a finite set of non-representative benchmarks and on typical machines and software setup. So, such experiments do not provide general guarantee for the efficiency of a method.



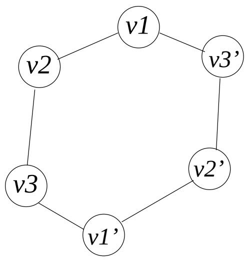
(a) Initial Circular Graph



(b) Initial Interference Graph



(c) Circular Graph after unrolling once



(d) Interferences after unrolling once

Fig. 4. Relationship between the Maximal Clique and the Width of a Circular Graph

2) According to the algorithmic theory, pseudo-polynomial algorithms have somehow exponential algorithmic complexity. Algorithmic theory says that, unless we do not have the choice, polynomial time algorithms are to be preferred to pseudo-polynomial methods.

The next section investigates the problem of minimal register need in periodic schedules.

IV. Computing the Periodic Register Sufficiency

The literature contains many techniques about reducing the periodic register need for a given fixed II. In this section, we want to compute the minimal register need for any valid SWP independently of II. We call it *the periodic register sufficiency* to distinguish it from the classical register sufficiency in basic blocks. We define it as:

$$PRS(G) = \min_{\sigma \in \Sigma(G)} RN_{\sigma}(G) \qquad (2)$$

Computing the periodic register sufficiency allows us for instance to determine if spill code cannot be avoided for a given loop: if R is the number of available registers, and if $PRS(G) > R$ then there are not enough registers to allocate to any loop schedule. Spill code has to be introduced necessarily, independently of II.

The complexity of computing the register sufficiency in ILP codes (regardless if they are basic blocks or loops) remains an open problem. It was proved that computing an instruction ordering that minimizes the number of required registers is NP-complete in the case of sequential codes [19], i.e., when we compute a strict sequential execution order. If we do not restrict the schedule to be sequential, the problem is different. It was proved in [4] that the problem of (parallel) scheduling under register constraints is NP-complete under the condition that the total schedule time is bounded. As far as we know, there is no known results about the problem of scheduling parallel operations to minimize the number of registers (without spill, without resource constraints) without bounding the total schedule time.

This section tries to give a formula that allows to compute PRS for any SWP schedule. Since we are not able to compute the register need independent of II, an obvious method would be to compute the minimal register need under a fixed II, and then iterate over all possible values of II until reaching a limit, or when $II = L$ (the maximal allowed II). First, such method is complex because computing the minimal periodic register need under a fixed II is NP-complete [4]. Second, it requires solving many optimization problems (one for each considered II).

The first step toward our goal is to give a sufficient condition such that the minimal register need under a fixed II would be greater than or equal to the one computed with $II + 1$. The next section investigates this aspect.

A. Minimal Register Need vs. Initiation Interval

It is intuitive that, the lower the initiation interval II, the higher the register pressure, since more parallelism requires more memory. If we succeed in finding a software pipelined schedule which needs R registers, and without assuming any resource conflicts, then it is possible to get another software pipelined schedule which needs no more than R registers with a higher II; until now, such assertion has not been proved. We show here that increasing the maximal duration L is a sufficient condition.

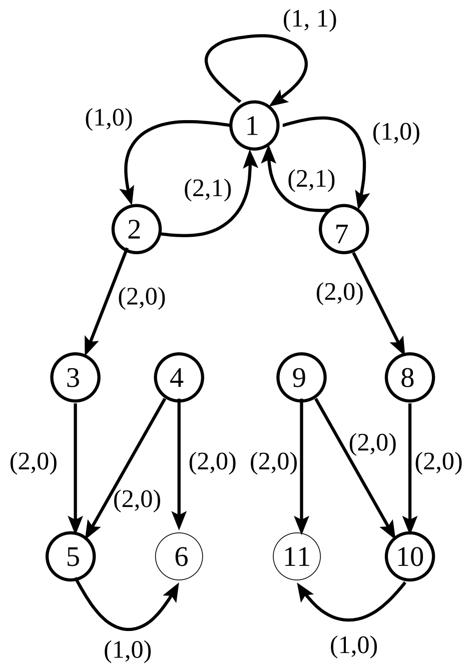
Theorem 2: Let $G = (V, E, \delta, \lambda)$ be a DDG with zero delays in accessing registers. If there exists a software pipelining $\sigma = (rn, cn, II)$ which needs R registers having a duration at most L, then there exists a software pipelining $\sigma' = (rn', cn', II + 1)$

which needs R registers too having a duration at most $L' = L + 1 + \lfloor L/II \rfloor$. Formally:

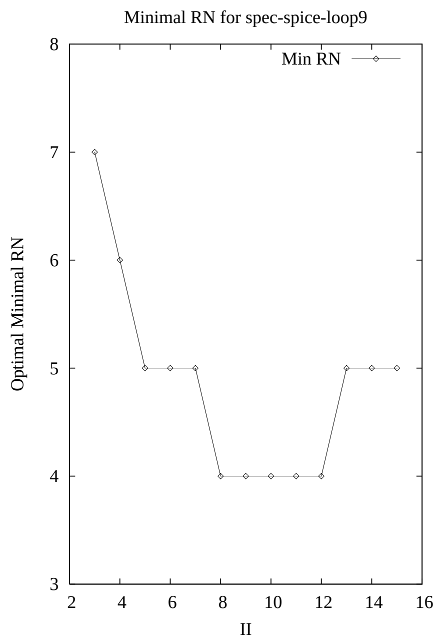
$$\forall \sigma = (rn, cn, II) \in \Sigma_L(G),$$
$$\exists \sigma' = (rn', cn', II + 1) \in \Sigma_{L+1+\lfloor L/II \rfloor}(G) :$$
$$RN_{\sigma'}(G) = RN_{\sigma}(G)$$

Proof: Please consult Appendix II. ■

Theorem 2 gives a sufficient condition so that the minimal register need does not increase when increasing II. The usual intuition suggests that increasing II would decrease register pressure [9], [10], [15], [17], [24]. The extant algorithms often attempt to deal with excess register pressure by increasing II, since intuitively less parallelism would require fewer registers. Some algorithms implicitly allow L to increase as well (although sometimes L is kept bounded to avoid hurting performance for short trip counts or to avoid long prologue/epilogue code). However, we have a counterexample demonstrating that increasing II does not necessarily reduce the register need if L is not also allowed to increase; it may even increase. Figure 5 shows a counterexample. The first part presents the DDG of a loop extracted from the spice benchmark (spec 95). The label of each edge e is the pair $(\delta(e), \lambda(e))$. The second part of Figure 5 plots the minimal register need for different fixed II's. It has been computed using an exact optimal integer linear programming approach presented in [21] (without any resource constraints). The maximal duration L has been fixed to the sum of all the latencies, i.e., $L = 20$ in this example. As can be seen, if L is fixed for all II, then the minimal register need can increase when incrementing II. This example shows that the minimal register need may not be a decreasing function of II as commonly believed. We have to allow L to increase when incrementing II. For instance, when $II = 12$ we computed that the minimal register need is 4 registers ($L = 20$). If we want to guarantee that the minimal register need will not increase when $II = 12+1 = 13$, we have to set (according to Theorem 2) a new maximal duration $L' = L+1+\lfloor L/II \rfloor = 20+1+\lfloor 20/12 \rfloor = 22$, that is we have to increase L by two clock cycles to give more freedom to the SWP scheduler so that it can decrease (or keep constant) the register need. Otherwise, we would require at least 5 registers as plotted.



(a) DDG Example



(b) Optimal Minimal Periodic Register Need

Fig. 5. Example of Minimal Periodic Register Need vs. Initiation Interval

This section provided a relationship between the minimal register need and II that we will use in the next section to compute the register sufficiency .

B. Computing the Periodic Register Sufficiency

The PRS defined by Equation 2 is called *the absolute register sufficiency* because it is defined for all valid SWP schedules belonging to $\Sigma(G)$ (an infinite set). In this section, we will compute PRS for a finite subset $\Sigma_L(G) \subseteq \Sigma(G)$, i.e., for the set of SWP schedules such that the duration does not exceed L. This is because many practical SWP schedulers assume a bounded duration L in order to limit the prologue/epilogue size. As we will show later, one can choose a value for L such that:

$$PRS = \min_{\sigma \in \Sigma(G)} RN_{\sigma}(G) = \min_{\sigma \in \Sigma_L(G)} RN_{\sigma}(G)$$

Many techniques show how to determine the minimal register need given a fixed II [1], [5], [17], [21]. If we use such methods to compute PRS, we have to solve many combinatorial problems, one for each II, starting from MII to a maximal duration L. Fortunately, the following corollary states that it is sufficient to compute PRS by solving a *unique* optimization problem with $II = L$ if we increase the maximal duration (the new maximal duration is denoted L' to distinguish it from L). Let us start by the following lemma, which is a direct consequence of Theorem 2:

Lemma 1: Let $G = (V, E, \delta, \lambda)$ be a DDG with zero delays in accessing registers. The minimal register need of all the software pipelined schedules with an initiation interval II assuming duration at most L is greater or equal to the minimal register need of all the software pipelined schedules with an initiation interval $II' = II + 1$ assuming duration at most $L' = L + 1 + \lfloor L/II \rfloor$. Formally,

$$\min_{\sigma=(rn,cn,II)\in\Sigma_L(G)} RN_\sigma(G)$$

$$\geq \min_{\sigma'=(rn',cn',II+1)\in\Sigma_{L+1+\lfloor L/II \rfloor}(G)} RN_{\sigma'}(G)$$

Proof: It is a direct consequence of Theorem 2. If we increment II by one clock cycle, we have to increment L by $1 + \lfloor L/II \rfloor$ in order to guarantee the existence of at least one valid software pipelined schedule with the same register need:

$$\forall\sigma = (rn, cn, II) \in \Sigma_L(G) | II \leq L,$$

$$\exists\sigma' = (rn', cn', II + 1) \in \Sigma_{L+1+\lfloor L/II \rfloor}(G) :$$

$$RN_\sigma(G) = RN_{\sigma'}(G) \Longrightarrow RN_\sigma(G) \geq RN_{\sigma'(G)}$$

by implicit evidence. ■

Corollary 1: Let $G = (V, E, \delta, \lambda)$ be a DDG with zero delays in accessing registers. Then, the exact periodic register sufficiency of G assuming duration at most L is greater or equal to the minimal register need with $II = L$ assuming duration at most $L' \geq L$. L' is computed formally as follows:

$$\min_{\sigma=(rn,cn,II)\in\Sigma_L(G)} RN_\sigma(G) \geq \min_{\sigma=(rn,cn,L)\in\Sigma_{L'}(G)} RN_\sigma(G)$$

where L' is the $(L - MII)^{th}$ term of the following recurrent sequence ($L' = U_L$):

$$\begin{cases} U_{MII} & = & L \\ U_{II+1} & = & U_{II+1+\lfloor U_{II}/II \rfloor} \end{cases}$$

Proof: It is a direct consequence of Lemma 1:

$$\min_{\sigma=(rn,cn,II)\in\Sigma_L(G)} RN_\sigma(G) \geq$$

$$\geq \min_{\sigma=(rn,cn,II+1)\in\Sigma_{L+1+\lfloor L/II \rfloor}(G)} RN_\sigma(G) \geq ...$$

$$... \geq \min_{\sigma=(rn,cn,L)\in\Sigma'(G)} RN_\sigma(G)$$

where $\Sigma'(G) = \Sigma_{U_{L-II}=U_{L-II-1}+1+\lfloor U_{L-II-1}/(L-II-1)\rfloor}(G)$. That is, we relax (increase) the maximal duration L' when we increment II, starting from MII to L, i.e., $(L-II)$ times. Starting from $II = MII$ amounts to increase the maximal duration $(L - MII)$ times, as follows.

$$\min_{\sigma=(rn,cn,MII)\in\Sigma_{L=U_{MII}}(G)} RN_\sigma(G) \geq$$

$$\geq \min_{\sigma=(rn,cn,MII+1)\in\Sigma_{U_{MII+1}=L+1+\lfloor L/II \rfloor}(G)} RN_\sigma(G) \geq ...$$

$$... \geq \min_{\sigma=(rn,cn,L)\in\Sigma'(G)} RN_\sigma(G)$$

where $\Sigma'(G) = \Sigma_{U_L=U_{L-1}+1+\lfloor U_{L-1}/(L-1)\rfloor}(G)$ ■

In other words, Corollary 1 proves the following implication:

$$\begin{cases} \min RN_\sigma(G) \\ II = L \\ \sigma_u \leq L', \forall u \in V \end{cases} \Longrightarrow \begin{cases} \min RN_\sigma(G) \\ MII \leq II \leq L \\ \sigma_u \leq L, \forall u \in V \end{cases}$$

where the value of L' is given by Corollary 1.

Corollary 1 enables us to solve a unique problem of minimal register need under a fixed $II = L$; the maximal duration must be increased with the proved recurrent sequence. If the initial L is sufficiently large, the computed PRS with $II = L$ is equal to the absolute PRS, i.e., the minimal register need of any valid SWP of the loop. If L isn't sufficiently large, then we compute the PRS of the subset $\Sigma_L(G)$ which may be greater than the absolute PRS. Figure 6 draws the theoretical asymptotic curves to explain the meanings of Corollary 1. If we fix L as a maximal duration for all values of II, the minimal register need is not always a decreasing function of II. At a certain value of II, the minimal register need may increase if the duration L is not relaxed (evidence shown in Figure 5).

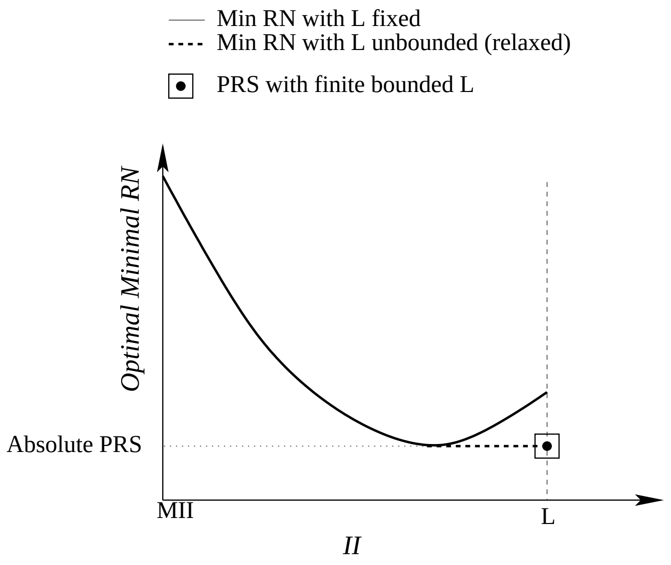


Fig. 6. Minimal Periodic Register Need and Periodic Register Sufficiency

We prove in Theorem 2 that the curve is a non-increasing function if the maximal duration L' is increased when we increment II (Lemma 1 and Corollary 1). If L is sufficiently large, the minimal register need at the point $II = L$ is exactly the absolute PRS. Such appropriate L is necessarily finite because PRS is a finite integer and hence there exists necessarily a SWP schedule that requires PRS registers. Computing formally a suitable finite large L remains an open problem. We think that $L = \sum_{u\in V} latency(u)$ would be convenient. It corresponds to the case when lifetime intervals may constitute a sequence of chains inside the SWP kernel. However, in practical cases, L should be bounded by the compiler (or by the user) in order to bound the prologue/epilogue code size. Thus, Corollary 1 gives us the way to compute the periodic register sufficiency for the class of schedules that belong to $\Sigma_L(G)$.

V. Stage Scheduling under Register Constraints

Stage scheduling, as studied in [3], is an approach that periodically schedules loop operations given a fixed II and a fixed reservation table (i.e., after satisfying resource constraints). In other terms, the problem is to compute the minimal register need given a fixed II and fixed row numbers (rn), while column numbers (cn) are left free (i.e., variables to optimize). This problem has been proved NP-complete by Huard in [7]. A careful study of his proof allows to deduce that the complexity of this problem comes from the fact that the last users of the values are not known before scheduling the loop. Mangione-Smith in [13] proved that stage scheduling under register constraints has a polynomial time complexity in the case of data dependence trees and forest of trees. This section proves a more general case than [13] by showing that if the killer is known before scheduling, as in the case of expression trees, then stage scheduling under register constraints is a polynomial problem. We will see that we are can deduce it by using the formula of register need given in Equation 1 (page 5). Before proving this general case, we first start by proving it for the case of trees (for clarity).

Let us begin by writing the formal problem of SWP with register need minimization:

$$\begin{cases} \text{Minimize} & RN_\sigma(G) \\ \\ \text{Subject to:} & \\ \sigma_v - \sigma_u \geq \delta(e) - II \times \lambda(e) & ,\forall e = (u,v) \in E \end{cases} \tag{3}$$

This standard problem has been proved NP-complete in [4], even for trees and chains. Eichenberger *et al.* studied a modified problem by considering a fixed reservation table. By considering the row and column numbers ($\sigma_u = rn(u) + II \times cn(u)$), fixing the reservation table amounts to fixing row numbers while letting column numbers as free integral variables. Thus, by considering the given row numbers as conditions, Problem 3 becomes:

$$\begin{cases} \text{Minimize} & RN_\sigma(G) \\ \\ \text{Subject to:} & \\ II \times cn(v) - II \times cn(u) & \\ \geq \delta(e) - II \times \lambda(e) - rn(v) + rn(u), & \forall e = (u,v) \in E \end{cases} \tag{4}$$

That is,

$$\begin{cases} \text{Minimize} & RN_\sigma(G) \\ \\ \text{Subject to:} & \\ cn(v) - cn(u) \geq & \\ \frac{\delta(e) - II \times \lambda(e) - rn(v) + rn(u)}{II}, & \forall e = (u,v) \in E \end{cases} \tag{5}$$

It is clear that the constraints matrix of Problem 5 constitutes an incidence matrix of the graph G. If we succeed in proving that the objective function $RN_\sigma(G)$ is a linear function of the cn variables, then Problem 5 becomes an integer linear programming system with a totally unimodular constraints matrix, and consequently, it can be solved with polynomial time algorithms [18]. Since the problem of stage scheduling defined by Problem 5 has been proved NP-complete, it is evident that $RN_\sigma(G)$ cannot be expressed as a linear function of cn for an arbitrary DDG. In this section, we restrict ourselves to the case of DDGs where each value $u \in V_R$ has a unique possible killer $k(u)$, such as the case of expression trees. In an expression tree, each value $u \in V_R$ has a unique killer k_u that belongs to the same original iteration, i.e., $\lambda\left((u, k_u)\right) = 0$. With this latter assumption, we will prove in the remaining of this section that $RN_\sigma(G)$ is a linear function of column numbers.

Let us begin by recalling the formula of $RN_\sigma(G)$ (see Page 5)

$$RN_\sigma(G) = \Big(\sum_{(l,r,p) \in C(G)} p\Big) + w\Big(\overline{C}(G)\Big) \tag{6}$$

The first term corresponds to the total number of turns around the circle, while the second term corresponds to the maximal fractional intervals simultaneously alive (the width of the circular fractional graph) . We set $P = \sum_{(l,r,p) \in C(G)} p$ and $W = w\Big(\overline{C}(G)\Big)$.

We know that $\forall (l,r,p) \in C(G)$ the circular interval of a value $u \in V_R$, its number of turns around the circle is $p = \left\lfloor \frac{lifetime_\sigma(u)}{II} \right\rfloor = \left\lfloor \frac{d_\sigma(u) - \sigma_u - \delta_w(u)}{II} \right\rfloor$.

Since each value u has a unique possible killer k_u belonging to the same original iteration (case of expression trees),

$$p = \left\lfloor \frac{\sigma_{k_u} - \sigma_u - \delta_w(u)}{II} \right\rfloor =$$
$$cn(k_u) - cn(u) + \left\lfloor \frac{rn(k_u) + \delta_r(k_u) - rn(u) - \delta_w(u)}{II} \right\rfloor$$

Here, we succeed in writing $P = \sum p$ as a linear function of column numbers cn, since rn and II are constants in Problem 5. Now, let's explore W. The fractional graph contains the fractional intervals $\{(l,r) | (l,r,p) \in C(G)\}$. Each fractional interval (l,r) of a value $u \in V_R$ depends only on the row numbers and II as follows:

- $l = (\sigma_u + \delta_w(u)) \bmod II = (rn(u) + II \times cn(u) + \delta_w(u)) \bmod II = (rn(u) + \delta_w(u)) \bmod II$;
- $r = d_\sigma(u) \bmod II = (\sigma_{k_u} + \delta_r(k_u)) \bmod II = (rn(k_u) + II \times cn(k_u) + \delta_r(k_u)) \bmod II = (rn(k_u) + \delta_r(k_u)) \bmod II$.

As can be seen, the fractional intervals depends only on row numbers and II which are constants in Problem 5. Hence, W, the width of the circular fractional graph is a constant too. From all the previous formulas, we deduce that:

$$RN_\sigma(G) = P + W =$$
$$\sum_{u \in V_R} cn(k_u) - cn(u) +$$
$$+ \left\lfloor \frac{rn(k_u) + \delta_r(k_u) - rn(u) - \delta_w(u)}{II} \right\rfloor + W$$

yielding to:

$$RN_\sigma(G) = \sum_{u \in V_R} cn(k_u) - cn(u) + constant \tag{7}$$

Equation 7 rewrites Problem 5 as the following integer linear programming system (by neglecting the constants in the objective function):

$$\begin{cases} \text{Minimize } \sum_{u \in V_R} cn(k_u) - cn(u) \\ \\ \text{Subject to:} \\ cn(v) - cn(u) \geq \left\lfloor \frac{\delta(e) - II \times \lambda(e) - rn(v) + rn(u)}{II} \right\rfloor, \\ \forall e = (u,v) \in E \end{cases} \tag{8}$$

The constraints matrix of System 8 describes an incidence matrix, so it is totally unimodular. It can be solved with a polynomial time algorithm.

This sections proves that stage scheduling of expression trees is a polynomial problem. Now, we can consider the larger case of the DDGs assigning a unique possible killer k_u for each value u. Such killer can belong to a different iteration $\lambda_k = \lambda(u, k_u)$. Then, the problem of stage scheduling in this class of loops remains also polynomial, as follows.

1) if the DDG is acyclic, then we can apply a loop retiming [11] to bring all the killers to the same iteration. Thus, we

come back to the case similar to expression trees studied in this section;

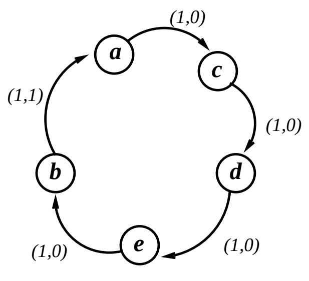
2) if the DDG contains cycles, it is not always possible to shift all the killers to the same iteration. Thus, by including the constants λ_k in the formula P becomes equal to:

$$\begin{aligned} P &= \sum_{u \in V_R} cn(k_u) - cn(u) + \lambda_k + \\ &\quad + \left\lfloor \frac{rn(k_u) + \delta_r(k_u) - rn(u) - \delta_w(u)}{II} \right\rfloor \\ &= \sum_{u \in V_R} cn(k_u) - cn(u) + constant \end{aligned}$$

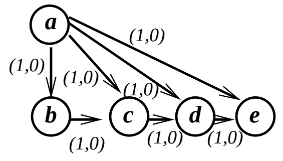
Since II and row numbers are constants, W remains a constant as proved by the following formulas of fractional intervals:

- $l = \sigma_u + \delta_w(u) \bmod II = (rn(u) + II \times cn(u) + \delta_w(u)) \bmod II = (rn(u) + \delta_w(u)) \bmod II$;
- $r = d_\sigma(u) \bmod II = \sigma_{k_u} + II \times \lambda_k + \delta_r(k_u) \bmod II = (rn(k_u) + II \times cn(k_u) + II \times \lambda_k + \delta_r(k_u)) \bmod II = (rn(k_u) + \delta_r(k_u)) \bmod II$.

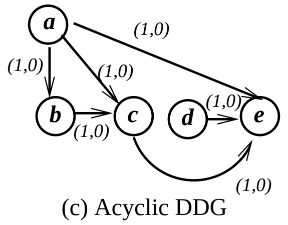
Consequently, $RN_\sigma(G)$ remains a linear function of column numbers, which means that System 8 can still be solved via polynomial time algorithms (usually with network flow algorithms).



(a) Cyclic DDG



(b) Acyclic DDG



(c) Acyclic DDG

Fig. 7. Examples of DDGs with Unique Possible Killer per Value

Our result in this section is more general than expression trees. We extend the previous result [13] in two ways. Figure 7 shows some examples, where all edges are flow dependences labeled by the pairs $(\delta(e), \lambda(e))$.

1) **Cyclic DDGs**: Our result takes into account cyclic DDGs with a unique killer per value. As an example, Figure 7(a) is a cyclic DDG with a unique possible killer per value. Such DDG is not considered in [13] because it is cyclic while it is neither a tree nor an acyclic DDG.
2) **Acyclic DDG**: Our result also takes into account acyclic DDGs with a unique possible killer per value, which are not necessarily trees or forest of trees. For instance, Figure 7(b) and Figure 7(c) are examples of acyclic DDG where every node has a unique possible killer (because of the transitive relationship between nodes). These DDGs are not trees. Analysing such unique killer relationship in general acyclic DDGs can be done using the so-called *potential killing relation* which has been formally defined in [20]. In Figure 7(b), we have the following unique killers: $k(a) = e, k(b) = c, k(c) = d, k(d) = e$. In Figure 7(c), we have the following unique killers: $k(a) = e, k(b) = c, k(c) = e, k(d) = e$. All these killing relationships can be deduced by analysing the potential killing relation of the DAG [20].

VI. Conclusion

The work presented in this article uses formal methods and reasoning to prove new interesting assertions in the problem of minimizing periodic register need in periodic scheduling. The first contribution brings a novel polynomial method for computing the exact register need (RN) of an already scheduled loop ($\mathcal{O}(n \lg n)$, where n is the number of statements). The complexity of the existing methods depends on II, which is a pseudo-polynomial factor.

Our second contribution provides a sufficient condition so that the minimal register need under a fixed II does not increase when incrementing II. We give an example to show that it is sometimes possible that the minimal register need increases when II is incremented. Such situation may occur when the maximal duration L is not relaxed (increased). This fact contradicts the general thought that incrementing II would require fewer registers (unless the constraint on L is loosened).

Guaranteeing that register need is a non-increasing function vs. II when relaxing the maximal duration allows now to easily write the formal problem of scheduling under register constraints instead of scheduling with register minimization as usually done in the literature. Indeed, according to our results, we can finally apply a binary search on II. If we have R, a fixed number of available registers, and since we know how to increase L so as the curve of RN vs. II becomes non-increasing, we can use successive binary search on II until reaching a RN below R. The number of such binary search steps is at most $log_2(L)$.

Our third contribution in this paper proves that computing the minimal register need with a fixed $II = L$ is exactly equal to the periodic register sufficiency if L sufficiently large, i.e., the minimal register need of all valid SWP schedules. Computing the periodic register sufficiency (PRS) allows to check for instance if introducing spill code is unavoidable when PRS is greater than the number of available registers.

While stage scheduling under registers constraints for arbitrary loops is an NP-complete problem, our fourth and last contribution proves that stage scheduling with register minimization is a polynomial problem in the special case of expression trees, and generally in the case of DDGs providing a unique possible killer per value. This generalization is made possible thanks to our new polynomial method of RN computation.

This article proposes new open problems. First, an interesting open question would be to provide a necessary condition so that the register need would be a non-increasing function of II. Second, in the presence of architectures with non-zero delays in accessing registers, is Theorem 2 still valid ? In other words, can we provide any guarantee that minimal register need in such architectures does not increase when incrementing II ? Third, we have shown that there exists a finite value of L such that the periodic register sufficiency assuming a maximal duration L is equal to the absolute periodic register sufficiency without assuming any bound on the duration. The open question is how to compute such appropriate value of maximal duration. Fourth and last, we require a DDG analysis algorithm to check whether each value has one and only one possible killer. We already have published such algorithm for the case of directed acyclic graphs [20], but the problem here is to extend it to cyclic graphs.

ACKNOWLEDGEMENT

Most of the research results of the current article (except Section V) were found thanks to the valuable support of Christine EISENBEIS from INRIA. The result of Section V was found thanks to the support of Pr. William JALBY from UVSQ. I wish to thank Alain DARTE from ENS-Lyon for his scientific influence. Finally, great advices and corrections have been made by colleagues and anonymous reviewers. This work has been partially supported by the French National Research Agency (ANR MOPUCE project).

REFERENCES

[1] E. Altman. *Optimal Software Pipelining with Functional Units and Registers*. PhD thesis, McGill University, Montreal, Oct. 1995.

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

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

[4] C. Eisenbeis, F. Gasperoni, and U. Schwiegelshohn. Allocating Registers in Multiple Instruction-Issuing Processors. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT'95*, pages 290–293. ACM Press, June 27–29, 1995.

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

[6] M. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Acadmeic Press, New York, 1980.

[7] Guillaume Huard. *Algorithmique du décalage d'instructions*. PhD thesis, École Normale Supérieure, Lyon, France, Dec. 2001.

[8] L. J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji. A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs. *Lecture Notes in Computer Science*, 641:176–??, 1992.

[9] R. Huff. Lifetime-Sensitive Modulo Scheduling. In *PLDI 93*, pages 258–267, Albuquerque, New Mexico, June 1993.

[10] J. Janssen. *Compilers Strategies for Transport Triggered Architectures*. PhD thesis, Delft University, Netherlands, 2001.

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

[12] S. Lelait. *Contribution à l'Allocation de Registres dans les Boucles*. PhD thesis, Université d'Orléans, France, Jan. 1996.

[13] W. Mangione-Smith, S. G. Abraham, and E. S. Davidson. Register Requirements of Pipelined Processors. In *International Conference on Supercomputing, Washington, DC*, pages 260–271, New York, NY 10036, USA, July 1992. ACM Press.

[14] J. Müller, D. Fimmel, and R. Merker. Optimal Loop Scheduling with Register Constraints Using Flow Graphs. In *ISPAN*, pages 180–186. IEEE, 2004.

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

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

[17] A. Sawaya. *Pipeline Logiciel: Découplage et Contraintes de Registres*. PhD thesis, Université de Versailles Saint-Quentin-En-Yvelines, Apr. 1997.

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

[19] R. Sethi. Complete register allocation problems. *SIAM Journal on Computing*, 4(3):226–248, 1975.

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

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

[22] A. Tucker. Coloring a Family of Circular Arcs. *SIAM Journal on Applied Mathematics*, 29(3):493–502, Nov. 1975.

[23] J. Wang, C. Eisenbeis, M. Jourdan, and B. Su. DEcomposed Software Pipelining: A new perspective and a new approach. *International Journal of Parallel Programming*, 22(3):351–373, June 1994.

[24] J. Wang, A. Krall, M. A. Ertl, and C. Eisenbeis. Software Pipelining with Register Allocation and Spilling. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 95–99, San Jose, California, Nov. 1994. ACM SIGMICRO and IEEE Computer Society TC-MICRO.

Sid-Ahmed-Ali TOUATI received a computer engineer diploma in 1997 from the "Institut National d'Informatique d'Alger", and received a master degree in computer science in 1998 from the "École Normale Supérieure de Lyon" (France). Then, he joined the research team of Christine EISENBEIS at the INRIA French Laboratory during four years. He got a Ph.D. in 2002 from the University of Versailles under the direction of Pr. William JALBY. Sid TOUATI is currently an assistant professor at the University of Versailles. His main scientific interests are fundamental and practical research topics about code analysis and optimisation, instruction level parallelism, and compilation techniques for embedded processors. At present, he is managing some research and Ph.D. projects on these topics, with the active collaboration of INRIA and ST-microelectronics.

Appendix I
Proof of Theorem 1

Let $\underline{C}(G)$ be a circular fractional graph (no complete turns around the circle exists). For each circular fractional interval $(l,r) \in C(G)$, we associate the two corresponding acyclic intervals I and I'. The cardinality of any maximal clique in the interference graph of all these acyclic intervals is equal to the width of $\underline{C}(G)$

Proof: In this section, we prove that the width of the acyclic fractional intervals graph after unrolling the kernel once is equal to the width of $\underline{C}(G)$, i.e., the maximal number of simultaneously alive values in the kernel. Figure 8 gives an example to help the understanding of this proof (note that the intervals here are plotted top to bottom instead of left to right).

After unrolling the kernel, each circular interval (l,r) becomes two acyclic intervals I and I'. We denote by G_a the graph of all these acyclic intervals. Unrolling the kernel once does not change the width of the interval graph. This is because the unrolling does not break the periodic schedule, it only exhibits the lifetime intervals during two successive kernels instead of one. Note that for each circular interval $(l,r) \in \underline{C}(G)$, its corresponding acyclic intervals I and I' cannot interfere because their lengths are less than II. Furthermore, the interval I precedes I' (i.e., $I \prec I'$) necessarily because:

- if $r \geq l$, i.e., $I =]l,r]$ and $I' =]l+II, r+II]$, then $(l+II)-r \geq 0$ because the length $r-l < II$;
- if $r < l$ i.e. $I =]l, r+II]$ and $I' =]l+II, r+2\times II]$, then the length $r+II-l < II$ and hence $(l+II)-(r+II) \geq 0$.

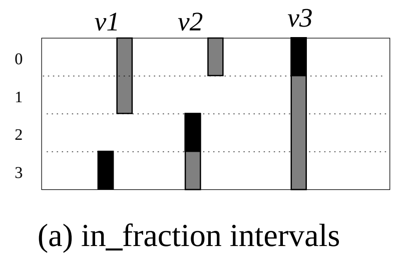
All the acyclic fractional intervals belong to a whole window $[0, 3\times II[$, as satisfied by the following inequalities: $0 \leq l, r < II$ and $r + 2\times II \leq 3\times II$.

Now, let us examine the whole window $[0, 3\times II[$, as shown in Figure 8(b). We consider the acyclic intervals graph G_a that we truncate into three parts (three windows). Let G_1 be the graph that contains the truncated acyclic intervals belonging to $[0, II[$, G_2 be the graph that contains the truncated acyclic intervals belonging to $[II, 2\times II[$ and G_3 be the graph that contains the truncated acyclic intervals belonging to $[2\times II, 3\times II[$. Below, we analyze the width of each of the intervals graphs G_1, G_2 and G_3.

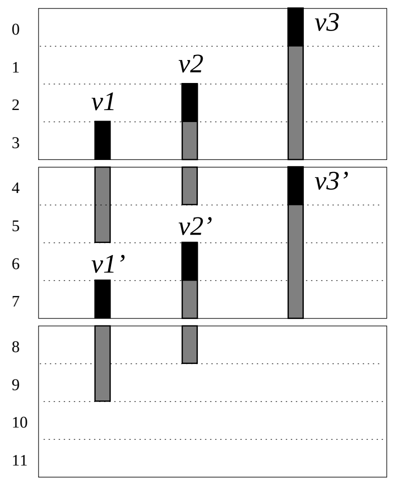
a) For the width of G_1, we consider the window $[0, II[::$ $\forall(l,r) \in \underline{C}(G)$ we have,

- $r \geq l \Longrightarrow I =]l,r] \wedge I \subseteq [0, II[\wedge I' =]l+II, r+II] \wedge I' \cap [0, II[= \emptyset$
- $r < l \Longrightarrow I' =]l+II, r+2\times II] \wedge I' \cap [0, II[= \emptyset$. The interval $I =]l, r+II]$ can be decomposed into two parts I_1 and I_2:
  1) $I_1 =]l, II[\wedge I_1 \cap \subseteq [0, II[$
  2) $I_2 = [II, r+II] \wedge I_2 \cap [0, II[= \emptyset$

G_1 then contains all the acyclic intervals $]l,r]$ when $r \geq l$, and all the acyclic intervals $]l, II[$ otherwise. In other words, the window $[0, II[$ contains the intervals of the SWP kernel except the right parts which go through the II barrier (for instance, the right parts of v_1 and v_2 in Figure 8(b)). Since G_1 is a subset of $\underline{C}(G)$, then $w(G_1) \leq w(\underline{C}(G))$. So the register need in this window is less than or equal to the register need of the SWP kernel.



(a) in_fraction intervals



(b) acyclic in_fraction intervals

Fig. 8. Acyclic Fractional Intervals

b) For the width of G_2, we consider the window $[II, 2\times II[::$ $\forall(l,r) \in \underline{C}(G)$ we have,

- $r \geq l \Longrightarrow I =]l,r] \wedge I \cap [II, 2\times II[= \emptyset \wedge I' =]l+II, r+II] \wedge I' \subseteq [II, 2\times II[$
- $r < l \Longrightarrow I =]l, r+II] \wedge I' =]l+II, r+2\times II]$. Each of I and I' can be decomposed into two parts ($I = I_1 \cup I_2$, $I' = I'_1 \cup I'_2$) as follows:
  1) $I_1 =]l, II[\wedge I_1 \cap [II, 2\times II[= \emptyset \wedge I_2 = [II, r+II[\wedge I_2 \subseteq [II, 2\times II[$
  2) $I'_1 =]l+II, 2\times II[\wedge I'_1 \subseteq [II, 2\times II[\wedge I'_2 = [II, r+2\times II[\wedge I'_2 \cap [II, 2\times II[= \emptyset$

G_2 then contains all the acyclic intervals $]l+II, r+II]$ when $r \geq l$, and all the acyclic intervals $[II, r+II]$ and $]l+II, 2\times II[$ when $r < l$. In other words, the window $[II, 2\times II[$ contains exactly all the intervals of the SWP kernel. Thus, $w(G_2) = w(\underline{C}(G))$; the register need in this window is equal to the register need of the SWP kernel.

c) For the width of G_3, we consider the window $[2\times II, 3\times II[::$ $\forall(l,r) \in \underline{C}(G)$ we have,

- $r \geq l \Longrightarrow I =]l,r] \wedge I \cap [2\times II, 3\times II[= \emptyset \wedge I' =]l+II, r+II] \wedge I' \cap [2\times II, 3\times II[= \emptyset$
- $r < l \Longrightarrow I =]l, r+II] \wedge I \cap [2\times II, 3\times II[= \emptyset$. $I' =]l+II, r+2\times II]$ can be decomposed into two parts I'_1 and I'_2 as follows:
  1) $I'_1 =]l+II, 2\times II[\wedge I'_1 \cap [2\times II, 3\times II[= \emptyset$
  2) $I'_2 =]2\times II, r+2\times II[\wedge I'_2 \subseteq [2\times II, 3\times II[$

G_3 then contains all the acyclic intervals $[2 \times II, r + 2 \times II[$ when $r < l$. In other words, the window $[2 \times II, 3 \times II[$ contains the intervals of the original kernel except the left parts of the intervals coming from the previous window $[II, 2 \times II[$ (for instance, the left parts of v'_1 and v'_2 in Figure 8(b)). Since G_3, is a subset of $\underline{\overline{C}}(G)$, then $w(G_3) \leq w(\underline{\overline{C}}(G))$. Thus, the register need of this window is less than or equal to the register need of the SWP kernel.

From the last three paragraphs, we conclude: $w(G_a) = \max(w(G_1), w(G_2), w(G_3)) \implies w(G_a) = w(\underline{\overline{C}}(G))$. In other words, in the window $[0, 3 \times II[$, the maximal number of simultaneously alive values is exactly the same as in the original SWP kernel. Since the width of the acyclic intervals graph G_a is equal to the cardinality of any maximal clique in the interference graph of these acyclic intervals, then consequently, the cardinality of any maximal clique in the interference graph of these acyclic intervals is equal to $w(\underline{\overline{C}}(G))$ the width of the circular fractional graph. ■

APPENDIX II
PROOF OF THEOREM 2

Let $G = (V, E, \delta, \lambda)$ be a DDG with zero delays in accessing registers. If there exists a SWP schedule $\sigma = (rn, cn, II)$ which needs R registers having a duration at most L, then there exists a SWP schedule $\sigma' = (rn', cn', II{+}1)$ which also needs R registers having a duration at most $L' = L + 1 + \lfloor L/II \rfloor$. Formally:

$$\forall \sigma = (rn, cn, II) \in \Sigma_L(G),$$

$$\exists \sigma' = (rn', cn', II + 1) \in \Sigma_{L+1+\lfloor L/II \rfloor}(G) :$$

$$RN_{\sigma'}(G) = RN_\sigma(G)$$

.

Proof: This theorem is only proved for the case of architecturally invisible delays in reading from and writing into registers (as superscalar semantics, EPIC, and VLIW with zero reading/writing delays). The general case (VLIW with non-zero reading/writing delays) is more difficult to prove, it remains an open problem (explained at the end of the section).

First, recall that row numbers $rn(u) = \sigma(u) \bmod II$ and column numbers $cn(u) = \lfloor \frac{\sigma_u}{II} \rfloor$.

Let $\sigma = (rn, cn, II)$ be a valid software pipelined schedule for G. In this proof, we show how to construct another schedule $\sigma' = (rn', cn', II{+}1)$ which needs the same number of registers as σ. Since we assume that reading and writing delays are equal to zero ($\delta_r = \delta_w = 0$), consequently the lifetime intervals begin and end exactly at schedule dates.

We use Figure 9 to illustrate this proof: Figure 9(c) is the kernel of a valid σ for the loop of Figure 9(a) with null writing and reading delays, where the execution rate is $II = 4$. Figure 9(b) shows the pipelined execution of the loop. The register need is 2 as shown by the lifetime intervals in the SWP kernel in Figure 9(d). In order to construct another σ' with $II' = 5$, we proceed by inserting a complete row of static nops (no-operations) in the kernel of σ without changing the maximal number of simultaneously alive values, while preserving the validity of the schedule.

In order to add the row of nops in the kernel of σ, we choose an arbitrary clock cycle $0 \leq c < II$. In Figure 9(d), we take for instance the clock cycle 1, which is an excessive cycle. Then, we "shift" downwards by one clock cycle all the statements scheduled by σ during or after the row c, as illustrated in Figure 9(e). We let the other statements unchanged (those scheduled strictly before the row c). The new kernel $\sigma' = (rn', cn', II + 1)$ is formally defined as follows.

Definition 3 (Incremented SWP): Let $G = (V, E, \delta, \lambda)$ be a DDG loop and $\sigma \in \Sigma(G)$ a valid SWP for it. We define $\sigma' = (rn', cn', II + 1)$ an incremented SWP of σ as

- its initiation interval is $II' = II + 1$ by definition.

$$\max_{u \in V}(rn'(u)) = \max_{u \in V}(rn(u) + 1) =$$

$$= \max_{u \in V}(rn(u)) + 1 \leq II + 1 = II'$$

- $\forall u \in V, cn'(u) = cn(u)$
- $\forall u \in V$

$$rn'(u) = \begin{cases} rn(u) & \text{if } rn(u) < c \\ rn(u) + 1 & \text{otherwise} \end{cases}$$

Now, we prove the following three main assertions:

1) the constructed schedule σ' is valid. See Lemma 2
2) the new maximal duration $L' = L + 1 + \lfloor L/II \rfloor$. See Lemma 3
3) $RN_\sigma(G) = RN_{\sigma'}(G)$

The two first points are proved in their corresponding lemmas, we prove the third point in the following paragraph.

d) The register need remains unchanged:: As can be observed in the example of Figure 9, we have:

- the simultaneously alive values during the row c in the previous kernel are exactly the same in the new kernel;
- the interferences between the lifetime intervals remain unchanged in the new kernel, see Figure 9(f): our transformation is similar to considering that the clock cycle c has been decomposed into two virtual clock cycles.

In order to prove that the register need of σ is equal to the register need of σ', we prove that both the complete turns around the circle do not change, and the interferences between the circular fractional intervals remain identical.

Let us begin by proving that the number of turns around the circle (distinct copies of each value) do not change. Let $(l, r, p) \in C(G)$ be a circular lifetime interval of a value $u \in V_R$ according to the initial schedule σ. We have:

$$p = \left\lfloor \frac{lifetime_\sigma(u)}{II} \right\rfloor = \left\lfloor \frac{d_\sigma - \sigma_u}{II} \right\rfloor$$

Let k_u be one of the killers of u. And let $e = (u, k_u)$. Then:

$$p = \left\lfloor \frac{rn(k_u) + II \times (cn(k_u) + \lambda(e)) - rn(u) - II \times cn(u)}{II} \right\rfloor$$

$$= \left\lfloor \frac{rn(k_u) - rn(u) + II \times (cn(k_u) - cn(u) + \lambda(e))}{II} \right\rfloor$$

$$= \left\lfloor \frac{rn(k_u) - rn(u)}{II} \right\rfloor + cn(k_u) - cn(u) + \lambda(e)$$

Since $\forall u \in V, 0 \leq rn(u) < II \implies -II < rn(k_u) - rn(u) < II$, we deduce that

$$\left\lfloor \frac{rn(k_u) - rn(u)}{II} \right\rfloor = 0$$

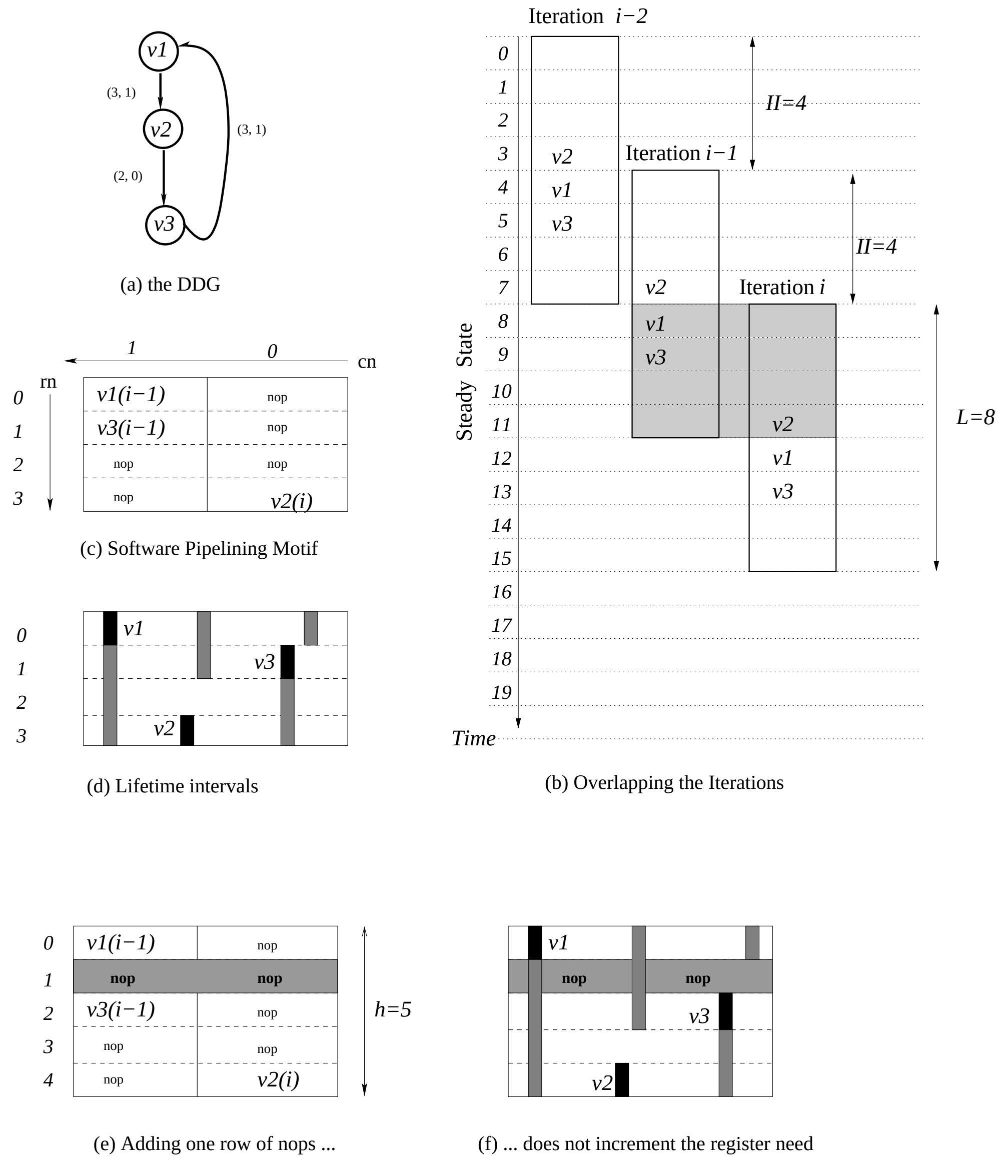


Fig. 9. Adding a Row of Nops Does Not Change the Register Need

Consequently,

$$p = cn(k_u) - cn(u) + \lambda(e) \tag{9}$$

Now, let us compute p', the new complete turn around the circle according the new schedule σ'.

For the clarity and the fluidity of the actual proof, assume at this point that a killer k_u of u according to σ is still a killer of u according to σ'. This assertion is proved outside this context in Lemma 4 following the actual proof. Now, an initial circular interval $(l, r, p) \in C(G)$ is transformed to (l', r', p') where:

$$p' = \left\lfloor \frac{rn'(k_u) + II' \times (cn'(k_u) + \lambda(e)) - rn'(u) - II' \times cn'(u)}{II'} \right\rfloor$$

$$= \left\lfloor \frac{rn'(k_u) - rn'(u) + II' \times (cn'(k_u) - cn'(u) + \lambda(e))}{II'} \right\rfloor$$

$$= \left\lfloor \frac{rn'(k_u) - rn'(u)}{II'} \right\rfloor + cn'(k_u) - cn'(u) + \lambda(e)$$

Since $\forall u \in V, 0 \leq rn'(u) < II' \Longrightarrow -II' < rn(k_u) - rn(u) < II'$, we deduce that

$$\left\lfloor \frac{rn'(k_u) - rn'(u)}{II'} \right\rfloor = 0$$

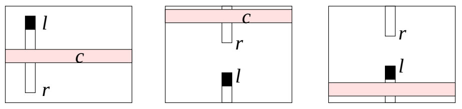
Consequently, and since $cn' = cn$ by definition of σ', we have

$$p' = cn(k_u) - cn(u) + \lambda(e) \tag{10}$$

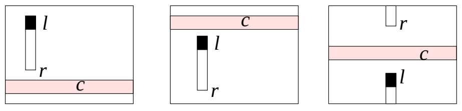
From Equation 9 and Equation 10, we deduce that $\forall (l, r, p) \in C(G)$ transformed to (l', r', p') by construction of the new kernel σ', we have

$$\sum_{(l',r',p'))} p' = \sum_{(l,r,p) \in C(G)} p \tag{11}$$

Now, after proving that the number of complete turns around the circle of σ' is identical to the one of σ, we now have to prove that the interferences between the fractional intervals remain identical. By considering the clock cycle c where we added a row of nops to build the new schedule σ', we have to prove that:



(a) c belongs to the fractional interval



(b) c does not belong to the fractional interval

Fig. 10. Adding a row c of nops does not alter the interferences between the intervals

1) the circular fractional intervals containing the date c in the previous kernel are the same in the new kernel. That is, if c belongs to a circular fractional interval in the SWP kernel of σ, it still belongs to the same circular fractional interval in the SWP kernel of σ'.
2) the interferences between the circular fractional intervals remain unchanged in the new kernel. That is, if c does not belong to a circular fractional interval in the SWP kernel of σ, it still does not belongs to the same circular fractional interval in the SWP kernel of σ'.

Let $(l, r) \in \underline{C}(G)$ be a circular fractional interval in the SWP kernel of σ that corresponds to the lifetime interval of a value $u \in V_R$. Since $\delta_r = \delta_w = 0$, we have $l = rn(u)$ and $r = rn(k_u)$. By construction of σ', (l, r) is transformed to (l', r') such that:

$$l' = \begin{cases} rn(u) & \text{if } rn(u) < c \\ l+1 & \text{otherwise} \end{cases}$$

$$\Longrightarrow l' = \begin{cases} l & \text{if } l < c \\ l+1 & \text{otherwise} \end{cases}$$

$$r' = \begin{cases} rn(k_u) & \text{if } rn(k_u) < c \\ rn(k_u)+1 & \text{otherwise} \end{cases}$$

$$\Longrightarrow r' = \begin{cases} r & \text{if } r < c \\ r+1 & \text{otherwise} \end{cases}$$

First, suppose that c initially belongs to the fractional interval (l, r). We have three distinct cases (see Figure 10(a)):

1) the case where $l < r$ and $l < c \le r$, see the first part of Figure 10(a). Then, $(l', r') = (l, r+1)$. Since $l' = l \wedge r' = r+1 \wedge l' < c < r'$, then c still belongs to (l', r').
2) if $l \ge r$ and $0 \le c \le r$, see the second part of Figure 10(a). Then, $(l', r') = (l+1, r+1)$. Since $l' = l+1 \wedge r' = r+1 \wedge l' \ge r' \wedge 0 \le c < r'$, then c still belongs to (l', r').
3) if $l \ge r$ and $l < c < II$, see the last part of Figure 10(a). Then, $(l', r') = (l, r)$. Since $l' = l \wedge r' = r \wedge l' < c < II+1$, then c still belongs to (l', r').

Second, suppose that c does not initially belong to the fractional interval (l, r). We have three distinct cases (see Figure 10(b)):

1) the case where $l < r$ and $r < c < II$, see the first part of Figure 10(b). Then, $(l', r') = (l, r)$. Since $l' = l \wedge r' = r \wedge l' < r' \wedge r' < c < II+1$, then c still does not belong to (l', r').
2) if $l < r$ and $0 \le c \le l$, see the second part of Figure 10(b). Then, $(l', r') = (l+1, r+1)$. Since $l' = l+1 \wedge r' = r+1 \wedge l' < r' \wedge 0 \le c < l'$, then c still does not belong to (l', r').
3) if $l \ge r$ and $r < c \le l$, see the last part of Figure 10(b). Then, $(l', r') = (l+1, r)$. Since $l' = l+1 \wedge r' = r \wedge l' > r' \wedge r' < c < l+1$, then c still does not belong to (l', r').

From all the above cases, we deduce that inserting a row of nops during a considered clock cycle c does not modify the interferences between the circular fractional intervals. Consequently, and from Equation 11, we deduce that:

$$RN_{\sigma'}(G) = RN_{\sigma}(G)$$

Finally, the reason why the case of VLIW (with non-zero reading/writing delays) is more difficult to prove is that such architectures exhibit to the compiler the reading and writing delays (offsets) from/into registers. Adding a row of nops inside a VLIW basic bloc may violate some flow dependences through registers because they are architecturally visible. ■

Lemma 2: Let $G = (V, E, \delta, \lambda)$ be a DDG loop and $\sigma \in \Sigma(G)$ a valid SWP for it. If $\sigma' = (rn', cn', II+1)$ is an incremented SWP of σ, then σ' is a valid SWP (it does not violate any edge of G).

Proof: The original σ is assumed a valid schedule which means according to [17]: $\forall e = (u, v)$,

$$rn(v) - rn(u) + II \times (\lambda(e) + cn(v) - cn(u)) \ge \delta(e)$$

$$\text{and } \lambda(e) + cn(v) - cn(u) \ge 0.$$

For the constructed schedule $\sigma' = (rn', cn', II')$, we have to check that: $\forall e = (u, v) \in E$,

$$rn'(v) - rn'(u) + (II+1)(\lambda(e) + cn'(v) - cn'(u)) \ge \delta(e) \quad (12)$$

There are four cases:

1) if $rn(u) < c$ and $rn(v) < c$ then $rn(u)' = rn(u)$ and $rn'(v) = rn(v)$. Since $cn' = cn$, Equation 12 becomes:
$$rn(v) - rn(u) + II \times (\lambda(e) + cn(v) - cn(u)) +$$
$$+ (\lambda(e) + cn(v) - cn(u)) \ge \delta(e)$$
which is satisfied because $\lambda(e) + cn(v) - cn(u) \ge 0$ and σ is valid;
2) if $rn(u) < c$ and $rn(v) \ge c$ then $rn(u)' = rn(u)$ and $rn'(v) = rn(v) + 1$. Equation 12 becomes:
$$rn(v) - rn(u) + II \times (\lambda(e) + cn(v) - cn(u)) +$$
$$+ (\lambda(e) + cn(v) - cn(u)) + 1 \ge \delta(e)$$
which is satisfied because $\lambda(e) + cn(v) - cn(u) \ge 0$ and σ is valid;
3) if $rn(u) \ge c$ and $rn(v) < c$ then $rn(u)' = rn(u) + 1$ and $rn'(v) = rn(v)$. The fact that $rn(u) > rn(v)$ means that the dependence $e = (u, v)$ is necessarily an inter kernel one (inter-iteration), i.e., it is satisfied by the sequential execution of kernel iterations. Then, $\lambda(e) + cn(v) - cn(u) > 0$ necessarily [17]. Equation 12 becomes:
$$rn(v) - rn(u) + II \times (\lambda(e) + cn(v) - cn(u)) +$$
$$+ (\lambda(e) + cn(v) - cn(u)) - 1 \ge \delta(e)$$
which is satisfied because $\lambda(e) + cn(v) - cn(u) > 0$ and σ is valid;
4) if $rn(u) \ge c$ and $rn(v) \ge c$ then $rn'(u) = rn(u) + 1$ and $rn'(v) = rn(v) + 1$. Equation 12 becomes:
$$rn(v) - rn(u) + II \times (\lambda(e) + cn(v) - cn(u)) +$$

$$+(\lambda(e)+cn(v)-cn(u))+1-1\geq\delta(e)$$

which is satisfied because $\lambda(e)+cn(v)-cn(u)\geq 0$ and σ is valid;

■

Lemma 3: Let $G=(V,E,\delta,\lambda)$ be a DDG loop and $\sigma\in\Sigma_L(G)$ a valid SWP for it with a maximal duration equal to L. Le $\sigma'=(rn',cn',II+1)$ be an incremented SWP. Then, the maximal duration of σ' is equal to $L'=L+1+\lfloor L/II\rfloor$

Proof:

let us compute the maximal duration L' with the constructed schedule σ'. $\forall u\in V$:

$$\begin{aligned}\sigma'_u &= rn'(u)+cn'(u)\times II'\\ &= rn'(u)+cn(u)\times(II+1)\\ &\leq rn(u)+cn(u)\times II+1+cn(u)\\ &\leq \sigma_u+1+\lfloor L/II\rfloor\\ &\leq L+1+\lfloor L/II\rfloor=L'\end{aligned}$$

■

Lemma 4: Let $G=(V,E,\delta,\lambda)$ be a DDG of a loop with zero delays in accessing registers. Let $\sigma=(rn,cn,II)\in\Sigma(G)$ be a valid SWP for G, and $\sigma'=(rn',cn',II')\in\Sigma(G)$ another SWP constructed from σ by adding a row of nops during a clock cycle c inside the kernel ($0\leq c<II$). σ' is formally defined as follows:

- $II'=II+1$
- $\forall u\in V, cn'(u)=cn(u)$
- $\forall u\in V$,

$$rn'(u)=\begin{cases}rn(u) & \text{if } rn(u)<c\\ rn(u)+1 & \text{otherwise}\end{cases}$$

Therefore, $\forall u\in V_R$, if k_u is a killer of u according to σ, then it is also a killer of u according to σ'.

Proof: First, the proof of Theorem 2 shows that if σ is a valid SWP then σ' is also a valid SWP. We recall that, $Cons(u)$ is the set of consumers (readers) of the value $u\in V_R$. Let denote by $killers_\sigma(u)\subseteq Cons(u)$, the set of all the killers of the value u when considering the schedule σ. We have to prove that: $k_u\in killers_\sigma(u)\implies k_u\in killers_{\sigma'}(u)$. Suppose the contrary is true, that is: $k_u\in killers_\sigma(u)\wedge k_u\notin killers_{\sigma'}(u)$. We want to finish with a contradiction.

$$k_u\in killers_\sigma(u)\implies\forall v\in Cons(u)|e=(u,k_u)\in E_R,$$
$$e'=(u,v)\in E_R:\sigma_{k_u}+II\times\lambda(e)\geq\sigma_v+II\times\lambda(e')$$
$$\implies rn(k_u)+II\times(cn(k_u)+\lambda(e))\geq$$
$$\geq rn(v)+II\times(cn(v)+\lambda(e'))$$
$$\implies rn(k_u)-rn(v)\geq II\times(-cn(k_u)+cn(v)-\lambda(e)+\lambda(e'))$$

Since $II>rn(k_u)-rn(v)$, then $II>II\times(-cn(k_u)+cn(v)-\lambda(e)+\lambda(e'))$. And since $II>0$, we have $1>-cn(k_u)+cn(v)-\lambda(e)+\lambda(e')$ which means:

$$k_u\in killers_\sigma(u)\implies\forall v\in Cons(u),$$
$$-cn(k_u)+cn(v)-\lambda(e)+\lambda(e')\leq 0\qquad(13)$$

Now, consider our assumption that $k_u\notin killers_{\sigma'}(u)$ which implies that:

$$\exists v\in Cons(u)-\{k_u\}|e=(u,k_u)\in E_R,e'=(u,v)\in E_R:$$
$$\sigma'_{k_u}+II'\times\lambda(e)<\sigma'_v+II'\times\lambda(e')$$
$$\implies rn'(k_u)+II'\times(cn'(k_u)+\lambda(e))<$$
$$<rn'(v)+II'\times(cn'(v)+\lambda(e'))$$
$$\implies rn'(k_u)-rn'(v)<$$
$$<II'\times(-cn'(k_u)+cn'(v)-\lambda(e)+\lambda(e'))$$

Since $-II'<rn'(k_u)-rn'(v)$, then $-II'<II'\times(-cn'(k_u)+cn'(v)-\lambda(e)+\lambda(e'))$. And since $II'>0\wedge cn=cn'$, we have $-1<-cn(k_u)+cn(v)-\lambda(e)+\lambda(e')$ which means:

$$k_u\notin killers_{\sigma'}(u)\implies\exists v\in Cons(u)-\{k_u\},$$
$$-cn(k_u)+cn(v)-\lambda(e)+\lambda(e')\geq 0\qquad(14)$$

From Equation 13 and Equation 14, we deduce that

$$k_u\in killers_\sigma(u)\wedge k_u\notin killers_{\sigma'}(u)$$
$$\implies\exists v\in Cons(u)-\{k_u\},$$
$$-cn(k_u)+cn(v)-\lambda(e)+\lambda(e')=0\qquad(15)$$

Consider now $v\in Cons(u)-\{k_u\}$ that satisfies Equation 15.

$$k_u\in killers_\sigma(u)\implies$$
$$rn(k_u)+II\times(cn(k_u)+\lambda(e))\geq rn(v)+II\times(cn(v)+\lambda(e'))$$

By using Equation 15, we deduce:

$$k_u\in killers_\sigma(u)\implies rn(k_u)\geq rn(v)\qquad(16)$$

$$k_u\notin killers_{\sigma'}(u)\implies$$
$$rn'(k_u)+II\times(cn'(k_u)+\lambda(e))<rn'(v)+II\times(cn'(v)+\lambda(e'))$$

Since $cn'=cn$ and $II'>0$, and by using Equation 15, we deduce:

$$k_u\notin killers_{\sigma'}(u)\implies rn'(k_u)<rn'(v)\qquad(17)$$

Depending on the values of $rn'(k_u)$ and $rn'(v)$, we have four distinct cases:

1) $rn'(k_u)=rn(k_u)\wedge rn'(v)=rn(v)\implies rn(k_u)<rn(v)$. Contradiction with Equation 16.
2) $rn'(k_u)=rn(k_u)+1\wedge rn'(v)=rn(v)+1\implies rn(k_u)+1<rn(v)+1$. Contradiction with Equation 16.
3) $rn'(k_u)=rn(k_u)+1\wedge rn'(v)=rn(v)\implies rn(k_u)+1<rn(v)\implies rn(k_u)<rn(v)$. Contradiction with Equation 16.
4) $rn'(k_u)=rn(k_u)\wedge rn'(v)=rn(v)+1\implies rn(k_u)<rn(v)+1\implies rn(k_u)\leq rn(v)$. By using Equation 16, we deduce that $rn(k_u)=rn(v)$ necessarily. This is is impossible because, by definition of σ':

$$rn'(k_u)=rn(k_u)\wedge rn'(v)=rn(v)+1\implies$$
$$rn(k_u)<c\leq rn(v)\implies rn(k_u)\neq rn(v)$$

Finally we find that,$k_u\in killers_\sigma(u)\wedge k_u\notin killers_{\sigma'}(u)\implies$ contradiction! Consequently, $k_u\in killers_\sigma(u)\implies k_u\in killers_{\sigma'}(u)$ ■