Periodic register saturation in innermost loops

Sid Touati, Zsolt Mathe

Periodic Register Saturation in Innermost Loops

Sid-Ahmed-Ali TOUATI - University of Versailles Saint-Quentin en Yvelines, France
Zsolt MATHE - INRIA-Saclay Laboratory, France

February 3, 2009

Abstract

This article treats register constraints in high performance embedded VLIW computing, aiming to decouple register constraints from instruction scheduling. It extends the register saturation (RS) concept to periodic instruction schedules, i.e., software pipelining (SWP). We formally study an approach which consists in computing the exact upper-bound of the register need for all the valid SWP schedules of an innermost loop (without overestimation), independently of the functional unit constraints. We call this upper-limit the *periodic register saturation* (PRS) of the data dependence graph (DDG). PRS is well adapted to situations where spilling is not a favourite solution for reducing register pressure compared to ILP scheduling: spill operations request memory data with a higher energy consumption. Also, spill code introduces unpredictable cache effects and makes Worst-Case-Execution-Time (WCET) estimation less accurate. PRS is a computer science concept that has many possible applications. First, it provides compiler designers new ways to generate better codes regarding register optimisation by avoiding useless spilling. Second, its computation can help architectural designers to decide about the most suitable number of available registers inside an embedded VLIW processor; such architectural decision can be done with full respect to instruction level parallelism extraction, independently of the chosen functional units configuration. Third, it can be used to verify prior to instruction scheduling that a code does not require spilling. In this paper, we write an appropriate mathematical formalism for the problem of PRS computation and reduction in the case of loops where data dependence graphs may be cyclic. We prove that PRS reduction is NP-hard and we provide optimal and approximate methods. We have implemented our methods, and our experimental results demonstrate the practical usefulness and efficiency of the PRS approach.

1 Introduction

The register saturation is a computer science concept initially studied in [17], intended for decoupling register constraints from resource constraints. In that paper, the author considers basic blocks only where data dependence graphs and instruction schedules are acyclic. In the current article, we extend the concept to periodic schedules of cyclic data dependence graphs (innermost loops), which is a more general and complex problem.

Register saturation (RS) is a formal solution to decouple register constraints from instruction scheduling concerns. It is well adapted to situations where spilling is not a favourite or a possible solution for reducing register pressure compared to ILP scheduling: spill operations request memory data with a higher energy consumption. Also, spill code introduces unpredictable cache effects: it makes WCET estimation less accurate and add difficulties to ILP scheduling (because spill operations latencies are unknown). Register Saturation (RS) is concerned about register maximisation not minimisation, and has some major proved mathematical characteristics [17]:

- As in the case of WCET research, the RS is an exact upper-bound of the register requirement of all possible valid instruction schedules. This means that the register requirement is not over-estimated. RS should not be overestimated, otherwise it would waste hardware registers for embedded VLIW designers, and would produce useless spilling strategies for compiler designers. Contrary to WCET where an exact estimation is hard to model, the RS computation and reduction are exactly modelled problems and can be optimally solved.

- The RS is a *reachable* exact upper-bound of the register requirement for any functional units configuration. This means that, for any resource constraints of the underlying processor (even sequential ones), there is always an instruction schedule that requires RS registers: this is a mathematical fact proved by Lemma 3 in [17]. This is contrary to the well known register sufficiency, which is a minimal bound of register requirement. Such minimal bound is *not* always reachable, since it is tightly correlated to the resource constraints. A practical demonstration is provided in chapter 5 of [16] proving that the register sufficiency is not a reachable lower bound of register need, and hence cannot be used to decouple register constraints from functional units constraints.

Currently, register saturation (RS) is not only intended for general purpose interactive compilers such as *gcc* or *icc*. We are also targeting embedded systems, where compilation time is allowed to last longer in order to produce highly optimised codes. Our target applications are codes where the core of computation is spent in *small* innermost one-dimensional loops (DSP filters, multimedia applications, BLAS codes, vectorial loops, etc.): optimising compilers in this area are allowed to use more aggressive code optimisation techniques (either with static or iterative compilation). Currently, we are not interested yet in studying multidimensional scheduling (loop nests), we discuss this issue in a further section.

There are many practical motivations that convince us to carry on fundamental studies on RS:

- *High performance VLIW computing*. Embedded systems in general cover a wide area of activities which differ in terms of stakes and objectives. In particular, embedded high performance VLIW computing requires cheap and fast VLIW processors to cover the computation budget of telecommunications, video and audio processing, with a tight energy consumption. Such embedded VLIW processors are designed to execute a typical set of applications. Usually, the considered set of typical applications is rarely represented by the set of common

benchmarks (mibench, spec, mediabench, BDTI, etc.), but is given by the industrial client. Then, the constructor of the embedded processor considers only such applications (which are not public) for the hardware design. Nowadays, some embedded VLIW processors (such as ST2xx family) have 32 or 64 registers, and the processor designers have no idea whether such number is adequate or not. Computing the RS of the considered embedded codes allows the hardware designers to precisely gauge with a static method the maximal amount of required registers without worrying about how much functional units they should put on the VLIW processor. RS provides the mathematical guarantee that this maximal register need limit is reachable for any VLIW configuration.

- *Circuit Synthesis*. As studied in [14], optimal cyclic scheduling under resource constraints is currently used to design dynamic reconfigurable circuits with FPGA. In that study, storage and registers are not considered because of practical resolution complexity. Thanks to the RS concept, register constraints can be satisfied prior to the cyclic scheduling problem, with a formal guarantee of providing enough registers for any cyclic schedule.

- *Embedded code optimisation and verification.* As done in [17], computing RS allows to guide instruction scheduling heuristics inside backend compilers. For instance, if RS is below $\mathcal{R}$ the number of available registers, then we can guarantee that the instruction scheduling process can be carried on without considering register constraints. If RS is greater than $\mathcal{R}$ the number of available registers, then RS reduction methods could be used.

- *High Performance Computing*. RS may be used to control high-level loop transformations such as loop unrolling without causing low level register spilling. In practice, this means that the unrolling degree is chosen so that RS remains below $\mathcal{R}$.

- *Just-in-time (JIT) compilation*. The compiler can generate a bytecode with a bounded RS. This means that the generated bytecode holds RS metrics as static annotations, providing information about the maximal register need for any underlying processor characteristics. At program execution, when the processor is known, the JIT can access such static annotations (present in the bytecode) and eventually schedule operations at run-time under only resource constraints without worrying about registers and spilling.

For all the above applications, we can have many solutions and strategies, and the literature is rich with many articles about the topics. The RS concept is not the unique and main strategy. It is a concept that may be used in conjunction and complementary with other strategies. RS is helpful thanks to two characteristics:

1. The RS concept can give a *formal* guarantee of avoiding useless spilling in some codes (case when RS$\leq$ R). Avoiding useless spilling allows to reduce the amount of memory requests and cache effects, which may save power and increase performance.

2. Since RS is a *static* metric, it does not require program execution or simulation. Usually, the results provided with such methods are not formally guaranteed and always depend on input data, on functional units configurations, on the precision of the simulator, on the presence or not of a processor prototype, etc. Also, such dynamic techniques are usually demonstrated for a set of benchmarks only, and such benchmarks are not really representative in some situations. Since RS is a mathematically proved static metric, it can be safely said that it is correct and valid for *all* possible codes meeting the model (not only benchmarks), and for *all* possible program executions (not only for a subset of input data), and this independently of the functional units configuration.

In this article, background and notations refer directly to our previous results published in [18]. We assume that the reader is familiar with [18]. Otherwise, we strongly suggest a first reading of sections 2 and 3 of [18], since they are related to many notions in the current contributions.

Our current article focus on PRS, which is the specialisation of RS in the context of periodic schedules. Our paper is organised as follows. Section 2 studies the problem of PRS computation. If PRS is $\geq R$, Section 3 studies the problems of PRS reductions: we prove that this problem is NP-hard and we provide exact and approximate solutions. We present our experimental results in Section 4. This section also presents some new experiments related to [18]. Section 5 discusses some related work, then we conclude.

2 Computing the Optimal Periodic Register Saturation

Let $G = (V, E, \delta, \lambda)$ be a loop. The periodic register saturation (PRS) is the maximal register requirement for all valid software pipelined schedules:

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

where $RN_{\sigma}(G)$ is the periodic register need for the SWP schedule σ. A software pipelined schedule which needs the maximum number of registers is called a *saturating SWP schedule*. Note that it may not be unique.

In this section, we show that our new method of computing $RN_{\sigma}(G)$ [18] is useful to write an exact modelling of PRS computation. In the current case, we are faced with a difficulty: for computing the periodic register sufficiency as done in[18], we are requested to minimise a *maximum* (minimise MAXLIVE), which a common optimisation problem in operational research; however, PRS computation requires to *maximise* a maximum, namely to maximise MAXLIVE. Maximising a maximum is a less conventional linear optimisation problem. It requires the writing of an exact equation of the maximum, which has been defined by Equation (1) in [18].

In practice, we need to consider loops with a bounded code size. That is, we should bound the duration L. This yields to computing the PRS by considering a subset of possible SWP schedules $\Sigma_L(G) \subseteq \Sigma(G)$: we compute the maximal register requirement in the set of all valid software pipelined schedules with the property that the duration does not exceed a fixed limit L and $MII \geq 1$. Bounding the schedule

space has the consequence to bound the values of the scheduling function as follows: $\forall u \in V, 0 \leq \sigma(u) \leq L$.

Computing the optimal register saturation is proved as an NP-complete problem in [17]. Now, let's study how we exactly compute the periodic register saturation using integer linear programming (intLP). Our intLP formulation expresses the logical operators ($\Longrightarrow$, $\vee$, $\Longleftrightarrow$) and the max operator ($max(x, y)$) by introducing extra binary variables. However, expressing these additional operators requires that the domain of the integer variables should be bounded, as explained in details in [17].

Next, we present our intLP formulation that computes a saturating schedule $\sigma \in \Sigma_L(G)$ considering a *fixed* II. Fixing a value for the initiation interval is necessary to have linear constraints in the intLP system. As far as we know, computing the exact periodic register need (MAXLIVE) of a SWP schedule with a non fixed II is not a mathematically defined problem (because a SWP schedule is defined according to a fixed II).

Basic Integer Variables

1. For the lifetime intervals, we define:
   - one schedule variable $\sigma_u \geq 0$ for each $u \in V$;
   - one variable which contains the killing date $k_u \geq 0$ for each statement $u \in V_R$.
2. For the periodic register need, we define:
   - $p_u \geq 0$ the number of the instances of $u \in V_R$ simultaneously alive, which is the number of complete periods around the circle produced by the cyclic lifetime interval of $u \in V_R$;
   - $l_u \geq 0$ and $r_u \geq 0$ the left and the right of the cyclic lifetime interval of $u \in V_R$;
   - the two acyclic fractional intervals $I_u =]a_u, b_u]$ and $I'_u =]a'_u, b'_u]$ after unrolling the kernel once.
3. For a maximal clique in the interference graph of the fractional acyclic intervals, we define:
   - interference binary variables $s_{I,J}$ for all the fractional acyclic intervals I, J: $s_{I,J} = 1$ iff I and J interfere with each other;
   - a binary variable x_I for each fractional acyclic interval: $x_I = 1$ iff I belongs to a maximal clique.

Linear Constraints

1. Periodic scheduling constraints: $\forall e = (u, v) \in E, \qquad \sigma_u - \sigma_v \leq +\lambda(e) \times II - \delta(e)$

2. The killing dates are computed by:

$$\forall u \in V_R, \qquad k_u = \max_{\substack{v \in Cons(u) \\ e=(u,v) \in E_R}} \left(\sigma_v + \delta_r(v) + \lambda(e) \times II\right)$$

We use the linear constraints of the *max* operator as defined in [17]. k_u is bounded by $\underline{k_u}$ and $\overline{k_u}$ where:

- $\underline{k_u} = \min_{v \in Cons(u)} \left(\delta_r(v) + \max_{e=(u,v) \in E_R} \lambda(e) \times II\right)$
- $\overline{k_u} = \max_{v \in Cons(u)} \left(L + \delta_r(v) + \max_{e=(u,v) \in E_R} \lambda(e) \times II\right)$

3. The number of interfering instances of a value (complete turns around the circle) is the integer division of its lifetime by II. We introduce an integer variable $\alpha_u \geq 0$ which holds the rest of the division:

$$\begin{cases} k_u - \sigma_u - \delta_w(u) = II \times p_u + \alpha_u \\ \alpha_u < II \\ \alpha_u \in \mathbb{N}^+ \end{cases}$$

4. The lefts [18] of the circular intervals are the rest of the integer division of the birth date by II. We introduce an integer variable $\beta_u \geq 0$ which holds the integral quotient of the division:

$$\begin{cases} \sigma_u + \delta_w(u) = II \times \beta_u + l_u \\ l_u < II \\ \beta_u \in \mathbb{N}^+ \end{cases}$$

5. The rights [18] of the circular intervals are the rest of the integer division of the killing date by II. We introduce an integer variable $\gamma_u \geq 0$ which holds the integer quotient of the division:

$$\begin{cases} k_u = II \times \gamma_u + r_u \\ r_u < II \\ \gamma_u \in \mathbb{N}^+ \end{cases}$$

6. The fractional acyclic intervals are computed by considering an unrolled kernel once (they are computed depending on whether the cyclic interval crosses the kernel barrier):

$$\begin{cases} a_u = l_u \\ r_u \geq l_u \Longrightarrow b_u = r_u \\ \text{case when the cyclic interval crosses } II\text{:} \\ r_u < l_u \Longrightarrow b_u = r_u + II \\ a'_u = a_u + II \\ b'_u = b_u + II \end{cases}$$

Since the variable domains are bounded, we can use the linear constraints of implication defined in [17]: we know that $0 \leq l_u < II$, so $0 \leq a_u < II$ and $II \leq a'_u < 2 \times II$. Also, $0 \leq l_u < II$ so $0 \leq b_u < 2 \times II$ and $II \leq b'_u < 3 \times II$.

7. For any pair of distinct fractional acyclic intervals I, J, the binary variable $s_{I,J} \in \{0,1\}$ is set to 1 if the two intervals are non empty and interfere with each other. It is expressed in the intLP by adding the following constraints.

   $\forall$ acyclic intervals I, J:

$$s_{I,J} = 1 \Longleftrightarrow \quad \begin{array}{l} \big[(length(I) > 0) \\ \wedge\ (length(J) > 0) \\ \wedge\ \neg(I \prec J\ \vee\ J \prec I)\big] \end{array}$$

   where $\prec$ denotes the usual relation *before* in the interval algebra. Assuming that $I =]a_I, b_I]$ and $J =]a_J, b_J]$, $I \prec J$ means that $b_I \leq a_J$, and the above constraints are written as follows. $\forall$ acyclic intervals I, J,

$$s_{I,J} = 1 \Longleftrightarrow \left\{ \begin{array}{ll} b_I - a_I > 0 & (i.e.,\ length(I) > 0) \\ b_J - a_J > 0 & (i.e.,\ length(J) > 0) \\ b_I > a_J & (i.e.,\ \neg(I \prec J)) \\ b_J > a_I & (i.e.,\ \neg(J \prec I)) \end{array} \right.$$

8. A maximal clique in the interference graph is an independent set in the complementary graph. Then, for two binary variables x_I and x_J, only one is set to 1 if the two acyclic intervals I and J do not interfere with each other:

$$\forall \text{ acyclic intervals } I, J: \qquad s_{I,J} = 0 \Longrightarrow x_I + x_J \leq 1$$

9. In order to guarantee that our objective function maximises the interferences between the non-zero length acyclic intervals, we add the following constraint:

$$\forall \text{ acyclic intervals } I, \qquad length(I) = 0 \Longrightarrow x_I = 0$$

   Since $length(I) = b_I - a_I$, it amounts to:

$$\forall \text{ acyclic intervals } I, \qquad b_I - a_I = 0 \Longrightarrow x_I = 0$$

Linear Objective Function A saturating SWP schedule can be obtained by maximising the value of:

$$\sum_{\text{acyclic fractional interval } I} x_I + \sum_{u \in V_R} p_u$$

Solving the above intLP model yields a solution $\overline{\sigma}$ for the scheduling variables, which define a saturating SWP, such that $PRS(G) = RN_{\overline{\sigma}}(G)$. Once $\overline{\sigma}$ computed by intLP, then $RN_{\overline{\sigma}}(G)$ is equal to the value of the objective function. Finally, $PRS = \max_{MII \leq II \leq L} RN_{\overline{\sigma}}(G)$.

The size of our intLP model is $\mathcal{O}(|V_R|^2)$ variables and $\mathcal{O}(|E| + |V_R|^2)$ constraints. The coefficients of the constraints matrix are all bounded by $\pm L \times \lambda_{max} \times II$, where λ_{max} is the maximal dependence distance in the loop. To compute the PRS, we scan all the admissible values of II, i.e., we iterate II the initiation interval from MII to L and then we solve the intLP system for each value of II. The PRS is finally the maximal register need among of all the ones computed by all the intLP systems. As can be remarked, the size of out intLP model is polynomial (quadratic) on the size of the input DDG.

3 Reducing Periodic Register Saturation

This section studies how to build an extended DDG, *i.e* how to add serial edges to a given DDG $G = (V, E, \delta, \lambda)$ such that its periodic register saturation is limited by a strictly positive integer $\mathcal{R}$ under a desired minimal initial interval (critical cycle) constraint $\overline{MII}$. This allows us to guarantee that any software pipelining of the new graph does not require more registers than those available. Consequently, we can always build a valid register allocation without spilling after the SWP process.

Problem 1 (ReducePRS) *Given a DDG $G = (V, E, \delta, \lambda)$, is there an extended DDG $\overline{G}$ of G such that $PRS(\overline{G}) \leq \mathcal{R}$ and $MII \leq \overline{MII}$?*

The following theorem has two main practical implications. First, it proves that we cannot have optimal solutions in practice unless we use exponential algorithms, or unless $P = NP$. That is, the usage of a sub-optimal heuristics in practice is unavoidable. Second, its formal proof gives us key hints about building optimal and approximate methods for PRS reduction.

Theorem 1 *Reducing the Periodic Register Saturation is NP-hard.*

Proof:
We prove that ReducePRS can be reduced from the problem of instruction scheduling under register constraints (SRC). We take the same instance for both problems. Let us start by defining the latter problem.

Problem 2 (SRC problem) *Let $G = (V, E, \delta, \lambda)$ be a DDG, $\mathcal{R}$ and $\overline{MII}$ two positive integers. Does it exist a valid SWP schedule $\sigma \in \Sigma_L(G)$ such that:*

$$RN_\sigma(G) \leq \mathcal{R} \ \wedge \ II \leq \overline{MII}$$

where II is the initiation interval of σ?

The SRC problem is proved NP-hard [5].

1. **ReducePRS $\Longrightarrow$ SRC**
Let $\overline{G}$ be a solution for the ReducePRS problem. Then, we can build a SWP schedule $\sigma \in \Sigma_L(\overline{G})$ in polynomial time complexity under only the serial constraints with $II = MII \leq \overline{MII}$.

2. **SRC $\Longrightarrow$ ReducePRS**
Let σ be a solution for SRC, i.e., $RN_\sigma(G) \leq \mathcal{R}$ and $II \leq \overline{MII}$. As an example, let us consider the DDG example of [18], that we redraw in Figure 1(a). The PRS of this DDG has been computed as a practical example in [16] (Chapter 8) and is equal to 8 registers. We would like to reduce the PRS to 4 registers based on the cyclic schedule of Figure 1(b). That schedule has 4 simultaneously alive values.

We have to build an extended DDG $\overline{G}$ such that we guarantee that any SWP schedule $\sigma' \in \Sigma(\overline{G})$ produces the same cyclic relative order between the circular lifetime intervals as defined by σ. If a lifetime interval $LT_\sigma(u(i))$ is before lifetime interval

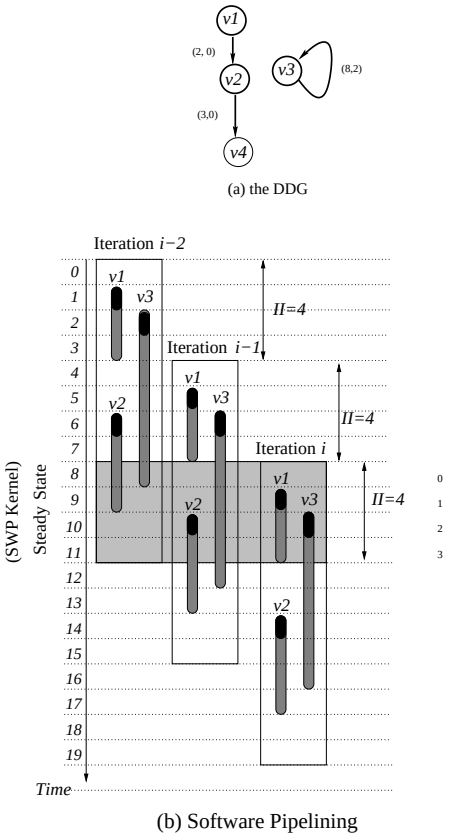


Figure 1: The DDG Example of [18]

$LT_\sigma(v(i+\alpha))$, then we must guarantee that any SWP schedule σ' makes $LT_{\sigma'}(u(i))$ before $LT_{\sigma'}(v(i+\alpha))$, α is a distance to be defined.

We model the cyclic ordering between the circular lifetime intervals by a graph $O = (V_R, E_\prec, \alpha)$: $e = (u, v) \in E_\prec$ means that the value produced by the operation $u(i)$ is killed before the definition of the value $v(i + \alpha(e))$ (It is not necessary to have u distinct from v). $\alpha(e)$ is chosen so that the killing date of $u(i)$ is as close as possible to the definition date of $v(i+\alpha(e))$, i.e., both of the two dates must be inside a window of size II. Since the schedule times of the distinct instances of the statement v are separated by II clock cycles, there is a unique distance α that defines the cyclic order between $LT_\sigma(u(i))$ and $LT_\sigma(v(i+\alpha))$ in a window of size II. The constraints that define such distance α between $u(i)$ and $v(i+\alpha)$ are:

$$LT_\sigma(u(i)) \prec LT_\sigma(v(i+\alpha)) \tag{1}$$

$$\sigma(v(i+\alpha)) + \delta_w(v) - k_{u(i)} < II \tag{2}$$

where $k_{u(i)}$ is the killing date of $u(i)$. Since

$$\begin{array}{lll}(1) & \Longleftrightarrow & k_{u(i)} \leq \sigma(v(i+\alpha)) + \delta_w(v) \\ & \Longleftrightarrow & k_u \leq \sigma_v + II \times \alpha + \delta_w(v)\end{array}$$

and

$$(2) \Longleftrightarrow \sigma_v + II \times \alpha + \delta_w(v) - k_u < II$$

(1) and (2) amount to:

$$0 \leq \sigma_v + II \times \alpha + \delta_w(v) - k_u < II$$

Then, α is the unique integer that belongs to the interval:

$$\frac{k_u - \sigma_v - \delta_w(v)}{II} \leq \alpha < 1 + \frac{k_u - \sigma_v - \delta_w(v)}{II}$$

$$\Longrightarrow \alpha = \left\lceil \frac{k_u - \sigma_v - \delta_w(v)}{II} \right\rceil$$

Now, we have completely defined the cyclic ordering graph $O = (V_R, E_{\prec}, \alpha)$. Note that the edges belonging to $E_{\prec}$ are defined from each value u to v (u not necessarily distinct from v), since a periodic schedule makes circular all the lifetime intervals: for any $(u, v) \in V_R^2$, there exists a unique α (under the constraints just defined above) such that $LT_\sigma(u(i)) \prec LT_\sigma(v(i+\alpha))$. As an illustration, Figure 2(b) shows the cyclic relative ordering between the values deduced from the schedule of Figure 1(b). For instance, $LT_\sigma(v_2(i)) \prec LT_\sigma(v_1(i+2))$, thus there is a cyclic ordering edge $e = (v_2, v_1)$ in Figure 2(a) with $\alpha(e) = 2$. Also, $LT_\sigma(v_1(i)) \prec LT_\sigma(v_1(i+1))$, thus there is a cyclic ordering edge $e = (v_1, v_1)$ in Figure 2(a) with $\alpha(e) = 1$.

Now, let us see how to build an extended DDG $\overline{G}$ based on this cyclic ordering, i.e., how to report cyclic precedence relations between the circular lifetime intervals. For each order $e = (u, v) \in E_{\prec}$ between two values u and v, we must guarantee that the killing date of u is always performed before the definition date of $v(i + \alpha(e))$:

$$k_u \leq \sigma\left(v(i + \alpha(e))\right) + \delta_w(v)$$

This means that $\forall u' \in Cons(u)$,

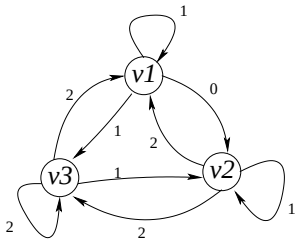
$$\sigma\left(u'(i + \lambda\left((u, u')\right)\right) + \delta_r(u') \leq \sigma\left(v(i + \alpha(e))\right) + \delta_w(v)$$

$$\Longleftrightarrow \sigma\left(u'(i)\right) + \delta_r(u') - \delta_w(v) \leq \sigma\left(v(i + \alpha(e) - \lambda\left((u, u')\right)\right)$$

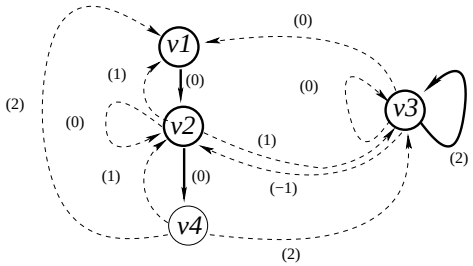
in which $\lambda\left((u, u')\right)$ is the distance of the flow dependence between u and its consumer u'. This is done by adding a serial edge e' to G from each consumer $u' \in Cons(u)$ to v with:

$$\delta(e') = \delta_r(u') - \delta_w(v) \qquad \text{and} \qquad \lambda(e') = \alpha(e) - \lambda\left((u, u')\right)$$

Figure 2(b) is the extended graph that has a periodic register saturation equal to 4. In that figure, the added serial edges appear with dashed lines and only tagged with the



(a) The Circular Ordering between Circular Lifetime Intervals



(b) Reducing the Periodic Register Saturation

Figure 2: Cyclic Ordering

distances. As an example, there is an order between v_1 and v_3 with a distance $\alpha = 1$. Since v_2 consumes v_1 with distance $\lambda = 0$, we add a serial edge from v_2 to v_2 with a distance $\alpha - \lambda = 1$.

Note that some added serial edges may be redundant and do not cause any typical restriction for instruction scheduling. As an illustration, there is an order between v_3 and itself with a distance $\alpha = 2$. Since v_3 consumes itself with a distance $\lambda = 2$, this produces a serial edge in G from v_3 to itself with $\alpha - \lambda = 0$. This serial edge is always satisfied by any schedule and can always be safely removed from $\overline{G}$.

By adding all these serial edges, we build an extended DDG $\overline{G}$ that has the following characteristics.

- Any SWP schedule σ' of $\overline{G}$ produces a circular order between the circular lifetime intervals as defined by σ. So, σ' cannot need more registers than σ. This is because if two lifetime intervals do not interfere with each other according to σ , they cannot interfere with each other according to σ'.

  1. The number of distinct interfering instances (turns around the circle) of each statements u with σ' cannot exceed the number p_u of distinct interfering instances with σ. This is because we have, according to σ, $LT_\sigma(u(i)) \prec LT_\sigma(v(i + p_u + 1))$. Since we report the cyclic order $e = (u, u)$ with $p_u \leq \alpha(e) = p_u + 1$ in the extended DDG $\overline{G}$, at most p_u instances of u

may interfere according to a schedule σ' of $\overline{G}$.

2. The fractional intervals inside the SWP kernel are constrained to satisfy the same precedence relation as defined by σ. If two fractional intervals (l, r) and (l', r') do not interfere with each other according to σ, then they cannot interfere according to σ'. Otherwise it means that σ' violates one of the added serial edges.

- σ is a valid software pipelined schedule for $\overline{G}$ since it satisfies all the introduced serial edges. Then, the extended DDG remains schedulable.
- Since the initiation interval II of σ is lower than or equal to $\overline{MII}$, a possible introduced critical cycle in $\overline{G}$ is not greater than $\overline{MII}$. Otherwise it means that σ is not a valid software pipelined schedule for $\overline{G}$.

From above, we deduce: $\forall \overline{\sigma} \in \Sigma_L(\overline{G}), \qquad RN_{\overline{\sigma}}(\overline{G}) \leq RN_\sigma(G)$ and hence $PRS(\overline{G}) \leq RN_\sigma(G) \leq \mathcal{R}$

⌟

From the previous proof, we deduce that the optimal reduction of periodic register saturation is equivalent to finding a software pipelined schedule with a minimal initiation interval which does not require more than $\mathcal{R}$ registers (but without considering any resource constraints). There exist many algorithms (optimal or heuristics) in the literature that compute a SWP schedule minimising MAXLIVE under a fixed II (a complete survey is done in [16]), any such method is suitable for use in this context. However, they would not bring efficient solutions for PRS reduction, since the purpose here is not to necessarily minimise the register requirement, but to not exceed the limit $\mathcal{R}$. So, we should use a SWP scheduling method that does not necessarily minimise the register requirement at the lowest possible level. As far as we know, the only method that allows this opportunity is the SIRA SWP technique presented in [19]. However, other SWP scheduling techniques under register constraints (without considering resources) may be used if they do not minimise the register requirement at the lowest possible value. We can then assume that we have such module. If the module computes an optimal SWP under register constraints (an NP-complete problem), then the PRS reduction solution is necessarily optimal. If the module is a heuristic, then the PRS reduction provides a sub-optimal solution. In both cases, using such module yields two possible situations for PRS reduction:

1. If the module computes a SWP schedule σ such that $RN_\sigma(G) \leq \mathcal{R}$, then, we add serial edges to the DDG as described in the previous proof. The critical cycle of the extended DDG is lower than or equal to II.
2. If the module fails to find a SWP schedule of initiation II with $RN_\sigma(G) \leq \mathcal{R}$, then we cannot reduce the periodic register saturation with respect to the critical cycle $MII \leq II$. We have to increment II (in binary search between $II_{min} = II$ and $II_{max} = L$), until reaching a solution or not. If no solution exists, spill code must be introduced. Introducing spill code is another interesting problem which is outside the scope of the paper. Introducing and minimising spill code

is indeed an NP-complete problem studied in the literature [15], but it is still not well understood in case of ILP scheduling (because of cache effects).

4 Experiments

We have developed a complete tool based on the research results presented in this article. It implements the integer linear program that computes the periodic register saturation of a DDG, and reduces its PRS if it exceeds $\mathcal{R}$. We use a PC under linux, equipped with a dual core Pentium D (3.4 Ghz), and 1 GB of memory. We did thousands of experiments on several DDGs extracted from different benchmarks (SPEC, Whetstone, Livermore, Linpac, DSP filters). The size of our DDG goes from 2 nodes and 2 edges, to 20 nodes and 26 edges. They represent the typical small loops intended to be analysed and optimised using the PRS concept. However, we also experiment larger DDGs produced by loop unrolling, resulting in DDGs with size $|V| + |E|$ reaching 460.

4.1 Computing RN

In [18], we provided method for RN computation of already scheduled loops in $O(|V| \times \ln(|V|))$, which is a good complexity in theory. Here, we provide experiments to demonstrate its efficiency in practice. For each DDG, we computed a valid loop schedule and we measure the time spent to compute its MAXLIVE (in ms). We report here the speedup of RN computation (compilation time speedup, do not confuse with speedup of benchmarks), measured as the integral ratio between RN computation time using the existing pseudo-polynomial method and RN computation time using our $O(|V| \ln |V|)$ algorithm. Figure 3 plots the speedup of RN computation obtained with our method: we plot here few DDG examples with various unrolling degrees. As can be seen, when II increases our method is faster and scales better since it has not a pseudo-polynomial complexity (till $\approx 70\times$ faster at best case). Even when we vary the DDG size (measured as the number of nodes and edges), Figure 4 shows our method is still faster and scales in a better way (till $\approx 70\times$ faster at best case). In this figure, we plot the maximal speedup obtained for any value of II. All these first experiments show that, when compilation time is an important issue, our new method of RN computation exhibits better execution times, especially when compiling large loops with big II.

4.2 Optimal PRS Computation

From the theoretical perspective, PRS is unbounded. However, as shown in Table 1, the PRS is bounded and finite, because the duration L is bounded in practice: in our experiments, we took $L = \sum_{e \in E}$, which is a convenient upper bound. Figure 5 provides some plots on maximal periodic register need vs. initiation intervals of many DDG examples. These curves have been computed using optimal intLP resolution using CPLEX. The plots do not start nor end at the same points because the parameters MII (starting point) and L (ending point) differ from one loop to another. Given a DDG, its PRS is equal to the maximal value of RN for any II. As can be seen, this

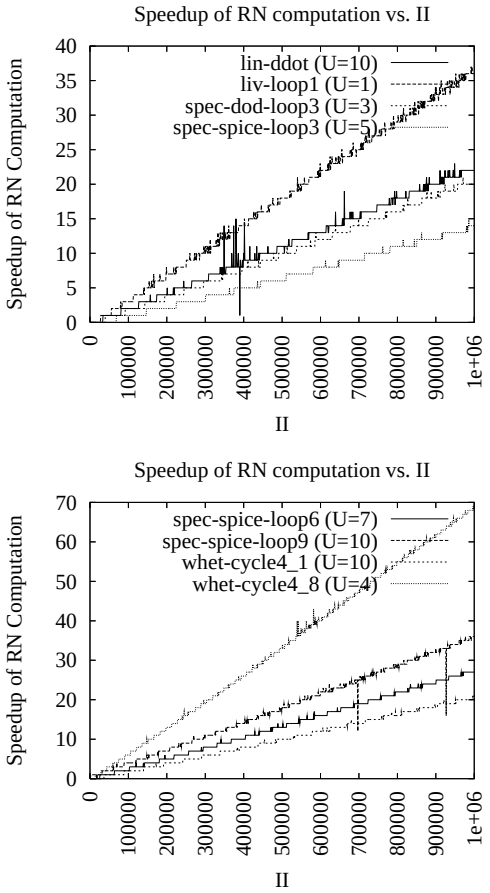


Figure 3: Speedup of RN Computation vs. Initiation Interval (II)

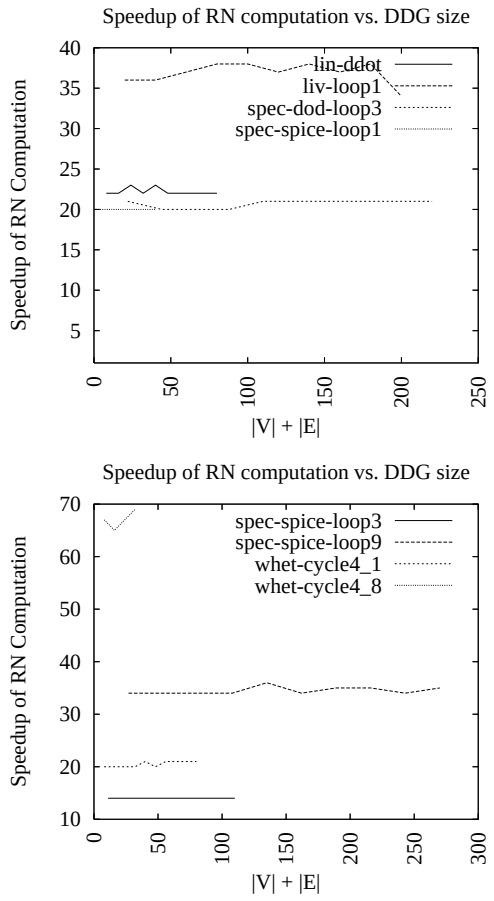


Figure 4: Speedup of RN Computation vs. DDG size ($|V| + |E|$)

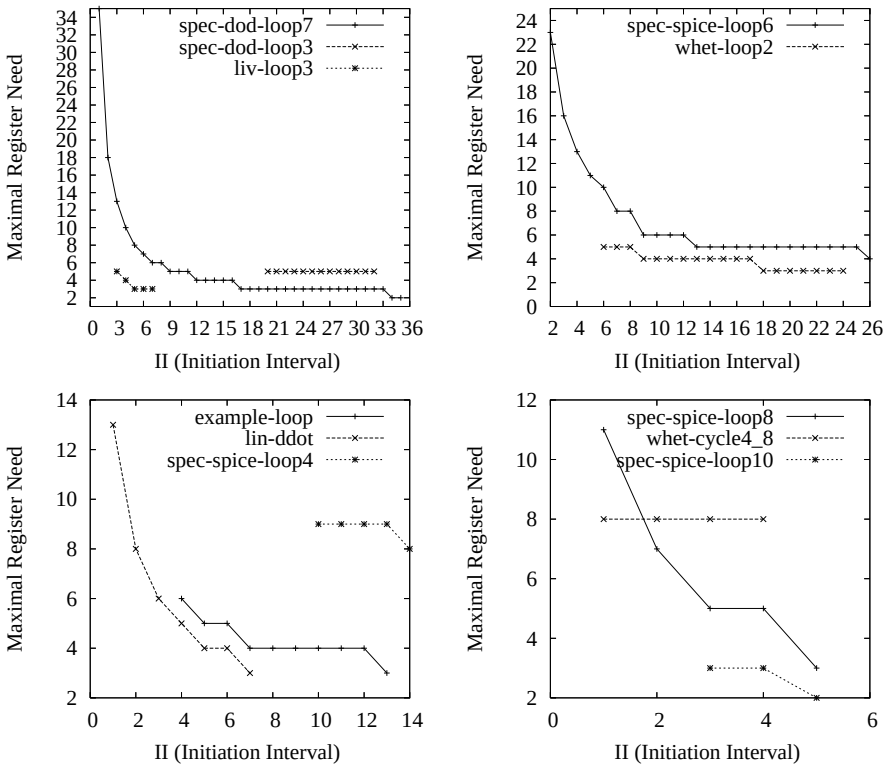


Figure 5: Maximal Periodic Register Need vs. Initiation Interval

maximal value of RN always holds for $II = MII$. This result is intuitive, since the lower is the II, the higher is ILP degree, and consequently the higher is the register need. The asymptotic plots of Figure 5 show that maximal RN vs. II describe non-increasing functions. Indeed, the maximal RN is either a constant or a decreasing function. Depending on $\mathcal{R}$ the number of available registers, PRS computation allows to deduce that register constraints are irrelevant in many cases (when $PRS \leq \mathcal{R}$)

We should recall an interesting mathematical property of register saturation [17]: for each computed maximal RN, there is a formal guarantee about the existence of at least one valid SWP schedule requiring that maximum, and thus for any functional units or ILP constraints. For instance, the case of spec-dod-loop7 has a PRS equal to 35. It means that there is always a SWP requiring exactly 35 registers, for any ILP or sequential processor. And, there is not another SWP schedule requiring more than 35 registers, unless the parameter L is larger. This is an interesting property which does not hold for the usual register sufficiency concept: indeed, as shown in [17], the register sufficiency (the minimal register need) is tightly related to the underlying resource constraints.

Optimal PRS computation using intLP resolution may be intractable because the

underlying problem is NP-complete. In order to be able to compute an approximate PRS for larger DDGs, we use a heuristics with the CPLEX solver. Indeed, the operational research community brings efficient ways to deduce heuristics based on exact intLP formulation. When using CPLEX, we can use a generic branch and bound heuristics for intLP resolution, tuned with many CPLEX parameters. In the current paper, we choose a first satisfactory heuristic by bounding the resolution with a real time limit (say 5 or 1 seconds). The intLP resolution stops when time goes out and returns the best feasible solution found. Of course, in some cases, if the given time limit is not sufficiently high, the solver may not find a feasible solution (as in any heuristic targeting an NP-complete problem). Using such CPLEX generic heuristics for intLP resolution avoids the need of designing new heuristics. Table 1 shows the results of PRS computation in both the case of optimal PRS, and approximate PRS (with time limits of 5 and 1 seconds). As can be seen, in most cases, this simple heuristic computes the optimal results. The more time we give to CPLEX computation, the closer it will be to the optimal one.

Benchmark	Loop	PRS	PRS (5 s)	PRS (1 s)
SPEC - SPICE	loop1	4	4	4
	loop2	28	28	28
	loop3	2	2	2
	loop4	9	9	NA
	loop5	1	1	1
	loop6	23	23	23
	loop8	11	11	11
	loop9	21	21	NA
	loop10	3	3	3
	tom-loop1	11	NA	NA
SPEC - DODUC	loop1	11	NA	NA
	loop2	6	6	5
	loop3	5	5	5
	loop7	35	35	35
SPEC - FPPP	fp-loop1	4	4	4
Linpac	ddot	13	13	NA
Livermoore	loop1	8	8	NA
	loop5	5	5	5
	loop23	31	NA	NA
Whetstone	loop1	6	5	NA
	loop2	5	5	5
	loop3	4	4	4
	cycle4-1	1	1	1
	cycle4-2	2	2	2
	cycle4-4	4	4	4
	cycle4-8	8	8	8
Figure 1 DDG	loop1	6	6	6
TORSHE	van-Dongen	10	10	9
DSP filter	WDF	6	6	6

Table 1: Optimal vs. Approximate PRS

We will use this kind of heuristics in order to compute approximate PRS for larger DDGs in the next section.

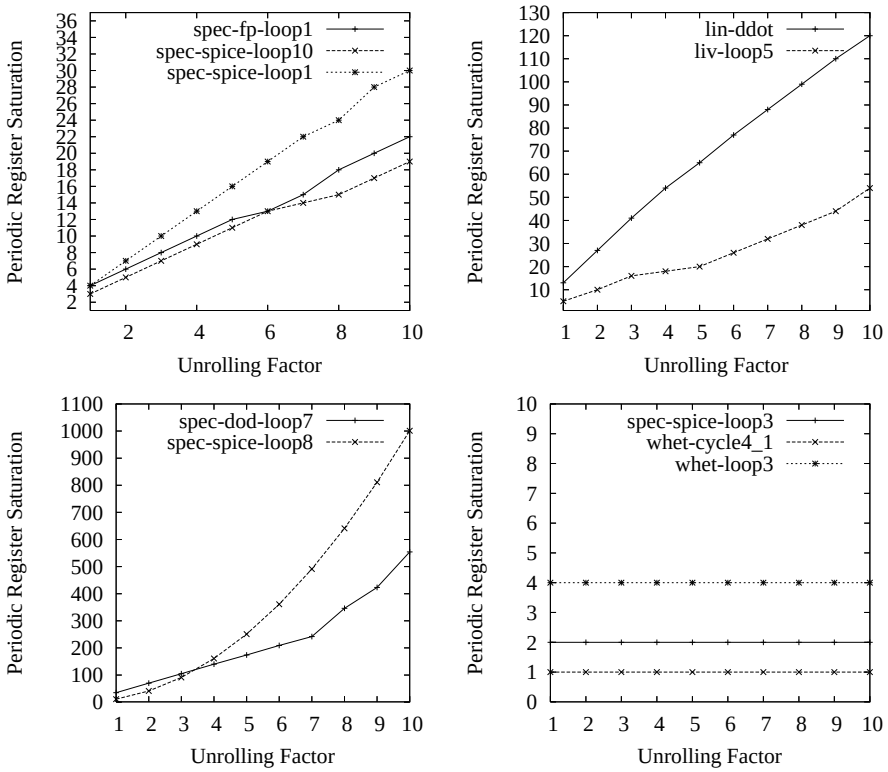


Figure 6: Periodic Register Saturation in Unrolled Loops

4.3 Approximate PRS Computation with Heuristic

We use loop unrolling to produce larger DDGs (up to 200 nodes and 260 edges). As can be seen in some cases (spec-spice-loop3, whet-loop3,whet-cycle-4-1), the PRS stays constant because the cyclic data dependence limit the inherent ILP, and hence PRS remains constant irrespective of unrolling degrees. In other cases (lin-ddot, spec-fp-loop1, spec-spice-loop1), PRS increases as a sub-linear function of unrolling degree. In other cases (spec-dod-loop7), PRS increases as a super-linear function of unrolling degree. This is because unrolling degree produces bigger durations L, which increase the PRS with a factor greater than the unrolling degree.

4.4 Optimal PRS Reduction

If PRS is used in the context of code optimisation, we may need to reduce it when it exceeds $\mathcal{R}$. We developed PRS reduction (optimal and approximate) based on the SIRA framework [19]. SIRA gives us the opportunity to reduce PRS below $\mathcal{R}$ without minimising it at the lowest possible level. This is useful for saving ILP. We did hun-

Benchmark	Loop	$\overline{PRS}$ ($\mathcal{R}$=16)	$\overline{PRS}$ ($\mathcal{R}$=32)
SPEC - SPICE	loop1	4 (0%)	4 (0%)
	loop2	16 (0%)	28 (0%)
	loop3	2 (0%)	2 (0%)
	loop4	9 (0%)	9 (0%)
	loop5	1 (0%)	1 (0%)
	loop6	16 (0%)	23 (0%)
	loop8	11 (0%)	11(0%)
	loop9	16 (0%)	21 (0%)
	loop10	3 (0%)	3 (0%)
	tom-loop1	11 (0%)	11 (0%)
SPEC - DODUC	loop1	11 (0%)	11 (0%)
	loop2	6 (0%)	6 (0%)
	loop3	5 (0%)	5 (0%)
	loop7	16 (66.66%)	32 (50%)
SPEC - FPPP	fp-loop1	4 (0%)	4(0%)
Linpac	ddot	13 (0%)	13 (0%)
Livermoore	loop1	8 (0%)	8 (0%)
	loop5	5 (0%)	5 (0%)
	loop23	16 (0%)	32 (0%)
Whetstone	loop1	6 (0%)	6 (0%)
	loop2	5 (0%)	5 (0%)
	loop3	4 (0%)	4 (0%)
	cycle4-1	1 (0%)	1 (0%)
	cycle4-2	2 (0%)	2 (0%)
	cycle4-4	4 (0%)	4 (0%)
	cycle4-8	8 (0%)	8 (0%)
DDG of Figure 1	loop1	6 (0%)	6 (0%)
TORSHE	van-Dongen	10 (0%)	10 (0%)
DSP filter	WDF	6 (0%)	6 (0%)

Table 2: Optimal PRS Reduction

dreds of experiments on hundreds of DDGs, with many values for $\mathcal{R}$ (8,16,32,64,128) and II. In all cases, the PRS approach allows to check whether a DDG is not constrained by registers. If $PRS \leq \mathcal{R}$, no edge is introduced in the DDG, resulting in a maximal ILP extraction under resource constraints. When PRS exceeds $\mathcal{R}$, some edges are introduced to reduce PRS to a new value $\overline{PRS}$ with taking care of MII if possible. As in [17], we measure the ILP loss after PRS reduction as equal to $1 - MII/\overline{MII}$, where MII is the initial critical cycle, and $\overline{MII}$ is the new critical cycle after PRS reduction. Table 2 shows the results of optimal PRS reduction when considering 16 and 32 available registers. The ILP loss is expressed in terms of percentage (numbers between brackets). As can be seen, if PRS exceeds $\mathcal{R}$, optimal PRS reduction can always reduce it to $\mathcal{R}$. The ILP loss is almost equal to zero, except in the case of spec-dod-loop7, because of large static operation latencies (17 cycles), which yields higher register pressure.

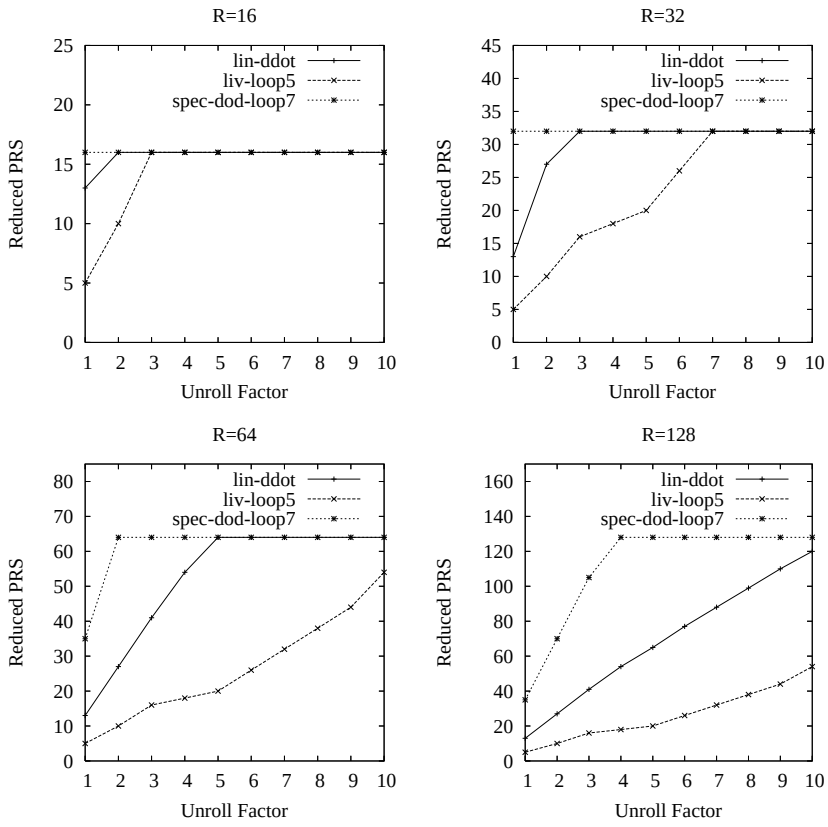


Figure 7: Approximate PRS Reduction in Unrolled Loops

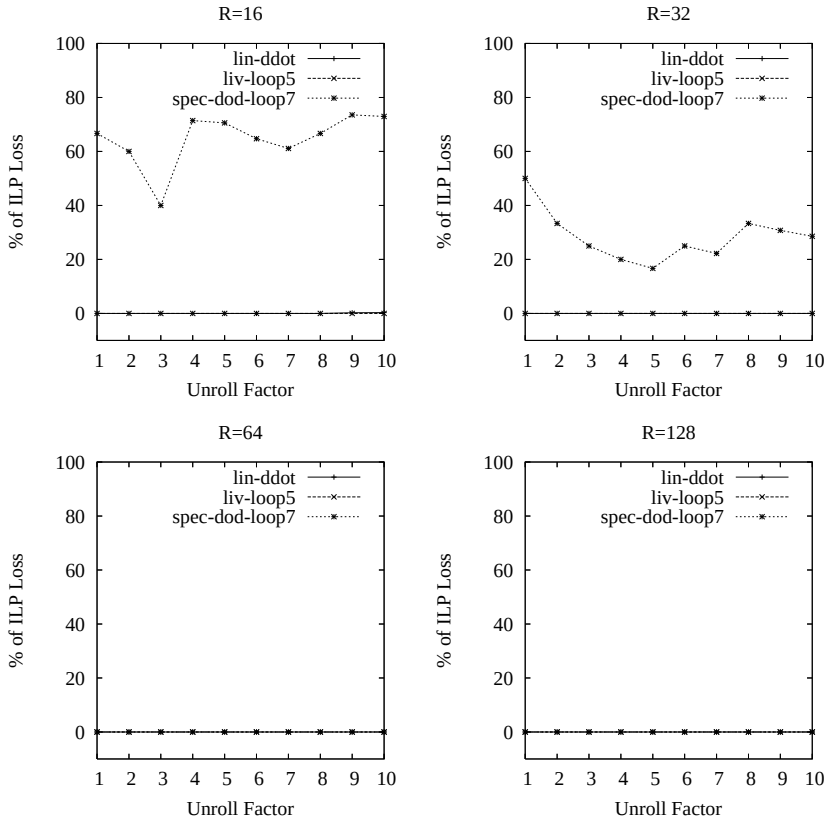


Figure 8: Percentage of ILP Loss in Unrolled Loops

4.5 PRS Reduction with Heuristic

Since PRS reduction is NP-hard, we cannot use optimal methods for larger DDG. So we use our heuristic based on SIRA. We have used loop unrolling to produce larger DDGs. Figure 7 plots the reduced PRS of many DDG with various unrolling (from 1 to 10) degrees and various numbers of available registers ($\mathcal{R} = 16, 32, 64, 128$). As can be seen, our heuristic succeeds in reducing PRS below $\mathcal{R}$. Figure 8 shows that the ILP loss is saved. It is equal to zero in most cases, while it may be much greater if the register pressure is high.

5 Related Work

The introduction of instruction level parallelism (ILP) has rendered inadequate the classical techniques of register allocation (RA) for sequential code semantics. As well known in backend compilation, there is a phase ordering problem between classical register allocation techniques and instruction scheduling. If a classical RA is done early, the introduced false dependences inhibit instruction scheduling from extracting a schedule with high amount of ILP. However, this conclusion does not prevent a compiler from effectively performing an early RA, with the condition that the allocator is sensitive to the scheduler. Such schedule sensitive register allocation methods have been studied in [8, 11, 13]. Until now, the problem of optimal spill insertion in ILP codes is not understood yet: the cache effects of memory operations on ILP scheduling are still not understood. However, optimal spill code insertion inside static issue slots can be applied [15], but the generated code is not necessarily optimal because memory operations have unknown static latencies (cache effects).

ILP scheduling is a special case of the general k-periodic multidimensional scheduling problem. Indeed, researchers in this area studied the special case when the scheduling period is unique and integral as done in [12] improved recently in [7]. In case of cyclic scheduling under register constraints, most of the approaches try to build a SWP schedule with a minimised MAXLIVE, see [4, 6, 22]. Then, in a second step, cyclic register allocation can be applied using the methods described in [9] improved later in [3]. We can also do a cyclic register allocation sensitive to SWP as done in [19]. All these previous techniques try to minimise the register requirement, not to maximise it as in the PRS approach.

The case of multidimensional memory storage optimisation is also interesting if we target regular loop nests for high performance codes [1, 20, 21]. A heuristic in case of registers is presented in [10]. However, such approaches are not considered yet in our specific embedded computing for many reasons: 1) our target loops are one-dimensional 2) our one-dimensional embedded loops contain enough ILP, so we do not need to optimise the whole loop nest 3) exploiting ILP and registers in multidimensional loop nests requires larger code size [2] than exploiting the ILP in innermost loops, while code size is an important optimisation aspect in embedded codes 4) the problem of optimal register allocation in multidimensional loops is still an open problem; a sub-optimal heuristic for this problem is presented in [10].

6 Conclusion

The register saturation is the exact maximal register need of any valid instruction schedule of a data dependence graph. If such DDG represents a Directed Acyclic Graph (DAG) of a basic block, then this study has been done in [17]. If such DDG represents the data dependences of an innermost loop (with possible recurrences), then our current article shows how do we extend the theoretical study. Indeed, the case of loops is more complex since it requires to consider periodic instruction scheduling (software pipelining).

Many practical applications may profit from PRS computation: 1) for compiler technology, PRS calculation provides new opportunities for avoiding and/or verifying useless spilling; 2) for JIT compilation, PRS metrics may be embedded in the generated byte-code as static annotations, which may help the JIT to dynamically schedule instructions without worrying about register constraints; 3) for helping hardware designers, PRS computation provides a static analysis of the exact maximal register requirement.

We show that our formula of computing the MAXLIVE (Equation 1 in [18]) is useful to compute the register sufficiency (minimise of maximum [18]) and to compute the register saturation (maximise a maximum) in this article. Furthermore, our current experiments demonstrate that our formula of MAXLIVE computation is more efficient, and scales better than the commonly used technique.

If the computed register saturation exceeds the number of available registers, we can bring a method to reduce this maximal register need in a sufficient way to just bring it below the limit without minimising it at the lowest possible level. Register saturation reduction must take care of MII, i.e., it should not increase the critical cycle if possible. We proved that this problem is NP-hard, and we provided optimal and approximate methods.

In theory, and contrary to the acyclic case [17], the periodic register saturation can be unbounded when dealing with loops scheduled with periodic schedules without resource constraints. However, in practice, experiments with SWP and many DDGs show that the register saturation is bounded. Consequently, it is useful and efficient to decouple register constraints from resource constraints. Our methods of PRS reduction do not introduce new edges in many cases. When edges are introduced to reduce PRS, our method takes care of not increasing MII, because it does not minimise PRS at its lowest possible value.

Acknowledgement

This research result has been partially funded by the ANR MOPUCE project (ANR number 05-JCJC-0039) and the European HIPEAC network of excellency. We would like to thank Alain DARTE from ENS-Lyon for his helpful remarks. This research result would not succeed without the valuable support of the University of Versailles Saint-Quentin en Yvelines, INRIA-Rocquencourt and INRIA-Saclay in France.

References

[1] Alain Darte and Robert Schreiber and Gilles Villard . Lattice-Based Memory Allocation . *IEEE Transactions on Computers*, pages 1242–1257, October 2005.

[2] Cédric Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *PACT*, pages 7–16. IEEE Computer Society, 2004.

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

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

[5] Christine Eisenbeis, Franco Gasperoni, and Uwe 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.

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

[7] Florent Blachot and Benot Dupont-de-Dinechin and Guillaume Huard. SCAN: A Heuristic for Near-Optimal Software Pipelining. In *Euro-Par*, 2006.

[8] Ramaswamy Govindarajan, Hongbo Yang, José N. Amaral, Chihong Zhang, and Guang R. Gao. Minimum Register Instruction Sequencing to Reduce Register Spills in Out-of-Order Issue Superscalar Architecture. *IEEE Transactions on Computers* , pages 4–20, 2003.

[9] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji. A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs. *Lecture Notes in Computer Science*, 641:176–??, 1992.

[10] Hongbo Rong and Alban Douillet and Guang R. Gao. Register Allocation for Software Pipelining Multi-dimensional Loops. *ACM SIGPLAN Notices*, 40(6):154–167, June 2005.

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

[12] Monica S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *PLDI*, pages 318–328, 1988.

[13] Schlomit S. Pinter. Register Allocation with Instruction Scheduling: A New Approach. *SIGPLAN Notices*, 28(6):248–257, June 1993.

[14] Premysl Sucha and Zdenek Hanzálek. Scheduling of Tasks with Precedence Delays and Relative Deadlines - Framework for Time-optimal Dynamic Reconfiguration of FPGAs. In *IPDPS*, pages 1–8. IEEE, 2006.

[15] Santosh G. Nagarakatte and R. Govindarajan. Register Allocation and Optimal Spill Code Scheduling in Software Pipelined Loops Using 0-1 Integer Linear Programming Formulation. In *CC'07*, volume 4420 of *LNCS*, pages 126–140. Springer, 2007.

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

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

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

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

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

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

[22] Jian Wang, Andreas Krall, and M. Anton Ertl. Decomposed Software Pipelining with Reduced Register Requirement. In *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT95*, pages 277 – 280, Limassol, Cyprus, June 1995.