On the Optimality of Register Saturation

Sid Touati

On the Optimality of Register Saturation

Sid-Ahmed-Ali TOUATI

University of Versailles, PRiSM laboratory, France
touati@prism.uvsq.fr

Abstract

In an optimizing compiler, the register allocation process is still a crucial phase since it allows to reduce spill code that damages the performances. The register constraints are generally taken into account during the instruction scheduling phase of an acyclic data dependence graph (DAG) : any schedule must minimize the register requirement. However, in a previous work [14], we introduced and mathematically studied the register saturation (RS) concept. It consists of computing the exact upper-bound of the register need for all the valid schedules, independently of the functional unit constraints. The goal of RS is to decouple register constraints from instruction scheduling.

In this paper, we continue our theoretical efforts and we present two main results. First, we give an exact solution with integer linear programming for both the problems of computing the RS of a DAG and reducing it. Our integer program brings a new way to model register constraints that allows us to produce the lowest number of constraints and variables in the literature (till now). Indeed, given a DAG with n nodes and m arcs, we need $\mathcal{O}(n^2)$ integer variables and $\mathcal{O}(m+n^2)$ linear constraints, which is better than the actual size complexity in the literature that model register constraints. Second, we prove that the problem of reducing the register saturation is NP-hard. Our detailed experiments in this paper show that our previous heuristics [14] are nearly optimal. We provide a discussion too in order to argument why the RS approach should be better that minimizing the register requirement.

Key words: Register Pressure, Instruction Level Parallelism, Integer Linear Programming, Optimizing Compilation.

1 Introduction

Because of the introduction of instruction level parallelism (ILP), the classical techniques of register allocation for sequential code semantics are not adapted any more. Thus, the old graph coloring techniques should be reconsidered to be efficient in optimizing compilers for modern architectures. In [5], the authors showed that there is a phase ordering problem between the old register allocation techniques and ILP instruction scheduling. If a classical register allocation is done early, the introduced false dependences inhibit a good further ILP extraction. However, this

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

conclusion does not prevent any compiler from performing effectively an early register allocation, but with the condition that the used allocator should be sensitive to the scheduler as done in [12,11,8].

Some other studies [13,10,5] claim that it is better to combine instruction scheduling and register allocation in a single complex pass, argumenting the fact that applying each method separately has a negative influence on the efficiency of the other. However, this phase ordering problem arises only if the applied first pass (ILP scheduler or register allocator) is "selfish". Indeed, we still can effectively decouple register constraints from instruction scheduling if enough care is taken. In this paper, we show how we can treat register constraints before scheduling and we explain why we should do it.

The principal reason for handling register constraints before instruction scheduling our believe that the register allocation is more important as an optimization issue than code scheduling. This is because, usually, the code performances are far more sensitive to memory accesses than to fine-grain scheduling (memory gap) : a cache miss may inhibit the processor from achieving a high dynamic ILP, even if the scheduler has extracted it at compile time. Even if someone would expect that spill codes exhibit high locality, and hence would likely produce cache hits, we cannot assert it at compile time since memory access latencies are non predictable at compile time (we cannot guarantee where the date would be located). Furthermore, memory requests, even if they are data independent, exhibit high potential conflicts because of micro-architectural restrictions and simplifications in the memory disambiguation mechanisms (load/store queues) and possible banking structure in cache levels [7]. These possible conflicts may cause severe performance degradation even if enough ILP exist, and even if the data is located in the cache. Of course, our claim that spill code is more damaging is appropriate for the architectures where the memory access delay is very long compared to the delay of calculation. This is the case in almost all high performance processors. If memory access delay is not critical, the register saturation concept may be less useful.

Another reason for handling register constraints prior to ILP scheduling is that register constraints are much more complex than resource constraints. Scheduling under resource constraints is a performance issue. Given a data dependence graph (DDG), we are sure to find at least one valid schedule for any underlying hardware properties (a sequential schedule in extreme case, i.e., no ILP). However, scheduling a DDG with a limited number of registers is more complex. We cannot guarantee the existence of at least one schedule. In some cases, we must introduce spill code and hence we change the problem (the input DDG). Also, a combined pass of scheduling with register allocation presents an important drawback if not enough registers are available. During scheduling, we may need to insert load-store operations if no enough free registers exist. We cannot guarantee the existence of a valid issue time for these introduced memory access in an already scheduled code; resource or data dependence constraints may prevent from finding a valid issue slot inside an already scheduled code. This fact forces to iteratively apply scheduling followed by spilling until reaching a solution.

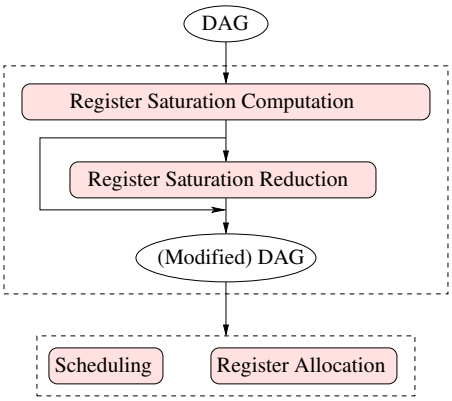

Fig. 1. Early Register Pressure Management

All the above arguments make us re-think new ways of handling register pressure before starting the scheduling process, so that the scheduler would be free from register constraints and would not suffer from excessive serializations. For this reason, we presented in [14] our register saturation (RS) concept that prevent a DAG from producing an excessive number of values simultaneously alive for all the valid schedules. Our pre-pass analyzes a DAG (with respect to control flow) to deduce the maximal register need for all schedules. We call this limit *the register saturation* (RS) because the register need can reach this limit but never exceed it. If RS exceeds the number of available registers, we introduce new arcs to reduce it, see Figure 1. In this paper, we provide exact (optimal) methods to both the problems of computing RS and reducing it. After our RS analysis pass, the DAG is free from register constraints and can be sent to the scheduler and the register allocator.

This article is organized as follows. Section 2 presents our DAG and processor model which can be used for most of existing ILP architectures (superscalar, VLIW, EPIC/IA64). Computing the optimal RS by intLP is given in Section 3. Our intLP formulation use the linear writing of logical formulas ($\Longrightarrow$, $\Longleftrightarrow$, $\vee$) and the max operator ($max(x, y)$) by introducing extra binary variables, as previously described in [15]. The optimal solution of reducing RS is provided in Section 4. Our large range of experiments show that our initial heuristics [14] are nearly optimal in Section 5. Before concluding, we make a discussion in Section 6 to argument why the RS concept is a better way for handling register constraints prior to ILP scheduling.

2 DAG and Processor Model

A DAG $G = (V, E, \delta)$ in our study represents the data dependences between the operations and any other serial constraints. The DAG is defined by its set of operations V, its set of edges $E = \{(u, v) / \ u, v \in V\}$, and δ such that $\delta(e)$ is the latency of the edge e in terms of processor clock cycles. Let n be the number of nodes and m the number of arcs.

A schedule σ of G is a function which gives an integer execution (issue) time for each operation :

$$\sigma \text{ is valid} \Longleftrightarrow \forall e = (u, v) \in E, \quad \sigma(v) - \sigma(u) \geq \delta(e)$$

The set of all valid acyclic schedules of G is denoted by $\Sigma(G)$.

We consider a target RISC-style architecture with multiple register types, where $\mathcal{T}$ denotes the set of register types (for instance, $\mathcal{T} = \{int, float\}$). Some statements and some precedence constraints of a DAG have more attributes than others, depending if they refer to values to be stored in registers or not. $V_{R,t}$ is the set of values to be stored in registers of type $t \in \mathcal{T}$. We consider that each statement $u \in V$ writes into at most one register of a type $t \in \mathcal{T}$. The statements which define multiple values with different types are accepted in our model if they do not define more than one value of a certain type [1]. $E_{R,t}$ is the set of flow dependence edges through a value of type $t \in \mathcal{T}$. The set of consumers (readers) of a value u^t is then the set :

$$Cons(u^t) = \{v \in V \mid (u, v) \in E_{R,t}\}$$

Some values in may not be consumed in the considered DAG. In order to model such exit values, we assume that the considered DAG contains a bottom node $\perp$ that is the sink of the flow dependences of these exit values. Also, there is a serial arc from any other node of the DAG to this bottom node. The latency of such virtual arc is equal to the latency of the source operation. The bottom node $\perp$ is always the last scheduled node of the DAG.

In order to consider static issue VLIW and EPIC/IA64 processors in which the hardware pipeline steps are visible to compilers (we consider dynamically scheduled 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 (architectural visible). We define two delay (offset) functions δ_r and δ_w in which : the read cycle of u^t from a register of type t is $\sigma(u) + \delta_r(u)$, and the the write cycle of u^t into a register of type t is $\sigma(u) + \delta_w(u)$.

For instance, in superscalar and EPIC/IA64 processors, δ_r and δ_w are equal to zero.

When a schedule is fixed, we can easily compute how much register we need of each register type t in order to build a valid register allocation. It is the standard concept of the maximal number of values of type t simultaneously alive, that is also equal to the maximal clique in the interference graph. To recall, two variables are said to be simultaneously alive iff their lifetime intervals interfere, and thus they cannot share the same register. The register requirement (or register need) of type t for a DAG G given a fixed schedule σ is noted $RN^{\sigma}_{t}(G)$.

After defining the DAG and processor model, the next section shows how we compute the exact optimal register saturation (RS).

[1] This model limitation, that has little impact on actual ILP processors, allows us to have some graph characteristics used for formal proofs in [15,14].

3 Computing the Optimal Register Saturation

First of all, if $|V_{R,t}|$, the total number of values of type t, is less than or equal to $\mathcal{R}_t$, the number of available registers of type t, then we are sure that any schedule cannot require more than $|V_{R,t}| \leq \mathcal{R}_t$ registers. Otherwise, we must analyze the register saturation (RS).

The RS of a register type t for a DAG G is the maximal register need for all valid schedules of this DAG :

$$RS_t(G) = \max_{\sigma \in \Sigma(G)} RN_t^{\sigma}(G)$$

We proved in [14] that computing this parameter is an NP-complete problem and we provided a heuristics. Below, we give the set of variables and constraints of an exact intLP for computing $RS_t(G)$. Our intLP formulation use the linear writing of logical formulas ($\Longrightarrow$, $\vee$, $\Longleftrightarrow$) and the max operator ($max(x, y)$) by introducing extra binary variables, as previously described in [15]. However, that linear writing of logical and max operators requires to bound the domain set of the integer variables.

3.0.1 Scheduling Variables

For all operations $u \in V$, we define the integer variable $\sigma_u \geq 0$ that holds the schedule time. Note that these schedule variables do not represent the final schedule under resource constraints (that will be computed after our RS pass), but they only represent intermediate variables for our intLP formulation. The first linear constraints are those that describe precedence relations, so we write into the intLP system :

$$\forall e = (u, v) \in E, \qquad \sigma_v - \sigma_u \geq \delta(e)$$

There are $\mathcal{O}(n)$ scheduling variables and $\mathcal{O}(m)$ linear scheduling constraints. In order to bound the domain set of our variables, we define T a worst possible schedule time. We choose T sufficiently large, where for instance $T = \sum_{e \in E} \delta(e)$ is a suitable worst total schedule time (case of no ILP). Then, we write the following constraint :

$$\forall u \in V, \sigma_u \leq T$$

As a consequence, we deduce for any $u \in V$:

- $\sigma_u \geq \underline{\sigma_u} = LongestPathTo(u)$ is the "as soon as possible" schedule time;
- $\sigma_u \leq \overline{\sigma_u} = T - LongestPathFrom(u)$ is the "as late as possible" schedule time according to the worst total schedule time T.

3.0.2 Register Need Constraints

Interference Graph

The lifetime interval of a value u^t of type t is (given a schedule σ)

$$LT_{\sigma}(u^t) =]\sigma_u + \delta_w(u), \max_{v \in Cons(u^t)} \big(\sigma_v + \delta_r(v)\big)]$$

That is, we assume that a value written at instant c in a register is available one step later (the lifetime interval is left open). Thus, if an operation u reads from a register at instant c while another operation v is writing in it at the same time, u does not get v's result but gets the value previously stored in this register. Note that this is a choice and not a limitation of the model.

We define for each value u^t the variable $k_{u^t} \geq 0$ that computes its killing date (the last time that this value is read). The number of such defined variables is$\mathcal{O}(n)$. Since our variable domains are bounded (assuming a finite T), we know that k_{u^t} is bounded by the two following finite schedule times :

$$\forall t \in \mathcal{T},\ \forall u^t \in V_{R,t}\ : \qquad \underline{k_{u^t}} \leq k_{u^t} \leq \overline{k_{u^t}}$$

where

- $\underline{k_{u^t}} = \underline{\sigma_u} + \delta_w(u)$ is the first possible definition date of u^t;
- $\overline{k_{u^t}} = \max_{v \in Cons(u^t)} \big(\overline{\sigma_v} + \delta_r(v)\big)$ is the latest possible killing date of u^t.

We use the linear constraints of the max operator to compute k_{u^t} as explained in [15]. We write into the intLP system :

$$\forall u^t \in V_{R,t}\ : \qquad k_{u^t} = \max_{v \in Cons(u^t)} \big(\sigma_v + \delta_r(v)\big)$$

The total complexity to define all killing dates for all registers types is bounded by $\mathcal{O}(n^2)$ variables and $\mathcal{O}(n^2)$ constraints.

Now, we can consider H_t the undirected interference graph of G for the register type t. For any couple of distinct values $u^t, v^t \in V_{R,t}$, we define a binary variable $s^t_{u,v} \in \{0,1\}$ such that it is set to 1 if the two lifetimes intervals of type t interfere : $\forall t \in \mathcal{T},\ \forall$ couple $u^t, v^t \in V_{R,t}$:

$$s^t_{u,v} = \begin{cases} 1 \text{ if } LT_\sigma(u^t) \cap LT_\sigma(v^t) \neq \phi \\ 0 \qquad\qquad \text{otherwise} \end{cases}$$

The number of variables $s^t_{u,v}$ is the number of combinations of 2 values among $|V_{R,t}|$, i.e., $\big(|V_{R,t}| \times (|V_{R,t}| - 1)\big)/2$.

$LT_\sigma(u^t) \cap LT_\sigma(v^t) = \phi$ means that one of the two lifetime intervals is "before" the other, i.e., $\big(LT_\sigma(u^t) \prec LT_\sigma(v^t)\big) \vee \big(LT_\sigma(v^t) \prec LT_\sigma(u^t)\big)$, where $\prec$ denotes the "before" relation in the interval algebra. Then, we have to express the following constraints :

$$s^t_{u,v} = 1 \Longleftrightarrow \neg\big(LT_\sigma(u^t) \prec LT_\sigma(v^t) \vee LT_\sigma(v^t) \prec LT_\sigma(u^t)\big)$$

where $LT_\sigma(u^t) \prec LT_\sigma(v^t)$ iff $k_{u^t} \leq \sigma_v + \delta_w(v)$. The negation of this constraint is $k_{u^t} > \sigma_v + \delta_w(v)$,i.e.,
$k_{u^t} - \sigma_v - \delta_w(v) - 1 \geq 0$. Since $s^t_{u,v} \in \{0,1\}$, these variables are constrained as

follows :

$$s^t_{u,v} \geq 1 \Longleftrightarrow \begin{cases} k_{u^t} - \sigma_v - \delta_w(v) - 1 \geq 0 \\ k_{v^t} - \sigma_u - \delta_w(u) - 1 \geq 0 \end{cases}$$

Given three logical expressions (P, Q, S), $(P \Longleftrightarrow (Q \wedge S))$ is equivalent to the expression $(P \wedge Q \wedge S) \vee (\neg P \wedge \neg Q) \vee (\neg P \wedge \neg S)$. We write these two disjunctions with linear constraints by introducing binary variables (see [15]) and by computing the finite lower bounds of the linear functions. The complexity of computing all the $s^t_{u,v}$ variables is bounded by $\mathcal{O}(n^2)$ binary variables and constraints.

Maximal Clique in the Interference Graph

The maximum number of values of type t simultaneously alive corresponds to a maximal clique in $H_t = (V_{R,t}, \mathcal{E}_t)$, where $(u^t, v^t) \in \mathcal{E}_t$ iff their lifetime intervals interfere ($s^t_{u,v} = 1$). For simplicity, rather than considering the interference graph itself, we prefer to consider its complementary graph $H'_t = (V_{R,t}, \mathcal{E}'_t)$ where $(u^t, v^t) \in \mathcal{E}'_t$ iff their lifetime intervals do *not* interfere ($s^t_{u,v} = 0$). Then, the maximum number of values of type t simultaneously alive corresponds to a maximal independent set in H'_t.

To write the constraints that describe independent sets (IS), we define a binary variable $x_{u^t} \in \{0, 1\}$ for each value $u^t \in V_{R,t}$ such that $x_{u^t} = 1$ iff u^t belongs to some IS of H'_t. We express in the model the following linear constraints :

$$\forall x_{u^t}, x_{v^t} \in V_{R,t} : \qquad s^t_{u,v} = 0 \Longrightarrow x_{u^t} + x_{v^t} \leq 1$$

This equations means that if two nodes u and v are connected in H', then one and only one of them may belong to an IS. The number of variables x_{u^t} is $\mathcal{O}(n)$. The number of introduced binary variables to express all the implications is bounded by $\mathcal{O}(n^2)$. The number of linear constraints to define the IS is bounded by $\mathcal{O}(n^2)$.

3.0.3 Linear Function of Register Need

The register requirement of type t is a maximal IS in H'_t, i.e., the maximal $\sum_{u^t \in V_{R,t}} x_{u^t}$. Thus, the register saturation of type t is computed by :

$$\text{Maximize} \sum_{u^t \in V_{R,t}} x_{u^t}$$

The total number of integer variables in our whole intLP is bounded by $\mathcal{O}(|V|^2)$, and the total number of constraints is at most $\mathcal{O}(m + n^2)$. Note that our intLP formulation may be optimized by considering that:

- an edge $e = (u, v)$ in the initial DAG is redundant for the scheduling constraints and can be safely ignored if $lp(u, v) > \delta(e)$ where $lp(u, v)$ denotes the longest path from u to v (with the condition that this arc doesn't belong to this longest path);

- two values $(u^t, v^t) \in V_{R,t}$ can never be simultaneously alive iff for all the possible schedules, one value is always defined after the killing date of the other. This is the case if any of the two following conditions is satisfied :

$$\begin{aligned} \forall v' \in Cons(v^t) : \quad & lp(v', u) \geq \delta_r(v') - \delta_w(u) \\ \vee \ \forall u' \in Cons(u^t) : \quad & lp(u', v) \geq \delta_r(u') - \delta_w(v) \end{aligned}$$

After computing the optimal RS, the next section shows how we reduce it if it exceeds a limit.

4 Optimal Register Saturation Reduction

In the case where the register saturation $RS_t(G)$ exceeds the number of available registers $\mathcal{R}_t$ of the type t, then we must add extra serial arcs into the DAG G to reduce $RS_t(G)$ below this limit. The new added arc must save ILP as much as possible by taking care of the critical path. We note by $\overline{E}$ the set of extra edges that we add to G to build a new extended DAG, namely $\overline{G} = G\backslash^{\overline{E}}$, such that $RS_t(\overline{G}) \leq \mathcal{R}_t$. We want to solve the formal problem stated below.

Definition 4.1 [ReduceRS Problem] Let $G = (V, E, \delta)$ be a DAG. Let $\mathcal{R}_t$ and $\mathcal{P}$ be two positive integers. Does there exist an extended DDG $\overline{G} = G\backslash^{\overline{E}}$ of G such that :

$$RS_t(\overline{G}) \leq \mathcal{R}_t$$

and

$$CriticalPath(\overline{G}) \leq \mathcal{P}$$

Theorem 4.2 *ReduceRS problem is NP-hard.*

Proof.

We prove that ReduceRS problem reduces from the problem of scheduling under register constraints. Let us start by defining the latter problem. For the sake of clarity of this proof, we assume that the considered register type t is implicit (we do not include t in our notations inside this proof).

Definition 4.3 [SRC problem]

Let $G = (V, E, \delta)$ be a DAG, $\mathcal{R}$ be a positive integer, and $\mathcal{P}$ be a length. Does there exist a valid schedule $\sigma \in \Sigma(G)$ such that :

$$RN^{\sigma}(G) \leq \mathcal{R}$$

and

$$\text{total schedule time } \leq \mathcal{P}$$

SRC problem has been proven NP-hard in [4]. Now we prove that both ReduceRS and SRC problems are equivalent in terms of computational complexity.

1. **ReduceRS $\Longrightarrow$ SRC**

Let $\overline{G}$ be a solution for the ReduceRS problem. Then trivially, any as soon as possible schedule $\sigma \in \Sigma(\overline{G})$ is a solution for SRC.

2. **SRC $\Longrightarrow$ ReduceRS**

Let σ be a solution for SRC, i.e., $RN^{\sigma}(G) \leq \mathcal{R}$ and the total schedule time is $\leq \mathcal{P}$. We build an extended DDG $\overline{G}$ by adding serial arcs to impose value lifetimes of any schedule of $\overline{G}$ to have the same precedence relations as defined by σ. $\forall u, v \in V_R / LT_{\sigma}(u) \prec LT_{\sigma}(v)$ then we add the following arcs :

- if $v \in Cons(u)$, then add serial arcs from the other u's readers (except v) to v; the set of added arcs is :

$$\Big\{e = (u', v) / \ u' \in Cons(u) - \{v\}\Big\}$$

- else, add serial arcs from all u's readers to v; the set of added arcs is :

$$\Big\{e = (u', v) / \ u' \in Cons(u)\Big\}$$

The latency of these added arcs has to be chosen depending on the target codes. We have two cases,

(i) in the case of superscalar codes, the semantics is sequential. So, the latency of each added arc is set to 1;

(ii) in the case of VLIW or EPIC/IA64, there exist reading and writing offsets [2]. Thus, for each added arc $e = (u', v)$, the latency is set to $\delta(e) = \delta_r(u') - \delta_w(v)$.

Indeed, the added arcs and the chosen latencies force the following assertion :

$$LT_{\sigma}(u) \prec LT_{\sigma}(v) \Longrightarrow \forall \sigma' \in \Sigma(\overline{G}) : \quad LT_{\sigma'}(u) \prec LT_{\sigma'}(v)$$

Then, for all values non simultaneously alive according to σ, there is no schedule σ' of $\overline{G}$ that makes them simultaneously alive. Formally, it is written :

$$\neg\Big(\exists u, v \in V_R, \ LT_{\sigma}(u) \prec L_{\sigma}(v), \ \exists \sigma' \in \Sigma(\overline{G}) / \ LT_{\sigma'}(u) \cap LT_{\sigma'}(v) \neq \phi\Big)$$

In other words, we ensure that any schedule of $\overline{G}$ will guarantee the precedence relations between the lifetime intervals of G according σ. Consequently, any schedule σ' of $\overline{G}$ cannot need more than the register need of σ and

$$RS(\overline{G}) = RN^{\sigma}(G) \leq \mathcal{R}$$

A solution for SRC problem may create a circuit in the solution of ReduceRS. We are sure that if any circuit is introduced in $\overline{G}$, then it must be non-positive because there exists at least the valid schedule $\sigma \in \Sigma(\overline{G})$. Consequently, a solution

[2] On EPIC/IA64 architectures, a writer and a reader can be scheduled at the same instruction group, so the writing delay is statically considered as zero.

of the ReduceRS problem may produce a cyclic DDG. We will see later how to eliminate these solutions.

With regard to the critical path of $\overline{G}$, the introduced serial arcs ensure that at least $\sigma \in \Sigma(\overline{G})$. Since there exists such a schedule with a total time $\leq \mathcal{P}$, the critical path of $\overline{G}$ cannot be longer than $\mathcal{P}$. □

The proof of Theorem 4.2 gives the intuition for optimal solution of the ReduceRS problem using integer programming. It is computed in two steps :

(i) we first compute a valid schedule σ such that the register need of type t is maximized and does not exceed $\mathcal{R}_t$, while the total schedule time is bounded. Again, this schedule is different from the final one to be computed under resource constraints;

(ii) then, we add serial arcs as described by the proof of Theorem 4.2. This results in an extended DDG that has a bounded register saturation with a minimized critical path.

To compute such a schedule, we use our intLP formulation previously defined in Section 3 that maximizes the register need. We keep all the constraints and variables of Section 3, except those that compute a maximal independent set. Now, we use a binary variable $x^i_{u^t}$ which is set to 1 if the value u^t is stored in the register i. Since there are $\mathcal{R}_t$ available registers, we have at most $|V| \times \mathcal{R}_t$ variables. Since $\mathcal{R}_t$ is a constant is our problem (the number of registers in the target machine), the number of these variables is $\mathcal{O}(|V|)$.

The intLP system tries to build a coloring of the interference graph with exactly $\mathcal{R}_t$ colors (the maximal number of available registers). If no solution can be found with $\mathcal{R}_t$ registers, then solve another intLP after decrementing $\mathcal{R}_t$ (until to 1). If no final solution can be found when reaching one available register, then the register saturation cannot be reduced and spilling is unavoidable. The variables $x^i_{u^t}$ are computed using following constraints.

- a value u^t is stored in only one register of type t :

$$\forall t \in \mathcal{T}, \forall u^t \in V_{R,t} : \quad \sum_{i=1}^{\mathcal{R}_t} x^i_{u^t} = 1$$

- if two values interfere, then they cannot share the same register :

$$\forall t \in \mathcal{T}, \forall \text{ couple } u^t, v^t \in V_{R,t} : \quad s^t_{u,v} \geq 1 \Longrightarrow \left(x^i_{u^t} + x^i_{v^t} \leq 1, \quad \forall i = 1, ..., \mathcal{R}_t\right)$$

There are at most $\mathcal{O}(V^2 \times \mathcal{R}_t) = \mathcal{O}(V^2)$ such constraints.

- The objective function minimizes the total schedule time :

$$\text{Minimise } \sigma_\perp$$

We add at most $\mathcal{O}(|V|^2)$ constraints to the previous intLP system of Section 3.

As explained before, our DAG and processor model include writing and reading offsets. Consequently, in some cases, the optimal RS reduction may need to introduce non-positive circuits into the original DAG. Even if such non-positive circuits do not prevent the graph from being scheduled, they still violate the DAG property and impose hard scheduling constraints that may not be satisfiable under resource constraints in the subsequent pass of instruction scheduling. We must eliminate such optimal solutions as explained in [15]. Basically, we add at most $\mathcal{O}(n^3)$ variables and $\mathcal{O}(m + n^3)$ constraints to guarantee the existence of a topological sort for extended DDG. Again, these constraints have only to be added for VLIW and EPIC codes, not for superscalar ones.

5 Experiments

We did an extensive set of experiments on some scientific codes extracted from SpecFP, whetstone, livermore and linpack We used CPLEX to solve our intLP programs. Since all the problems of RS computation and reduction are NP-hard, reaching the optimal solutions were very time consuming (from many seconds to many days). The experimented DAGs are simply some loop bodies (excluding branches). Detailed numerical results and plots are shown in [15]. Basically, all our heuristics presented in [14] are nearly optimal.

Regarding RS computation, the maximal empirical error is one register (in very few cases). For RS reduction, we have to explore two parameters : the reduced RS and the critical path. We note $\overline{RS}$ and $\overline{ILP}$ the RS reduction and ILP loss resulted from optimal intLP programs; we note $\overline{RS^*}$ and $\overline{ILP^*}$ the RS reduction and ILP loss resulted from our heuristics. Then, our experiments are divided into the following sets.

(i) In the case where $\overline{RS} = \overline{RS^*}$, our algorithm succeeds in optimally reducing RS. Then, the ILP loss may be :
  (a) $\overline{ILP} = \overline{ILP^*}$ (72.22% of all the results). Our algorithm succeeds in optimally reducing RS with the optimal ILP loss.
  (b) $\overline{ILP} < \overline{ILP^*}$ (18.5% of all the results). Our algorithm succeeds in optimally reducing RS but with sub-optimal ILP loss.
  (c) $\overline{ILP} > \overline{ILP^*}$ impossible !

(ii) In the case where $\overline{RS} > \overline{RS^*}$, our algorithm did not succeed in optimally reducing RS. Then, the ILP loss may be :
  (a) $\overline{ILP} = \overline{ILP^*}$ (4.63% of all the results). Our algorithm has sub-optimal RS reduction but optimal ILP loss.
  (b) $\overline{ILP} < \overline{ILP^*}$ (less than 1% of all the results). Our algorithm has sub-optimal RS reduction with sub-optimal ILP loss.
  (c) $\overline{ILP} > \overline{ILP^*}$ (3.7% of all the results). Our algorithm has sub-optimal RS reduction but with *super*-optimal ILP loss. This case is interesting : since our algorithm has sub-optimal RS reduction, then it gets extra registers which allow him to exploit more ILP.

(iii) The case where $\overline{RS} < \overline{RS^*}$ is impossible because our heuristics computes a valid RS^*.

Clearly, our RS reduction algorithm is very efficient : it, in most of times, optimally reduces RS with optimal ILP loss. Sub-optimal ILP loss is, in most of times, accompanied with optimal RS reduction, while sub-optimal RS reduction is mostly accompanied with *super*-optimal ILP loss. We get both sub-optimal ILP loss and sub-optimal RS reducing in less than 1% of the cases.

6 Discussion : to minimize or to saturate the register need ?

The literature contains a lot of techniques about minimizing the register requirement in superscalar (sequential) codes that are sensitive to ILP scheduling [6,8]. Others prefer to combine ILP scheduling with register allocation [2,5,9]. All these techniques try to minimize the register requirement. In our method, we use the contrary approach : we maximize the register requirement in order to minimize the amount of added arcs as previously done by Berson in [1]. We provided in [14] a comparison with his method to show its limitations that we fixed in our work.

Minimizing the register requirement is inherently a worse technique than saturating it because of many reasons. We explain them below.

Case where register pressure is zero

Given a DAG, we do not need to add serial arcs if RS does not exceed the number of available registers. While the minimization approach add extra arcs, our method doesn't. For instance, look at Figure 2, where bold circles are the values to be stored in registers and bold arcs are the flow dependences. The initial DAG has clearly a register saturation equal to 4 : this is because we can schedule the 4 operations $\{a, b, c, d\}$ so as to produce 4 values simultaneously alive. If the processor has at least 4 registers, then the DAG is let as it is for the subsequent scheduler. However, with a minimization approach, the new DAG in Part (b) is restricted to not require more than 2 registers [3], regardless the number of available ones. As can be seen, the DAG in Part (b) is more restrictive than the initial one that is let free by the RS analysis pass.

How much arcs we introduce

If the inherent data dependences of a considered DAG produce a restrictive register pressure on a further ILP scheduler (case where RS exceeds the number of available registers), the minimization approach add more arcs than the RS reduction approach. This is because our method introduce only the sufficient amount of arcs to reduce RS below the considered limit. However, the minimization approaches try to reduce the register need at the lowest possible level, which is not the appropriate method since it does not fully utilize the available registers. For instance, look at

[3] Here, we minimize the register requirement under critical path constraints.

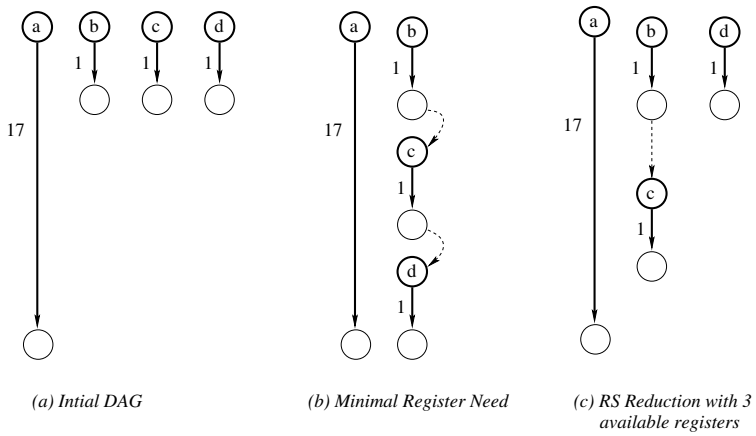

Fig. 2. RS Reduction vs. Minimal Register Requirement

Figure 2 and assume we have 3 registers available. Part (c) shows the new DAG produced by the RS reduction pass : here, RS is reduced from 4 to 3, and hence we have less arcs than what produces the minimization approach in Part (b). For the former, the final allocator would use 1, 2 or 3 registers depending on the schedule; for the latter, we would use only 1 or 2 registers , which is more restrictive. Hence, the RS concept helps to better take benefit from available registers.

When both methods are equivalent

If the target processor is a superscalar out of order, and if its dynamic scheduler is optimal and the register renaming hardware support has an infinite number of hidden registers, both methods (RS and register need minimization) should be equivalent. With limited number of hidden registers for renaming, and a sub-optimal runtime scheduler, our RS method is likely to produce better codes because it makes better use of the available registers.

Our method apply for explicit reading/writing offsets

Our DAG and processor model admit explicit reading from and writing to registers. Thus, our method is more generic than the existing techniques, and can be applied for superscalar, VLIW and EPIC codes. For the two later cases, a special care must be taken when reducing RS : we must prohibit non-positive circuits in the resulted DAGs.

In the case of a global scheduler

Our model assumes that there is only one possible definition per value. This assumption is correct inside a basic bloc (BB), i.e., if the code does not contain branches. In the case of a global control flow graph (CFG), a static data dependence analysis may provide for some values more than one definition because it cannot determine which execution path is taken. Actually, we have shown in [15] how to extend RS analysis to a global acyclic CFG (excluding loops) and its interaction with a global instruction scheduler that may move operations from one BB

to another. However, we must be aware that in global register allocation, the number of allocated registers may be superior to MAXLIVE, because of the possible insertion of extra "move" operations. We think that we can prove that the optimal difference is at most one extra register by using the theoretical results of [3]. So, we can still use the register saturation concept in a global acyclic CFG by taking into account this possible extra register. For example, this can be done by decrementing $\mathcal{R}$ the number of available registers, so that the final register allocation cannot exceed $\mathcal{R}$, even if move operations have been inserted.

Similar Work

Our theoretical framework is an extension to the previous study done by Berson [1]. A better explanation of the differences between our heuristics [14] and Berson's one can be retrieved from [14]. To summarize :

- our work is formal, so we could prove all our assertions. We are able to guarantee that the generated code will be correct ;
- we showed in [14] that Berson's work contained some mistakes and incomplete proofs. For instance, the killing relations between the nodes have to be carefully chosen, otherwise the computed register saturation would be incorrect. This important aspect wasn't covered in [1], so its proof of the NP-completeness of computing RS was incomplete.
- our work is an extension to VLIW and EPIC codes, with multiple register types ;
- our experiments show that our heuristics are nearly optimal, while Berson's heuristics wasn't proved so.

Pinter's work [12] has some relation to ours, because she computes the maximal interference graph (with graph coloring) for all possible schedules of the basic blocks. However, first, her technique works only for superscalar codes, and does not fit neither for VLIW nor EPIC codes. Second, her model does not take into account multiple register types. Third and last, she didn't prove that her method is close to the optimal.

7 Conclusion

In this paper, we continue our early work about the register saturation (RS) notion to manage register pressure and to avoid spill code before scheduling and register allocation steps. We believe that register constraints must be taken into account before ILP scheduling, but by using the RS concept and not by the existing minimization strategies. Otherwise, the subsequent ILP scheduler would be restricted even if enough registers exist.

Computing the register saturation of a DAG is NP-complete. An intLP exact formulation is presented. Our formal mathematical modeling and theoretical study in [14] enables us to give nearly optimal heuristics. In the presence of branches, global RS of an acyclic CFG is brought back to RS in DAGs (basic blocs) by

inserting entry and exit values with the corresponding flow arcs (see [15]).

If RS exceeds the number of available registers, we must reduce it while minimizing the increase of critical path. This is an NP-hard problem. An optimal exact RS reduction method based on integer programming is presented. If we assume writing offsets (VLIW and EPIC codes), some optimal solutions may require to insert non-positive circuits in the original DAG. These circuits may prevent the extended DDG from being scheduled in the presence of resource constraints. A sufficient and necessary condition to overcome this problem is to guarantee the existence of a topological sort for the extended graph. This is done by adding new constraints to the intLP formulation. We have also proved that our initial algorithmic heuristics for RS reduction is very efficient compared to the optimal solutions.

An important problem (let for a future work) is the minimal spill code insertion in data dependence graphs. The existing studies insert spill operations either in sequential codes (regardless on FUs usage), or by iterating ILP scheduling followed by spilling. We think that this problem must be taken into account at the data dependence graph level in order to break this iterative problem.

References

[1] David A. Berson. *Unification of Register Allocation and Instruction Scheduling in Compilers for Fine-Grain Parallel Architecture*. PhD thesis, Pittsburgh University, 1996.

[2] Thomas S. Brasier. FRIGG: A New Approach to Combining Register Assignment and Instruction Scheduling. Master thesis, Michigan Technological University, 1994.

[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] 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.

[5] S. M. Freudenberger and J. C. Ruttenberg. Phase Ordering of Register Allocation and Instruction Scheduling. In *Code Generation – Concepts, Tools, Techniques. Proceedings of the International Workshop on Code Generation*, pages 146–172, London, 1992. Springer-Verlag.

[6] R. Govindarajan, H. Yang, C. Zhang, J. N. Amaral, and G. R. Gao. Minimum Register Instruction Sequence Problem: Revisiting Optimal Code Generation for DAGs. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS-01)*, pages 26–26, Los Alamitos, CA, April 23–27 2001. IEEE Computer Society.

[7] William Jalby and Christophe Lemuet. WBTK: A New Set of Microbenchmarks to Explore Memory System Performance. In *Los Alamos Computer Science Institute (LACSI) Symposium*, October 2002.

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

[9] D. Kaestner and M. Langenbach. Code Optimization by Integer Linear Programming. *Lecture Notes in Computer Science*, 1575:122–136, 1999.

[10] Waleed M. Meleis. Dural-Issue Scheduling for Binary Trees with Spills and Pipelined Loads. *SIAM J. Comput.*, 30(6):1921–1941, March 2001.

[11] Cindy Norris and Lori L. Pollock. A Scheduler-Sensitive Global Register Allocator. In IEEE, editor, *Supercomputing 93 Proceedings: Portland, Oregon*, pages 804–813, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, November 1993. IEEE Computer Society Press.

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

[13] Raúl Silvera, Jian Wang, Guang R. Gao, and R. Govindarajan. A Register Pressure Sensitive Instruction Scheduler for Dynamic Issue Processors. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques (PACT-97)*, pages 78–89, San Francisco, California, November 1997. IEEE Computer Society Press.

[14] Sid-Ahmed-Ali Touati. Register Saturation in Superscalar and VLIW Codes. In *Proceedings of The International Conference on Compiler Construction*, Lecture Notes in Computer Science. Springer-Verlag, April 2001.

[15] 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.