Register Saturation in Superscalar and VLIW Codes

Sid Touati

Register Saturation in Superscalar and VLIW Codes

Sid-Ahmed-Ali TOUATI

INRIA, Domaine de Voluceau, BP 105. 78153, Le Chesnay cedex, France
Sid-Ahmed-Ali.Touati@inria.fr

Abstract. The registers constraints can be taken into account during the scheduling phase of an acyclic data dependence graph (DAG): any schedule must minimize the register requirement. In this work, we mathematically study and extend the approach which consists of computing the exact upper-bound of the register need for all the valid schedules, independently of the functional unit constraints. A previous work (URSA) was presented in [5, 4]. Its aim was to add some serial arcs to the original DAG such that the worst register need does not exceed the number of available registers. We write an appropriate mathematical formalism for this problem and extend the DAG model to take into account delayed read from and write into registers with multiple registers types. This formulation permits us to provide in this paper better heuristics and strategies (nearly optimal), and we prove that the URSA technique is not sufficient to compute the maximal register requirement, even if its solution is optimal.

1 Introduction and Motivation

In Instruction Level Parallelism (ILP) compilers, code scheduling and register allocation are two major tasks for code optimization. Code scheduling consists of maximizing the exploitation of the ILP offered by the code. One factor that inhibits such use is the registers constraints. A limited number of registers prohibits an unbounded number of values simultaneously alive. If the register allocation is carried out before scheduling, false dependencies are introduced because of the registers reuse, producing a negative impact on the available ILP exposed to the scheduler. If scheduling is carried out before, spill code might be introduced because of an insufficient number of registers. A better approach is to make code scheduling and register allocation interact with each other in a complex combined pass, making the register allocation and the scheduling heuristics very correlated.

In this article, we present our contribution to avoiding an excessive number of values simultaneously alive for all the valid schedules of a DAG, previously studied in [4, 5]. 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. We provide better heuristics to compute and reduce it if it exceeds the number

of available registers by introducing new arcs. Experimental results show that in most cases our strategies are nearly optimal.

This article is organized as follows. Section 2 presents our DAG model which can be used for both superscalar and VLIW processors (UAL and NUAL semantics [15]). The RS problem is theoretically studied in Sect. 3. If it exceeds the number of available registers, a heuristic for reducing it is given in Sect. 4. We have implemented software tools, and experimental results are described in Sect. 5. Some related work in this field is given in Sect. 6. We conclude with our remarks and perspectives in Sect. 7.

2 DAG Model

A DAG $G = (V, E, \delta)$ in our study represents the data dependences between the operations and any other serial constraints. Each operation u has a strictly positive latency $lat(u)$. The DAG is defined by its set of operations V, its set of arcs $E = \{(u, v)/\ u, v \in V\}$, and δ such that $\delta(e)$ is the latency of the arc e in terms of processor clock cycles.

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

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

We note by $\Sigma(G)$ the set of *all* the valid schedules of G. Since writing to and reading from registers could be delayed from the beginning of the operation schedule time (VLIW case), we define the two delay functions δ_r and δ_w such that $\delta_w(u)$ is the write cycle of the operation u, and $\delta_r(u)$ is the read cycle of u. In other words, u reads from the register file at instant $\sigma(u) + \delta_r(u)$, and writes in it at instant $\sigma(u) + \delta_w(u)$.

To simplify the writing of some mathematical formulas, we assume that the DAG has one source ($\top$) and one sink ($\bot$). If not, we introduce two fictitious nodes ($\top, \bot$) representing nops (evicted at the end of the RS analysis). We add a virtual serial arc $e_1 = (\top, s)$ to each source with $\delta(e_1) = 0$, and an arc $e_2 = (t, \bot)$ from each sink with the latency of the sink operation $\delta(e_2) = lat(t)$. The total schedule time of a schedule is then $\sigma(\bot)$. The null latency of an added arc e_1 is not inconsistent with our assumption that latencies must be strictly positive because the added virtual serial arcs no longer represent data dependencies. Furthermore, we can avoid introducing these virtual nodes without any consequence on our theoretical study since their purpose is only to simplify some mathematical expressions.

When studying the register need in a DAG, we make a difference between the nodes, depending on whether they define a value to be stored in a register or not, and also depending on which register type we are focusing on (int, float, etc.). We also make a difference between edges, depending on whether they are flow dependencies through the registers of the type considered :

- $V_R \subseteq V$ is the subset of operations which define a value of the type under consideration (int, float, etc.), we simply call them *values*. We assume that

at most one value of the type considered can be defined by an operation. The operations which define multiple values are taken into account if they define at most one value of the type considered.
- $E_R \subseteq E$ is the subset of arcs representing true dependencies through a value of the type considered. We call them *flow* arcs.
- $E_S = E - E_R$ are called *serial* arcs.

Figure 1.b gives the DAG that we use in this paper constructed from the code of part (a). In this example, we focus on the floating point registers: the values and flow arcs are shown by bold lines. We assume for instance that each read occurs exactly at the schedule time and each write at the final execution step ($\delta_r(u) = 0$, $\delta_w(u) = lat(u) - 1$).

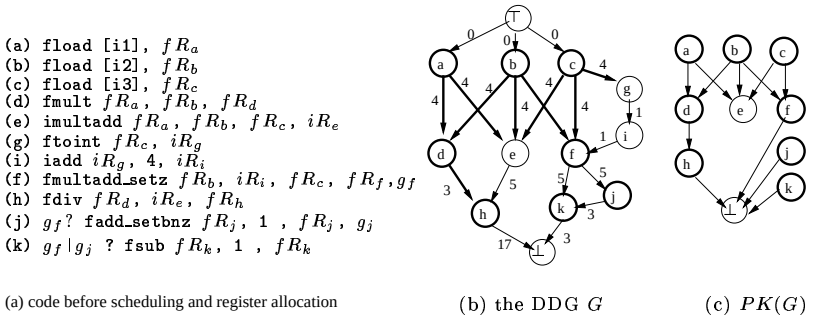


Fig. 1. DAG model

Notation and Definitions on DAGs

In this paper, we use the following notations for a given DAG $G = (V, E)$:

- $\Gamma_G^+(u) = \{v \in V / (u,v) \in E\}$ successors of u;
- $\Gamma_G^-(u) = \{v \in V / (v,u) \in E\}$ predecessors of u;
- $\forall e = (u,v) \in E \quad source(e) = u \wedge target(e) = v$. u, v are called *endpoints*;
- $\forall u, v \in V : \; u < v \iff \exists$ a path $(u, \ldots, v)$ in G;
- $\forall u, v \in V : \; u || v \iff \neg(u < v) \wedge \neg(v < u)$. u and v are said to be *parallel*;
- $\forall u \in V \quad \uparrow u = \{v \in V / v = u \vee v < u\}$ u's ascendants including u;
- $\forall u \in V \quad \downarrow u = \{v \in V / v = u \vee u < v\}$ u's descendants including u.
- two arcs e, e' are *adjacent iff* they share an endpoint;
- $A \subseteq V$ is an antichain in $G \iff \forall u, v \in A \quad u || v$;
- AM is a *maximal* antichain $\iff \forall A$ antichain in $G \quad |A| \leq |AM|$;
- the *extended* DAG $G\backslash^{E'}$ of G generated by the arcs set $E' \subseteq V^2$ is the graph obtained from G after adding the arcs in E'. As a consequence, any valid schedule of G' is necessarily a valid schedule for G:

$$G' = G\backslash^{E'} \Longrightarrow \Sigma(G') \subseteq \Sigma(G)$$

– let $I_1 = [a_1, b_1] \subset \mathbb{N}$ and $I_2 = [a_2, b_2] \subset \mathbb{N}$ be two integer intervals. We say that I_1 is before I_2, noted by $I_1 \prec I_2$, *iff* $b_1 < a_2$.

3 Register Saturation

3.1 Register Need of a Schedule

Given a DAG $G = (V, E, \delta)$, a value $u \in V_R$ is alive just after the writing clock cycle of u until its last reading (consumption). The values which are not read in G or are still alive when exiting the DAG must be kept in registers. We handle these values by considering that the bottom node $\perp$ consumes them. We define the set of consumers for each value $u \in V_R$ as

$$Cons(u) = \begin{cases} \{v \in V / (u,v) \in E_R\} & \text{if } \exists (u,v) \in E_R \\ \perp & \text{otherwise} \end{cases}$$

Given a schedule $\sigma \in \Sigma(G)$, the last consumption of a value is called the killing date and noted :

$$\forall u \in V_R \quad kill_\sigma(u) = \max_{v \in Cons(u)} \left(\sigma(v) + \delta_r(v)\right)$$

All the consumers of u whose reading time is equal to the killing date of u are called the killers of u. We assume that a value written at instant t in a register is available one step later. That is to say, if operation u reads from a register at instant t while 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. Then, the *life interval* L_u^σ of a value u according to σ is $]\sigma(u) + \delta_w(u), kill_\sigma(u)]$.

Given the life intervals of all the values, the register need of σ is the maximum number of values simultaneously alive :

$$RN_\sigma(G) = \max_{0 \leq i \leq \sigma(\perp)} |vsa_\sigma(i)|$$

where $vsa_\sigma(i) = \{u \in V_R / i \in L_u^\sigma\}$ is the set of values alive at time i

Building a register allocation (assign a physical register to each value) with R available registers for a schedule which needs R registers can be easily done with a polynomial-complexity algorithm without introducing spill code nor increasing the total schedule time [16].

3.2 Register Saturation Problem

The RS is the maximal register need for all the valid schedules of the DAG :

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

We call σ a *saturating schedule iff* $RN_\sigma(G) = RS(G)$. In this section, we study how to compute $RS(G)$. We will see that this problem comes down to

answering the question "*which operation must kill this value ?*" When looking for saturating schedules, we do not worry about the total schedule time. Our aim is only to prove that the register need can reach the RS but cannot exceed it. Minimizing the total schedule time is considered in Sect. 4 when we reduce the RS. Furthermore, for the purpose of building saturating schedules, we have proven in [16] that to maximize the register need, looking for only one suitable killer of a value is sufficient rather than looking for a group of killers: for any schedule that assigns more than one killer for a value, we can obviously build another schedule with at least the same register need such that this value is killed by only one consumer. So, the purpose of this section is to select a suitable killer for each value to saturate the register requirement.

Since we do not assume any schedule, the life intervals are not defined so we cannot know at which date a value is killed. However, we can deduce which consumers in $Cons(u)$ are impossible killers for the value u. If $v_1, v_2 \in Cons(u)$ and $\exists$ a path $(v_1 \cdots v_2)$, v_1 is always scheduled before v_2 with at least $lat(v_1)$ processor cycles. Then v_1 can never be the last read of u (remember that we assume strictly positive latencies). We can consequently deduce which consumers can "potentially" kill a value (possible killers). We note $pkill_G(u)$ the set of the operations which can kill a value $u \in V_R$:

$$pkill_G(u) = \left\{ v \in Cons(u) / \downarrow v \cap Cons(u) = \{v\} \right\}$$

One can check that all operations in $pkill_G(u)$ are parallel in G. Any operation which does not belong to $pkill_G(u)$ can never kill the value u.

Lemma 1. *Given a DAG $G = (V, E, \delta)$, then $\forall u \in V_R$*

$$\forall \sigma \in \Sigma(G) \quad \exists v \in pkill_G(u): \quad \sigma(v) + \delta_r(v) = kill_\sigma(u) \tag{1}$$

$$\forall v \in pkill_G(u) \quad \exists \sigma \in \Sigma(G): \quad kill_\sigma(u) = \sigma(v) + \delta_r(v) \tag{2}$$

Proof. A complete proof is given in [17], page 13.

A *potential killing DAG* of G, noted $PK(G) = (V, E_{PK})$, is built to model the potential killing relations between operations, (see Fig. 1.c), where:

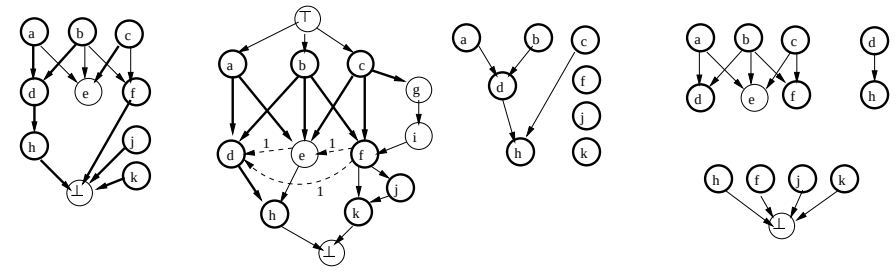
$$E_{PK} = \{(u, v) / \ u \in V_R \wedge v \in pkill_G(u)\}$$

There may be more than one operation candidate for killing a value. Let us begin by assuming a *killing function* which enforces an operation $v \in pkill_G(u)$ to be the killer of $u \in V_R$. If we assume that $k(u)$ is the unique killer of $u \in V_R$, we must always verify the following assertion:

$$\forall v \in pkill_G(u) - \{k(u)\} \quad \sigma(v) + \delta_r(v) < \sigma\big(k(u)\big) + \delta_r\big(k(u)\big) \tag{3}$$

There is a family of schedules which ensures this assertion. To define them, we extend G by new serial arcs that enforce all the potential killing operations of each value u to be scheduled before $k(u)$. This leads us to define an extended DAG associated to k noted $G_{\rightarrow k} = G \backslash^{E_k}$ where:

$$E_k = \left\{ e = (v, k(u)) / u \in V_R \ v \in pkill_G(u) - \{k(u)\} \text{ with } \delta(e) = \delta_r(v) - \delta_r\big(k(u)\big) + 1 \right\}$$



(a) $PK(G)$ with k (b) $G_{\rightarrow k}$ (c) $DV_k(G)$ (d) $\mathcal{B}(G)$

Fig. 2. Valid killing function and bipartite decomposition

Then, any schedule $\sigma \in \Sigma(G_{\rightarrow k})$ ensures Property 3. The condition of the existence of such a schedule defines the condition of a *valid killing function*:

$$k \text{ is a valid killing function} \iff G_{\rightarrow k} \text{ is acyclic}$$

Figure 2 gives an example of a valid killing function k. This function is shown by bold arcs in part (a), where each target kills its sources. Part (b) is the DAG associated to k.

Provided a valid killing function k, we can deduce the values which can never be simultaneously alive for any $\sigma \in \Sigma(G_{\rightarrow k})$. Let $\downarrow_R (u) = \downarrow u \cap V_R$ be the set of the descendant values of $u \in V$.

Lemma 2. *Given a DAG $G = (V, E, \delta)$ and a valid killing function, then:*

1. *the descendant values of $k(u)$ cannot be simultaneously alive with u:*

$$\forall u \in V_R \ \forall \sigma \in \Sigma(G_{\rightarrow k}) \ \forall v \in \downarrow_R k(u) \quad L^u_\sigma \prec L^v_\sigma \tag{4}$$

2. *there exists a valid schedule which makes the other values non descendant of $k(u)$ simultaneously alive with u, i.e. $\forall u \in V_R \ \exists \sigma \in \Sigma(G_{\rightarrow k})$:*

$$\forall v \in \left(\bigcup_{v' \in pkill_G(u)} \downarrow_R v' \right) - \downarrow_R k(u) \quad L^u_\sigma \cap L^v_\sigma \neq \phi \tag{5}$$

Proof. A complete proof is given in [17], page 23.

We define a DAG which models the values that can never be simultaneously alive according to k. The *disjoint value DAG* of G associated to k, and noted $DV_k(G) = (V_R, E_{DV})$ is defined by:

$$E_{DV} = \left\{ (u, v) / u, v \in V_R \ \wedge \ v \in \downarrow_R k(u) \right\}$$

Any arc (u, v) in $DV_k(G)$ means that u's life interval is always before v's life interval according to any schedule of $G_{\rightarrow k}$, see part Fig. 2.c[1]. This definition permits us to state in the following theorem that the register need of any schedule of $G_{\rightarrow k}$ is always less than or equal to a maximal antichain in $DV_k(G)$. Also, there is always a schedule which makes all the values in this maximal antichain simultaneously alive.

Theorem 1. *Given a DAG $G = (V, E, \delta)$ and a valid killing function k then:*

- $\forall \sigma \in \Sigma(G_{\rightarrow k}) : RN_\sigma(G) \leq |AM_k|$
- $\exists \sigma \in \Sigma(G_{\rightarrow k}) : RN_\sigma(G) = |AM_k|$

where AM_k is a maximal antichain in $DV_k(G)$

Proof. A complete proof is given in [17], page 18.

Theorem 1 allows us to rewrite the RS formula as

$$RS(G) = \max_{k \text{ a valid killing function}} |AM_k|$$

where AM_k is a maximal antichain in $DV_k(G)$. We refer to the problem of finding such a killing function as the *maximizing maximal antichain* problem (MMA). We call each solution for the MMA problem a *saturating killing function*, and AM_k its *saturating values*. Unfortunately, we have proven in [17] (page 24) that finding a saturating killing function is NP-complete.

3.3 A Heuristic for Computing the RS

This section presents our heuristics to approximate an optimal k by another valid killing function k^*. We have to choose a killing operation for each value such that we maximize the parallel values in $DV_k(G)$. Our heuristics focus on the potential killing DAG $PK(G)$, starting from source nodes to sinks. Our aim is to select a group of killing operations for a group of parents to keep as many descendant values alive as possible. The main steps of our heuristics are:

1. decompose the potential killing DAG $PK(G)$ into connected bipartite components;
2. for each bipartite component, search for the best saturating killing set (defined below);
3. choose a killing operation within the saturating killing set (defined below).

We decompose the potential killing DAG into connected bipartite components (CBC) in order to choose a common saturating killing set for a group of parents. Our purpose is to have a maximum number of children and their descendants values simultaneously alive with their parents values. A CBC $cb = (S_{cb}, T_{cb}, E_{cb})$ is a partition of a subset of operations into two disjoint sets where:

[1] This DAG is simplified by transitive reduction.

- $E_{cb} \subseteq E_{PK}$ is a subset of the potential killing relations;
- $S_{cb} \subseteq V_R$ is the set of the parent values, such that each parent is killed by at least one operation in T_{cb};
- $T_{cb} \subset V$ is the set of the children, such that any operation in T_{cb} can potentially kill at least a value in S_{cb}.

A bipartite decomposition of the potential killing graph $PK(G)$ is the set (see Fig. 2.d)

$$\mathcal{B}(G) = \{cb = (S_{cb}, T_{cb}, E_{cb}) / \ \forall e \in E_{PK} \ \exists cb \in \mathcal{B}(G) : \ e \in E_{cb}\}$$

Note that $\forall cb \in \mathcal{B}(G) \ \forall s, s' \in S_{cb} \ \forall t, t' \in T_{cb} \quad s||s' \wedge t||t'$ in $PK(G)$.
A saturating killing set $SKS(cb)$ of a bipartite component $cb = (S_{cb}, T_{cb}, E_{cb})$ is a subset $T'_{cb} \subseteq T_{cb}$ such that if we choose a killing operation from this subset, then we get a maximal number of descendant values of children in T_{cb} simultaneously alive with parent values in S_{cb}.

Definition 1 (Saturating Killing Set). *Given a DAG $G = (V, E, \delta)$, a saturating killing set $SKS(cb)$ of a connected bipartite component $cb \in \mathcal{B}(G)$ is a subset $T'_{cb} \subseteq T_{cb}$, such that:*

1. *killing constraints:*

$$\bigcup_{t \in T'_{cb}} \Gamma^-_{cb}(t) = S_{cb}$$

2. *minimizing the number of descendant values of T'_{cb}*

$$\min | \bigcup_{t \in T'_{cb}} \downarrow_R t|$$

Unfortunately, computing a SKS is also NP-complete ([17], page 81).

A Heuristic for Finding a SKS Intuitively, we should choose a subset of children in a bipartite component that would kill the greatest number of parents while minimizing the number of descendant values. We define a cost function ρ that enables us to choose the best candidate child. Given a bipartite component $cb = (S_{cb}, T_{cb}, E_{cb})$ and a set Y of (cumulated) descendant values and a set X of non (yet) killed parents, the cost of a child $t \in T_{cb}$ is :

$$\rho_{X,Y}(t) = \begin{cases} \frac{|\Gamma^-_{cb}(t) \cap X|}{|\downarrow_R t \cup Y|} & \text{if } \downarrow_R t \cup Y \neq \phi \\ |\Gamma^-_{cb}(t) \cap X| \text{ otherwise} \end{cases}$$

The first case enables us to select the child which covers the most non killed parents with the minimum descendant values. If there is no descendant value, then we choose the child that covers the most non killed parents.

Algorithm 1 gives a modified greedy heuristic that searches for an approximation SKS^* and computes a killing function k^* in polynomial time. Our heuristic has the following properties.

Algorithm 1 Greedy-k: a heuristics for the MMA problem

Require: a DAG $G = (V, E, \delta)$
  for all values $u \in V_R$ **do**
    $k^*(u) = \perp$ {all values are initially non killed}
  end for
  build $\mathcal{B}(G)$ the bipartite decomposition of $PK(G)$.
  for all bipartite component $cb = (S_{cb}, T_{cb}, E_{cb}) \in \mathcal{B}(G)$ **do**
    $X := S_{cb}$ {all parents are initially uncovered}
    $Y := \phi$ {initially, no cumulated descendant values}
    $SKS^*(cb) := \phi$
    while $X \neq \phi$ **do** {build the SKS for cb}
      select the child $t \in T_{cb}$ with the maximal cost $\rho_{X,Y}(t)$
      $SKS^*(cb) := SKS^*(cb) \cup \{t\}$
      $X := X - \Gamma^-_{cb}(t)$ {remove covered parents}
      $Y := Y \cup \downarrow_R t$ {update the cumulated descendent values}
    end while
    for all $t \in SKS^*(cb)$ **do** {in decreasing cost order}
      for all parent $s \in \Gamma^-_{cb}(t)$ **do**
        if $k^*(s) = \perp$ **then** {kill non killed parents of t}
          $k^*(s) := t$
        end if
      end for
    end for
  end for

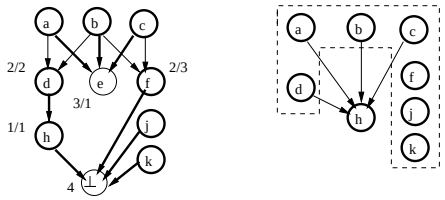
Theorem 2. *Given a DAG $G = (V, E, \delta)$, then:*

1. *Greedy-k always produces a valid killing function k^* ;*
2. *PK(G) is an inverted tree $\Longrightarrow$ Greedy-k is optimal.*

Proof. Complete proofs for both (1) and (2) are given in [17], pages 31 and 44 resp.

Since the approximated killing function k^* is valid, Theorem 1 ensures that we can always find a valid schedule which requires exactly $|AM_{k^*}|$ registers. As consequence, our heuristic does not compute an upper bound of the optimal register saturation and then the optimal RS can be greater than the one computed by Greedy-k. A conservative heuristic which computes a solution exceeding the optimal RS cannot ensure the existence of a valid schedule which reaches the computed limit, and hence it would imply an obsolete RS reduction process and a waste of registers. The validity of a killing function is a key condition because it ensures that there exists a register allocation with exactly $|AM_{k^*}|$ registers. As summary, here are our steps to compute the RS :

1. apply Greedy-k on G. The result is a valid killing function k^* ;
2. construct the disjoint value DAG $DV_{k^*}(G)$;
3. find a maximal antichain AM_{k^*} of $DV_{k^*}(G)$ using Dilworth decomposition [10] ; Saturating values are then AM_{k^*} and $RS^*(G) = |AM_{k^*}| \leq RS(G)$.



(a) $PK(G)$ with k^* (b) $DV_{k^*}(G)$

Fig. 3. Example of computing the register saturation

Figure 3.a shows a saturating killing function k^* computed by Greedy-k: bold arcs denote that each target kills its sources. Each killer is labeled by its cost ρ. Part (b) gives the disjoint value DAG associated to k^*. The Saturating values are $\{a, b, c, d, f, j, k\}$, so the RS is 7.

4 Reducing the Register Saturation

In this section we build an extended DAG $\overline{G} = G\backslash^{\overline{E}}$ such that the RS is limited by a strictly positive integer (number of available registers) with the respect of the critical path. Let $\mathcal{R}$ be this limit. Then :

$$\forall \sigma \in \Sigma(\overline{G}) : RN_\sigma(\overline{G}) \leq RS(\overline{G}) \leq \mathcal{R}$$

We have proven in [16] that this problem is NP-hard. In this section we present a heuristic that adds serial arcs to prevent some saturating values in AM_k (according to a saturating killing function k) from being simultaneously alive for any schedule. Also, we must care to not increase the critical path if possible.

Serializing two values $u, v \in V_R$ means that the kill of u must always be carried out before the definition of v, or vice-versa. A value serialization $u \rightarrow v$ for two values $u, v \in V_R$ is defined by :

- if $v \in pkill_G(u)$ then add the serial arcs $\big\{e = (v', v)/$

$$v' \in pkill_G(u) - \{v\} \quad \text{with } \delta(e) = \delta_r(v') - \delta_w(v)\big\}$$

- else add the serial arcs $\big\{e = (u', v)/$

$$u' \in pkill_G(u) \wedge \neg(v < u') \quad \text{with } \delta(e) = \delta_r(u') - \delta_w(v)\big\}$$

To do not violate the DAG property (we must not introduce a cycle), some serializations must be filtered out. The condition for applying $u \rightarrow v$ is that $\forall v' \in pkill_G(u) : \quad \neg(v < v')$. We chose the best serialization within the set of all the possible ones by using a cost function $\omega(u \rightarrow v) = (\omega_1, \omega_2)$, such that :

- $\omega_1 = \mu_1 - \mu_2$ is the prediction of the reduction obtained within the saturating values if we carry out this value serialization, where

- μ_1 is the number of saturating values serialized after u if we carry out the serialization ;
- μ_2 is the predicted number of u's descendant values that can become simultaneously alive with u ;

– ω_2 is the increase in the critical path.

Our heuristic is described in Algorithm 2. It iterates the value serializations within the saturating values until we get the limit $\mathcal{R}$ or until no more serializations are possible (or none is expected to reduce the RS). One can check that if there is no possible value serialization in the original DAG, our algorithm exits at the first iteration of the outer while-loop. If it succeeds, then any schedule of $\overline{G}$ needs at most $\mathcal{R}$ registers. If not, it still decreases the original RS, and thus limits the register need. Introducing and minimizing the spill code is another NP-complete problem studied in [8, 3, 2, 9, 14] and not addressed in this work.

Now, we explain how to compute the prediction parameters μ_1, μ_2, ω_2. We note $\overline{G_i}$ the extended DAG of step i, k_i its saturating function, and AM_{k_i} its saturating values and $\downarrow_{R_i} u$ the descendant values of u in $\overline{G_i}$:

1. $(u \rightarrow v)$ ensures that $k_{i+1}(u) < v$ in $\overline{G_{i+1}}$. According to Lemma 2, $\mu_1 = |\downarrow_{R_i} v \cap AM_{k_i}|$ is the number of saturating values in $\overline{G_i}$ which cannot be simultaneously alive with u in $\overline{G_{i+1}}$;
2. new saturating values could be introduced into $\overline{G_{i+1}}$: if $v \in pkill_{\overline{G_i}}(u)$, we force $k_{i+1}(u) = v$. According to Lemma 2,

$$\mu_2 = \left| \left(\bigcup_{v' \in pkill_{\overline{G_i}}(u)} \downarrow_{R_i} v' \right) - \downarrow_{R_i} v \right|$$

   is the number of values which could be simultaneously alive with u in $\overline{G_{i+1}}$. $\mu_2 = 0$ otherwise ;
3. if we carry out $(u \rightarrow v)$ in $\overline{G_i}$, the introduced serial arcs could enlarge the critical path. Let $lp_i(v', v)$ be the longest path going from v' to v in $\overline{G_i}$. The new longest path in $\overline{G_{i+1}}$ going through the serialized nodes is :

$$\max_{\substack{\text{introduced } e=(v',v) \\ \delta(e) > lp_i(v',v)}} lp_i(\top, v') + lp_i(v, \bot) + \delta(e)$$

   If this path is greater than the critical path in $\overline{G_i}$, then ω_2 is the difference between them, 0 otherwise.

At the end of the algorithm, we apply a general verification step to ensure the potential killing property proven in Lemma 1 for the original DAG. We have proven in Lemma 1 that the operations which do not belong to $pkill_G(u)$ cannot kill the value u. After adding the serial arcs to build $\overline{G}$, we might violate this assertion because we introduce some arcs with negative latencies. To overcome this problem, we must guarantee the following assertion : $\forall u \in V_R,\ \forall v' \in Cons(u) - pkill_{\overline{G}}(u)$

$$\exists v \in pkill_{\overline{G}}(u) / v' < v \text{ in } \overline{G} \Longrightarrow lp_{\overline{G}}(v', v) > \delta_r(v') - \delta_r(v) \qquad (6)$$

Algorithm 2 Value Serialization Heuristic

Require: a DAG $G = (V, E, \delta)$ and a strictly positive integer $\mathcal{R}$
  $\overline{G} := G$
  compute AM_k, saturating values of $\overline{G}$;
  while $|AM_k| > \mathcal{R}$ **do**
    construct the set U_k of all admissible serializations between saturating values in AM_k with their costs (ω_1, ω_2);
    if $\nexists (u \rightarrow v) \in U / \omega_1(u \rightarrow v) > 0$ **then** {no more possible RS reduction}
      exit;
    end if
    $X := \{(u \rightarrow v) \in U / \omega_2(u \rightarrow v) = 0\}$ {the set of value serializations that do not increase the critical path}
    if $X \neq \phi$ **then**
      choose a value serialization $(u \rightarrow v)$ in X with the minimum cost $\mathcal{R} - \omega_1$;
    else
      choose a value serialization $(u \rightarrow v)$ in X with the minimum cost ω_2;
    end if
    carry out the serialization $(u \rightarrow v)$ in $\overline{G}$;
    compute the new saturating values AM_k of $\overline{G}$;
  end while
  ensure potential killing operations property {check longest paths between pkill operations}

In fact, this problem occurs if we create a path in $\overline{G}$ from v' to v where $v, v' \in pkill_G(u)$. If assertion (6) is not verified, we add a serial arc $e = (v', v)$ with $\delta(e) = \delta_r(v') - \delta_r(v) + 1$ as illustrated in Fig. 4.

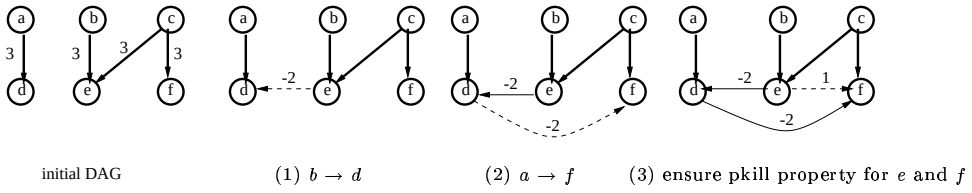


Fig. 4. Check Potential Killers Property

Example 1. Figure 5 gives an example for reducing the RS of our DAG from 7 to 4 registers. We remind that the saturating values of G are $AM_k = \{a, b, c, d, f, j, k\}$. Part (a) shows all the possible value serializations within these saturating values. Our heuristic selects $a \rightarrow f$ as a candidate, since it is expected to eliminate 3 saturating values without increasing the critical path. The maximal introduced longest path through this serialization is $(\top, a, d, f, k, \bot) = 8$, which is less than the original critical path (26). The extended DAG $\overline{G}$ is presented in part (b) where the value serialization $a \rightarrow f$ is introduced: we add the the serial arcs

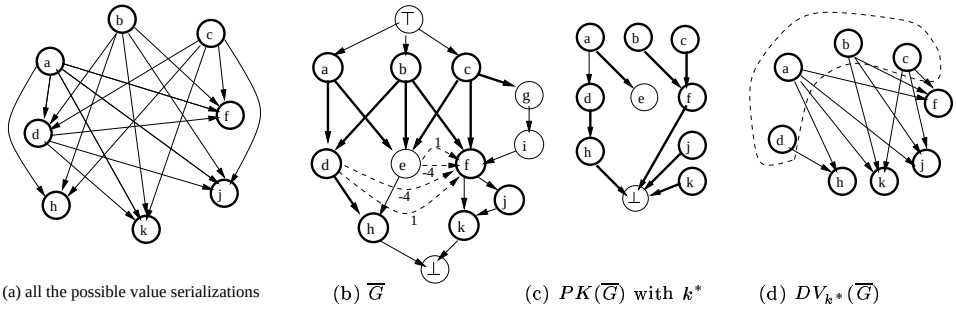


Fig. 5. Reducing register saturation

(e, f) and (d, f) with a -4 latency. Finally, we add the serial arcs (e, f) and (d, f) with a unit latency to ensure the $pkill_{\overline{G}}(b)$ property. The whole critical path does not increase and RS is reduced to 4. Part (c) gives a saturating killing function for $\overline{G}$, shown with bold arcs in $PK(\overline{G})$. $DV_k(\overline{G})$ is presented in part (d) to show that the new RS becomes 4 floating point registers.

5 Experimentation

We have implemented the RS analysis using the LEDA framework. We carried out our experiments on various floating point numerical loops taken from various benchmarks (livermore, whetsone, spec-fp, etc.). We focus in these codes on the floating point registers. The first experimentation is devoted to checking Greedy-k efficiency. For this purpose, we have defined and implemented in [16] an integer linear programming model to compute the optimal RS of a DAG. We use CPLEX to resolve these linear programming models. The total number of experimented DAGs is 180, where the number of nodes goes up to 120 and the number of values goes up to 114. Experimental results show that our heuristics give quasi-optimal solutions. The worst experimental error is 1, which means that the optimal RS is in worst case greater by one register than the one computed by Greedy-k.

The second experimentation is devoted to checking the efficiency of the value serialization heuristics in order to reduce the RS. We have also defined and implemented in [16] an integer linear programming model to compute the optimal reduction of the RS with a minimum critical path increase (NP-hard problem). The total number of experimented DAGs is 144, where the number of nodes goes up to 80 and the number of values goes up to 76. In almost all cases, our heuristics manages to get the optimal solutions. Optimal reduced RS was in the worst cases less by one register than our heuristics results. Since RS computation in the value serialization heuristics is done by Greedy-k, we add its worst experimental error (1 register) which leads to a total maximal error of two registers. All optimal vs. approximated results are fully detailed in [16].

Since our strategies result in a good efficiency, we use them to study the RS behavior in DAGs. Experimentation on only loop bodies shows that the RS is low, ranging from 1 to 8. We have unrolled these loops with different unrolling factors going up to 20 times. The aim of such unrolling is to get large DAGs, increase the registers pressure and expose more ILP to hide memory latencies. We carried out a wide range of of experiments to study the RS and its reduction with with various limits of available registers (going from 1 up to 64). We experimented 720 DAGs where the number of of nodes goes up to 400 and the number of values goes up to 380. Full results are detailed in [17, 16].

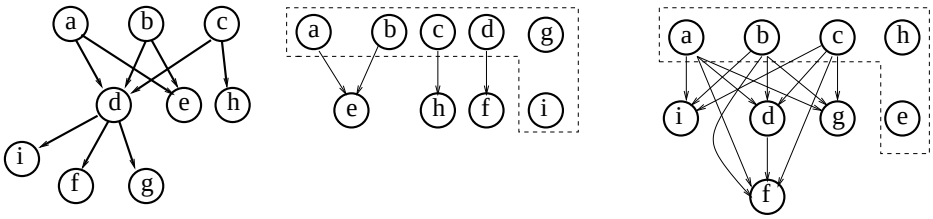
The first remark deduced from our full experiments is that the RS is lower than the number of available registers in a lot of cases. The RS analysis makes it possible to avoid the registers constraints in code scheduling: most of these codes can be scheduled without any interaction with the register allocation, which decreases the compile-time complexity. Second, in most cases our heuristics succeeds in reducing it until reaching the targeted limit. In a few cases we lose some ILP because of the intrinsic register pressure of the DAGs: but since spill code decreases the performance dramatically because of the memory access latencies, a tradeoff between spilling and increasing the overall schedule time can be done in few critical cases. Finally, in the cases where the RS is lower than the number of available registers, we can use the extra non used registers by assigning to them some global variables and array elements with the guarantee that no spill code could be introduced after by the scheduler and the register allocator.

6 Related Work and Discussion

Combining code scheduling and register allocation in DAGs was studied in many works. All the techniques described in [11, 6, 13, 7, 12] used their heuristics to build an optimized schedule without exceeding a certain limit of values simultaneously alive. The dual notion of the RS, called the register sufficiency, was studied in [1]. Given a DAG, the authors gave a heuristic which found the minimum register need; the computation was $O(log^2|V|)$ factor of the optimal. Note that we can easily use the RS reduction to compute the register sufficiency. This is done in practice by setting $\mathcal{R} = 1$ as the targeted limit for the RS reduction.

Our work is an extension to URSA [4, 5]. The minimum killing set technique tried to saturate the register requirement in a DAG by keeping the values alive as late as possible: the authors proceeded by keeping as many children alive as possible in a bipartite component by computing the minimum set which killed all the parent's values. First, since the authors did not formalize the RS problem, we can easily give examples to show that a minimum killing set does not saturate the register need, even if the solution is optimal [17]. Figure. 6 shows an example where the RS computed by our heuristics (Part b) is 6 where the optimal solution for URSA yields a RS of 5 (part c). This is because URSA did not take into account the descendant values while computing the killing sets. Second, the validity of the killing functions is an important condition to compute the RS

and unfortunately was not included in URSA. We have proven in [17] that non valid killing functions can exist if no care is taken. Finally, the URSA DAG model did not differentiate between the types of the values and did not take into account delays in reads from and writes into the registers file.



(a) original DAG

(b) $DV_k(G)$ such $\{e, h\}$ is the saturating killing set

(c) $DV_k(G)$ such $\{d\}$ is the optimal minimum killing set

Fig. 6. URSA drawback

7 Conclusion and Future Work

In our work, we mathematically study and define the RS notion to manage the registers pressure and avoid spill code before the scheduling and register allocation passes. We extend URSA by taking into account the operations in both Unit and Non Unit Assumed Latencies (UAL and NUAL [15]) semantics with different types (values and non values) and values (float, integer, etc.). The formal mathematical modeling and theoretical study permit us to give nearly optimal strategies and prove that the minimum killing set is insufficient to compute the RS. Experimentations show that the registers constraints can be obsolete in many codes, and may therefore be ignored in order to simplify the scheduling process. The heuristics we use manage to reduce the RS in most cases while some ILP is lost in few DAGs. We think that reducing the RS is better than minimizing the register need: this is because minimizing the register need increases the register reuse, and the ILP loss must increase as consequence. Our DAG model is sufficiently general to meet all current architecture properties (RISC or CISC), except for some architectures which support issuing dependent instructions at the same clock cycle, which would require representation using null latency. Strictly positive latencies are assumed to prove the pkill operation property (Lemma 1) which is important to build our heuristics. We think that this restriction should not be a major drawback nor an important factor in performance degradation, since null latency operations do not generally contribute to the critical execution paths. In the future, we will extend our work to loops. We will study how to compute and reduce the RS in the case of cyclic schedules like software pipelining (SWP) where the life intervals become circular.

References

1. A. Agrawal, P. Klein, and R. Ravi. Ordering Problems Approximated : Register Sufficiency, Single Processor Scheduling and Interval Graph Completion. internal research report CS-91-18, Brown University, Providence, Rhode Island, Mar. 1991.
2. P. Bergner, P. Dahl, D. Engebretsen, and M. O'Keefe. Spill Code Minimization via Interference Region Spilling. *ACM SIG-PLAN Notices*, 32(5):287–295, May 1997.
3. D. Bernstein, D. Q. Goldin, M. C. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R. Y. Pinter. Spill Code Minimization Techniques for Optimizing Compilers. *SIGPLAN Notices*, 24(7):258–263, July 1989. Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.
4. D. Berson, R. Gupta, and M. Soffa. URSA: A unified ReSource allocator for registers and functional units in VLIW architectures. In *Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 243–254, Orlando, Florida, Jan. 1993.
5. D. A. Berson. *Unification of Register Allocation and Instruction Scheduling in Compilers for Fine-Grain Parallel Architecture*. PhD thesis, Pittsburgh University, 1996.
6. D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrating Register Allocation and Instruction Scheduling for RISCs. *ACM SIGPLAN Notices*, 26(4):122–131, Apr. 1991.
7. T. S. Brasier. FRIGG: A New Approach to Combining Register Assignment and Instruction Scheduling. Master thesis, Michigan Technological University, 1994.
8. D. Callahan and B. Koblenz. Register Allocation via Hierarchical Graph Coloring. *SIGPLAN Notices*, 26(6):192–203, June 1991. Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.
9. G. J. Chaitin. Register allocation and spilling via graph coloring. *ACM SIG-PLAN Notices*, 17(6):98–105, June 1982.
10. P. Crawley and R. P. Dilworth. *Algebraic Theory of Lattices*. Prentice Hall, Englewood Cliffs, 1973.
11. J. R. Goodman and W.-C. Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. In *Conference Proceedings 1988 International Conference on Supercomputing*, pages 442–452, St. Malo, France, July 1988.
12. C. Norris and L. 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, Nov. 1993. IEEE Computer Society Press.
13. S. S. Pinter. Register Allocation with Instruction Scheduling: A New Approach. *SIGPLAN Notices*, 28(6):248–257, June 1993.
14. M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, Sept. 1999.
15. Schlansker, B. Rau, and S. Mahlke. Achieving High Levels of instruction-Level Parallelism with Reduced Hardware Complexity. Technical Report HPL-96-120, Hewlet Packard, 1994.
16. S.-A.-A. Touati. Optimal Register Saturation in Acyclic Superscalar and VLIW Codes. Research Report, INRIA, Nov. 2000. ftp.inria.fr/INRIA/Projects/a3/touati/optiRS.ps.gz.
17. S.-A.-A. Touati and F. Thomasset. Register Saturation in Data Dependence Graphs. Research Report RR-3978, INRIA, July 2000. ftp.inria.fr/INRIA/publication/publi-ps-gz/RR/RR-3978.ps.gz.