Register Saturation in Instruction Level Parallelism

Sid-Ahmed-Ali TOUATI
University of Versailles, PRiSM laboratory, France
touati@prism.uvsq.fr

Abstract

The registers constraints are usually taken into account during the scheduling pass of an acyclic data dependence graph (DAG): any schedule of the instructions inside a basic block must bound the register requirement under a certain limit. In this work, we show how to handle the register pressure before the instruction scheduling of a DAG. We mathematically study an approach which consists in managing the exact upper-bound of the register need for all the valid schedules of a considered DAG, independently of the functional unit constraints. We call this computed limit the register saturation (RS) of the DAG. Its aim is to detect possible obsolete register constraints, i.e., when RS does not exceed the number of available registers. If it does, we add some serial edges to the original DAG such that the worst register need does not exceed the number of available registers. We propose an appropriate mathematical formalism for this problem. Our generic processor model takes into account superscalar, VLIW and EPIC/IA64 architectures. Our deeper analysis of the problem and our formal methods enable us to provide nearly optimal heuristics and strategies for register optimization in the face of ILP.

Keywords Register Requirement, Register Pressure, Instruction Level Parallelism, Integer Linear Programming, Optimizing Compilation.

1 Introduction

The introduction of instruction level parallelism (ILP) has rendered the classical techniques of register allocation for sequential code semantics inadequate. In [16], the authors showed that there is a phase ordering problem between classical register allocation techniques and ILP instruction scheduling. If a classical register allocation 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 register allocation, with the condition that the allocator is sensitive to the scheduler, as done in [2, 18, 20, 23, 25, 26].

Some other studies [5, 8, 16, 24, 28] claim that it is better to combine instruction scheduling and register allocation in a single complex pass, arguing that applying each method separately has a negative influence on the efficiency of the other. However, we think that this phase ordering problem arises only if the applied first pass (ILP scheduler or register allocator) is “selfish”. Indeed, we 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 think that our methods provide better techniques than the existing solutions.

The principal reason for handling register constraints before instruction scheduling is that register allocation is more important as an optimization issue than code scheduling. This is because performance is 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 we expect spill code to exhibit high locality, and hence likely produces cache hits, this cannot be asserted at compile time. It is very hard for a compiler to guarantee the existence of data inside a memory hierarchy level. Consequently, it is difficult to really guarantee the latency of a memory operation at compile time. So, the schedule of the instructions computed by the compiler wouldn’t act in harmony with the dynamic execution of

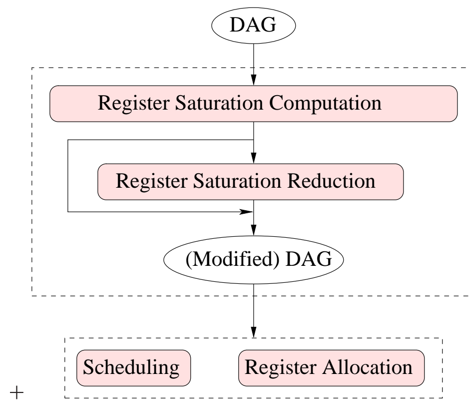


Figure 1. Early Register Pressure Management

the hardware. The authors in [15] relate that about 66% of application execution times are spent satisfying memory requests. Furthermore, memory requests exhibit a high potential for conflicts, even if they are data independent. These conflicts are due to micro-architectural restrictions and simplifications in the memory disambiguation mechanisms (load/store queues) and possible banking structure in cache levels [21]. These possible conflicts may cause severe performance degradation even in the presence of high levels of ILP, and even if the data is located in the cache [22]. Of course, our claim that spill code is more damaging than a weak static ILP extraction is more appropriate for those architectures where memory access latency 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 useless.

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. Unless we generate superscalar codes with sequential semantics, we cannot guarantee in the case of VLIW 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 not enough free registers exist. We cannot guarantee the existence of a valid issue time for these introduced memory accesses in already scheduled code; resource or data dependence constraints may prohibit all possible issue slots inside the scheduled code. This fact forces an iterative process of scheduling followed by spilling until reaching a solution. Even if we can experimentally reduce the backtracking as in [31], this iterative aspect adds a high algorithmic complexity factor to a pass integrating both register allocation and scheduling. As far as we know, there is no formal solution that effectively solves this problem.

The above arguments suggest that we consider new ways of handling register pressure before starting the scheduling process. The scheduler should be freed from register constraints so that the schedule does not suffer from excessive serialization. This article synthesizes our contributions from [29, 30]. We study the concept of register saturation (RS), which prevents a DAG from producing an excessive number of simultaneously live values for all the valid schedules. Our pre-pass analyzes a DAG (with respect to control flow) to deduce the maximum register need among 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 edges in the DAG to reduce RS, as illustrated in Figure 1. In this paper, we give some theoretical results on RS and provide exact (optimal) and approximate methods for the problems of computing and reducing RS. 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). Section 3 provides some theoretical results on computing the RS that prove the NP-completeness of this problem. Section 4 presents an algorithmic heuristics for computing RS. Computing

the optimal RS by integer linear programming (intLP) is given in Section 5. 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. If the RS exceeds the number of available registers, RS must be reduced. Section 6 proves that this problem is NP-hard. An algorithmic heuristics for reducing RS is given in Section 7 and an exact optimal solution is presented in Section 8. Section 9 presents our large range of experiments, which show that our heuristics are nearly optimal in practice. Before concluding, in Section 10 we discuss why the RS concept is a better way to handle register constraints prior to ILP scheduling compared to register minimization. To enhance readability, only the most important formal proofs are presented in this paper. The complete theoretical proofs are provided in the cited references.

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. Each operation u has a strictly positive latency $lat(u)$. The DAG is defined by its set of nodes (operations) V, its set of edges (data dependences and serial constraints) $E = \{(u, v)|\ u, v \in V\}$, and δ such that $\delta(e)$ is the latency of the edge e in terms of processor clock cycles. We assume that the initial DAG contains only edges with positive latencies. This assumption is useful for some formal proofs. However, we will see in later sections (when reducing RS) that we can insert new edges with non-positive latencies.

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

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

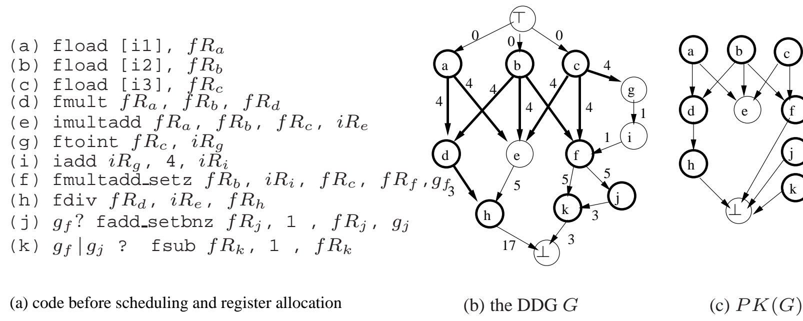
The set of all valid acyclic schedules of G is denoted by $\Sigma(G)$.

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 edge $e_1 = (\top, s)$ to each source with $\delta(e_1) = 0$, and an edge $e_2 = (t, \bot)$ 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 edge e_1 is not inconsistent with our assumption that latencies must be strictly positive because the added virtual serial edges do not exist in the original DAG. Furthermore, we can avoid introducing these virtual nodes without any impact on our theoretical study, since their purpose is only to simplify some mathematical expressions.

We consider a target RISC-style architecture with multiple register types, where $\mathcal{T}$ denotes the set of register types (for instance, $\mathcal{T} = \{int, float\}$). We differentiate between statements and precedence constraints, based on whether they refer to values to be stored in registers or not.

- $V_{R,t} \subseteq V$ is the set of statements (operations) which define values to be stored in registers of type $t \in \mathcal{T}$. We simply call such statements *values*. We assume that each statement $u \in V_{R,t}$ writes into at most one register of a type $t \in \mathcal{T}$. Statements which define multiple values with different types are accepted in our model if they do not define more than one value of a single type. We denote by u^t the value of type t defined by the operation u.
- $E_{R,t} \subseteq E$ is the set of data flow dependence edges through a value of type $t \in \mathcal{T}$. We call them *flow* edges.
- All the edges in $E - E_{R,t}$, *i.e.* edges which are not data flow dependences, are called *serial* edges.

Basically, there are three types of ILP codes : superscalar, VLIW and EPIC. Superscalar codes can be simply considered as linear sequential programs. Even if the compiler try to generate efficient superscalar codes, the processor is the unique responsible for dynamically extracting ILP at execution time. So, code generation for such ILP codes write sequential ones, as if they would be executed by a sequential processor. However, VLIW codes contain information about parallel operations. The compiler has the task of statically extracting ILP and then generating the code by compacting the parallel operations into *Very Long Instructions Words*. The processor executes such instructions (containing many independent operations) sequentially. So, the compiler has the complete control of the dynamic execution of VLIW codes (except dynamic events, such as cache misses, exceptions, etc.). EPIC codes have a semantics that may be considered as a mixture between VLIW and superscalar: while the compiler include information about ILP in the code, the processor can use such ILP information at execution time, or can simply execute sequentially the program. From the compiler point of view, an EPIC processor can be viewed as a sequential (superscalar) or as VLIW processor.



(a) code before scheduling and register allocation (b) the DDG G (c) $PK(G)$

Figure 2. DAG Model

To accommodate static issue VLIW and EPIC/IA64 processors in which the hardware pipeline steps are visible to compilers (we allow for dynamically scheduled superscalar processors as well), we assume that reading from and writing into a register may be delayed from the beginning of the schedule time, and these delays are visible to the compiler (architecturally visible). We define two delay (offset) functions $\delta_{r,t}$ and $\delta_{w,t}$ in which: the read cycle of u^t from a register of type t is $\sigma(u) + \delta_{r,t}(u)$, and the write cycle of u^t into a register of type t is $\sigma(u) + \delta_{w,t}(u)$. By definition, we have $\delta_{r,t}(u) \leq \delta_{w,t}(u) < lat(u)$. For instance, according to superscalar and EPIC/IA64 code semantics, $\delta_{r,t}$ and $\delta_{w,t}$ are equal to zero. This is because, according to the semantics provided by the vendors, such codes can be considered as sequential (linear). Any register written by operation u at time slot c in the code, that register is assumed as busy at the program point c (no delay is architecturally visible). The same remark holds when reading from registers.

Figure 2.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 edges are illustrated by bold lines. We assume for instance that each read occurs exactly at the schedule time and each write at the final execution step ($\delta_r(u) = 0$, $\delta_w(u) = lat(u) - 1$). The nodes with non-bold lines are any other operations that do not write into registers (as stores), or write into registers of unconsidered types. The edges with non-bold lines represent the precedence constraints that are not flow dependences through registers, such as data dependences through memory, or through registers of unconsidered types, or any other serial constraints.

Notation and Definitions on DAGs

In this paper, we use the following notations for a given DAG $G = (V, E)$ (as those usually used in lattices and orders algebra):

- $\Gamma^+_G(u) = \{v \in V | (u, v) \in E\}$ successors of u in the graph G;
- $\Gamma^-_G(u) = \{v \in V | (v, u) \in E\}$ predecessors of u in the graph G;
- $\forall e = (u, v) \in E \quad source(e) = u \wedge target(e) = v$. u, v are called *endpoints*;
- $\forall u, v \in V : \ u < v \Longleftrightarrow \exists$ a path $(u, \ldots, v)$ in G;
- $\forall u, v \in V : \ u || v \Longleftrightarrow \neg(u < v) \wedge \neg(v < u)$. u and v are said to be *parallel*;
- $\forall u \in V \ \uparrow u = \{v \in V | v = u \vee v < u\}$ u's ascendants including u. In other terms, a node u is an ascendant of a node v iff $u = v$ or if there exists a path from u to v;
- $\forall u \in V \ \downarrow u = \{v \in V | v = u \vee u < v\}$ u's descendants including u. In other terms, a node u is a descendant of a node v iff $u = v$ or if there exists a path from v to u;
- two edges e, e' are *adjacent* iff they share an endpoint;
- $A \subseteq V$ is an antichain iff all nodes belonging to A are parallel. Formally, $A \subseteq V$ is an antichain in G iff $\forall u, v \in A, \ u || v$;

- AM is a *maximal* antichain iff its size in terms of number of nodes is maximal. Formally, AM is a *maximal* antichain $\forall A$ antichain in $G, \quad |A| \leq |AM|$;
- the *extended* DAG $G\backslash^{E'}$ of G generated by the edges set $E' \subseteq V^2$ is the DAG obtained from G after adding the edges in E'. As a consequence, any valid schedule of G' is necessarily a valid schedule for G:

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

- an extended graph has a similar definition as above, but it is not restricted to be a DAG;
- 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 $b_1 < a_2$. We say that I_1 finishes I_2 iff $b_1 = b_2$.

3 Some Theoretical Results on Computing Register Saturation

In this section, we study some formal properties of register saturation in order to help us compute it algorithmically. For clarity and without loss of generality, let us focus on a single register type. Accordingly, our notation becomes V_R for the set of values of the implicit type we consider, E_R for the set of flow edges through a register of that type, and δ_r and δ_w for reading/writing delays. Also, we use the notation u for both the operation u and the value it produces.

3.1 Register Need of a Schedule

Given a DAG $G = (V, E, \delta)$, a value $u \in V_R$ is alive from the point just after the writing clock cycle of u until the point of its last use (consumption). Values which are not read in G or are still alive when exiting the DAG are assumed to be kept in registers as exit values. We model these exit 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 :

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

All the consumers of u whose reading time is equal to the killing date of u are called the killers of u[1]. We assume that a value written at instant i in a register is available one step later. That is to say, if operation u reads from a register at instant i while operation v is writing in the same register at the same time, u does not get v's result but, gets the value previously stored in the register. Then, the *lifetime interval* $LT_\sigma(u)$ of a value u according to σ is $]\sigma(u) + \delta_w(u), kill_\sigma(u)]$. This interval is left-open by convention only and can be changed without any consequence on our mathematical study.

Given the lifetime 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 $vsa_\sigma(i) = \{u \in V_R | i \in LT_\sigma(u)\}$ is the set of values alive at time step i. A maximal set of values simultaneously alive are called *excessive values*. In other terms, if the register need at time step i is maximal, then all the values alive at this time step are called excessive values. Figure 3 is an example of a valid schedule for the previous DAG that needs three FP registers. The bars represent the lifetime intervals. $\{e, f\}$ are the killers of b. $\{a, b, d\}$ is a set of FP excessive values since they are the maximum number of values simultaneously alive of type float. 9 is a FP excessive clock cycle since at this time there are three FP values simultaneously alive. Note that we may have more than one set of excessive values, since the register need may be defined with many sets of values simultaneously alive.

[1] While it is evident that a killer is unique in the case of linear codes (superscalar), VLIW codes may leads to multiple killers per value.

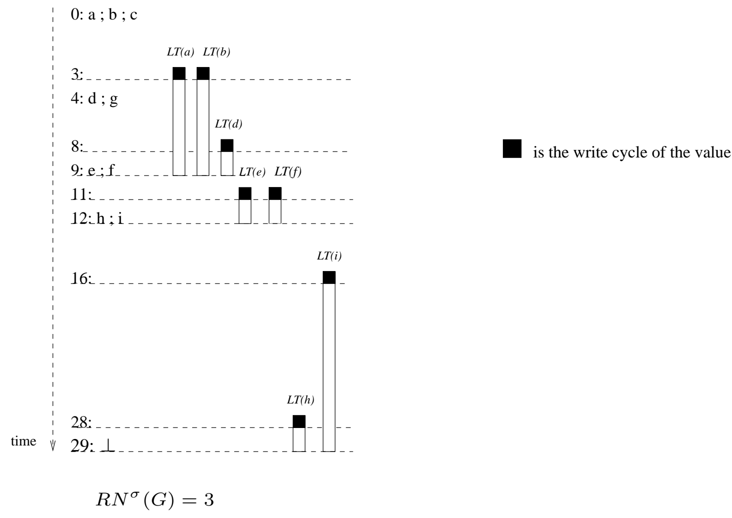


Figure 3. Register Need of Acyclic Schedules

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 Section 6 when we reduce the RS. Furthermore, we will prove that, for the purpose of maximizing 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 u, we can build another schedule with at least the same register need such that this value u is killed by only one consumer. Therefore, the purpose of this section is to select a suitable killer for each value in order to saturate the register requirement.

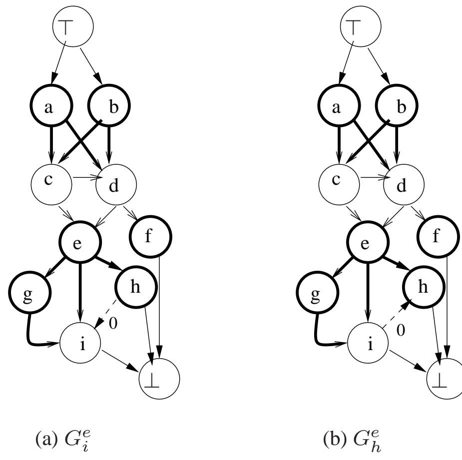
Since we do not assume any schedule, the lifetime intervals are not defined yet, so we cannot know at which date a value is killed. However, we can deduce which consumers in $Cons(u)$ are impossible killers for the value u. If $v_1, v_2 \in Cons(u)$ and $\exists$ a path $(v_1 \cdots v_2)$, v_1 is always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

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

A potential killing operation for a value u is simply a consumer of u that is neither a descendant nor an ascendant of another consumer of u. 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. The following lemma proves that for any value u and for any schedule σ, there exists a potential killer v that is a killer of u according to σ. Furthermore, for any potential killer v of a value u, there exists a schedule σ that makes v a killer of u.

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

$$\forall \sigma \in \Sigma(G), \quad \exists v \in pkill_G(u) : \quad \sigma(v) + \delta_r(v) = kill_\sigma(u) \tag{1}$$
$$\forall v \in pkill_G(u), \quad \exists \sigma \in \Sigma(G) : \quad kill_\sigma(u) = \sigma(v) + \delta_r(v) \tag{2}$$



(a) G_i^e (b) G_h^e

Figure 4. Each Potential Killing Operation can Kill the Value

Proof :

The proof of (1) is directly derived from the definition of $pkill$. Since

$$v \in pkill(u) \Longrightarrow \nexists v' \in Cons(u) \quad v < v'$$

then the killing date of u must be the schedule date of some operations in $pkill(u)$. Let us prove that

$$\forall u \in V_R,\ \nexists v' \in Cons(u) - pkill(u),\ \exists \sigma \in \Sigma(G): \quad kill_\sigma(u) = \sigma(v') + \delta_r(v')$$

Suppose the converse is true.

$$\exists v' \in Cons(u) - pkill(u) \Longrightarrow \exists v \in pkill(u) | v' < v$$

Let $lp(v', v)$ be the longest path from v' to v.

$$\text{since } lp(v', v) \geq lat(v') > \delta_r(v') \Longrightarrow \sigma(v) - \sigma(v') > \delta_r(v')$$

Since $\delta_r(v) \geq 0$:

$$\sigma(v) + \delta_r(v) - \sigma(v') > \delta_r(v') \Longrightarrow \sigma(v) + \delta_r(v) > \sigma(v') + \delta_r(v')$$

Then

$$kill_\sigma(u) \geq \sigma(v) + \delta_r(v) > \sigma(v') + \delta_r(v')$$

In order to prove (2) we create an extended DAG $G_v^u = G\backslash^{E'}$, for each $v \in pkill(u)$, to enforce v to be the last read of the value u. $\forall v' \in pkill(u) - \{v\}$, we add a serial edge e from v' to v with latency $\delta(e) = \delta_r(v') - \delta_r(v)$. Then, any schedule $\sigma \in \Sigma(G_v^u)$ ensures $\sigma(v) + \delta_r(v) \geq \sigma(v') + \delta_r(v')$ which means $kill_\sigma(u) = \sigma(v)$. Let's prove that G_v^u is still a DAG. Suppose the converse is true, *i.e.*, $\exists u \in V_R,\ \exists v \in pkill(u)$ such that G_v^u is cyclic. Let $C = (v, \cdots, v', v)$ be this cycle where the introduced edge is (v', v). We know that all the potential killing operations $pkill(u)$ of a value u are parallel in G. However, before introducing this edge, a path $P = v \rightsquigarrow v'$ means that $v < v'$ in G which is a contradiction.

Figure 4 shows the two extended DAGs associated with e. The original DAG is presented in Figure 4. Here, we assume that all read delay are null. e has two potential killing operations $\{h, i\}$, so we have two extended DAG: G_i^e ensures that i kills e, and G_h^e that ensures that h kills e.

⌟

A *potential killing DAG* of G, noted $PK(G) = (V, E_{PK})$, is built to model the potential killing relations between the operations, (see Figure 2.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. Next, we prove that looking for a unique suitable killer for each value is sufficient for maximizing the register need: the next theorem proves that for any schedule that assigns more than one killer for a value, we can build another schedule with at least the same register need such that this value is killed by only one consumer. Consequently, our formal study will look for a unique killer for each value instead of looking for a group of killers.

Theorem 1 *Let $G = (V, E, \delta)$ be a DAG and a schedule $\sigma \in \Sigma(G)$. If there is at least one excessive value that has more than one killer according to σ, then there exists another schedule $\sigma' \in \Sigma(G)$ such that:*

$$RN^{\sigma'}(G) \geq RN^{\sigma}(G)$$

and each excessive value is killed by a unique killer according to σ'.

Proof :

We suppose that there exists a schedule $\sigma \in \Sigma(G)$ with at least one excessive value that has more than one killer:

$$\exists \sigma \in \Sigma(G), \ \exists u \in EV^{\sigma}(G): \quad |killers_{\sigma}(u)| > 1$$

where $EV^{\sigma}(G)$ is a set of excessive values assuming σ as a schedule for G. We show in this proof how to build a new schedule $\sigma' \in \Sigma(G)$ such that u is killed by a unique killer and σ' needs at least as many registers as σ does.

Suppose that u has j killers according to σ, and we note them:

$$killers_{\sigma}(u) = \{k_1, ..., k_j\}$$

with $kill_{\sigma}(u) = \sigma(k_1) + \delta_r(k_1) = \cdots = \sigma(k_j) + \delta_r(k_j)$. We choose one killer within this set to be the only one killer of u according to σ', say k_1. We build σ' by "shifting" down k_1 and all its descendants with a strictly positive factor, say 1:

$$\forall v \in V \quad \sigma'(v) = \begin{cases} \sigma(v) + 1 & \text{if } v \in \downarrow k_1 \\ \sigma(v) & \text{otherwise} \end{cases}$$

Now, we prove that σ' is valid and needs at least as many registers as σ does, and that k_1 is the only killer of u according to σ'.

σ' is valid: we can easily check that any dependence $\forall e = (v_1, v_2) \in E$ is satisfied by σ':

1. if both $v_1, v_2 \notin \downarrow k_1$, then
$$\sigma'(v_2) - \sigma'(v_1) = \sigma(v_2) - \sigma(v_1) \geq \delta(e)$$
2. in the case where $v_1 \notin \downarrow k_1 \ \wedge \ v_2 \in \downarrow k_1$
$$\sigma'(v_2) - \sigma'(v_1) = \sigma(v_2) + 1 - \sigma(v_1) > \delta(e)$$
3. the case of $v_1 \in \downarrow k_1 \ \wedge \ v_2 \notin \downarrow k_1$ is impossible because the edge $e = (v_1, v_2)$ exists;
4. in the case where both $v_1, v_2 \in \downarrow k_1$, then
$$\sigma'(v_2) - \sigma'(v_1) = \sigma(v_2) + 1 - \sigma(v_1) - 1 \geq \delta(e)$$

$RN^{\sigma'} \geq RN^{\sigma}$: let t be an excessive clock cycle according to σ, *i.e*, a clock cycle t where the excessive values are simultaneously alive during it:

$$\forall v \in EV^{\sigma}(G): \qquad t \in LT_{\sigma}(v)$$

$$\Longrightarrow \forall v \in EV^{\sigma}(G): \qquad \sigma(v) + \delta_w(v) < t \leq kill_{\sigma}(v)$$

Here, we want to prove that these excessive values according to σ are still alive during t according to σ'. Any value $v \in EV^{\sigma}(G)$ has the same definition date in σ' as in σ, this is because only $\downarrow k_1$ nodes have been shifted down and :

$$\forall v \in EV^{\sigma}(G) - \{u\}: \quad v \notin \downarrow k_1$$

otherwise $LT_{\sigma}(u) \prec LT_{\sigma}(v)$ which is in contradiction with $u, v \in EV^{\sigma}(G)$. Then

$$\forall v \in EV^{\sigma}(G): \quad \sigma'(v) = \sigma(v)$$

However, the killing date of any excessive value $v \in EV^{\sigma}(G)$ could be increased by the translation factor 1:

$$\forall v \in EV^{\sigma}(G): \quad kill_{\sigma}(v) \leq kill_{\sigma'}(v)$$

which gives

$$\forall v \in EV^{\sigma}(G): \quad \sigma'(v) < t \leq kill_{\sigma'}(v)$$

$$\Longrightarrow RN^{\sigma'} \geq |EV^{\sigma}(G)| = RN^{\sigma}(G)$$

k_1 is the unique killer of u: since $k_1 \in pkill_G(u)$, there is no other potential killer $k \in pkill(u) \wedge k \neq k_1$ such as $k \in \downarrow k_1$. Otherwise, k_1 cannot kill u (pkill operations property). In this case $\sigma'(k) = \sigma(k)$ while $\sigma'(k_1) = \sigma(k) + 1$. We conclude

$$\forall k \in pkill_G(u) - \{k_1\} \quad \sigma'(k_1) + \delta_r(k_1) > \sigma'(k) + \delta_r(k) \Longrightarrow killers_{\sigma'}(u) = \{k_1\}$$

Finally, generalizing to an arbitrary number of excessive values like u (those that have more than one killer and that are simultaneously alive with u) is obviously done by iteratively building new σ' schedule for each of these values. However, we must take a precaution. Indeed, if we treat an excessive value u_1 by shifting down one of its killers, and then we proceed to another excessive value u_2, we cannot guarantee that shifting down u_2's killer would not shift down other u_1 consumers (and hence, u_1 becomes killed by multiple consumers). To break this recursivity, we proceed as follows. When we treat an excessive value u by shifting down its killer $k(v)$, we add an edge to the DAG from each potential killer of u (except $k(u)$) to $k(u)$. Hence, when we iterate over the remaining excessive values, any shifting down action would always guarantee the existence of a unique killer for the previously treated values. The added edges does not introduce a cycle since they define a strict order between the potential killing nodes.

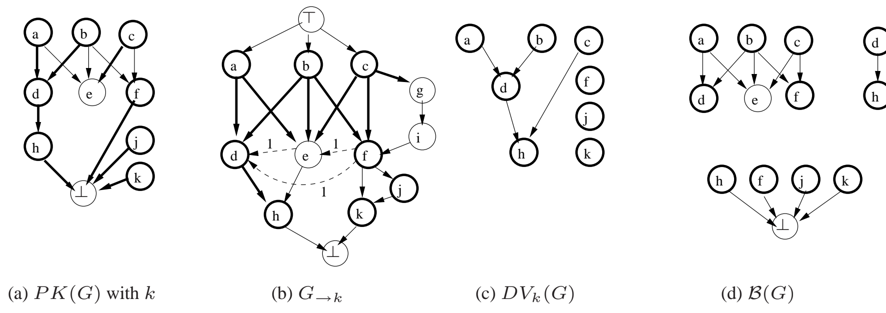
⌟

Corollary 1 *Given a DDG $G = (V, E, \delta)$, there is always a saturating schedule for G with the property that each saturating value has a unique killer.*

Proof :

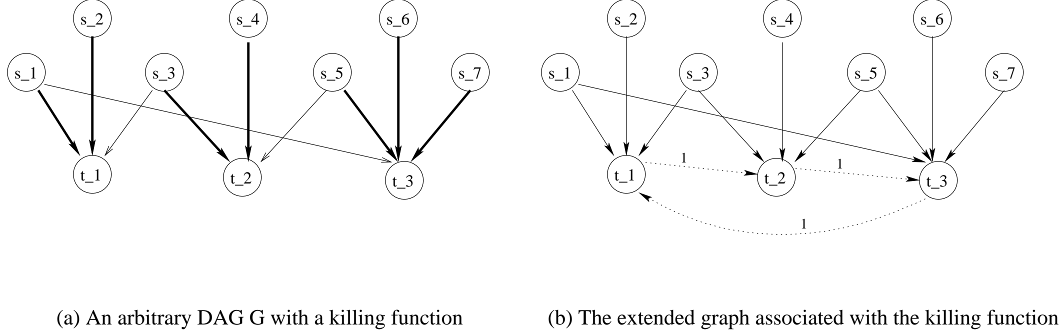
Direct consequence of Theorem 1.

⌟



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

Figure 5. Valid Killing Function and Bipartite Decomposition



(a) An arbitrary DAG G with a killing function (b) The extended graph associated with the killing function

Figure 6. An Example of an Invalid Killing Function

Let us begin by assuming a *killing function*, k, which guarantees that an operation $v \in pkill_G(u)$ is 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 that ensures this assertion. In order to define them, we extend G by new serial edges that force all the potential killing operations of each value u to be scheduled before $k(u)$. This leads us to define an extended DAG associated with k and denoted $G_{\to k} = G\backslash^{E_k}$ where:

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

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

$$k \text{ is a valid killing function} \Longleftrightarrow G_{\to k} \text{ is acyclic}$$

Figure 5 gives an example of a valid killing function k. This function is illustrated by bold edges in part (a), where each target of a bold edge kills its source. Part (b) is the DAG associated with k.

According to our definition, invalid killing functions may exist. Figure 6 is an example, where Part (a) illustrates an arbitrary DAG with a killing function (the source of each bold edge is killed by its sink). Part (b) shows that the extended graph associated with the killing function is cyclic. According to our definition, the killing function defined in Part (a) isn't valid.

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

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

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

$$\forall u \in V_R,\ \forall \sigma \in \Sigma(G_{\rightarrow k}),\ \forall v \in \downarrow_R k(u): \quad LT_\sigma(u) \prec LT_\sigma(v) \tag{4}$$

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

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

Proof :

A complete proof is given in [30], Appendix A, Section A.1.4, page 253.

┘

We define a DAG which models the values that can never be simultaneously alive when assuming k as a killing function. The *disjoint value DAG* of G associated with k, and denoted $DV_k(G) = (V_R, E_{DV})$ is defined by:

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

Any edge (u, v) in $DV_k(G)$ means that u's lifetime interval is always before v's lifetime interval according to any schedule of $G_{\rightarrow k}$, see Figure 5.c (this DAG is simplified by transitive reduction). This definition permits us to state Theorem 2 as follows.

Theorem 2 *Given a DAG $G = (V, E, \delta)$ and a valid killing function k, let AM_k be a maximal antichain in the disjoint value DAG $DV_k(G)$. Then:*

- *the register need of any schedule of $G_{\rightarrow k}$ is always less than or equal to the size of a maximal antichain in $DV_k(G)$. Formally,*

$$\forall \sigma \in \Sigma(G_{\rightarrow k}),\ RN^\sigma(G) \leq |AM_k|$$

- *there is always a schedule which makes all the values in this maximal antichain simultaneously alive. Formally,*

$$\exists \sigma \in \Sigma(G_{\rightarrow k}),\ RN^\sigma(G) = |AM_k|$$

Proof :

First property Let us begin by proving that:

$$\forall \sigma \in \Sigma(G_{\rightarrow k}) : RN^\sigma(G) \leq |AM_k|$$

$DV_k(G)$, the disjoint value DAG, models the order between value lifetime in any schedule of $G_{\rightarrow k}$. The definition of the disjoint value DAG states that $\forall \sigma \in \Sigma(G_{\rightarrow k}), \forall u, v \in V_R$:

$$u < v \text{ in } DV_k(G) \Longleftrightarrow u < k(u) \leq v \text{ in } G_{\rightarrow k}$$

If $v = k(u)$, then $\sigma(u) + \delta_w(u) < \sigma(v) + \delta_r(v)$, because of true data dependence. By hypothesis on DAG model we have $\delta_r(v) \leq \delta_w(v)$, then $\sigma(u) + \delta_w(u) < \sigma(v) + \delta_w(v)$. In the case where $v \neq k(u)$, any path from $k(u)$ to v is a data dependence path with strictly positive integer latencies. We deduce that:

$$\forall \sigma \in \Sigma(G_{\rightarrow k}) \quad \sigma\big(k(u)\big) + \delta_r\big(k(u)\big) \leq \sigma(v) + \delta_w(v)$$

That is,

$$kill_\sigma(u) \leq \sigma(v) + \delta_w(v)$$

We deduce that the following assertion is correct:

$$\forall \sigma \in \Sigma(G_{\rightarrow k}) \quad u \sim v \text{ in } DV_k(G) \implies LT_\sigma(u) \cap LT_\sigma(v) = \phi$$

We rewrite it: $\forall \sigma \in \Sigma(G_{\rightarrow k})$

$$\begin{aligned} LT_\sigma(u) \cap LT_\sigma(v) \neq \phi &\implies u||v \text{ in } DV_k(G) \\ &\implies \{u, v\} \in vsa^\sigma(c)\,, c \in LT_\sigma(u) \cap LT_\sigma(v) \end{aligned}$$

Then, any values simultaneously alive for $\sigma \in \Sigma(G_{\rightarrow k})$ belong to an antichain in $DV_k(G)$:

$$\forall 0 \leq c < \overline{\sigma},\ \exists A \text{ an antichain of } DV_k(G) \quad vsa^\sigma(c) \subseteq A$$

Since $RN^\sigma(G_{\rightarrow k}) = \max_{0 \leq c \leq \overline{\sigma}} |vsa^\sigma(c)|$ and $|vsa^\sigma(c)| \leq |AM_k|$, we conclude that $RN^\sigma(G) = \max_{0 \leq c \leq \overline{\sigma}} |vsa^\sigma(c)| \leq |AM_k|$.

Second Property Now, given a set of excessive values AM_k, we must prove that:

$$\exists \sigma \in \Sigma(G_{\rightarrow k}) : RN^\sigma(G) = |AM_k|$$

We have to build a schedule σ such that $RN^\sigma(G) = |AM_k|$. For this purpose, we consider $G_{\rightarrow k}$ in order to ensure the killing relation, and we add some serial edges to enforce the values in AM_k in order to be simultaneously alive. This leads us to a new extended DAG $G' = G_{\rightarrow k} \backslash^{E'}$ and

$$\forall \sigma \in \Sigma(G')\ \forall u, v \in AM_k : \qquad LT_\sigma(u) \cap LT_\sigma(v) \neq \phi$$

A sufficient condition that two values u, v in AM_k must satisfy to be simultaneously alive for any schedule of $G_{\rightarrow k}$ is

$$\begin{aligned} &\Big[v < u < k(v) \wedge lp(v, u) \geq \delta_w(v) - \delta_w(u)\ \wedge \\ &\qquad \wedge\ lp\big(u, k(v)\big) > \delta_w(u) - \delta_r\big(k(v)\big)\Big] && (6) \\ \vee\ &\Big[u < v < k(u)\ \wedge\ lp(u, v) \geq \delta_w(u) - \delta_w(v) \wedge \\ &\qquad \wedge\ lp\big(v, k(u)\big) > \delta_w(v) - \delta_r\big(k(u)\big)\Big] && (7) \\ \vee\ &\Big[k(u) = k(v)\Big] && (8) \end{aligned}$$

with $lp(u, v)$ for $u, v \in V$ denoting the longest path from u to v.

These conditions ensure that $\forall \sigma \in \Sigma(G_{\rightarrow k})\ \forall u, v \in V_R$:

$$\begin{aligned} u, v \text{ satisfy (6)} &\implies \sigma(u) + \delta_w(u) \geq \sigma(v) + \delta_w(v) \\ &\qquad \wedge\, \sigma(k(v)) + \delta_r(k(v)) > \sigma(u) + \delta_w(u) \\ u, v \text{ satisfy (7)} &\implies \sigma(v) + \delta_w(v) \geq \sigma(u) + \delta_w(u) \\ &\qquad \wedge\, \sigma(k(u)) + \delta_r(k(u)) > \sigma(v) + \delta_w(v) \\ u, v \text{ satisfy (8)} &\implies kill_\sigma(u) = kill_\sigma(v) \end{aligned}$$

Then, by using usual interval order algebra notations:

$$\begin{aligned} u, v \text{ satisfy Cond. (6)} &\implies \neg(LT_\sigma(u)\ \prec\ LT_\sigma(v)\ \vee\ LT_\sigma(u)\ \succ\ LT_\sigma(v)) \\ u, v \text{ satisfy Cond. (7)} &\implies \neg(LT_\sigma(u)\ \succ\ LT_\sigma(v)\ \vee\ LT_\sigma(u)\ \prec\ LT_\sigma(v)) \\ u, v \text{ satisfy Cond. (8)} &\implies LT_\sigma(u) \text{ finishes } LT_\sigma(v) \end{aligned}$$

If two values in $u, v \in AM_k$ do not satisfy any of these conditions, then we use Algorithm 1 to enforce them. This algorithm uses the boolean function $vsa_{G'}(u, v)$ to check if two values u, v satisfy one of the above conditions. We add iteratively serial edges until all values in AM_k satisfy one of these conditions. The added serial edges do not introduce a cycle and any schedule σ of G' has $RN^\sigma(G') = |AM_k|$. All this is proved by Lemma 3, as follows.

┘

Algorithm 1 Extended $G_{\to k}$ to enforce values to be simultaneously alive

Require: a valid killing function k
  construct the extended graph $G_{\to k}$ associated with k
  $G' \leftarrow G_{\to k}$ {the final extended graph is initialized}
  search for a maximal antichain AM_k in the disjoint value DAG $DV_k(G)$
  for all $u \in AM_k$ **do**
    for all $v \in AM_k | \ u \neq v$ **do**
      if $\neg vsa_{G'}(u, v)$ **then**
        if $u || v$ in G' **then**
          if $\neg(k(u) < v)$ **then**
            add the serial edges $e = (u, v), e' = (v, k(u))$ to G' with $\delta(e) = \delta_w(u) - \delta_w(v)$ and $\delta(e') = \delta_w(v) - \delta_r\big(k(u)\big) + 1$
          else $\{\neg(k(v) < u)$ certainly$\}$
            add the serial edges $e = (v, u), e' = (u, k(v))$ to G' with $\delta(e) = \delta_w(v) - \delta_w(u)$ and $\delta(e') = \delta_w(u) - \delta_r\big(k(v)\big) + 1$
          end if
        else
          if $v < u$ **then**
            add the serial edges $e = (v, u)$ and $e' = (u, k(v))$ to G' with $\delta(e) = \delta_w(v) - \delta_w(u)$ and $\delta(e') = \delta_w(u) - \delta_r\big(k(v)\big) + 1$
          else $\{u < v\}$
            add the serial edges $e = (u, v)$ and $e' = (v, k(u))$ to G' with $\delta(e) = \delta_w(u) - \delta_w(v)$ and $\delta(e') = \delta_w(v) - \delta_r\big(k(u)\big) + 1$;
          end if
        end if
      end if
    end for
  end for

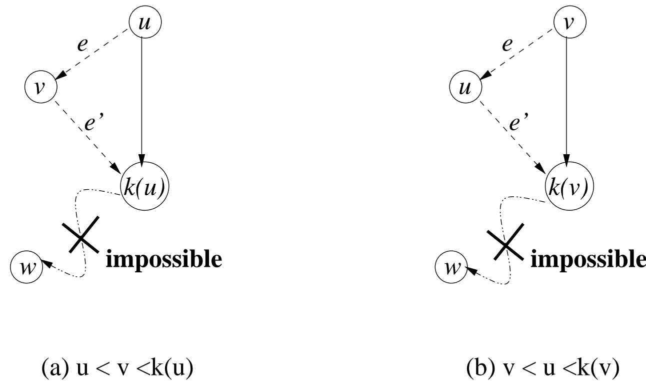
Lemma 3 *Let $G = (V, E, \delta)$ be a DAG. Let k be a killing function and AM_k be a maximal antichain in the disjoint value DAG $DV_k(G)$. The extended graph $G' = G_{\to k} \backslash^{E'}$ produced by Algorithm 1 has the two following properties:*

1. *it a DAG;*
2. *for any schedule σ of G', the lifetime intervals of any two values belonging to the maximal antichain AM_k interfere. Formally,*

$$\forall u, v \in AM_k,\ \forall \sigma \in \Sigma(G') : \qquad LT_\sigma(u) \cap LT_\sigma(v) \neq \phi$$

Proof :

We proceed by induction. We prove that after exiting Algorithm 1, G' is still a DAG. We also prove that the algorithm makes all values in AM_k satisfying one of the conditions (6), (7) or (8). For this last condition, if two values do not satisfy it in the DAG $G_{\to k}$, they cannot satisfy it in G': this is because the killing operations



(a) u < v <k(u) (b) v < u <k(v)

Figure 7. Making Values Simultaneously Alive

has been fixed in $G_{\rightarrow k}$. So, if u, v do not satisfy Condition (8), Algorithm 1 can only force them to satisfy Condition (6) or Condition (7).

We prove also the following property

$$\forall u, v \in AM_k \quad \neg\big(k(v) < u \vee k(u) < v\big) \text{ in } G'$$

which is the same as proving that Algorithm 1 guarantees that all values in AM_k are forced to be simultaneously alive in G':

$$\nexists u, v \in AM_k | u \sim v \text{ in } DV_k(G')$$

Initially, this is correct because $u, v \in AM_k \Longrightarrow u \notin\downarrow_R k(v) \wedge v \notin\downarrow_R k(u)$. In this proof, we note G'_i the graph built after exiting iteration i. Suppose that after exiting iteration $i-1$, G'_{i-1} is still a DAG and

$$\forall u, v \in AM_k \quad \neg\big(k(v) < u \ \vee \ k(u) < v\big) \text{ in } G'_{i-1}$$

Let u_i and v_i be the two chosen values at iteration i which do not satisfy any of the conditions. Let us prove now that G'_i is still a DAG and the two chosen values $u_i, v_i \in AM_k$ satisfy one of the conditions after exiting iteration i. Furthermore, we prove that after exiting this iteration

$$\nexists w \in AM_k | k(u_i) < w \ \vee \ k(v_i) < w \text{ in } G'_i$$

Our algorithm introduces serial edges in four cases:

1. $u_i || v_i$ in G'_{i-1}, then
   - if $\neg(k(u_i) < v_i)$, the two introduced edges $e = (u_i, v_i), e' = (v_i, k(u_i))$ cannot introduce a cycle, because $u_i < k(u_i)$ in G'_{i-1}, see Figure 7.a. Now they are satisfying Cond. (7). Also, after introducing these edges, the following property is satisfied:

     $$\nexists w \in AM_k | k(u_i) < w \ \vee \ k(v_i) < w \text{ in } G'_i$$

     Suppose the converse is true, *i.e.*,

     $$\exists w \in AM_k | k(u_i) < w \ \vee \ k(v_i) < w \text{ in } G'_i$$

     If $k(u_i) < w$ in G'_i, $\Longrightarrow k(u_i) < w$ in G'_{i-1} because we have not introduced a serial edge from $k(u_i)$, which is impossible because of induction hypothesis.
     If $k(v_i) < w$ in G'_i, $\Longrightarrow k(v_i) < w$ in G'_{i-1} because we have not introduced a serial edge from $k(v_i)$, which is also impossible because of induction hypothesis;

- else $\neg(k(v_i) < u_i)$ certainly, because otherwise

$$v_i < k(v_i) < u_i \wedge u_i < k(u_i) < v_i \Longrightarrow u_i < v_i \wedge v_i < u_i \text{ in } G'_{i-1} \text{ (impossible)}$$

  Then the introduced edges $e = (v_i, u_i), e' = (u_i, k(v_i))$ cannot introduce any cycle because $v_i < k(v_i)$ in G'_{i-1}, see Figure 7.b. Now they are satisfying Cond. (6). Also, after introducing these edges, the following property is satisfied:

$$\nexists w \in AM_k | k(u_i) < w \ \vee \ k(v_i) < w \text{ in } G'_i$$

  The proof is similar to the above case;

2. if $v_i < u_i$ in G'_{i-1}, then by induction hypothesis $\neg(k(v_i) < u_i)$ in G'_{i-1}. The two introduced edges $e = (v_i, u_i)$ and $e' = (u_i, k(v_i))$ cannot cause any cycle. Now they are satisfying Cond. (6). Also, after introducing these edges,

$$\nexists w \in AM_k | k(u_i) < w \ \vee \ k(v_i) < w \text{ in } G'_i$$

   The proof is similar to the case above;

3. $u_i < v_i$ in G'_{i-1}, this case is similar to above. Now they are satisfying Cond. (7).

After $n = |AM_k|^2$ iterations, we conclude that :

$$\forall u, v \in AM_k \quad u, v \text{ satisfy one of the conditions (6), (7) or (8)}$$

and then $\forall u, v \in AM_k \ \forall \sigma \in \Sigma(G') \qquad LT_\sigma(u) \cap LT_\sigma(v) \neq \phi$

┘

Theorem 2 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*. A saturating killing function means a killing function that produces a saturated register need. The saturating values are the values that are simultaneously alive, and their number reaches the maximal possible register need. Unfortunately,

Theorem 3 *Given a DAG $G = (V, E, \delta)$, computing a saturating killing function is NP-complete.*

Proof :

A complete proof is given in [30], Appendix A, Section A.1.5, page 253.

┘

Corollary 2 *Given a DAG $G = (V, E, \delta)$, computing the register saturation is NP-complete.*

Proof :

A complete proof is given in [30], Appendix A, Section A.1.6, page 257.

┘

4 A Heuristics for Computing the RS

This section presents our heuristics to approximate an optimal k by another valid killing function k^*. An optimal k is simply a killing function that defines the optimal register saturation. We have to choose a killing operation for each value such that we maximize the parallel values in $DV_k(G)$. Our heuristics compute a valid killing function by focusing 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 that keeps 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 descendant's values simultaneously alive with their parent's values. A CBC $cb = (S_{cb}, T_{cb}, E_{cb})$ is a partition of a subset of operations into two disjoint sets where:

- $E_{cb} \subseteq E_{PK}$ 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 one value in S_{cb}.

A bipartite decomposition of the potential killing graph $PK(G)$ is the set (see Figure 5.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 the parents, as well as the children, are parallel inside the potential killing DAG$PK(G)$. Formally,

$$\forall cb \in \mathcal{B}(G)\ \forall s, s' \in S_{cb}\ \forall t, t' \in T_{cb} : \quad s||s' \wedge t||t' \text{ in } PK(G)$$

A saturating killing set $SKS(cb)$ of a bipartite component $cb = (S_{cb}, T_{cb}, E_{cb})$ is a subset of children $T'_{cb} \subseteq T_{cb}$. Such subset provides a unique killer for each value present in the set S_{cb} of parents. Such unique killer is chosen so as to minimize the number of descendant values of all the killers in T_{cb}. The dual consequence is to get a maximal number of values simultaneously alive with the parent values belonging to 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: each parent must be killed*

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

2. *objective function: minimize 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 (the proof is the same as Theorem 3's proof).

A Heuristics for Finding a SKS Intuitively and according to Lemma 2, 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 not (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 greatest number of non-killed parents, with a corresponding minimum number of descendant values. If there are no descendant values, then we choose the child that covers the most non-killed parents.

Algorithm 2 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

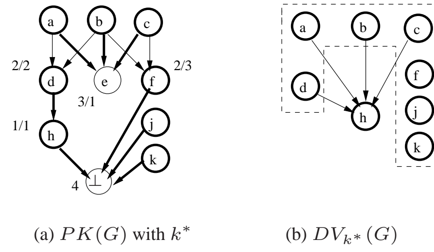
Algorithm 2 gives a greedy heuristics that searches for an approximation SKS^* and computes a killing function k^* in polynomial time. Our heuristics has the following property.

Corollary 3 *Let $G = (V, E, \delta)$ be a DAG. If $PK(G)$ is a tree, then Greedy-k computes an optimal register saturation with a polynomial time complexity.*

Proof :

Trivially, each value has at most one possible killer, *i.e.*, there is only one choice for the killing function. Then, the saturating values are simply the sources of the potential killing DAG $PK(G)$. Expression trees for instances belong to this class of DAGs, because their potential killing DAGs are trees.

┘



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

Figure 8. Example of Computing the Register Saturation

Since the approximated killing function k^* is valid, Theorem 2 ensures that we can always find a valid schedule which requires exactly $|AM_{k^*}|$ registers. Consequently, our heuristics do not compute an upper bound of the optimal register saturation, and 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 unnecessary RS reduction process and a waste of registers. The validity of the killing function is a key condition because it ensures the existence of a register allocation requiring exactly $|AM_{k^*}|$ registers. As a 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 [11]. The approximated set of saturating values is the nodes belonging to AM_{k^*}. The approximated RS is equal to $RS^*(G) = |AM_{k^*}| \leq RS(G)$.

Figure 8.a shows a saturating killing function k^* computed by Greedy-k: bold edges mean that each source is killed by its sink. Each killer is labeled by its cost ρ. Part (b) gives the disjoint value DAG associated with k^*. The Saturating values are $\{a, b, c, d, f, j, k\}$, so the RS is 7.

5 Exact Register Saturation Computation

First, 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 compute the register saturation (RS).

Let $RN_t^\sigma(G)$ denote the register need of register type t given a schedule $\sigma \in \Sigma(G)$, which is equal to the maximal number of values of type t simultaneously alive. The RS of a register type t for a DAG G is the maximal register need of type t among all valid schedules of G:

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

Below, we give the set of variables and constraints of an exact integer linear programming (intLP) formulation for computing the optimal $RS_t(G)$. 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 we bound the domain of the integer variables, as explained below.

5.1 Expressing Logical Operators by Integer Programming

In [17], the authors show how to model the disjunctive operator $\vee$. Consider the problem:

$$\begin{cases} \text{maximize (or minimize) } f(x) \\ \text{subject to : } g(x) \geq 0 \ \vee \ h(x) \geq 0 \end{cases}$$

By introducing a binary variable $\alpha \in \{0, 1\}$, this disjunction is equivalent to:

$$\begin{cases} g(x) \geq \alpha \underline{g} \\ h(x) \geq (1 - \alpha)\underline{h} \end{cases}$$

where $\underline{g}$ and $\underline{h}$ are two known non null finite lower bounds for g and h respectively. We deduce the linear constraints of any other logical operator:

1. $g(x) \geq 0 \Longrightarrow h(x) \geq 0$ can be written $g(x) < 0 \vee h(x) \geq 0$
2. $g(x) \geq 0 \Longleftrightarrow h(x) \geq 0$ can be written $\big(g(x) \geq 0 \wedge h(x) \geq 0\big) \vee \big(h(x) < 0 \wedge g(x) < 0\big)$

Also, $z = max(x, y)$ can be written $\begin{cases} x \geq y \Longrightarrow z = x \\ y \geq x \Longrightarrow z = y \end{cases}$.

Thanks to the use of binary variables for expressing logical operators, our intLP formulation of register constraints contains a polynomial number of variables and constraints, *i.e*, it depends only on the number of nodes and edges of the input DAG. Unfortunately, this is not the case of the existing techniques in the literature where the number of variables and constraints is pseudo-polynomial, since this number depends on the total schedule time. The following section presents our intLP formulation of RS computation.

5.2 Scheduling Variables

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

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

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 (the extreme case of a sequential schedule, i.e, no ILP). Then, we write the following constraint:

$$\sigma_\perp \leq T$$

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

- $\sigma_u \geq \underline{\sigma_u} = LongestPathTo(u)$ is the shortest schedule time;
- $\sigma_u \leq \overline{\sigma_u} = T - LongestPathFrom(u)$ is the longest schedule time according to the worst total schedule time T.

5.3 Register Need Constraints

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

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

That is, we assume that a value written at instant c in a register is available one step later. Thus, if an operation u reads from a register r at instant c while another operation v is writing to the r at the same time, u does not get v's result, but rather gets the value previously stored in r. Note that these semantics are explicitly chosen and encoded in the definition of $LT_\sigma(u^t)$, and are not a limitation of the model.

We define for each value u^t the variable $k_{u^t} \geq 0$ which computes its killing date (the last time that u^t is read). 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:

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

where

- $\underline{k_{u^t}} = \underline{\sigma_u} + \delta_{w,t}(u)$ is the first possible definition date of u^t;
- $\overline{k_{u^t}} = \max_{v \in Cons(u^t)} \left(\overline{\sigma_v} + \delta_{r,t}(v)\right)$ 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 Section 5.1. We write into the intLP system:

$$\forall u^t \in V_{R,t}: \qquad k_{u^t} = \max_{v \in Cons(u^t)} \left(\sigma_v + \delta_{r,t}(v)\right)$$

Now, we can consider H_t the undirected interference graph of G for the register type t. For any pair 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 $u^t, v^t \in V_{R,t}$:

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

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

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

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

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

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

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 [30]) and by computing the finite lower bounds of the linear functions.

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 $x_{u^t} \in V_{R,t}$ such that $x_{u^t} = 1$ if 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 a given IS.

5.4 Linear Objective Function and General Remarks

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 the intLP formulation is bounded by $\mathcal{O}(|V|^2)$, and the total number of constraints is at most $\mathcal{O}(|E| + |V|^2)$. 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 edge 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:

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

The next section explores the problem of reducing RS if it exceeds the number of available registers.

6 The Complexity of 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 edges into the DAG G to reduce $RS_t(G)$ below this limit. The added edges 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 first solve the formal problem stated below.

Definition 2 (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

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

Note that an extended DDG may contain a cycle (as we will see later), while an extended DAG is restricted to stay a DAG.

Theorem 4 *The ReduceRS problem is NP-hard.*

Proof:

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

Definition 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

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

The SRC problem has been proven NP-hard in [13]. Now we prove the equivalence of ReduceRS and SRC 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.*, $RN^{\sigma}(G) \leq \mathcal{R}$ with a total schedule time of $\leq \mathcal{P}$. We build an extended DDG $\overline{G}$ by adding serial edges to impose the same precedence relations as defined by σ on the value lifetimes of any schedule of $\overline{G}$. Then, $\forall u, v \in V_R | LT_{\sigma}(u) \prec LT_{\sigma}(v)$ we add the following edges:

- If $v \in Cons(u)$, add serial edges from the readers of u (except v) to v; the set of added edges is:

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

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

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

The latency of these added edges has to be assigned based on the target architecture. There are two cases:

1. in the case of superscalar codes, there are sequential code semantics. So, the latency of each added edge is set to 1;
2. in the case of VLIW or EPIC/IA64, there are reading and writing offsets. Thus, for each added edge $e = (u', v)$, the latency is set to $\delta(e) = \delta_r(u') - \delta_w(v)$.

Indeed, the added edges 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 not simultaneously alive according to σ, there is no schedule σ' of $\overline{G}$ that makes them simultaneously alive. Formally, :

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

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 require more than the register need of σ and

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

A solution for the SRC problem may create a cycle in the solution of ReduceRS. We are sure that if any cycle 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 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 edges 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 next section provides an algorithmic heuristics that tries to reduce RS below a limit. This section follows the ideas and notations used in Section 4.

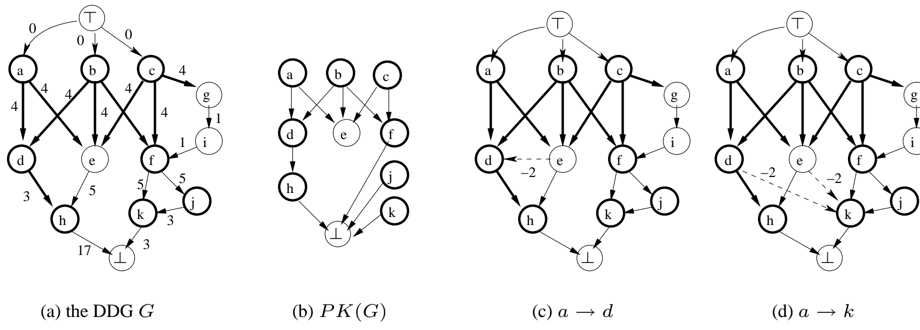
7 An Algorithmic Heuristics for Reducing the Register Saturation

For clarity and without loss of generality, let us focus on only one register type [2]. Then, our notations become V_R for the set of values of the implicit type we consider, E_R for the set of flow edges through a register of that type, δ_r and δ_w for reading/writing delays, and $RN^\sigma(G)$ for the register need of the type we consider. Also, we use the notation u for both the operation u and the value of the considered type it produces.

In this section we build an extended DAG $\overline{G} = G\backslash^{E}$ 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 :

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

[2] If more than one register type exists, we apply our algorithm on each type.



(a) the DDG G (b) $PK(G)$ (c) $a \to d$ (d) $a \to k$

Figure 9. Value Serialization

This section presents a heuristics that adds serial edges to prevent some saturating values in AM_k (according to a saturating killing function k) from being simultaneously alive for any schedule. Also, we take care not to increase the critical path, if possible.

Serializing two values $u, v \in V_R$ means that the killing of u must always be carried out before the definition of v, or *vice-versa*, as illustrated by Figure 9. A value serialization $u \to v$ for two values $u, v \in V_R$ is defined by:

- if $v \in pkill_G(u)$ then add the serial edges $\left\{e = (v', v) | v' \in pkill_G(u) - \{v\}\right\}$. Textually, this means that if v is a potential killer of u, the value serialization $u \to v$ means to add a serial edge from any potential killer of u (except v) to v itself, see Figure 9.c.
- otherwise add the serial edges $\left\{e = (u', v) | u' \in pkill_G(u) \wedge \neg(v < u')\right\}$ Textually, this means that if v is not a potential killer of u, the value serialization $u \to v$ means to add a serial edge from any potential killer of u to v itself, see Figure 9.d.

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

1. in the case of superscalar codes, the semantics is sequential. So, the latency of each added edge is set to 1;
2. in the case of VLIW or EPIC/IA64, there exist reading and writing offsets[3]. Thus, for each added edge $e = (u', v)$, the latency is set to $\delta(e) = \delta_r(u') - \delta_w(v)$.

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

- $\omega_1 = \mu_1 - \mu_2$ tries to predict how much RS would be reduced (in the best case) if we carry out this value serialization, where
  - μ_1 is the number of saturating values serialized after u if we carry out this value serialization $u \to v$;
  - μ_2 is the predicted number of u's descendant values that can become simultaneously alive with u;
- ω_2 is the predicted increase in the critical path.

Our heuristics is described in Algorithm 3. It iterates 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 [3, 4, 9, 10, 27] and is not addressed in this article.

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 $\overline{G_i}$:

[3]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.

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 edges 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.

Algorithm 3 Value Serialization Heuristics

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}

At the end of the algorithm, we apply a general check step to ensure the potential killing property proved in Lemma 1 (page 6) for the original DAG. Lemma 1 proves that the operations which do not belong to $pkill_G(u)$ cannot kill the value u. After adding the serial edges that build $\overline{G}$, we may violate this assertion because we introduce some edges with negative latencies. If this assertion is not verified, the computed RS may be incorrect. To overcome this problem, we must guarantee the following assertion: $\forall u \in V_R,\ \forall v' \in Cons(u) - pkill_{\overline{G}}(u)$:

$$\exists v \in pkill_{\overline{G}}(u) | v' < v \text{ in } \overline{G} \Longrightarrow lp_{\overline{G}}(v', v) > \delta_r(v') - \delta_r(v) \quad (9)$$

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 (9) is not verified, we add a serial edge $e = (v', v)$ with $\delta(e) = \delta_r(v') - \delta_r(v) + 1$ as illustrated in Figure 10: after two value serializations during step 1 and 2, assertion (9) is forced to be verified during step 3.

Example 1 *Figure 11 gives an example of reducing the RS of our initial DAG (Figure 2 page 4) from 7 to 4 registers. Remember 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 heuristics selects $a \rightarrow f$ as a candidate, since it is expected to eliminate 3 saturating values*

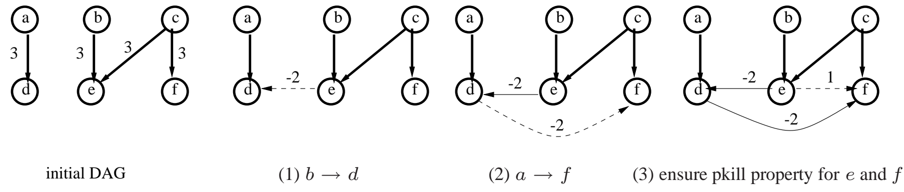


Figure 10. Checking the Potential Killers Property

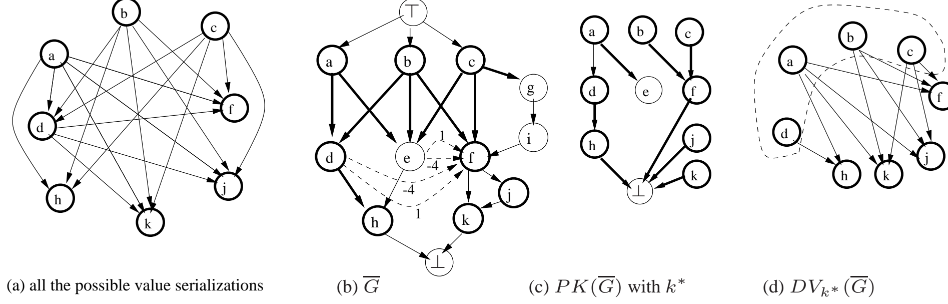


Figure 11. Register Saturation Reduction

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 serial edges* (e, f) *and* (d, f) *with a -4 latency. Finally, we add the serial edges* (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* $\overline{G}$*, presented with bold edges in* $PK(\overline{G})$. $DV_k(\overline{G})$ *is presented in part (d) to show that the new RS is 4 floating point registers.*

After providing an approximate algorithm for RS reduction, the next section presents an optimal exact method using integer linear programming.

8 An Optimal Method for RS Reduction

The proof of Theorem 4 gives the intuition for our optimal solution for the ReduceRS problem using integer programming. It is computed in two steps:

1. we first compute a valid schedule σ such that the register need of type t is maximized but 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;

2. then, we add serial edges as described by the proof of Theorem 4. This results in an extended DDG that has a bounded register saturation with a minimized critical path.

In order to compute such a minimal schedule that does not require more than $\mathcal{R}_t$ registers, we use our intLP formulation previously defined in Section 5 that maximizes the register need. We keep all the constraints and variables of Section 5, except those that compute a maximal independent set. Now, we use a binary variable $x^i_{u^t}$ for each value 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 in 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 the 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:

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

As explained before, our DAG and processor model includes writing and reading offsets. Consequently, in some cases, the optimal RS reduction may need to introduce non-positive cycles into the original DAG. Even if such non-positive cycles 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 instruction scheduling pass. We must eliminate such optimal solutions as explained in the following section.

Eliminating Cycles with Non-positive Latencies

As presented in the proof of Theorem 4, the latency of any added edge $e = (u', v)$ is equal to $\delta(e) = \delta_r(u') - \delta_w(v)$ in the case of VLIW code. Thus, if $\delta_r(u') \leq \delta_w(v)$ then $\delta(e)$ becomes non-positive, producing possible non-positive cycles.

Remember that the purpose of the register saturation analysis is to ensure in the first steps of compilation that any schedule of a given DAG will not require more registers than those available. The scheduling phase is mainly constrained by resources (functional units) of the target architecture. If the extended DDG produced by the register saturation reduction contains a non-positive cycle, we cannot guarantee the existence of a schedule under resource constraints. This is because non-positive cycles introduce some “not later than” scheduling constraints which may not be satisfied in the presence of resource constraints[4].

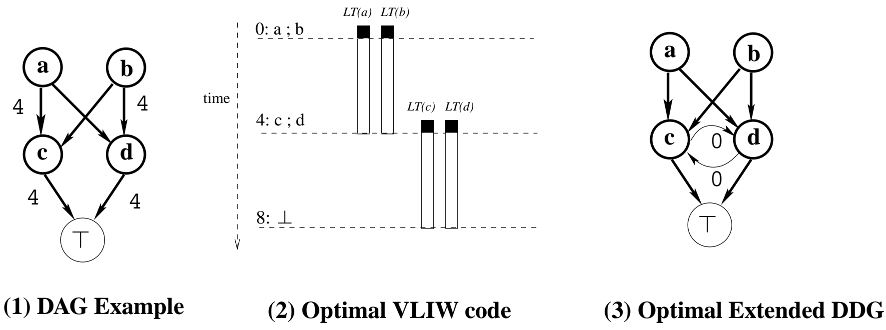
For instance, let us assume a zero weighted cycle between two operations u and v. Theoretically, any schedule such that $\sigma(u) = \sigma(v)$ satisfies this zero weighted cycle. However, if we have a resource constraint that prohibits these two operations from being scheduled at the same clock cycle, then there is no valid schedule that meets these constraints. When we reduce the register saturation, we must ensure than there is always a schedule for any resource constraints. The following example gives an illustration.

Example 2 *We use Figure 12 in this example. The register saturation of the DAG in Part (1) is equal to 3 (easy to see that we can schedule $\{a, b, c\}$ to be simultaneously alive). Here we assume that the reading and writing delays are equal to zero. Let ask the question: does there exist an extended DDG of the DAG in Part (1) with a RS equal to two while the critical path is equal to eight ? The answer is yes. The extended DDG is presented in Part (3). The VLIW schedule in Part (2) shows that it requires two registers while its total schedule time is equal to 8. As can be seen, the extended DDG constructed from this schedule has a null cycle between c and d. We can easily see that we cannot construct any extended DDG without a cycle, since the minimal register need of the DAG is 2: the lifetimes intervals of the values c and d must be necessary serialized after the intervals of a and b if we want to require only two registers. These two lifetime intervals serializations are responsible for introducing the null cycle between c and d. Now, if we accept the extended DDG of Part (3) as a solution, we cannot guarantee the existence of a schedule under any resource constraints. For instance, if c and d cannot be scheduled in parallel because of resource conflicts, then no valid schedule exists. We do admit such situation in the process of RS reduction.*

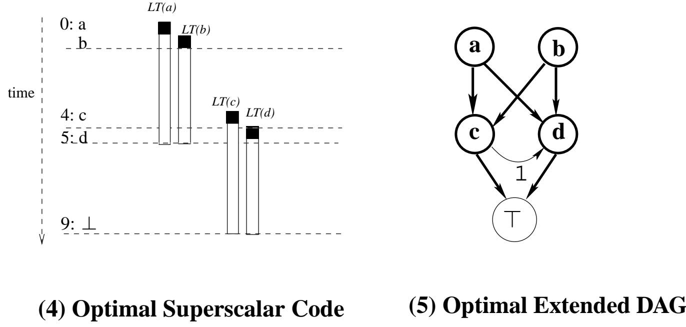
Note that the problem of non-positive cycles does not arise for superscalar (sequential) codes because all the introduced edges have a positive latency equal to 1. As example, the minimal register saturation (in the case of superscalar codes) of the DAG in Figure 12.(1) is equal to 3 (instead of 2 in the case of VLIW codes). The superscalar schedule is presented in Part (4) with its corresponding extended DAG in Part (5).

To eliminate this problem of non-positive cycles, we impose the restriction that the extended graph $\overline{G}$ must be a DAG. This is done by guaranteeing the existence of a topological sort for the extended graph. Therefore, we add some variables and constraints to the optimal intLP system.

[4]Such constraints are similar to real time constraints, which cannot always be satisfied.



(1) DAG Example (2) Optimal VLIW code (3) Optimal Extended DDG



(4) Optimal Superscalar Code (5) Optimal Extended DAG

Figure 12. Example of Non-positive Cycles

- We define integer variables that hold a topological ordering of the graph. For each $u \in V$, we associate an integer variable d_u, such that for any two nodes u and w, $d_u < d_w$ means that u is topologically sorted before w.
- We bound the topological sort by the number of nodes: $\forall u \in V: \quad d_u \leq |V|$
- We write the topological sort constraints for each edge in the original DAG: $\forall e = (u, v) \in E: \quad d_u < d_v$
- If we add a serial edge in the extended DDG, we must satisfy the topological sort constraints. If two lifetime intervals $LT_\sigma(u^t)$ and $LT_\sigma(v^t)$ do not interfere with each other, serial edges will be introduced. $\forall u, v \in V_{R,t}$:
  - if $v \in Cons(u^t)$, serial edges will be added from the u's other readers to v. We then write the constraints:

    $$LT_\sigma(u^t) \prec LT_\sigma(v^t) \Longrightarrow \Big(\forall u' \in Cons(u^t) - \{v\} : d_{u'} < d_v\Big)$$

    That is,

    $$\sigma_v + \delta_{w,t}(v) - k_{u^t} \geq 0 \Longrightarrow \Big(\forall u' \in Cons(u^t) - \{v\} : d_{u'} < d_v\Big)$$

  - if $v \notin Cons(u^t)$, serial edges will be added from all u's readers to v. We then write the constraints:

    $$LT_\sigma(u^t) \prec LT_\sigma(v^t) \Longrightarrow \Big(\forall u' \in Cons(u^t) : d_{u'} < d_v\Big)$$

    That is,

    $$\sigma_v + \delta_{w,t}(v) - k_{u^t} \geq 0 \Longrightarrow \Big(\forall u' \in Cons(u^t) : d_{u'} < d_v\Big)$$

Note that these constraints may be optimized by considering the fact that some values can never interfere, see Section 5.4.

We add at most $\mathcal{O}(|V|^3)$ variables and $\mathcal{O}(|V|^3 + |E|)$ constraints to guarantee that reducing RS always produces an acyclic extended DAG. Again, these constraints are only added for VLIW and EPIC codes, not for superscalar codes.

We continue in the next section with the result of our experimental implementation.

9 Experiments

This section presents our experimental results from some DDGs extracted from SpecFP, whetstone, livermore and linpack. Such graphs can be explored in [30]. These DDGs are those that have been used in the prior studies [12, 19]. In our experiments, we focus on floating point registers and we assume that we target superscalar codes. The DAGs used for the experiments are the loop bodies. This section presents our concluding analysis.

Before starting the presentation of our experiments, we would like to argue why we chose to evaluate our method using graphs instead of implementing it inside a real compiler. First, since our study focuses on register optimization in DDGs, we decide to check the efficiency of our heuristics on some realistic graphs extracted from real codes. This way of evaluation that does not require a complete implementation inside a compiler allows us to isolate our contribution by demonstrating the efficiency of our heuristics on DDGs. If we include our heuristics inside an existing optimizing compiler, it would be hard to isolate our contribution, since the optimizing compilation passes are numerous nowadays, and their interactions (whether they are with the hardware or with other compilation passes) are difficult to analyze. In other words, for any value of the resulted speedup (positive or negative), it is very hard to certify that the speedup gain or loss results directly and only from our heuristics. It is possible that the interaction with other compilation passes may inhibit or accentuate the performance gain. So, we think that it is better for us to concentrate our attention on graphs.

Second, we think that our experiments are realistic because our theoretical model takes into account VLIW, EPIC and superscalar codes. Usually, not all register optimization methods, even those implemented inside compilers, work for these three types of program semantics. Third and last, our experiments clearly demonstrates nearly optimal results, which is an important aspect in our case of combinatorial problems.

9.1 Computing RS

The first experiments check the efficiency of our Greedy-k algorithm compared to optimal RS (computed by integer programming). The next section summarizes our results.

9.1.1 Optimal vs. Approximated Methods

Let RS denote the optimal register saturation computed by intLP, and RS^* the approximated RS as computed by our heuristics. The experimental results show that our approximate algorithm is very efficient: in almost all cases, it computes the exact register saturation. The maximal experimental error is 1, *i.e.*, the optimal register saturation is one larger than the saturation computed by our heuristics. We have unrolled the loops to increase register pressure in order to study the efficiency of our heuristics in larger DAGs. DAGs are the bodies of these unrolled loops: the number of nodes in these unrolled loops ranges from 4 to 120.

Our approximated algorithm clearly computes nearly optimal solutions in polynomial time. In the 134 DAGs used in this study (up to 120 nodes per DAG), we do not reach RS optimality in only 7 cases. Our worst empirical error is 1, *i.e.*, $RS^* \leq RS \leq RS^* + 1$.

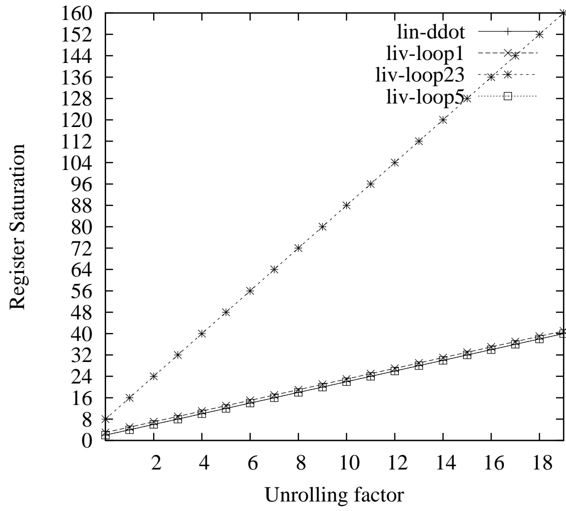
After evaluating the efficiency of our method, we use it to experimentally study the RS behavior in unrolled loops.

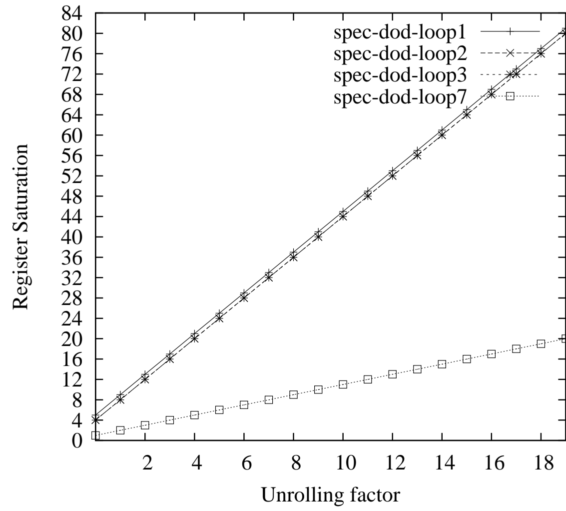
9.1.2 RS Behavior in Unrolled Loops

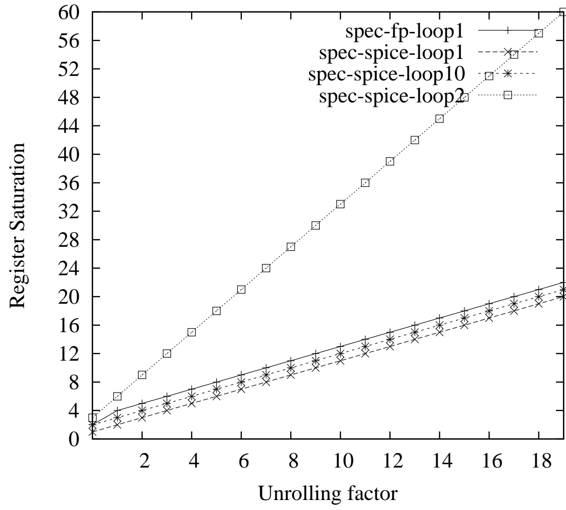
In this experiment, we study the RS evolution as a function of the unrolling degree in each loop. Figure 13 shows the plots of RS (computed by our heuristics) versus the unrolling degree. Loops are unrolled from 1 to 20 times, producing DAGs with between 4 to 400 nodes, which is sufficient to study the RS behavior in real applications. As we expect, RS is evidently a non-decreasing function: since unrolling a loop produces more values because of loop bodies duplication, RS could not decrease. The RS versus the unrolling factor produces a function that can be one of the two following cases:

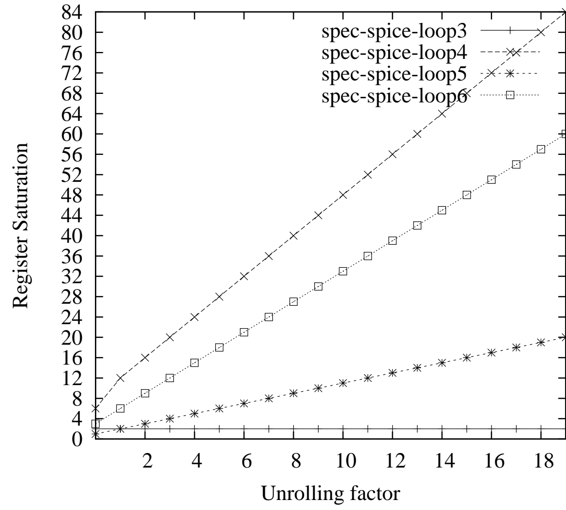
1. constant or non strictly increasing because of recurrent data dependences;
2. linear in the case of, for instance, fully parallel loops.

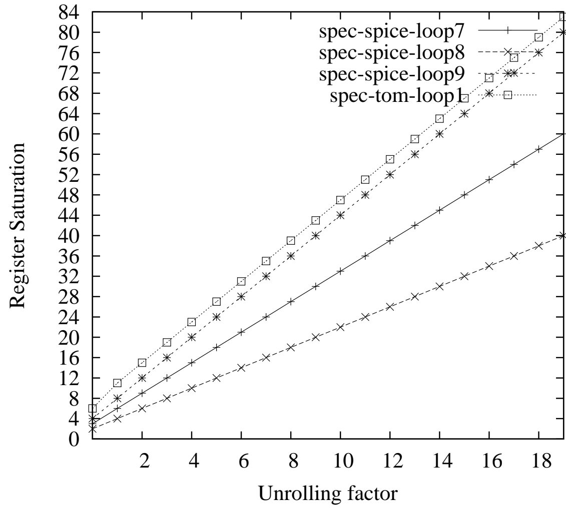
If the number of available registers is bounded, we must keep RS under control. The next section summarizes our results.











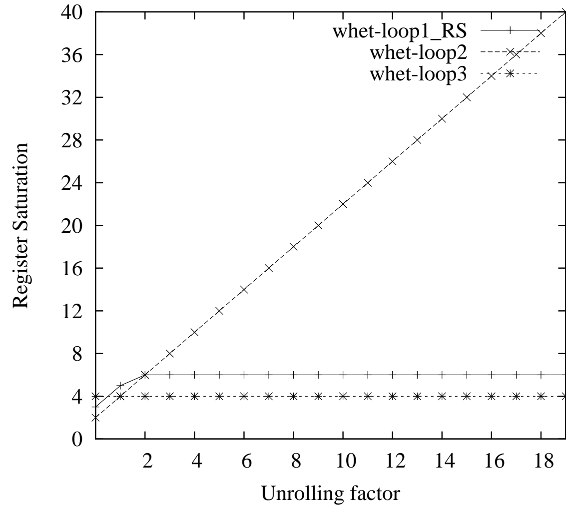


Figure 13. RS Evolution in Unrolled Loops

9.2 Reducing RS

In this section, we experimentally study our techniques for reducing RS under critical path constraints. At first, we investigate the efficiency of our heuristics versus the optimal results.

9.2.1 (Approximated) Value Serialization Heuristics versus Optimal RS Reduction

Let us begin by stressing our heuristics to check their limitations. We consider DAGs of loop bodies and try to reduce the register saturation to the lowest possible value. This is done by setting the number of available registers $\mathcal{R} = 1$. Our value serialization heuristics get sub-optimal results for only 7 of the 27 DAGs used in the experiment. The optimally reduced RS is less than our heuristics results by two registers in the worst case.

In the second set of experiments, we unroll the loops with multiple factors (up to 6, with up to 80 node DAGs) and we try to reduce their RS under a limit computed as the first power of 2 lower than the original RS. For example, if the original RS is 12 then we reduce it to 8, etc.

Here, we also get a maximal experimental error of 2 registers.

We didn't check for larger unrolling degrees because computing optimal RS reduction of larger DAGs is computational intractable. We think that the experiments that we have performed are sufficient to study the efficiency of our strategies (the number of nodes in all these unrolled loops ranges from 4 to 80).

After evaluating the efficiency of value serialization, we use it to investigate unrolled loops.

9.2.2 Value Serialization Heuristics Behavior in Unrolled Loops

We study the limit of RS reduction versus the degree of loop unrolling (we consider the DAG of the loop bodies after unrolling). Figure 14 plots RS reduced to 32 registers using our heuristics on various loops with unroll factors ranging from 1 to 20. In almost all practical cases, RS is maintained under the 32-register limit, except for Livermore-loop23. In that case, RS is maintained under 32 until the loop is unrolled by a factor of 12. After that, the register pressure is sufficiently high to always keep the register need above 32. The reason is shared by both intrinsic data dependences properties (intrinsic register pressure, *i.e.*, register sufficiency) and our heuristics limitations. If RS cannot be reduced below the limit, we have to insert spill operations, which is outside the scope of this paper. A special remark is that reduced RS in unrolled loops is not an increasing function. That is, if we reduce the RS to $\mathcal{R}_1 > R$ in the loop unrolled n times, and to $\mathcal{R}_2 > R$ in the loop unrolled $n+1$ times, this does not necessary mean that $\mathcal{R}_1 \leq \mathcal{R}_2$ (see Livermore-loop23 in Figure 14). The explanation is that as more independent nodes are available in a DAG, the more serialization opportunities are possible. Consequently, this results in more freedom and more choices for our heuristics.

9.3 ILP Loss after RS Reduction

In this last section, we study the ILP lost due to RS reduction. We evaluate the maximal theoretical ILP of a DAG $G = (V, E, \delta)$ as:

$$ILP(G) = \frac{|V|}{CriticalPath(G)}$$

The ratio used for expressing the ILP loss is

$$\frac{\text{original ILP} - \text{new ILP}}{\text{original ILP}}$$

We start by examining the efficiency of the value serialization heuristics in terms of ILP loss.

9.4 Optimal versus Approximated ILP Loss

Let us examine the ILP loss in our experiments. Results can be decomposed into five families, depending on the obtained RS and ILP loss after reduction. We denote by $\overline{RS}$ and $\overline{ILP}$ the RS reduction and ILP loss resulting from optimal intLP programs; we denote by $\overline{RS^*}$ and $\overline{ILP^*}$ the RS reduction and ILP loss resulting from our heuristics. Then, the five families of results are the following.

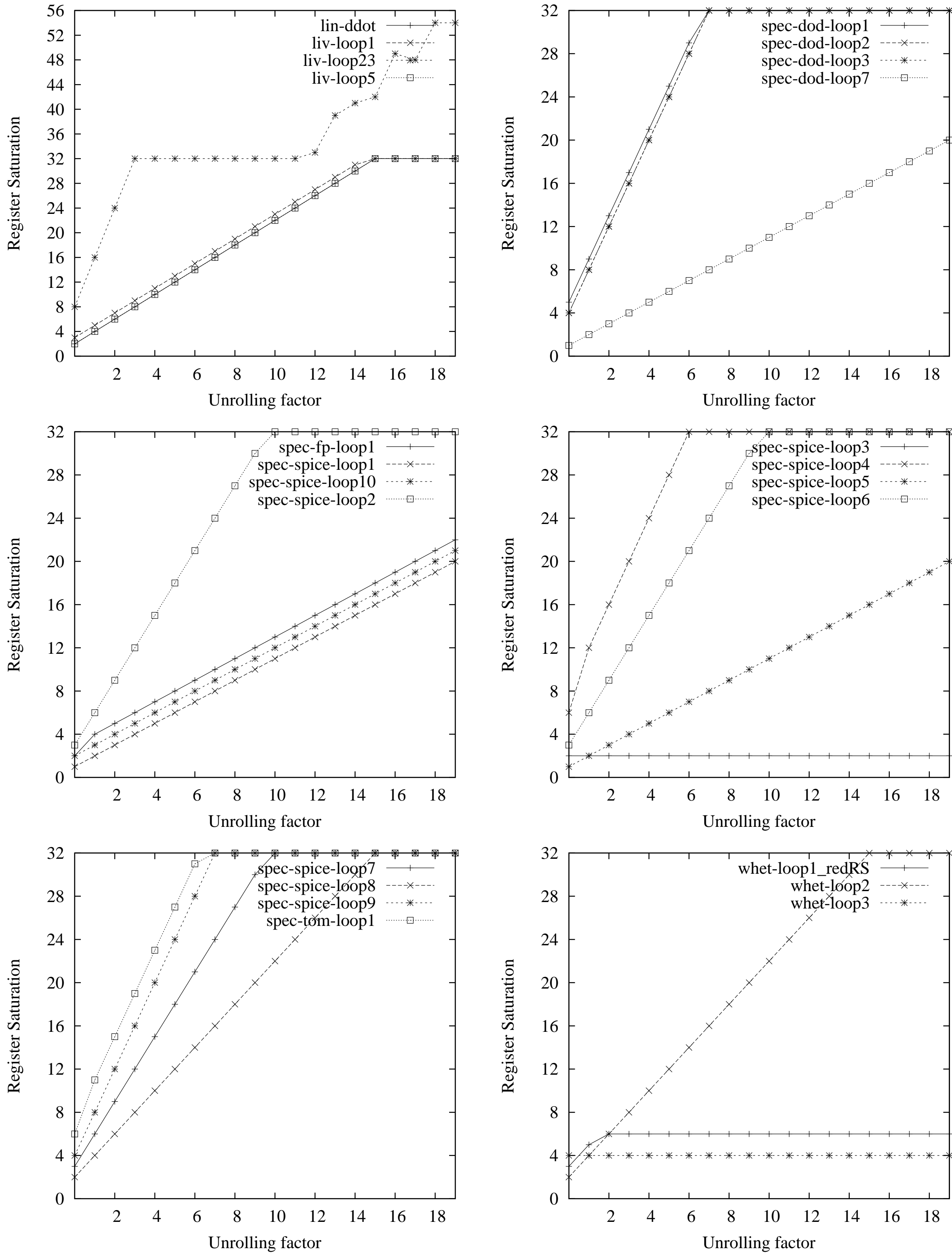


Figure 14. RS Reduction in Unrolled Loops ($\mathcal{R} = 32$)

1. In the case where $\overline{RS} = \overline{RS^*}$, our algorithm succeeds in optimally reducing RS. Then, the ILP loss may be:
   (a) $\overline{ILP} = \overline{ILP^*}$ (family 1). Our algorithm succeeds in optimally reducing RS with the optimal ILP loss. 72.22% of all the results belong to this family.
   (b) $\overline{ILP} < \overline{ILP^*}$ (family 2). Our algorithm succeeds in optimally reducing RS but with sub-optimal ILP loss. 18.5% of all the results belong to this family.
   (c) $\overline{ILP} > \overline{ILP^*}$ is not possible.
2. In the case where $\overline{RS} > \overline{RS^*}$, our algorithm did not succeed in optimally reducing RS. Then, the ILP loss may be:
   (a) $\overline{ILP} = \overline{ILP^*}$ (family 3). Our algorithm has sub-optimal RS reduction but optimal ILP loss. 4.63% of all the results belong to this family.
   (b) $\overline{ILP} < \overline{ILP^*}$ (family 4). Our algorithm has sub-optimal RS reduction with sub-optimal ILP loss. Less than 1% of all the results belong to this family.
   (c) $\overline{ILP} > \overline{ILP^*}$ (family 5). 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, it has extra registers which allow more ILP. 3.7% of all the results belong to this family.
3. The case where $\overline{RS} < \overline{RS^*}$ is impossible because our heuristics compute a valid RS^*.

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

Having established the efficiency of value serialization, we use it to study ILP loss in unrolled loops.

9.5 ILP Loss after RS reduction in Unrolled Loops

We unroll the loops up to 20 times to get larger DAGs (up to 400 nodes). We try to maintain their RS under 32 FP registers. Figure 15 plots ILP loss according to unrolling degree. In most cases, our heuristics do not produce a loss of ILP, *i.e.*, critical paths do not increase. However, in some cases, ILP loss exceeds 60% (the case of spec-spice-loop8) in order to maintain a RS under 32.

As in the RS reduction experiments, the ILP loss is not an increasing function. The explanation is that the more independent nodes are available in the DAG, the more lifetime interval serialization opportunities are possible. Our heuristics have more freedom to choose the best interval serialization that minimizes critical path growth. We note that, in these experiments, some operations have long specified latencies (up to 17 for an FP division). These long latencies can dramatically increases the critical path, since we may introduce serial edges that merge two long paths.

Before concluding, we wish to argue that the RS approach is a better way to satisfy register constraints before ILP scheduling than existing register need minimization approaches.

10 Related Work and Discussion

The literature contains a lot of techniques for minimizing the register requirement in superscalar (sequential) codes that are sensitive to ILP scheduling [2, 18, 20, 23, 25, 26]. Others prefer to combine ILP scheduling with register allocation [5, 8, 16, 24, 28]. 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 number of edges added to the DAG, as previously done by Berson [6]. Minimizing the register requirement is an inherently worse technique than saturating the register requirement for many reasons, which we explain below.

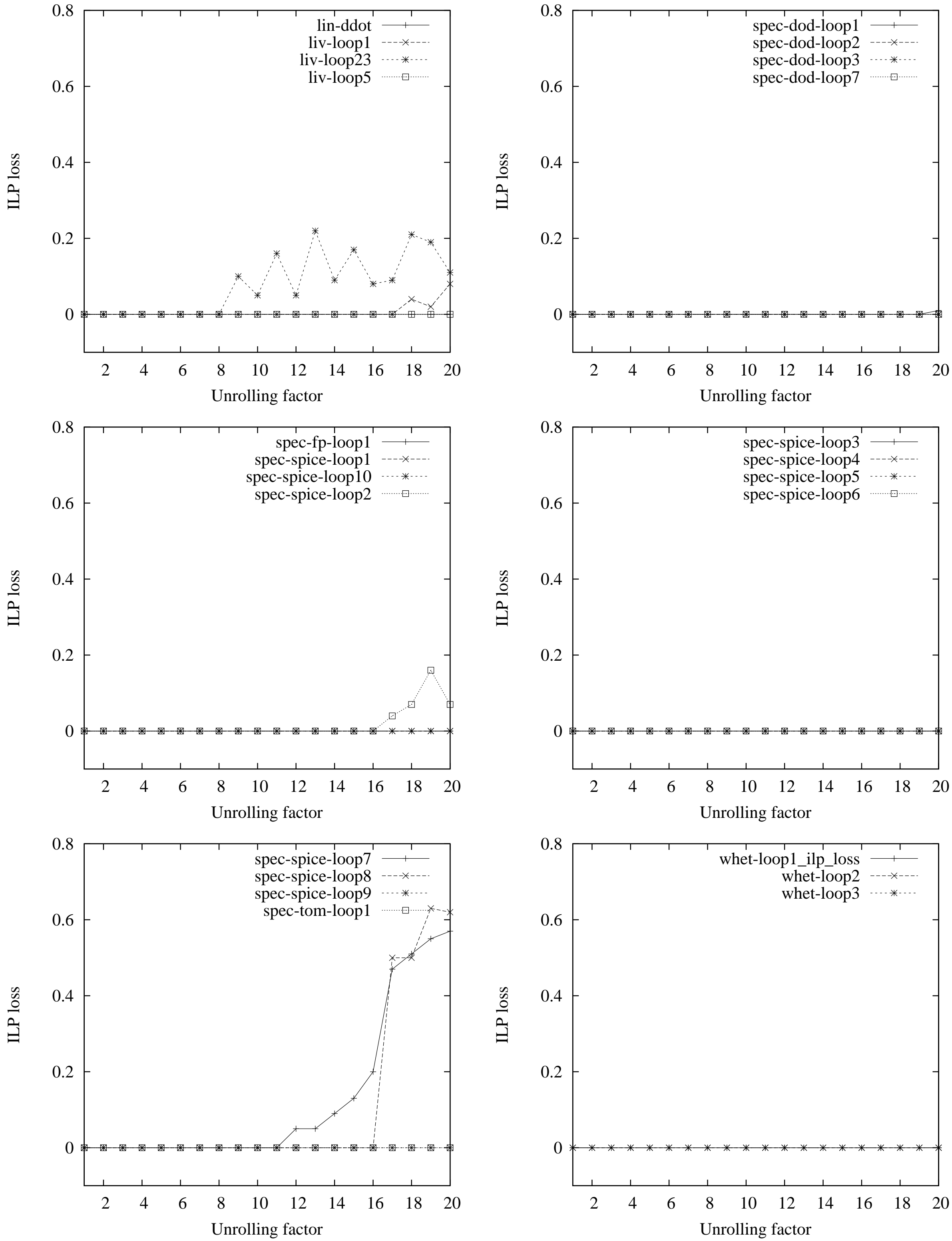


Figure 15. ILP Loss in Unrolled Loops ($\mathcal{R} = 32$)

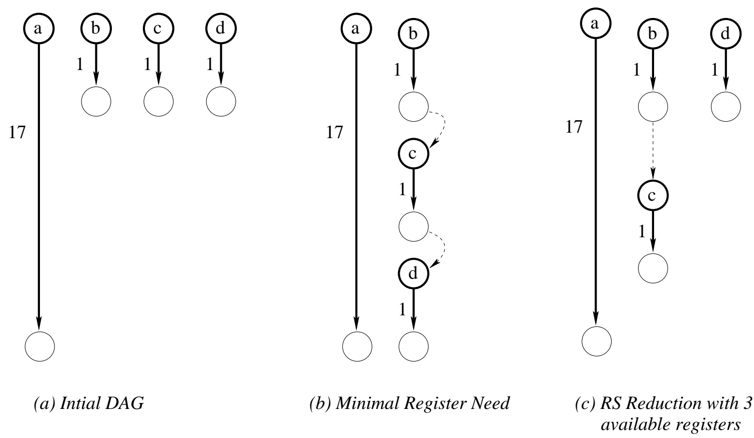


Figure 16. RS Reduction vs. Minimal Register Requirement

Case where register constraints are obsolete Given a DAG, we do not need to add serial edges if the RS does not exceed the number of available registers. Unfortunately, the minimization approach adds extra edges if the register requirement can be further reduced, even if RS does not exceed the limit. For instance, look at Figure 16, where bold circles are the values to be stored in registers and bold edges are the flow dependences. The initial DAG has 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 not modified before the scheduling pass. However, with a minimization approach, the new DAG in Part (b) is restricted to not require more than 2 registers[5], regardless the number of available registers. The DAG in Part (b) is more restrictive than the initial DAG, which is left unmodified by the RS analysis pass.

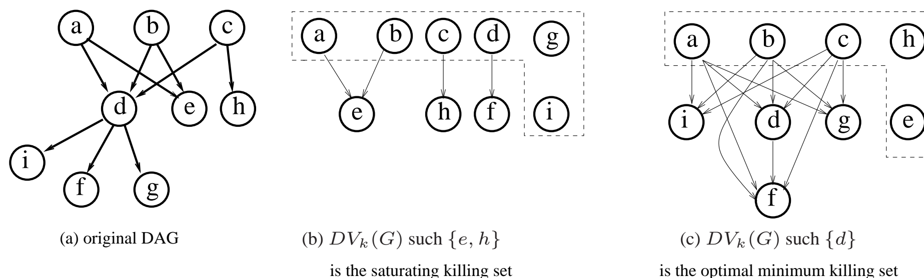
How many edges are introduced If the inherent data dependences of a DAG produce restrictive register pressure for an ILP scheduler (when RS exceeds the number of available registers), the minimization approach adds more edges than the RS reduction approach. This is because our method introduces only the necessary number of edges to reduce RS below the register limit. However, the minimization approaches tries to reduce the register need to the lowest possible level. This is not an appropriate approach, since it does not fully utilize the available registers. For instance, look at Figure 16 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 fewer serialization edges than those produced by the minimization approach presented in Part (b). Using the RS approach, the final allocator could use 1, 2 or 3 registers depending on the schedule. Using a register minimization approach, the scheduler could use only 1 or 2 registers. Hence, the RS approach helps the scheduler make better use of the available registers.

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

Our methods apply for explicit reading/writing offsets Our DAG and processor model allows for explicit delays when reading from and writing into registers. Thus, our method is more generic than existing techniques, and can be applied to superscalar, VLIW and EPIC architectures. For the last two cases, special care must be taken when reducing RS: we must prohibit non-positive cycles in the resulting 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),

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



(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

Figure 17. URSA Drawback

a static data dependence analysis may result in some values with more than one definition because it cannot determine which execution path is taken. We show in [30] 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.

Comparison to URSA Our work is an extension to URSA [6, 7]. Their minimum killing set technique tries to saturate the register requirement in a DAG by keeping the values alive as long as possible: the authors proceed by keeping as many children in a bipartite component alive as possible by computing the minimum set which kills 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. Figure. 17 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 function is an important condition to compute the RS and unfortunately is not included in URSA. We showed in Section 3 that invalid killing functions exist. So, the proof in [7] about the NP-completeness of RS computation is incomplete, since they did not prove the validity of the computed killing function. Finally, the URSA DAG model did not differentiate between the types of values and did not take into account delays in reading from and writing into the registers file.

Resource constraints Our experimental results are presented in the form of joint statements about critical path length and register requirement. Can anything formal be said about machines with finite resources ? Since our techniques assume infinite resources, it is theoretically possible that edges inserted to decrease register pressure might lead to unbalanced functional unit usage. Thus, edges might accidentally dictate bursts of all integer, all memory, or all floating point operations.

Let us answer this possible limitation. First, our work focus on data dependence graphs. Thus, a schedule can certainly be found on a machine with finite resources. Reporting resource conflicts at the graph level can only be done with simple resource descriptions (no structural hazards, *i.e.*, a FU is used during a contiguous interval of time), as done by Berson in [6] and Pinter in [26]. This strategy gives exactly the same solution as scheduling under resource and register constraints, *i.e.*, it is nothing but a combined approach of scheduling and register allocation. Second, the FU usage may be decreased especially if we try to minimize the register requirement. In our framework, we saturate the register requirement, thus the RS concept helps us reduce the number of serialization edges added to the DAG. Third and last, we give priority to register constraints over ILP scheduler (but we are still sensitive to this later) because we believe that spill code is more damaging to performance than a weak ILP extraction.

The methods of [6, 26] combine resource and register constraints. Their methods are only studied for superscalar codes with a unique register type, while our method works for VLIW, EPIC and superscalar codes with multiple register types. Furthermore, we did not read any experimental results that highlight if the heuristics of [6, 26] are near or far from the optimal, while we propose nearly optimal heuristics.

11 Conclusion

In this paper, we formally study the register saturation (RS) notion to manage register pressure in acyclic data dependence graphs (DAGs). RS helps to avoid inserting spill code before instruction scheduling and register allocation steps. We believe that register constraints must be taken into account before ILP scheduling, but by using the RS concept instead of the existing

strategies that minimize the register need. Otherwise, the subsequent ILP scheduler is restricted even if enough registers exist.

We give many fundamental results regarding the RS computation. First, we prove that choosing an appropriated unique killer is sufficient to saturate the register need. Second, we prove that fixing a unique killer per value allows to optimally compute the register saturation with polynomial time algorithms. If a unique killer is not fixed per value, we prove that computing the register saturation of a DAG is NP-complete in the general case (except for expression trees for instance). An exact formulation using integer programming and an efficient approximate algorithm are presented. Our formal mathematical modeling and theoretical study enable us to give nearly optimal heuristics.

Our experiments show that register constraints may be obsolete in many codes, and can therefore be ignored in order to simplify the instruction scheduling process. The heuristics we use manage to reduce RS in most cases while some ILP is lost in few DAGs.

If RS exceeds the number of available registers, we must reduce it while minimizing the increase to the critical path. We prove that this is an NP-hard problem. An optimal exact RS reduction method based on integer programming is presented, as well as an efficient approximate algorithm. If we assume writing offsets (such as those in VLIW and EPIC codes), some optimal solutions may require the insertion of non-positive cycles in the original DAG. These cycles 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.

The size complexity of our intLP formulations depends only the size of the input DAG (quadratic on the number of edges and nodes). This is better than the size complexity of the existing technique in the literature that model register constraints [1, 13, 14]. Indeed, these exact intLP systems have a size complexity that depends on a worst-case total schedule time factor, which does not depend on the size of the input DAG. Thus, the resulting size complexity is pseudo-polynomial, and not polynomial as in our intLP system.

An important problem (left for a future work) is the insertion of minimal spill code 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.

Acknowledgement

We would like to thank Alain Darte from École Normale Supérieur de Lyon and François Thomasset from INRIA-Rocquencourt for their help to improve this work.

References

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

[2] W. Ambrosch, M. A. Ertl, F. Beer, and A. Krall. Dependence-Conscious Global Register Allocation. *Lecture Notes in Computer Science*, 782:129–??, 1994.

[3] 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. Proceedings of Programming Language Design and Implementation (PLDI'97).

[4] 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.

[5] D. Bernstein, J. M. Jaffe, and M. Rodeh. Scheduling Arithmetic and Load Operations in parallel with No Spilling. *SIAM Journal on Computing*, 18(6):1098–1127, Dec. 1989.

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

[7] D. A. 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.

[8] T. S. Brasier, P. H. Sweany, S. J. Beaty, and S. Carr. CRAIG: A Practical Framework for Combining Instruction Scheduling and Register Assignment. In *Parallel Architectures and Compilation Techniques (PACT '95)*, 1995.

[9] 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.

[10] G. J. Chaitin. Register Allocation and Spilling via Graph Coloring. *ACM SIG-PLAN Notices*, 17(6):98–105, June 1982. Proceedings of the SIGPLAN '82 Symposium on Compiler Construction.

[11] P. Crawley and R. P. Dilworth. *Algebraic Theory of Lattices*. Prentice Hall, Englewood Cliffs, 1973.

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

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

[14] C. Eisenbeis and A. Sawaya. Optimal Loop Parallelization under Register Constraints. In *Sixth Workshop on Compilers for Parallel Computers CPC'96.* , pages 245–259, Aachen - Germany, Dec. 1996.

[15] W. fen Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM Latencies with an Integrated Memory Hierarchy Design. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, Nuevo Leone, Mexico, Jan. 2001.

[16] 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.

[17] R. S. Garfinkel and G. L. Nemhauser. *Integer Programming*. John Wiley & Sons, New York, 1972. Series in Decision and Control.

[18] 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.

[19] R. Govindarajan, E. R. Altman, and G. R. Gao. Minimizing Register Requirements under Resource-Constrained Rate-Optimal Software Pipelining. In *MICRO27*, pages 85–94, Dec. 1994.

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

[21] W. Jalby, C. Lemuet, and S.-A.-A. Touati. Improving Load/Store Queues Usage in Scientific Computing. In *Proceedings of the International Conference on Parallel Processing (ICPP'04).*, pages 38–45, Montréal, Canada, Aug. 2004. IEEE.

[22] W. Jalby, C. Lemuet, and S.-A.-A. Touati. An Efficient Memory Operations Optimization Technique for Vector Loops on Itanium 2 Processors. *Conucurrency and Computation: Practice and Experience*, 2004 (to appear). Wiley Interscience.

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

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

[25] 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.

[26] S. S. Pinter. Register Allocation with Instruction Scheduling: A New Approach. *SIGPLAN Notices*, 28(6):248–257, June 1993. Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation.

[27] M. Poletto and V. Sarkar. Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, Sept. 1999.

[28] R. Silvera, J. Wang, G. 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, Nov. 1997. IEEE Computer Society Press.

[29] S.-A.-A. Touati. Register Saturation in Superscalar and VLIW Codes. In *Proceedings of The International Conference on Compiler Construction*, Lecture Notes in Computer Science. Springer-Verlag, Apr. 2001.

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

[31] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Modulo Scheduling with Integrated Register Spilling for Clustered VLIW Architectures. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO-34)*, pages 160–169, Austin, Texas, Dec. 2001.