Maximizing for Reducing the Register Need in Acyclic Schedules

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

ABSTRACT

In this article, we give our heuristics to carry out the register allocation before acyclic code scheduling with the respect of the critical path. We proceed by "maximizing" the register need instead of minimizing it [15]. Intuitively, maximizing the register need implies a minimization of the false dependencies introduced by the register reuse. If the maximal register requirement exceeds the number of available register $\mathcal{R}$, we add fictitious arcs in the DAG to prohibit any schedule from needing more than $\mathcal{R}$ registers.

1. INTRODUCTION AND MOTIVATION

In Instruction Level Parallelism (ILP), code scheduling and register allocation are two major tasks for code optimization. If register allocation is carried out before scheduling, false dependencies could be introduced because of register reuse, producing a negative impact on available ILP exposed to the scheduler. If scheduling is carried out before, spill code might be introduced because of an insufficient number of registers. A better approach is to combine and make code scheduling and register allocation interact with each other. The aim is first to try to construct a schedule with a limited number of values simultaneously alive in order to avoid spill code when allocating a physical register to each value. Second, the register allocation should not increase the critical path when deciding for register reuse. The main problem with such an approach is its complexity and a combined approach makes the register allocation and scheduling heuristics correlated.

In this article, we present our contribution to avoiding an excessive number of values simultaneously alive for all valid schedules of a DAG, previously studied in the URSA framework [3] and proven insufficient in [15, 16]. The complete version of this work is given in [16, 14], where the reader can find theorems and their proofs asserted in this paper. Our approach is to analyze a DAG to deduce the maximal register need for all schedules. We call this limit **register saturation** (RS) because the register need (minimum number of registers needed to avoid spill code) can reach this limit but never exceed it.

There are two major reasons for dissociating registers constraints before code scheduling. First, we believe that the register allocation has more effects on code performance comparing to code scheduling, and hence we must give a higher priority to the register allocation : a disturbed register allocation process introduces avoidable spill code which may produce cache misses. If the register saturation does not exceed the number of available registers, we can do code scheduling without any interaction with register allocation, thereby reducing the compile time complexity. Second, the registers constraints models in modern processors are simpler than the resources constraints because there are few possible configurations : the number of registers can be 32, 64 or 128, and the register types that we generally focus on are classified as integer, float, guards, etc. However, resource constraints are less comparable. Each processor has its own properties. This obliges us to redo code scheduling for each target processor. With our approach, we can allocate registers in a DAG such that it satisfies the registers constraints for a whole set of processors with an number of available registers less or equal to register saturation. Our approach is portable since lot of compilers (gcc for instance) use the same first steps of compilation (parsing, intermediate code generation and optimization), but a different ILP backend for code optimization and generation for a the target processor. Our approach can be applied on the intermediate code level to decide for register allocation without increasing the critical path if possible. After that, the ILP backend can schedule and optimize the code for a specific target processor.

2. DAG MODEL

A DAG $G = (V, E, \delta)$ in our study represents a direct acyclic graph which defines data dependences between operations. Each operation u has a strictly positive latency $lat(u)$. The DAG is then defined by V its set of operations, E its set of arcs, and δ the delay function defined on the arcs. A schedule σ of G is a positive function that gives an integer execution (issue) time for each operation :

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

We note by $\Sigma(G)$ the set of all valid schedules for G, and $\overline{\sigma} = \max u\sigma(u) + lat(u)$ the last execution step.

Since writing and reading into and from registers could be

delayed from the beginning of the operation schedule time (VLIW case), we define the two delay functions δ_r and δ_w such that $\sigma(u) + \delta_w(u)$ is the instant when the operation u writes into the register, and $\sigma(u) + \delta_r(u)$ the instant when it reads from a register.

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

- $V_R \subseteq V$ is the subset of operations which define a value of the type under consideration, we simply call them **values**. We assume that at most one value of the type considered can be defined by an operation ;
- $E_R \subseteq E$ is the subset of arcs representing true dependencies through a value of the type considered. We call them **flow** arcs.
- $E_S = E - E_R$ are called **serial** arcs.

Notation and Definitions on DAGs

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

- $\Gamma_G^-(u) = \{v \in V / (v, u) \in E\}$ predecessors of u ;
- $\forall e = (u, v) \in E \quad source(e) = u \wedge target(e) = v$;
- $\forall u, v \in V : \ u < v \Longleftrightarrow \exists$ 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** ;
- $\forall u \in V \ \downarrow u = \{v \in V / v = u \vee u < v\}$ u's descendants including u.

We give also the following definitions :

- $A \subseteq V$ is an antichain in $G \Longleftrightarrow \forall u, v \in A \quad u || v$
- AM is a **maximal** antichain $\Longleftrightarrow \forall A$ antichain in G $|A| \leq |AM|$;
- the **extended** DAG $G\backslash^{E'}$ of G generated by the arcs set $E' \subseteq V^2$ is the graph obtained from G after adding the arcs in E'. As a consequence, any valid schedule of G' is necessarily a valid schedule for G.

3. REGISTER SATURATION

In this section, we study the register saturation notion from a theoretical perspective. We begin by recalling what the register need of a schedule is.

3.1 Register Need of a Schedule

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

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

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 u's consumers whose reading time is equal to u's killing date are called killers of u. We assume that a value written at instant t in a register is available one step later. Then, the **lifetime interval** L_u^σ of a value u according to σ is $]\sigma(u) + \delta_w(u), kill_\sigma(u)]$. Having all values lifetime intervals, the register need of σ, noted $RN_\sigma(G)$, is the maximum number of values simultaneously alive, which is the minimum number of registers needed to avoid spill code :

3.2 Register Saturation Problem

The register saturation is the maximal register need for all valid schedules $\sigma \in \Sigma(G)$:

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

We call σ a **saturating schedule** *iff* $RN_\sigma(G) = RS(G)$. In this section, we study how to compute $RS(G)$. This problem comes down to answering the question "*which operation must kill this value ?*". We have proven in [14] that to maximize the register need, looking for only one suitable killer of a value is sufficient rather than looking for a group of killers : for any schedule that assigns more than one killer for a value, we can obviously build another schedule with at least the same register need such that this value is killed by only one consumer.

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

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

One can check that all operations in $pkill_G(u)$ are parallel in G. Any operation that does not belong to $pkill_G(u)$ can never kill the value u. We have proven in [16] the following assertion :

$$\exists \sigma \in \Sigma(G) \ v \text{ is one of the killers of } u \Longleftrightarrow v \in pkill_G(u)$$

A **potential killing DAG** of G, noted $PK(G) = (V, E_{PK})$, is built to model the potential killing relations between op-

erations, where:

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

There may be more than one operation candidate for killing a value. Let us begin by assuming a function k that enforces an operation $k(u) = v \in pkill_G(u)$ to be the unique killer of $u \in V_R$ in order to build a saturating schedule. There is a family of schedules that ensures this assertion. To define them, we extend G by new serial arcs that enforce all potential killing operations of each value u to be scheduled before $k(u)$. The extended DAG associated to k noted $G_{\rightarrow k} = G\backslash^{E_k}$ is defined by inserting new arcs entering $k(u)$ from all the other potential killers:

$$E_k = \Big\{e = (v, k(u))/\delta(e) = \delta_r(v) - \delta_r\big(k(u)\big) + 1\Big\}$$

Then, any schedule $\sigma \in \Sigma(G_{\rightarrow k})$ ensures that $\forall u \in V_R$

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

The condition of the existence of such a schedule defines the condition of a **valid killing function**:

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

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

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

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

Any arc (u,v) in $DV_k(G)$ means that u's lifetime interval is always before v's lifetime interval according to any schedule of $G_{\rightarrow k}$. This definition permits us to state in the following theorem that the register need of any schedule of $G_{\rightarrow k}$ is always less than or equal to a maximal antichain in $DV_k(G)$.

THEOREM 3.1. *Given a DAG $G = (V, E, \delta)$ and a valid killing function k then:*

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

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

Theorem 3.1 allows us to rewrite the register saturation formula as

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

Unfortunately, the problem of finding such a killing function is NP-complete[16].

3.3 A Heuristic for Computing RS

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

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

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

- $E_{cb} \subseteq E_{PK}$ arcs are potential killing relations;
- $S_{cb} = \{s \in V_R/\ \exists e \in E_{cb} \wedge s = source(e)\}$ parent values;
- $T_{cb} = \{t \in V/\exists e \in E_{cb} \wedge s = target(e)\}$ children nodes;
- $cb = (S_{cb}, T_{cb}, E_{cb})$ is bipartite:

A bipartite decomposition of the potential killing graph $PK(G)$ is the set

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

There exists only one bipartite decomposition and

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

A saturating killing set $SKS(cb)$ of a bipartite component $cb = (S_{cb}, T_{cb}, E_{cb})$ is a subset $T'_{cb} \subseteq T_{cb}$ such that if we choose a killing operation from this subset, then we get a maximal number of descendant values of children in T_{cb} simultaneously alive with parent values in S_{cb}:

1. killing constraints: all parents in S_{cb} must be potentially killed by at least one child in T'_{cb}

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

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

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

Unfortunately, computing $SKS(cb)$ is also NP-complete [16].

Algorithm 1 Greedy-k: a heuristics for computing the RS

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

A Heuristic for Finding a SKS

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

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

The first case enables us to select the child which covers the most non killed parents with the minimum descendant values. If there is no descendant value, then we choose the child that covers the most non killed parents. Algorithm 1 gives a modified greedy heuristic that searches for an approximation SKS^* and computes a killing function k^* in polynomial time. Our heuristics have the following proven properties [16] :

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

4. REDUCING REGISTER SATURATION

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

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

We have proven in [14] that this problem is NP-hard. In this section we present a heuristic that adds serial arcs to prevent some saturating values in AM_k (according to a saturating killing function k) from being simultaneously alive for any schedule $\sigma \in \Sigma(\overline{G})$. Also, we must take care not to increase the critical path if possible.

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

- if $v \in pkill_G(u)$ then add the serial arcs $\{e = (v', v) / v' \in pkill_G(u) - \{v\}$ with $\delta(e) = \delta_r(v') - \delta_w(v)\}$
- else add the serial arcs $\{e = (u', v) / u' \in pkill_G(u) \wedge \neg(v < u')$ with $\delta(e) = \delta_r(u') - \delta_w(v)\}$

According to this definition, $u \rightarrow v$ cannot introduce cycles but may be impossible if we do not want to violate the DAG property. We call a value serialization $u \rightarrow v$ **admissible** *iff*: $\forall v' \in pkill_G(u) : \quad \neg(v < v')$.

We use this information to build, in a subsequent algorithm, the set of all admissible value serializations in order to choose the best candidate. For this aim, we define a cost function $\omega(u \rightarrow v) = (\omega_1, \omega_2)$ that selects the best candidate [16, 15], such that :

- $\omega_1 = \mu_1 - \mu_2$ which is the prediction of the reduction obtained within the saturating values if we carry out this value serialization, where
  - μ_1 is the number of saturating values serialized after u if we carry out the serialization ;
  - μ_2 is the predicted number of u's descendant values that can become simultaneously alive with u ;
- ω_2 is the increase in the critical path.

Our heuristic is presented in Algorithm 2. It iterates value serializations between saturating values until we get the limit $\mathcal{R}$ or until no more serializations are possible (either no more admissible value serializations or none is expected to reduce RS). One can check that if the 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 register saturation, and thus limits the register need. Introducing and minimizing spill code is another NP-complete problem studied in [6, 2, 1, 7] and not addressed in this work. All formulas of μ_1, μ_2 and ω_2 are defined in [16].

5. REGISTER ALLOCATION

In this section, we assume a DAG $G = (V, E, \delta)$ such that $RS(G) \leq \mathcal{R}$. We build a register allocation for this DAG as follows :

Algorithm 2 Value Serialization Heuristic

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

1. search for a saturating killing function k;
2. build $G_{\rightarrow k}$ the DAG associated to k. Any value $u \in V_R$ is killed by one node $k(u)$.
3. build the disjoint value DAG $DV_k(G_{\rightarrow k})$. According to Theorem 3.1, any chain in this DAG is a list of non interfering lifetimes in any schedule of $G_{\rightarrow k}$;
4. build a minimal chain decomposition [9] for $DV_k(G_{\rightarrow k})$;
5. allocate the same register to all the values in the same chain, but different registers for two different chains. According to Dilworth Theorem, we need $RS(G) \leq \mathcal{R}$ registers since a minimal chain decomposition is equal to the cardinality of the maximal antichain.

6. EXPERIMENTATION

We implemented the register saturation analysis using the LEDA framework [11]. We carried out our experiments on various loops taken from various benchmarks (livermore, whetsone, spec-fp, etc.). We focus for instance in these codes on floating point registers. The first experimentation is devoted to checking Greedy-k efficiency. For this purpose, we defined and implemented in [14] integer linear programming models to compute optimal register saturation for DAGs. We use cplex [8] to resolve these linear programming models. The total number of experimented DAGs is 180 (obtained by unrolling the loops with several unrolling factors), where the number of nodes goes up to 120 and the number of values goes up to 114. Experimental results show that our heuristics give quasi-optimal solutions. The worst experimental error is 1, which means that the optimal register saturation is in worst case greater by one register than the one computed by Greedy-k. But since Greedy-k always produce valid killing function, we are sure that we can build a register allocation with $R = RS^*(G) \leq RS(G)$ available registers.

The second experimentation is devoted to checking the efficiency of the value serialization heuristics in order to reduce register saturation. We also defined and implemented in [14] integer linear programming models to compute optimal reduction of register saturation with a minimum critical path increase (NP-hard problem). The total number of experimented DAGs is 144, where the number of nodes goes up to 80 and the number of values goes up to 76. In almost all cases, our heuristics manages to get optimal solutions. Optimal reduced RS was in the worst cases less by one register than our heuristics results. Since RS computation in the value serialization heuristics is done by Greedy-k, we add its worst experimental error (1 register) which leads to a total maximal error of two registers.

Since our strategies result in good efficiency, we use them to carry out register allocation in DAGs. Experimentation on only loop bodies shows that the 1 to 8 registers are sufficient. We have unrolled these loops with different unrolling factors going up to 20 times. The aim of such unrolling is to get large DAGs, increase the registers pressure and expose more ILP to hide memory latencies. We carried out a wide range of of experiments to study the RS, reducing RS and ILP loss with various limits of available registers (going from 1 up to 64). We experimented 720 DAGs where the number of of nodes goes up to 400 and the number of values goes up to 380. Full results are detailed in [16, 14].

7. RELATED WORK

Combining code scheduling and register allocation in DAGs has been studied in many works. All the techniques described in [10, 4, 13, 5, 12] use their heuristics to build an optimized schedule without exceeding a certain limit of values simultaneously alive in order to keep these values in physical registers. Our work is an extension to the URSA framework presented in [3]. The minimum killing set technique tries to saturate the register need in a DAG by keeping values alive as long as possible: the authors proceed by keeping as many children alive as possible by computing the minimum set that kills all the parent's values. First, since the authors do not formalize the register saturation 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, see [16].

8. CONCLUSION AND FUTURE WORK

In our work, we carry out register allocation before scheduling with the respect of critical path. A DAG in our model represents scheduling constraints for both superscalar and VLIW architectures with different types of registers. Our DAG model is sufficiently general to meet all current architecture properties (RISC or CISC), except for some architectures which support issuing dependent instructions at the same clock cycle, which would require representation using null latency. We think that this restriction should not be a major drawback. In the future, we will extend our work to loops. We will study how to compute and reduce register saturation in the case of cyclic schedules like software pipelining (SWP).

9. REFERENCES

[1] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew O'Keefe. Spill Code Minimization via Interference Region Spilling. *ACM SIG-PLAN Notices*, 32(5):287–295, May 1997.

[2] David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron 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.

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

[4] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating Register Allocation and Instruction Scheduling for RISCs. *ACM SIGPLAN Notices*, 26(4):122–131, April 1991.

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

[6] David Callahan and Brian 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.

[7] G. J. Chaitin. Register allocation and spilling via graph coloring. *ACM SIG-PLAN Notices*, 17(6):98–105, June 1982.

[8] CPLEX Optimization, Inc., Incline Village, Nevada. *Using the CPLEX Callable Library and CPLEX Mixed Integer Library*, 1993.

[9] Peter Crawley and Robert P. Dilworth. *Algebraic Theory of Lattices*. Prentice Hall, Englewood Cliffs, 1973.

[10] 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.

[11] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform of Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, England, January 1999.

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

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

[14] Sid-Ahmed-Ali Touati. Optimal Register Saturation in Acyclic Superscalar and VLIW Codes. Research Report, INRIA, November 2000. ftp.inria.fr/INRIA/Projects/a3/touati/optiRS.ps.gz.

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

[16] Sid-Ahmed-Ali Touati and François Thomasset. Register Saturation in Data Dependence Graphs. Research Report RR-3978, INRIA, July 2000. ftp.inria.fr/INRIA/publication/publi-ps-gz/RR/RR-3978.ps.gz.