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 registers $R$, we add fictitious arcs in the DAG to prohibit any schedule from needing more than $R$ 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 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 back-end 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 dependencies 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 $\delta$ the delay function defined on the arcs. A schedule $\sigma$ of $G$ is a positive function that gives an integer execution (issue) time for each operation:

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

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

Since writing and reading into and from registers could be
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

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

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

$$\forall u \in V_R \quad \text{killing date of } u = \max_{v \in \text{Cons}(u)} (\sigma(v) + \delta_u(v))$$

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 $f$ in a register is available one step later. Then, the lifetime interval $L^*_u$ of a value $u$ is the maximum value of the register need of $\sigma$, noted $RN_u(G)$, 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 $\sigma \in \Sigma(G)$:

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

We call $\sigma$ a saturating schedule if $RN_u(G) = RS(G)$. 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 $\text{Cons}(u)$ are impossible killers for the value $u$. If $v_1, v_2 \in \text{Cons}(u)$ and $\exists$ a path $(v_1 \ldots v_2)$, $v_2$ is always scheduled before $v_2$ with at least $\text{lat}(v_1)$ processor cycles. Then $v_2$ can never be the last read of $u$. We can consequently deduce which consumers can “potentially” kill a value (possible killers). We note $\text{kills}(u)$ the set of operations which can kill a value $u \in V_R$:

$$\text{kills}(u) = \{ v / v \in \text{Cons}(u) \cap v \cap \text{Cons}(u) = \{v\} \}$$

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

$$\forall \sigma \in \Sigma(G) \quad \text{there is one of the killers of } u \iff \exists v \in \text{kills}(u)$$

A potential killing DAG of $G$, noted $PK(G) = (V, E_{PK})$, is built to model the potential killing relations between op-
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;
- \( S_{cb} = \{ s \in V_R \mid \exists v \in E_{cb} \wedge s = source(v) \} \) parent values;
- \( T_{cb} = \{ t \in V \mid \exists v \in E_{cb} \wedge t = target(v) \} \) children nodes;
- \( cb = (S_{cb}, T_{cb}, E_{cb}) \) is bipartite.

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

\[
B(G) = \{ cb = (S_{cb}, T_{cb}, E_{cb}) \mid \forall v \in E_{PK} \exists cb \in B(G) : e \in E_{cb} \}
\]

There exists only one bipartite decomposition and

\[
\forall cb \in B(G), \forall s' \in S_{cb} \forall t' \in T_{cb} \forall s' \wedge t' \in PK(G)
\]

A saturating killing set \( SKS(cb) \) of a bipartite component \( cb = (S_{cb}, T_{cb}, E_{cb}) \) is a subset \( T_{cb}' \subseteq T_{cb} \) such that if we choose a killing operation from this subset, then we get a maximal number of descendant values of children in \( T_{cb} \) simultaneously alive with parent values in \( S_{cb} \):

1. killing constraints: all parents in \( S_{cb} \) must be potentially killed by at least one child in \( T_{cb}' \)
   \[
   \bigcup_{t \in T_{cb}'} \{ c \in E_{cb} \mid ts = c \} = S_{cb}
   \]
2. minimizing the number of descendant values of \( T_{cb}' \)
   \[
   \min \left| \bigcup_{t \in T_{cb}'} \{ c \in E_{cb} \mid ts = c \} \right|
   \]

Unfortunately, computing \( SKS(cb) \) is also NP-complete [16].
Algorithm 1 Greedy-k: a heuristic 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 $B(G)$ the bipartite decomposition of $PK(G)$.
for all bipartite component $cb = (S_0, T_0, E_0) \in B(G)$ do
  $X := S_0$ \{all parents are initially uncovered\}
  $Y := \phi$ \{initially, no cumulated descendant values\}
  $SK^*(cb) := \phi$
  while $X \neq \phi$ do 
    \{build the SKS for $cb$\}
    select the child $t \in T_0$ with the maximal cost $\rho_{X,Y}(t)$
    $SK^*(cb) := SK^*(cb) \cup \{t\}$
    $X := X - \Gamma_{cb}(t) \{\text{remove covered parents}\}$
    $Y := Y \cup \{t\}$ \{update the cumulated descendant values\}
  end while
  for all $t \in SK^*(cb)$ do \{in decreasing order of cost\}
    if $k^*(t) = \perp$ then \{kill non killed parents of $t$\}
      $k^*(t) := t$
    end if
  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 $\rho$ that enables us to choose the best candidate child. Given a bipartite component $cb = (S_0, T_0, E_0)$ and a set $Y$ of (cumulated) descendant values and a set $X$ of non (yet) killed parents, the cost of a child $t \in T_0$ is:

$$\rho_{X,Y}(t) = \begin{cases} \frac{\mu^*(t \cap X)}{|t \cap Y|} & \text{if } \exists t \cup Y \neq \phi \\ |t \cap X| & \text{otherwise} \end{cases}$$

The first case enables us to select the child that 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 $SK^*$ 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 $\implies$ Greedy-k computes an optimal RS.

4. REDUCING REGISTER SATURATION

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

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

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 $\delta(e) = \delta(v') - \delta(v)$\}
- else add the serial arcs $\{e = (u', v) / u' \in pkill_G(u) \cap \{v < u\} \}$ \{with $\delta(e) = \delta(u') - \delta(u)$\}

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: $\forall v' \in pkill_G(u) : -(v < v')$.

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
  - $\mu_1$ is the number of saturating values serialized after $u$ if we carry out the serialization;
  - $\mu_2$ is the predicted number of $u$'s descendant values that can become simultaneously alive with $u$;
- $\omega_2$ 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 $R$ 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 exists at the first iteration of the outer while loop. If it succeeds, then any schedule of $\overline{G}$ needs at most $R$ 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 $\mu_1, \mu_2$ and $\omega_2$ are defined in [16].

5. REGISTER ALLOCATION

In this section, we assume a DAG $G = (V, E, \delta)$ such that $RS(G) \leq R$. 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 $R$

$G := G$
compute $AM_b$, saturating values of $\overline{G}$;
while $|AM_b| > R$ do
construct the set $U$ of all admissible serializations between saturating values in $AM_b$ with their costs $(\omega_1, \omega_2)$;
if $\beta(u \to v) \in U/\omega_1(u \to v) > 0$ then {no more possible RS reduction}
exit;
end if
$X := \{(u \to v) \in U/\omega_2(u \to v) = 0\}$ {the set of value
serializations that do not increase the critical path}
if $X \neq \emptyset$ then
choose a value serialization $(u \to v)$ in $X$ with the minimum cost $R - \omega_1$;
else
choose a value serialization $(u \to v)$ in $X$ with the minimum cost $\omega_2$;
end if
carry out the serialization $(u \to v)$ in $\overline{G}$; compute the new saturating values $AM_b$ of $\overline{G}$;
end while

1. search for a saturating killing function $k$;
2. build $G \to k$ the DAG associated to $k$. Any $u \in V_R$ is killed by one node $k(u)$.
3. build the disjoint value DAG $DV_b(G \to k)$. According to
Theorem 3.1, any chain in this DAG is a list of non
interfering lifetimes in any schedule of $G \to k$;
4. build a minimal chain decomposition [9] for $DV_b(G \to k)$;
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 $RS(G) \leq R$
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 (liversmore,
whetstone, 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 reg-
ister 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 effi-
cency 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 re-
duction of register saturation with a minimum critical path
increase (NP-hard problem). The total number of exper-
imented DAGs is 144, where the number of nodes goes up to
80 and the number of values goes up to 76. In almost
cases, our heuristics manages to get optimal solutions.
Optimal reduced RS was in the worst cases less by one reg-
ister 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 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 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 de-
scribed 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 tech-
nique tries to saturate the register need in a DAG by keep-
ing 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 prob-
lem, 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 schedu-
ling with the respect of critical path. A DAG in our model
represents scheduling constraints for both supenacular
and VLIW architectures with different types of registers. Our
DAG model is sufficiently general to meet all current archi-
tecture properties (RISC or CISC), except for some archi-
tectures 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 regis-
ter saturation in the case of cyclic schedules like software
pipelining (SWP).
9. REFERENCES


