SIRA: Schedule Independent Register Allocation for Software Pipelining

Sid Ahmed Ali Touati, Christine Eisenbeis
A3 Project, INRIA Rocquencourt, 78153 Le Chesnay Cedex, France
Sid-Ahmed-Ali.Touati@inria.fr Christine.Eisenbeis@inria.fr

Abstract

The register allocation in loops is generally carried out after or during the software pipelining process. This is because doing the register allocation at first step without assuming a schedule lacks the information of interferences between values live ranges. The register allocator introduces extra false dependencies which reduces dramatically the original ILP (Instruction Level Parallelism). In this paper, we give a new formulation to carry out the register allocation before the scheduling process, directly on the data dependence graph by inserting some anti dependencies arcs (*reuse* edges). This graph extension is first constrained by minimizing the critical cycle and hence minimizing the ILP loss due to the register pressure. The second constraint is to ensure that there is always a cyclic register allocation with the set of available registers, and this for any software pipelining of the new graph. We give the exact formulation of this problem with linear integer programming.

1 Introduction

1.1 Background

The problem of minimizing the register requirement for vertical code[1] (single issue processors) is an old problem proven NP-complete in [Set75] for general DAGs. In the case where the DDG is a tree (arithmetic expression for instance), the optimal register allocation can be computed in polynomial complexity. The problem of minimizing the number of registers needed to evaluate an expression tree without spills was first resolved by Nakata [Nak67] and Redziejowski [Red69]. Sethi and Ullman extended that result in [SU70] to minimize the amount of spill code needed to evaluate an expression tree, given a fixed number of registers.

[1] No static ILP can be expressed by the generated code.

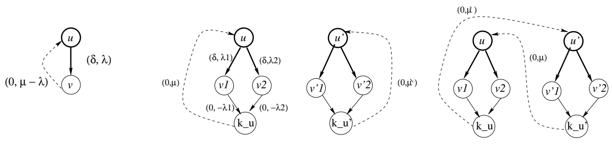
With the introduction of multiple issue processors (VLIW and superscalar processors), the register allocation became constrained by the total schedule time. Both the problems of minimizing the register requirement with a fixed total schedule time, or minimizing the total schedule time with a fixed number of registers was proven NP-hard [EGS95] , even for DAGs. A special case was described by Meleis in [MD99] where he gave a polynomial algorithm which produced an optimal schedule of a binary tree given a fixed number of registers : the tree could not contain unary operations, the latency of the operations must be 1, and the machine was restricted to issue no more than one memory operation and one arithmetic operation per time slot.

1.2 Motivation

This report address the problem of register pressure in cyclic data dependence graphs (DDGs), with multiple registers types and non unit assumed latencies. Our aim is to decouple the registers constraints and allocation from the scheduling process and analyse the trade-off between memory and parallelism constraints. The principal reason is that the register allocation process is more important, as an optimization issue, than the code scheduling. This is because the code performance is far more sensitive to the memory access than to the fine-grain scheduling : a cache miss inhibits the processor from achieving a high dynamic ILP, even if the scheduler has extracted it at compile time. Our approach is to take into account the registers constraints before the code scheduling without hurting the ILP or restricting the scheduler.

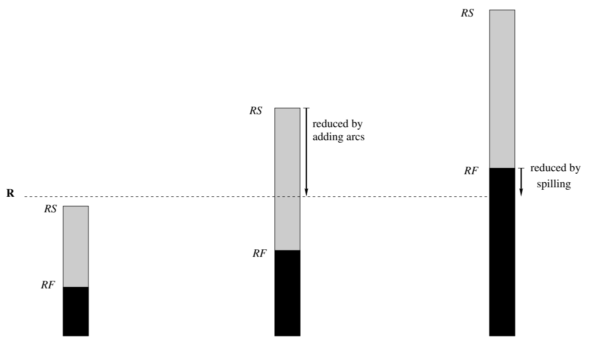
In our previous work [Tou01d, Tou01b], we have considered the register saturation problem for acyclic Data Dependency Graphs (DDG). Register saturation is based on analyzing the DDG and answering the question:

> What is the maximal number of registers required for any schedule of this code?



(a) Simple Reuse Circuit (b) Killing Tasks (c) Another Allocation Scheme

Figure 1: Steps of Register Pressure Analysis and Management



(a) Null Register Pressure (b) Possible Spill Code Insertion (c) Unavoidable Spill Code

Figure 2: Register Pressure Configurations

If this factor is less than the number of available registers R (see Figure 2.a), then the register pressure has no effects on the scheduler, and the DDG is left as it is. However, if the saturation exceeds R (see Figure 2.b), we add serial arcs into the DDG to reduce it below the limit R while minimizing the corresponding increasing of critical path .

In this paper we address the same issue for loops and base our analysis of the Data Dependency Graph of the loop. The starting point is based on the following idea, that can be thought as a variant of [SCFS98]:

When two variables of some program are allocated into the same register R, this creates an antidependency between the operation that reads the first variable from R and the operation that writes the second variable into R. This dependency is called a "reuse" dependency, also called "Universal Occupancy Vector"(UOV) in [SCFS98]. This dependency or UOV in turn affects the performance of the program, estimated in terms of critical path for acyclic DDG, critical cycle for simple loops or schedule length for more general programs.

Our work here considers software pipelining of simple loops without conditionals, represented as DDG. We define the notion of reuse dependencies for software pipelined loops and analyze their influence on register pressure and Initiation Interval. We can either fix II and find the minimal number of registers required or alternatively fix a number of registers and find the minimum II. In both cases we are sure that any software pipelined loop generated after this modified DDG will not use more than the given register count.

Our paper is organized as follows: next section introduces a motivating example, then we formalize the problem. Finally we give a formulation with linear integer programming techniques and conclude with related work. For every detail about loop software pipelining and cyclic register allocation, one can refer to [AJLA95, WEJS94, ELM95].

2 Motivating Example

Let us consider the following loop with two instructions u and v.

```
for {i=1; i <=n; i++}
{
  /* u */
A(i+3) = ...
  /* v */
... = A(i)
    }
```

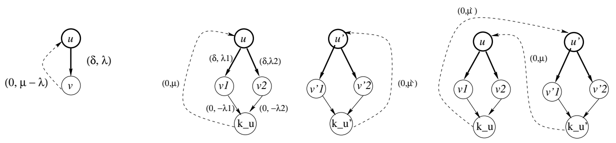
There is a flow dependency between u and v with distance $\lambda = 3$. This means that the operation v reads the value produced by u λ iterations earlier. This serialisation constraint is represented through an edge $u \rightarrow v$ with two labels: δ is the latency of operation u, λ is the distance of the dependence. This means that in the final schedule there should be at least δ clock cycles or time units between issue of $u(i)$ and $v(i+\lambda)$ for any iteration i.

Let us assume now that we use ρ different registers R_1, R_2, ... , R_ρ, cyclically for carrying this value, $A(4)$, written by $u(1)$, is stored in R_1, $A(5)$ in R_2, ... $A(\rho+3)$ in R_ρ, $A(\rho+4)$, written by $u(\rho+1)$, in R_1, ... (figure 3) Then $u(\rho+1)$ stores its result in the same register R_1. Hence $v(\lambda+1)$ should be scheduled before $u(\rho+1)$ and there is an anti-dependency between $v(\lambda+1)$ and $u(\rho+1)$ with a distance $\rho-\lambda$, see Figure 4.(a) where the values are shown with bold circles and flow arcs with bold lines. Dashed ones represent the anti-dependencies. Since u has some delay δ_w before writing into the result register, the latency of this anti-dependency is set to δ_w. This anti-dependency must in turn be counted when computing the new minimum initiation interval $MII \geq \left\lceil \frac{\delta-\delta_w}{\rho} \right\rceil$

Hence controlling register pressure means first determining which operations reuse registers killed by other operations (**where should antidependencies be added?**) and secondly determining variable lifetime, or equivalently register pressure (**how many iterations later (μ) should (direct) reuse occur**)? The lower the μ the lower the register pressure but also the larger the critical cycle.

When an operation creates a value that is read by more than one operation, we cannot know in advance which of these consumers would actually kill the value (which one would be scheduled to be the last reader), and hence we cannot know in advance when a register is freed. We propose a trick which defines for each value u^t of type t a fictitious killing task k_{u^t}. We insert an arc from each consumer $v \in Cons(u^t)$ to k_{u^t} to reflect the fact that this killing task is scheduled after the last scheduled consumer, see Figure 4.(b). The latency of this serial arc is $\delta_{r,t}(v)$, and we set its distance to $-\lambda$ where λ is the distance of the flow dependence between u and its consumer v. This means that the operation $k_{u^t}(i+\lambda-\lambda)$ i.e. $k_{u^t}(i)$ is scheduled when the value $u^t(i)$ is killed. Now, a register allocation scheme consists of defining the edges of reuse as defined just above. Hence defining for each u the task v which reuses the same register. We add then an edge from k_{u^t} to v (representing an anti-dependence from the killer of u to v) with a latency $-\delta_{w,t}(v)$ and a distance $\mu_{u,v}$ to be defined.

There are three main constraints that the resulting dependency graph must meet. First, the sum of distances along each circuit must be positive, else the problem



(a) Simple Reuse Circuit (b) Killing Tasks (c) Another Allocation Scheme

Figure 4: Examples of Register Reuse Schemes

would not have a solution. Second, the number of registers used by an allocation schema (decision) is $\sum \mu$ (we will prove this assertion in the next section) and must be lower or equal to the number of available registers. And finally - this is not a constraint - but a choice in our work -, a register released by an operation can be reused by only one operation, and each operation reuses only one register. This means that we have the same reuse pattern for every iteration, so that there is a one-to-one mapping between the killing nodes and the operations that have a result operand.

Let us consider the dependency graph of figure 4.(c). We can decide that u and v use disjoint sets of registers. u reuses only registers killed by k_u and v reuse only registers killed by k_v. In this case we have the following system (ρ is the number of register required):

$$\begin{cases} II & \geq & \frac{\delta}{\mu} \\ II & \geq & \frac{\delta\prime}{\mu\prime} \\ \mu & > & 0 \\ \mu\prime & > & 0 \\ \rho & = & \mu+\mu\prime \end{cases}$$

We can also decide that registers carry alternatively values from u and u' (figure 4.c) and we obtain the system:

$$\begin{cases} II & \geq & \frac{\delta+\delta\prime}{\nu+\nu\prime} \\ \nu+\nu\prime & > & 0 \\ \rho & = & \nu+\nu\prime \end{cases}$$

In both cases, we can fix II and minimize ρ or fix ρ and minimize II. It is easy to find out that the second choice is always better that the first one.

The reuse relation between the values are described by defining a new graph called *a reuse graph*. Figure 5.(a) shows the a first reuse decision where for instance u (v resp.) reuses the register used by itself μ_1 (μ_2 resp.) iterations earlier. Figure 5.(b) is the second reuse choice where u (v resp.) reuses the register used by v (u resp.) μ_1 (μ_2 resp.) iterations earlier. The resulted data dependency graph after adding the killing tasks and the anti-dependencies (Figure 4) to apply the register reuse decisions is called the *the DDG associated to a reuse graph*: Figure 4.(b) is the associated DDG to Figure 5.(a), and Figure 4.(c) is the one associated to Figure 5.(b). In the next section, we give a formal definition and modeling to the register allocation problem based on the reuse graphs.

```
{
   /* u, iteration 1 */
      R1 = ...
   /* v, iteration 1 */
... = A(1)
   /* u, iteration 2 */
      R2 = ...
   /* v, iteration 2 */
... = A(2)
   /* u, iteration 3 */
      R3 = ...
   /* v, iteration 3 */
... = A(3)
   /* u, iteration 4 */
      R4 = ...
   /* v, iteration 4 */
... = R1
   /* u, iteration 5 */
      R5 = ...
   /* v, iteration 5 */
... = R2
   /* u, iteration rho+1 (rho=5)*/
      R1 = ...
   /* v, iteration rho+1 */
... = R3
....

     }
```

Figure 3: Cyclic register allocation

(a) First Reuse Graph

(b) Second Reuse Graph

Figure 5: Reuse Graphs

3 Reuse Graphs for Register Allocation

In this section, we consider a Data Dependency Graph (DDG) $G = (V, E, \delta, \lambda)$. Nodes of the DDG are operations, that may generate output results. In that case, one register of some type $t \in \mathcal{T}$ is needed for storing this variable, and the operation is also called a *value* operation. Edges may be flow dependencies (in that case there are due to some variable that has to be stored into some register), or simple serialisation constraints due to other data dependencies or external reason. A software pipeline scheduling function assigns to each operation u a scheduling issue date σ_u, that satisfies the serialisation constraints: for each edge (u, v, δ, λ),

$$\sigma_u + \delta \leq \sigma_v + \lambda.II$$

.

A register allocation consists of choosing which operations reuses which released register. We define :

Definition 3.1 (Reuse Relation) *Let $G = (V, E, \delta, \lambda)$ be a DDG. A reuse relation for a register type $t \in \mathcal{T}$ is a bijection between $V_{R,t}$ and $V_{R,t}$ such that $reuse_t(u) = v$ iff the statement v reuses the register of type t released by the statement u. We note also $reuse_t^{-1}(v) = u$. We associate to this relation a reuse distance $\mu^t_{u,v}$ such that the operations $v(i + \mu^t_{u,v})$ reuses the register of type t released by the operation $u(i)$*

We represent the reuse relation by a graph (see Figure 5) :

Definition 3.2 (Reuse Graph) *Let $G = (V, E, \delta, \lambda)$ be a DDG and $reuse_t$ a reuse relation of a register type $t \in \mathcal{T}$. The reuse graph $G^r = (V_{R,t}, E_r, \mu)$ is defined by :*

$$E_r = \{e = (u^t, v^t) \,/\, reuse_t(u) = v \;\wedge\; \mu_t(e) = \mu^t_{u,v}\}$$

We call each arc in G^r a *reuse arc* , and each path in G^r a *reuse path*.

Lemma 3.1 *Each reuse path P constructed by inserting all the successive nodes u_i, u_{i+1} such that :*

$$reuse_t(u_i) = u_{i+1} \Longrightarrow u_{i+1} \in P$$

is an elementary circuit which we call a reuse circuit. And all the reuse circuits of G^r are disjoined :

$$\forall C \neq C' \text{ two reuse circuits} \qquad C \cap C' = \phi$$

We note $\mathcal{C}$ the set of all the reuse circuits of G^r.

Lemma 3.2 *Let $G^r = (V_{R,t}, E_r, \mu)$ be a reuse DDG according to a reuse relation $reuse_t$. Then, any value $u^t \in V_{R,t}$ of a register type $t \in \mathcal{T}$ belongs to a unique reuse circuit C in G^r.*

Let be $\mu_t(G^r)$ the sum of all the reuse distances between values of type t :

$$\mu_t(G^r) = \sum_{e=(u,v) \in E_r} \mu^t_{u,v}$$

and we note also $\mu_t(C)$ the sum of all the reuse distances between values of type t which belongs to the reuse circuit C :

$$\forall C \in \mathcal{C} \qquad \mu_t(C) = \sum_{e=(u,v) \in C} \mu^t_{u,v}$$

To report the register reuse decision in the DDG, we have to ensure that if $reuse_t(u) = v$ with a distance $\mu^t_{u,v}$ then $u^t(i)$ must be killed before the definition of $v^t(i + \mu^t_{u,v})$. The distance $\mu^t_{u,v}$ means that there is $\mu^t_{u,v}$ registers allocated between $u^t(i)$ and $v^t(i+\mu^t_{u,v})$. For this purpose, we define for each value u^t of type t a fictitious killing task k_{u^t} which corresponds to its killing date. We insert an anti-dependency arc between k_{u^t} and v iff $reuse_t(u) = v$.

Definition 3.3 (Killing Node) *Let $G = (V, E, \delta, \lambda)$ be a DDG and $\mathcal{T}$ a set of registers types. A killing node k_{u^t} of a value $u^t \in V_{R,t}$ of type t is a fictitious operation that corresponds to (an upper bound of) the killing date of u^t. It is defined by inserting in the DDG G the node k_{u^t} for all $u^t \in V_{R,t}$ with the the following serial arcs :*

- *add a serial arc $e = (v, k_{u^t})$ from each consumer $v \in Cons(u^t)$ to k_{u^t} ;*
- *for each inserted arc $e = (v, k_{u^t})$, set its latency to $\delta(e) = \delta_{r,t}(v)$, and its distance to $\lambda(e) = -d$ such that d is the distance of the flow dependence from u to v through a register of type t : $d = \lambda(e')$ with $e' = (u, v) \in E_{R,t}$.*

Note that the distance in terms of iterations of the path between each value and its killer is null. The set of all the killing nodes of type t is noted K_t :

$$K_t = \{k_{u^t} \;/\; u^t \in V_{R,t}\}$$

The resulted data dependency graph after adding the killing tasks and the anti-dependencies arcs is called the *DDG associated to the reuse relation*.

Definition 3.4 (DDG associated to a Reuse Relation) *Let $G = (V, E, \delta, \lambda)$ be a DDG with its inserted killing nodes K_t. The DDG associated to a reuse relation $reuse_t$ of a register type $t \in \mathcal{T}$ is an extended DDG of G such that we add an arc $e = (k_{u^t}, v)$ iff $reuse_t(u) = v$. We set its latency to $\delta(e) = -\delta_{w,t}(v)$, and its distance to $\lambda(e) = \mu^t_{u,v}$ (to be defined).*

Parts (b) and (c) of Figure 4 are two examples of the DDGs associated to the reuse relation defined in parts (a) and (b) of Figure 5 resp. We note the DDG associated to the reuse relation as $G_{\to r}$. One can remark that a reuse arc (u, v) is the counterpart of a path (u, v) in the meeting graph of any software pipelining schedule of $G_{\to r}$. Any arc (k_{u^t}, v) in $G_{\to r}$ according to a reuse relation ensures that the life interval of the value $u^t(i)$ ends before the definition of the value $v^t(i + \mu^t_{u,v})$.

Furthermore, a reuse distance $\mu^t_{u,v}$ defines an anti-dependence between the killer of u and v with a distance $\mu^t_{u,v} - \lambda \geq 0$ where λ is the distance between u and its killer.

Each reuse circuit has a counterpart in $G_{\to r}$ which we call an *image* of the reuse circuit :

$$C = (u_0, \cdots, u_n, u_0) \text{ a reuse circuit} \iff$$

$$C = (u_0, u'_0, k^t_{u_0}, \cdots, u_n, u'_n, k^t_{u_n}, u_0) \text{ a circuit in } G_{\to r}$$

where u'_i is a consumer of u_i. For instance, the reuse circuit (u, v, u) in Figure 5.(b) has an image $(u, v_1, k_u, u', v'_1, k_{u'}, u)$ in Figure 4.(c). Note that a reuse circuit can have more than one image in $G_{\to r}$ because a value can have more than one consumer : for instance, a second image for (u, v, u) in Figure 5.(b) is $(u, v_2, k_u, u', v'_2, k_{u'}, u)$ in Figure 4.(c). The distance of any image circuit is (see Figure 6) :

$$\lambda(C) = \mu_t(C) > 0 \tag{1}$$

There is some constraints that a reuse relation must meet in order to be valid : the existence of at least a software pipelining schedule for $G_{\to r}$ (i.e. all the introduced circuits must have a positive distance) defines the validity condition of the reuse relation.

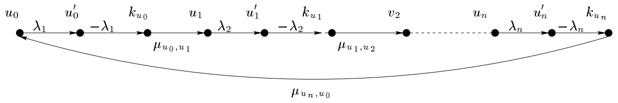
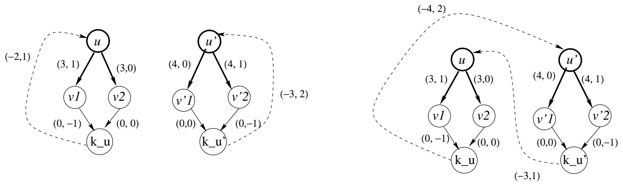


Figure 6: The Sum of Distances in the Reuse Circuits Images



(a) First Valid Reuse Scheme

(b) Second Valid Reuse Scheme

Figure 7: Valid Reuse Relations

Definition 3.5 (Valid Reuse Relation) *Let $G_{\to r}$ be a DDG associated to a reuse relation $reuse_t$. We say that $reuse_t$ is valid iff there exists a distance $\lambda(e) = \mu^t_{u,v}$ for each arc $e = (k_{u^t}, v)$ such that :*

$$\Sigma_L(G_{\to r}) \neq \phi \iff \forall \text{ circuit } C \text{ in } G_{\to r} \quad \lambda(C) > 0$$

$$\iff \forall C \text{ a reuse circuit } \sum_{(u,v) \in C} \mu^t_{u,v} > 0$$

Figure 7 shows two examples of DDGs associated to valid reuse relations. Note that the case of a circuit with null distance and negative latency $(\lambda(C) = 0 \;\wedge\; \delta(C) \leq 0)$ cannot exist because the anti-dependencies can never create a circuit with a null distance, otherwise it means that an operation (statement instance) reuses the register used by itself, which is impossible (no sense).

If a reuse relation is valid, we can build a cyclic register allocation in the DDG associated to it as explained in the following theorem.

Theorem 3.1 *Let $G_{\to r}$ be a reuse DDG according to a valid reuse relation $reuse_t$ such that there is only one reuse circuit in G^r. Then the unique reuse circuit C defines a cyclic register allocation for $G_{\to r}$ with exactly $\mu_t(C)$ registers if we unroll the loop $\rho = \mu_t(C)$ times.*

Proof :

Let unroll $G_{\to r}$ $\mu_t(C)$ times : each operation $u \in V$ has now ρ copies in the unrolled loop. We note by u_i the i^{th} copy of the operation $u \in V_{R,t}$. For the clarity of this proof, we

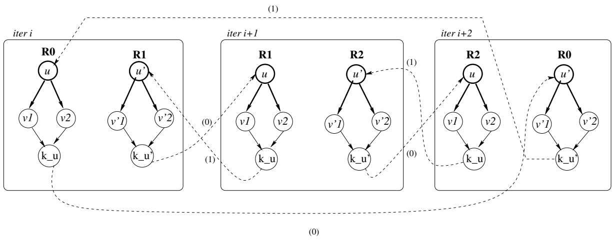


Figure 8: Cyclic Register Allocation with One Reuse Circuit

illustrate it by the example of Figure 8 which builds a cyclic register allocation with 3 registers for Figure 7.(b) : we have unrolled this loop 3 times. We allocate $\mu_t(C) = 3$ registers in the unrolled loop as follows :

1. for each reuse arc $e = (u, v)$, allocate $\mu_t(e)$ registers in the loop to the $\mu_t(e)$ values produced by u. For instance, the reuse arc (u, u') needs 2 registers R_1 and R_2 which we allocate to u_0 and u_1 resp. in Figure 8.
2. Since the reuse relation is valid, we are sure that for each reuse arc (u, v), the killing date of each value $u^t(i)$ is scheduled before the definition date of $v^t(i + \mu^t_{u,v})$. So, we allocate the same register to $v^t\big((i + \mu^t_{u,v}) \bmod \rho\big)$ as the one allocated to $u^t(i)$. For instance in Figure 8, we allocate the same register R_1 to u_1 and $u'((1 + 2) \bmod 3) = u'_0$.

Finally, we have allocated $\mu_t(C)$ registers in the unrolled loop to all the values. The dashed lines in Figure 8 represent the anti-dependencies with their corresponding distances after the unrolling.

⌟

Note that we can also build a cyclic register allocation with exactly $\mu_t(C)$ registers if we unroll the loop $k \times \rho$ times, where $\rho = \mu_t(C)$ and $k \in \mathbb{N}^+$ as follows :

1. unroll the loop ρ times and build a cyclic register allocation with $\mu_t(C)$ registers as explained in Theorem 3.1 ;
2. unroll the allocated loop k times.

At this point, we can state in the following theorem that the set of the reuse circuits define a cyclic register allocation with $\mu_t(G^r)$ registers.

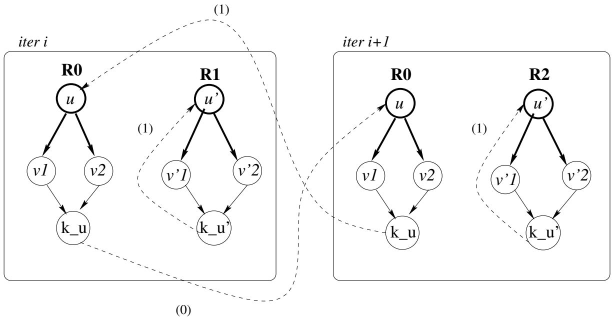


Figure 9: Cyclic Register Allocation

Theorem 3.2 *Let $G_{\rightarrow r}$ be a reuse DDG according to a valid reuse relation $reuse_t$ of a register type $t \in \mathcal{T}$. Then the reuse graph G^r defines a cyclic register allocation for $G_{\rightarrow r}$ with exactly $\mu_t(G^r)$ registers if we unroll the loop α times where :*

$$\alpha = lcm(\mu_t(C_1), \cdots, \mu_t(C_n))$$

with $\mathcal{C} = \{C_1, \cdots, C_n\}$ is the set of all the reuse circuits.

Proof :

It is a direct consequence of Theorem 3.1. The cyclic register allocation is build as follows :

1. unroll the loop α times. Each reuse circuit C is duplicated $\frac{\alpha}{\mu_t(C)}$ times ;
2. build a cyclic register allocation for each reuse circuit as explained in Theorem 3.1.

Figure 9 is an example of a cyclic register allocation of Figure 7.(a) which contains two reuse circuits ; (u, u) with a distance 1, and (u', u') wit a distance 2. The unrolling degree is hence $lcm(1, 2) = 2$. The dashed lines represent the anti-dependencies after unrolling the loop.

⌟

Corollary 3.1 *Let $G_{\rightarrow r}$ be a reuse DDG according to a valid reuse relation $reuse_t$ of a register type $t \in \mathcal{T}$. Then any valid software pipeline of $G_{\rightarrow r}$ admits a cyclic register allocation that uses less than $\mu_t(G^r)$.*

Proof :

According to Theorem 3.2, we can build a cyclic register allocation with $\mu_t(G^r)$ available registers. Then, the cyclic register requirement of any software pipeline cannot exceed $\mu_t(G^r)$.

$\lrcorner$

Corollary 3.2 *Let $G = (V, E, \delta, \lambda)$ be a loop with a set of values types $\mathcal{T}$. To each value type $t \in \mathcal{T}$ is associated a valid reuse relation $\mathcal{T}$ and a reuse graph G^r_t. The loop $G = (V, E, \delta, \lambda)$ can be allocated with $\mu_t(G^r)$ registers for each value type t if we unroll it α times, where*

$$\alpha = lcm(\alpha_{t_1}, \cdots, \alpha_{t_n})$$

with α_{t_i} is the valid unrolling degree of the value type t_i.

Proof:

Direct consequence of Theorem 3.2. The cyclic register allocation is build as follows:

1. unroll the loop α times. Each reuse circuit C_t of a register type t is duplicated $\alpha_t \times \frac{\alpha}{\mu_t(C)}$ times;
2. build a cyclic register allocation for each reuse circuit of each register type t as explained in Theorem 3.2.

$\lrcorner$

4 SIRA Problem Formulation

From the previous section, we deduce that doing a cyclic register allocation of a DDG is equivalent to find a valid reuse relation.

Problem 4.1 (SIRA) *Let $G = (V, E, \delta, \lambda)$ be a loop DDG and $\mathcal{R}_t$ the number of available registers of type t. Find a valid reuse relation $reuse_t$ such that the corresponding reuse graph $G^r = (V_{R,t}, E_r, \mu)$ has*

$$\mu_t(G^r) \leq \mathcal{R}_t$$

where the critical circuit in $G_{\rightarrow r}$ is minimized.

Theorem 4.1 *SIRA is NP-complete.*

Proof:

The SIRA decision problem can be formulated as:

Problem 4.2 (dec(SIRA)) *Let $G = (V, E, \delta, \lambda)$ be a loop DDG and $\mathcal{R}_t$ the number of available registers of type t, and k a positive integer. Does there exist a valid reuse relation $reuse_t$ such that*

$$\mu_t(G^r) \leq \mathcal{R}_t$$

and $MII \leq k$.

First, dec(SIRA) belongs to NP:

- to check if a reuse relation is valid, we check if the sum of distances of the reuse circuits are all strictly positive. The set of all the reuse circuits is simply done by looking for the strongly connected components of the reuse graph because the reuse circuits are elementary and disjoined;
- the tests
$$\mu_t(G^r) \leq \mathcal{R}_t$$
and $MII \leq k$ are linear.

Second, dec(SIRA) does not belong to P since it can be reduced easily to the problem of register allocation with a minimum number of registers under a fixed critical path, proven NP-complete in [EGS95].

$\lrcorner$

4.1 Two comments

There are however some more optimistic results. For instance it should be noted that when we enforce that any two operations do not share registers (this means that $reuse_t(u) = u$ for each operation u), then minimizing the register pressure amounts to minimizing buffers in the terminology of [NG93]. In the latter paper it is proven that minimizing buffers in software pipelined loops is a polynomial problem. Based on this property we conjecture than **when the reuse relation is fixed then we the problem of minimizing the register pressure is a polynomial problem**.

The second comment is based on the same observation like in the example of section 2. In terms of register pressure and II, it is always better to have one single reuse circuit instead of two or more. This means that we should only consider **Hamiltonian reuse circuits**. This is interesting because Hamiltonian reuse circuits result in register allocation schemes that can be directly implemented on rotating register files [ELM95, RLTS92]. This however may not be a good solution when also the unrolling degree for achieving such a register allocation is taken into account. In the case of an Hamiltonian circuit and no rotating register file on the processor, then the unrolling degree (and consequently the code size duplication factor) is exactly the register pressure.

4.2 ILP Formulation

In order to obtaining a basis for further experiments, we have considered formulating SIRA under the form of a linear integer program.rdietperform experiments Our model is built for a fixed execution rate II. We write the linear constraints which define a reuse relation for each register type.

Basic Variables

- a schedule variable σ_u for each operation $u \in V$ including one for each killing node k_{u^t} ;
- a binary variables $\theta^t_{u,v}$ for each $(u,v) \in V^2_{R,t}$ and for each register type $t \in \mathcal{T}$ which is set to 1 iff $reuse_t(u) = v$;
- a distance $\mu^t_{u,v}$ for the anti-dependence through the register type t between u and v for all $(u,v) \in V^2_{R,t}$.

Linear Constraints

Cyclic Scheduling Constraints

- bound the scheduling variables (we assume a worst schedule time of one iteration;

$$\forall u \in V \qquad \underline{\sigma_u} \le \sigma_u \le \overline{\sigma_u}$$

- bound the anti-dependence distance $\mu^t_{u,v}$ by the number of available registers:

$$\forall (u,v) \in V^2_{R,t} \qquad \mu^t_{u,v} \le \mathcal{R}_t$$

- data dependencies

$$\forall e = (u,v) \in E \qquad \sigma_u + \delta(e) \le \sigma_v + II \times \lambda(e)$$

- schedule killing nodes for consumed values: $\forall u^t \in V_{R,t}$

$$\forall v \in Cons(u^t) \qquad \sigma_{k_{u^t}} \ge \sigma_v + \delta_{r,t}(v) + ...$$

$$... \max_{e=(u,v)\in E_{R,t}} \lambda(e) \times II$$

- there is an anti-dependency between u and v iff $reuse_t(u) = v$. Then we add an arc from k_{u^t} to $v : \forall t \in \mathcal{T}\ \forall (u,v) \in V^2_{R,t}$

$$\theta^t_{u,v} = 1 \Longrightarrow \sigma_{k_{u^t}} - \delta_{w,t}(v) \le \sigma_v + II \times \mu_{u,v}$$

Since $\theta^t_{u,v}$ is binary, we write in the model:

$$\forall t \in \mathcal{T}\ \forall (u,v) \in V^2_{R,t} \qquad \theta_{u,v} \ge 1 \Longrightarrow$$

$$\sigma_{k_{u^t}} - \delta_{w,t}(v) \le \sigma_v + II \times \mu_{u,v}$$

We use the linear expression of the implication defined in [Tou00, Tou01a, Tou01c]

Reuse Relation Constraints The reuse relation must be a bijection:

- a register can be reused by only one operation:

$$\forall t \in \mathcal{T}\ \forall u \in V_{R,t} \qquad \sum_{v \in V_{R,t}} \qquad \theta^t_{u,v} = 1$$

- one value can reuse only one released register:

$$\forall t \in \mathcal{T}\ \forall u \in V_{R,t} \qquad \sum_{v \in V_{R,t}} \qquad \theta^t_{v,u} = 1$$

- if no register reuse between two values ($reuse_t(u) \ne v$), then set the anti-dependence distance to null: $\forall t \in \mathcal{T}\ \forall (u,v) \in V^2_{R,t}$

$$\theta^t_{u,v} = 0 \Longrightarrow \mu^t_{u,v} = 0$$

Since $\theta^t_{u,v}$ is binary, we write in the model:

$$\forall t \in \mathcal{T}\ \forall (u,v) \in V^2_{R,t} \qquad \theta^t_{u,v} \le 0 \Longrightarrow \begin{cases} \mu^t_{u,v} \ge 0 \\ \mu^t_{u,v} \le 0 \end{cases}$$

Objective Function We want to minimize the the number of registers required for the register allocation. so, we chose an arbitrary register type t which we use as an objective function:

$$\text{Minimize} \sum_{(u,v)\in V^2_{R,t}} \mu^t_{u,v}$$

The other registers types are bounded in the model by their respective number of available registers.

Summary The reuse relation produced is necessarily valid since we succeed in constructing a cyclic schedule. The complexity of the model is bounded by $\mathcal{O}(|V|^2)$ variables and $|E| + \mathcal{O}(|V|^2)$ constraints. To solve SIRA, we proceed by ;

1. begin with $II = MII$;

2. if the solution is greater than $\mathcal{R}_t$, then increment II or use a dichotomy between II and a maximum $II_{max} = L$;

3. if we reach the maximum II_{max} without finding a solution, there is no cyclic register allocation with R_t registers and hence spill code must be introduced.

Minimizing the unrolling degree is to minimize $lcm(\mu_i)$ the least common multiple of the anti-dependence distances of the reuse circuits. This problem is very difficult since there is no way to express linearly the least common multiple. We can consider two solutions :

1. limit the reuse distances with strictly positive constants $(\mu_1 \leq c_1, \cdots, \mu_n \leq c_n)$: the unrolling degree becomes limited by $c_1 \times \cdots \times c_n$. More these constants are sufficiently small, more the unrolling degree is minimized, more the critical circuit increases while the system becomes more difficult to solve. We think that this solution is non efficient and inaccurate ;

2. look for only one reuse (hamiltonian) circuit : the unrolling degree becomes limited by $\sum \mu_i \leq \mathcal{R}_t$.

We have already implemented this formulation directly and performed preliminary experiments. Up to now, we could solve the SIRA problem of minimizing the register pressure under the constraint of not increasing the minimum initiation interval for loops with number of operations up to 20.

5 Related Work

Scheduling with registers constraints tries to ensure that the number of values simultaneously alive does not exceed the number of available registers, guaranteeing the existence of a register allocation with the set of available registers.

In the field of cyclic scheduling, Huff was the first who proposed in [Huf93] a software pipelining heuristic which tried to minimize the values lifetimes, hoping that this would minimize the register requirement. Its technique was based on computing dynamically the scheduling interval of the operations. Ning and Gao defined an approximation of register requirement called buffers in [NG93, Nin93]. The difference between a buffer and a register is that if the life intervals of two values do not interfere, they can share a register but not a buffer. In fact, a buffer is a special register which passes the successive copies of the values produced from one software pipelining motif to the successive ones. The authors showed that in practical cases the buffers was a good approximation of the register requirement : they found in their treated example that the number of buffers is in worst case less by one than the register need. Wang *et al* [WKEE94, WKE95] proposed a software pipelining technique which tried to reduce the register requirement. They maintained dynamically a graph which reflected an approximation of the register requirement during scheduling. The RESIS methodology was studied in [SC96] which tried to minimize the maximal number of values simultaneously alive . The authors proceeded by changing the motif without incrementing the initiation interval. SWING [LVA95, LGAV96, Llo96] is a heuristic which constructed a motif with a reduced number of values lifetimes as the Huff's technique, and also tried to reduce the initiation interval.

Sawaya wrote an integer programming model which reduced the exact register requirement in [ES96a, ES96b]. The complexity of his model was $\mathcal{O}(|V| \times \lambda_{max} h)$ variables and $\mathcal{O}(|E| + |V| \times \lambda_{max} h)$ constraints. A better formulation was given in [EDA96] with $\mathcal{O}(|V| \times h)$ variables and $\mathcal{O}(|E| + |V| \times h)$ constraints. Our modeling is $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|E| + |V|^2)$ constraints which is better since it depends only on amount of input data (the size of the input DDG) and not on the data itself (specified operation latencies and critical cycle). Also we are able to control register pressure without directly looking for a solution, making our approach more flexible.

6 Conclusion

In this work we have presented a new clean framework for analyzing register pressure in software pipelined loops by working directly on the Data Dependency Graph instead of looking for a solution by any cheap heuristic or expensive exact method. In our framework trade off between register pressure and Initiation Interval is explicited.

We are able to transform the DDG so that register pressure is guaranteed. This DDG can then be processed by any software pipelining algorithm that does not have to consider register constraints. We have also introduced and implemented an ILP exact formulation, which provides a good starting point for further work in this framework where memory and parallelism constraints are decoupled. We expect that this work can be extended to other memory components such as cache or local memory.

References

[AJLA95] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995.

[Alt95] Eric Altman. *Optimal Software Pipelining with Functional Units and Registers*. PhD thesis, McGill University, Montreal, October 1995.

[EDA96] Alexandre E. Eichenberger, Edward S. Davidson, and Santosh G. Abraham. Minimizing Register Requirements of a Modulo Schedule via Optimum Stage Scheduling. *International Journal of Parallel Programming*, 24(2):103–132, April 1996.

[EGS95] Christine Eisenbeis, Franco Gasperoni, and Uwe Schwiegelshohn. Allocating Registers in Multiple Instruction-Issuing Processors. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT'95*, pages 290–293. ACM Press, June 27–29, 1995.

[ELM95] Christine Eisenbeis, Sylvain Lelait, and Bruno Marmol. The meeting graph: A new model for loop cyclic register allocation. In Lubomir Bic, Wim Böhm, Paraskevas Evripidou, and Jean-Luc Gaudiot, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 264–267, Limassol, Cyprus, June 27–29, 1995. ACM Press.

[ES96a] Christine Eisenbeis and Antoine Sawaya. Optimal Loop Parallelization under Register Constraints. In *Sixth Workshop on Compilers for Parallel Computers CPC'96.* , pages 245–259, Aachen - Germany, December 1996.

[ES96b] Christine Eisenbeis and Antoine Sawaya. Optimal Loop Parallelization under Register Constraints. Technical Report RR-2781, INRIA, January 1996. ftp://ftp.inria.fr/INRIA/publication/publi-ps-gz/RR/RR-2781.ps.gz.

[GAG94] R. Govindarajan, Erik R. Altman, and Guang R. Gao. Minimizing Register Requirements under Resource-Constrained Rate-Optimal Software Pipelining. In *MICRO27*, pages 85–94, December 1994.

[Huf93] R. Huff. Lifetime-Sensitive Modulo Scheduling. In *PLDI 93*, pages 258–267, Albuquerque, New Mexico, June 1993.

[LGAV96] J. Llosa, A. Gonzalez, E. Ayguadé, and M. Valero. Swing Modulo Scheduling: A Lifetime-Sensitive Approach. In *PACT 96*, Boston, Massachusetts, October 20-23 1996.

[Llo96] Josep Llosa. *Reducing the Impact of Register Pressure on Software Pipelined Loops*. PhD thesis, Universitat Politecnica de Catalunya (Spain), 1996.

[LVA95] J. Llosa, M. Valero, and E. Ayguadé. Hypernode Reduction Modulo Scheduling. In *micro28*, pages 350–360, Boston, Massachusetts, November 1995.

[MD99] Waleed M. Meleis and Edward S. Davidson. Dual-Issue Scheduling with Spills for Binary Trees. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 678–686, New York, January 17–19 1999. ACM-SIAM.

[Nak67] I. Nakata. On Compiling Algorithms for Arithmetic Expressions. *Communications of the ACM*, 10:492–494, July 1967.

[NG93] Qi Ning and Guang R. Gao. A Novel Framework of Register Allocation for Software Pipelining. In *Conference Record of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 29–42, Charleston, South Carolina, January 1993. ACM Press.

[Nin93] Qi Ning. *Optimal Register Allocation to Support Time Optimal Scheduling for Loops*. PhD thesis, School of Computer Science, McGill University, Montreal, Quebec, Canada, 1993.

[Red69] R. R. Redziejowski. On Arithmetic Expressions and Trees. *Communications of the ACM*, 12(2):81–84, February 1969.

[RLTS92] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register Allocation for Software Pipelined Loops. *SIGPLAN Notices*, 27(7):283–299, July 1992. Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.

[SC96] Fermin Sanchez and Jordi Cortadella. RESIS: A New Methodology for Register Optimization in Software Pipelining. In *Proceedings of Second International Euro-Par Conference, Euro-Par'96*, Lyon, France, August 1996.

[SCFS98] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Beth Simon. Schedule-independent storage mapping for loops. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 24–33, San Jose, California, October 3–7, 1998. ACM SIGARCH, SIGOPS, SIGPLAN, and the IEEE Computer Society.

[Set75] R. Sethi. Complete register allocation problems. *SIAM Journal on Computing*, 4(3):226–248, 1975.

[SU70] R. Sethi and J. D. Ullman. The Generation of Optimal Code for Arithmetic Expressions. *Journal of the ACM*, 17(4):715–728, 1970.

[Tou00] 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.

[Tou01a] Sid-Ahmed-Ali Touati. EquiMax: A New Formulation of Acyclic Scheduling Problem for ILP Processors. In *Interaction between Compilers and Computer Architectures*. Kluwer Academic Publishers, 2001.

[Tou01b] Sid-Ahmed-Ali Touati. Maximizing for Reducing Register Need in Acyclic Schedules. In *Proceedings of 5th International Workshop on Software and Compilers for Embedded Systems, SCOPES*, St Goar, Germany, March 2001.

[Tou01c] Sid-Ahmed-Ali Touati. Optimal Acyclic Fine-Grain Schedule with Cache Effects for Embedded and Real Time Systems. In *Proceedings of 9th nternational Symposium on Hardware/Software Codesign, CODES*, Copenhagen, Denmark, April 2001. ACM.

[Tou01d] Sid-Ahmed-Ali Touati. Register Saturation in Superscalar and VLIW Codes. In *Proceedings of The International Conference on Compiler Construction*, Lecture Notes in Computer Science. Springer-Verlag, April 2001.

[WEJS94] Jian Wang, Christine Eisenbeis, Martin Jourdan, and Bogong Su. DEcomposed Software Pipelining: A new perspective and a new approach. *International Journal of Parallel Programming*, 22(3):351–373, June 1994.

[WKE95] Jian Wang, Andreas Krall, and M. Anton Ertl. Decomposed Software Pipelining with Reduced Register Requirement. In *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT95*, pages 277 – 280, Limassol, Cyprus, June 1995.

[WKEE94] Jian Wang, Andreas Krall, M. Anton Ertl, and Christine Eisenbeis. Software Pipelining with Register Allocation and Spilling. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 95–99, San Jose, California, November 1994. ACM SIGMICRO and IEEE Computer Society TC-MICRO.