Early Control of Register Pressure for Software Pipelined Loops

Sid-Ahmed-Ali Touati, Christine Eisenbeis

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

Abstract. The register allocation in loops is generally performed after or during the software pipelining process. This is because doing a conventional register allocation at first step without assuming a schedule lacks the information of interferences between variable lifetime intervals. Thus, the register allocator may introduce an excessive amount of false dependences that reduce dramatically the ILP (Instruction Level Parallelism). We present a new framework for controlling the register pressure before software pipelining. This is based on inserting some anti-dependences edges (*register reuse* edges) labeled with *reuse distances*, directly on the data dependence graph. In this new graph, we are able to guarantee that the number of simultaneously alive variables in any schedule does not exceed a limit. The determination of register and distance reuse is parameterized by the desired critical circuit ratio (MII) as well as by the register pressure constraints - either can be minimized while the other one is fixed. After scheduling, register allocation is done cyclically on conventional register sets or on rotating register files. We give an optimal exact model, and another approximative one that generalizes the Ning-Gao [12] buffer optimization heuristics.

1 Introduction

This article addresses the problem of register pressure in simple loop data dependence graphs (DDGs), with multiple register types and non unit assumed latencies operations. Our aim is to decouple the registers constraints and allocation from the scheduling process and to analyze the trade-off between memory (register pressure) and parallelism constraints, measured as the critical ratio MII[1] of the DDG.

The principal reason is that we believe that register allocation is more important as an optimization issue than code scheduling. This is because the code performance is far more sensitive to memory accesses than to fine-grain scheduling (memory gap) : a cache miss may inhibit the processor from achieving a high dynamic ILP, even if the scheduler has extracted it at compile time. Even if someone would expect that spill codes exhibit high locality, and hence would likely produce cache hits, we cannot assert it at compile time. The authors in [5] related that about 66% of application execution times are spent to satisfying memory requests.

Another reason for handling register constraints prior to ILP scheduling is that register constraints are much more complex than resource constraints. Scheduling under

[1] We refer here to MII_{dep} since we will not consider any resource constraints.

resource constraints is a performance issue. Given a DDG, we are sure to find at least one valid schedule for any underlying hardware properties (a sequential schedule in extreme case, i.e., no ILP). However, scheduling a DDG with a limited number of registers is more complex. We cannot guarantee the existence of at least one schedule. In some cases, we must introduce spill code and hence we change the problem (the input DDG). Also, a combined pass of scheduling with register allocation presents an important drawback if not enough registers are available. During scheduling, we may need to insert load-store operations. We cannot guarantee the existence of a valid issue time for these introduced memory access in an already scheduled code; resource or data dependence constraints may prevent from finding a valid issue slot inside an already scheduled code. This forces to iteratively apply scheduling followed by spilling until reaching a solution.

All the above arguments make us re-think new ways of handling register pressure before starting the scheduling process, so that the scheduler would be free from register constraints and would not suffer from excessive serializations.

Existing techniques in this field usually apply register allocation after a step of software pipelining that is sensitive to register requirement. Indeed, if we succeed in building a software pipelined schedule that does not produce more than R values simultaneously alive, then we can build a cyclic register allocation with R available registers [2, 13]. We can use either loop unrolling [2], inserting move operations [7], or a hardware rotating register file when available [13][2]. Therefore, a great amount of work tries to schedule a loop such that it does not use more than R values simultaneously alive [8, 22, 12, 14, 11, 4, 15, 6, 9]. In this paper we directly work on the loop DDG and modify it in order to satisfy the register constraints for any further subsequent software pipelining pass. This idea is already present in [1] for DAGs and use the concept of *reuse* edge or vector developed in [17, 18].

Our article is organized as follows. Sect. 2 defines our loop model and a generic ILP processor. Sect. 3 starts the study with a simple example. The problem of cyclic register allocation is described in Sect. 4 and formulated with integer linear programming (intLP). The special case where a rotating register file (RRF) exists in the underlying processor is discussed in Sect. 5. In Sect.6, we present a polynomial subproblem. Finally, we synthesize our experiments in Sect. 7 before concluding.

2 Loop Model

We consider a simple innermost loop (without branches). It is represented by a graph $G = (V, E, \delta, \lambda)$, such that: V is the set of the statements in the loop body and E is the set of precedence constraints (flow dependences, or other serial constraints). We associate to each arc $e \in E$ a latency $\delta(e)$ in terms of processor clock cycles and a distance $\lambda(e)$ in terms of number of iterations. We denote by $u(i)$ the instance of the

[2] Insertion of *move* operations or using a rotating register file requires $R+1$ registers at most [2].

statement $u \in V$ of the iteration i. A valid schedule σ must satisfy :

$$\forall e = (u, v) \in E \,:\, \sigma\big(u(i)\big) + \delta(e) \leq \sigma\big(v(i + \lambda(e))\big)$$

We consider a target RISC-style architecture with multiple register types, where $\mathcal{T}$ denotes the set of register types (for instance, $\mathcal{T} = \{int, float\}$). We make a difference between statements and precedence constraints, depending if they refer to values to be stored in registers or not. $V_{R,t}$ is the set of values to be stored in registers of type $t \in \mathcal{T}$. We consider that each statement $u \in V$ writes into at most one register of a type $t \in \mathcal{T}$. The statements which define multiple values with different types are accepted in our model if they do not define more than one value of a certain type. $E_{R,t}$ is the set of flow dependence edges through a value of type $t \in \mathcal{T}$. The set of consumers (readers) of a value u^t is then the set :

$$Cons(u^t) = \{v \in V \mid (u, v) \in E_{R,t}\}$$

To consider static issue VLIW and EPIC/IA64 processors in which the hardware pipeline steps are visible to compilers (we consider dynamically scheduled superscalar processors too), we assume that reading from and writing into a register may be delayed from the beginning of the schedule time, and these delays are visible to the compiler (architectural visible). We define two delay (offset) functions $\delta_{r,t}$ and $\delta_{w,t}$ in which : the read cycle of u^t from a register of type t is $\sigma(u) + \delta_{r,t}(u)$, and the the write cycle of u^t into a register of type t is $\sigma(u) + \delta_{w,t}(u)$.

For superscalar and EPIC/IA64 processors, $\delta_{r,t}$ and $\delta_{w,t}$ are equal to zero.

A software pipelining is a function σ that assigns to each statement u a scheduling date (in terms of clock cycle) that satisfies at least the precedence constraints. It is defined by an initiation interval, noted II, and the scheduling date σ_u for the operations of the first iteration. The operation $u(i)$ of iteration i is scheduled at time $\sigma_u + (i-1) \times II$. For all edge $e = (u, v) \in E$, this periodic schedule must satisfy:

$$\sigma_u + \delta(e) \leq \sigma_v + \lambda(e).II$$

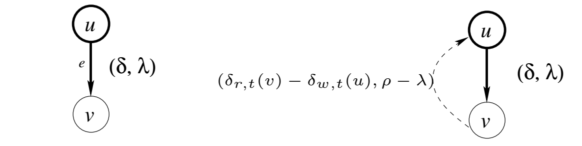
Classically, by adding all such inequalities on any circuit C of G, we find that II must be greater than or equal to $\max_C \frac{\sum_{e \in C} \delta(e)}{\sum_{e \in C} \lambda(e)}$, that we commonly denote as MII (minimal initiation interval).

We consider now a number of available registers ρ and all the schedules that have no more than ρ simultaneously alive variables. Any actual following register allocation will induce new dependences in the DDG. Hence, register pressure has influence on the expected II, even if we assume unbounded resources. What we want to analyze here is the minimum II that can be expected for any schedule using less than ρ registers. We will denote this value as $MII(\rho)$ and we will try to understand the relationship between $MII(\rho)$ and ρ. Let us start by an example to fix the ideas.

3 Basic Ideas

We give now more intuitions to the new edges that we add between couples of operations. These edges represent possible reuse by the second operation of the register released by the first operation. This can be viewed as a variant of [1] or [17, 18].

Let us consider a simple loop that consists of a unique flow dependence from u to v with distance $\lambda = 3$ (see Fig. 1.(a) where values to be stored in registers of the considered type are in bold circles, and flows are in bold edges). If we have an unbounded number of registers, all iterations of this loop can be run in parallel since there is no recurrence circuit in the DDG. At each iteration, operation u writes into a new register. Now, let us assume that we only have $\rho = 5$ available registers. The different instances of u can use only $\rho = 5$ registers to cyclically carry their results. In this case, the operation $u(i + \rho)$ writes into the same register previously used by $u(i)$. This fact creates an anti-dependence from $v(i + \lambda)$, which reads the value defined by $u(i)$, to $u(i + \rho)$; this means an anti-dependence in the DDG from v to u with a distance $\rho - \lambda = 2$. Since u actually writes into its destination register $\delta_{w,t}(u)$ clock cycles after it is issued and v reads it $\delta_{r,t}(v)$ after it is issued, the latency of this anti-dependence is set to $\delta_{r,t}(v) - \delta_{w,t}(u)$ for VLIW or EPIC codes, and to 1 for superscalar (sequential) codes. Consequently, the DDG becomes cyclic because of storage limitations (see Fig. 1.(b), where the anti-dependence is dashed). The introduced anti-dependence, also called "Universal Occupancy Vector' '(UOV) in [17], must in turn be counted when computing the new minimum initiation interval since a new circuit is created.



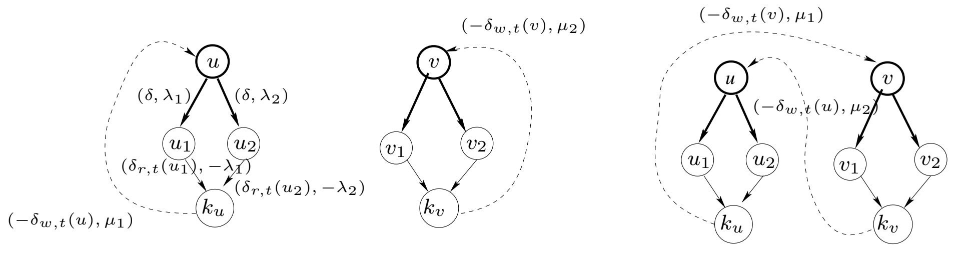
(a) Simple DDG (b) Antidependence

Fig. 1. Simple Example

When an operation defines a value that is read by more than one operation, we cannot know in advance which of these consumers actually kills the value (which is 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 edge 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 Fig. 2). The latency of this serial edge is set to $\delta_{r,t}(v)$ because of the reading delay, and we set its distance to $-\lambda$ where λ is the distance of the flow dependence between u and its consumer v. This is done to model the fact 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. The iteration number i of the killer of $u(i)$ is only a convention and can be changed by retiming [10], without changing the nature of the problem.

Now, a register allocation scheme consists of defining the edges and the distances of reuse. That is, we define for each $u(i)$ the operation v and iteration $\mu_{u,v}$ such that $v(i + \mu_{u,v})$ reuses the same destination register as $u(i)$. This reuse creates a new anti-dependence from k_u to v with latency equal to $-\delta_{w,t}(v)$ for VLIW or EPIC codes, and to 1 for sequential superscalar codes. The distance $\mu_{u,v}$ of this edge has to be defined.

We will see in a further section that the register requirement can be expressed in terms of $\mu_{u,v}$.


u
(δ, λ_1)
(δ, λ_2)
u_1
u_2
$(\delta_{r,t}(u_1), -\lambda_1)$
$(\delta_{r,t}(u_2), -\lambda_2)$
k_u
$(-\delta_{w,t}(u), \mu_1)$
$(-\delta_{w,t}(v), \mu_2)$
v
v_1
v_2
k_v
$(-\delta_{w,t}(v), \mu_1)$
u
v
$(-\delta_{w,t}(u), \mu_2)$
u_1
u_2
v_1
v_2
k_u
k_v

(a) First Reuse Decision (b) Another Allocation Scheme

Fig. 2. Killing Tasks

Hence, controlling register requirement means, first, determining which operation should reuse the register killed by another operation (*where should anti-dependences be added?*). Secondly, we have to determine variable lifetimes, or equivalently register requirement (*how many iterations later (μ) should reuse occur*)? The lower is the μ, the lower is the register requirement, but also the larger is the MII.

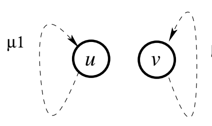
Fig. 2.(a) presents a first reuse decision where each statement reuses the register freed by itself. This is illustrated by adding an anti-dependence from k_u (resp. k_v) to u (resp. v) with an appropriate distance μ, as we will see later. Another reuse decision (see Fig. 2.(b)) may be that the statement u (resp. v) reuses the register freed by v (resp. u). This is illustrated by adding an anti-dependence from k_u (resp. k_v) to v (resp. u). In both cases, the register requirement is $\mu_1 + \mu_2$, but it is easy to see that the two schemes do not have the same impact on MII: intuitively it is better that the operations share registers instead of using two different pools of registers. The next section gives a formal definition of the problem and provides an exact formulation.

4 Problem Description

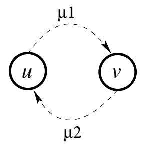
4.1 Data Dependences and Reuse Edges

The reuse relation between the values (variables) is described by defining a new graph called *a reuse graph* that we note $G^r = (V_{R,t}, E_r, \mu)$. Fig. 3.(a) shows the first reuse decision where u (v resp.) reuses the register used by itself μ_1 (μ_2 resp.) iterations earlier. Fig. 3.(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. Each edge $e = (u, v) \in E_r$ with a distance $\mu(e)$ in the reuse graph means that there is an anti-dependence between k_u and v with a distance $\mu(e)$. The resulted DDG after adding the killing tasks and the anti-dependences to apply the register reuse decisions is called the *DDG associated with a reuse decision* :

Fig. 2.(a) is the associated DDG with Fig. 3.(a), and Fig. 2.(b) is the one associated with Fig. 3.(b). We denote by $G_{\rightarrow r}$ the DDG associated to a reuse decision r.



(a) First Reuse Graph



(b) Second Reuse Graph

Fig. 3. Reuse Graphs

A reuse graph must verify some constraints to be valid : first, the resulting DDG must be schedulable; second, each value reuses only one freed register, and each register is reused by only one value. The second constraint means that the reuse scheme is the same at each iteration. Generalizing this condition by allowing different (but periodic) reuse schemes is beyond the scope of this paper. This condition results in the following lemma.

Lemma 1. *[21] Let $G^r = (V_{R,t}, E_r, \mu)$ be a valid reuse graph of type t associated with a loop $G = (V, E, \delta, \lambda)$. Then :*

- *the reuse graph only consists of elementary and disjoined circuits ;*
- *any value $u^t \in V_{R,t}$ belongs to a unique circuit in the reuse graph.*

Any circuit C in a reuse graph is called a *reuse circuit*. We note $\mu(C)$ the sum of the μ distances in this circuit. Then, to each reuse circuit $C = (u_0, u_1, .., u_n, u_0)$, there exists an image $C' = (u_0 \rightsquigarrow k_{u_0}, u_1, ..., u_n \rightsquigarrow k_{u_n}, u_0)$ for it in the associated DDG. For instance in Fig. 2.(a), $C' = (v, v_1, k_v, v)$ is an image for the reuse circuit $C = (v, v)$ in Fig. 3.(a). Such image may not be unique.

If a reuse graph is valid, we can build a cyclic register allocation in the DDG associated with it, as explained in the following theorem. We require $\mu(G^r)$ registers, in which $\mu(G^r)$ is the sum of all μ distances in the reuse graph G^r.

Theorem 1. *[21] Let $G = (V, E, \delta, \lambda)$ be a loop and $G^r = (V_{R,t}, E_r, \mu)$ a valid reuse graph of a register type $t \in \mathcal{T}$. Then the reuse graph G^r defines a cyclic register allocation for G with exactly $\mu_t(G^r)$ registers of type t 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 reuse circuits, and lcm is the least common multiple.

For a complete and detailed proof, please refer to [21].

As a corollary, we can build a cyclic register allocation for all register types.

Corollary 1. *[21] Let $G = (V, E, \delta, \lambda)$ be a loop with a set of register types $\mathcal{T}$. To each type $t \in \mathcal{T}$ is associated a valid reuse graph G^r_t. The loop can be allocated with $\mu_t(G^r)$ registers for each type t if we unroll it α times, where*

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

where α_{t_i} is the unrolling degree of the reuse graph of type t_i.

It should be noted that the fact that the unrolling factor may be significantly high is not related to our method and would happen only if we actually want to allocate the variables on this minimal number of registers with the computed reuse scheme. However, there may be other reuse schemes for the same number of registers, or there may be other available registers in the architecture. In that case, the meeting graph framework [2] can help to control or reduce this unrolling factor.

From all above, we deduce a formal definition of the problem of optimal cyclic register allocation with minimal ILP loss. We call it Schedule Independent Register Allocation (SIRA).

Problem 1 (SIRA). Let $G = (V, E, \delta, \lambda)$ be a loop and $\mathcal{R}_t$ the number of available registers of type t. Find a valid reuse graph for each register type such that the corresponding

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

and the critical circuit in G is minimized.

This problem can be reduced to the classical NP-complete problem of minimal register allocation [21]. The following section gives an exact formulation of SIRA.

4.2 Exact Formulation

In this section, we give an intLP model for solving SIRA. It is built for a fixed execution rate II (the new constrained MII). Note that II is not the initiation interval of the final schedule, since the loop is not already scheduled. II denotes the value of the new desired critical circuit. Here, we assume VLIW or EPIC codes. For superscalar ones, we only have to set the anti-dependence latency to 1.

Our SIRA exact model uses the linear formulation of the logical implication ($\Longrightarrow$) and equivalence ($\Longleftrightarrow$) by introducing binary variables, as previously explained in [19–21]. The size of our system is bounded by $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|E| + |V|^2)$ linear constraints.

Basic Variables

- a schedule variable $\sigma_u \geq 0$ 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 $t \in \mathcal{T}$. It is set to 1 iff (u, v) is a reuse edge of type t ;
- $\mu^t_{u,v}$ for reuse distance for all $(u, v) \in V^2_{R,t}$, and for each register type.

Linear Constraints

- data dependences (the existence of at least one valid software pipelining schedule, including killing tasks constraints)

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

- there is an anti-dependence between k_{u^t} and v if (u, v) is a reuse edge: $\forall t \in \mathcal{T}, \ \forall (u, v) \in V_{R,t}^2$:

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

- If there is no register reuse between two values $(reuse_t(u) \neq v)$, then $\theta_{u,v}^t = 0$. The anti-dependence distance $\mu_{u,v}^t$ must be set to 0 in order to not be cumulated in the objective function. $\forall t \in \mathcal{T}, \ \forall (u, v) \in V_{R,t}^2$:

$$\theta_{u,v}^t = 0 \Longrightarrow \mu_{u,v}^t = 0$$

The reuse relation must be a bijection from $V_{R,t}$ to $V_{R,t}$:

- a register can be reused by only one operation:

$$\forall t \in \mathcal{T}, \ \forall u \in V_{R,t} : \sum_{v \in V_{R,t}} \theta_{u,v}^t = 1$$

- one value can reuse only one released register:

$$\forall t \in \mathcal{T}, \ \forall u \in V_{R,t} : \sum_{v \in V_{R,t}} \theta_{v,u}^t = 1$$

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

$$\text{Minimize} \sum_{(u,v) \in V_{R,t}^2} \mu_{u,v}^t$$

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

$$\forall t' \in \mathcal{T} - \{t\} : \sum_{(u,v) \in V_{R,t'}^2} \mu_{u,v}^{t'} \leq \mathcal{R}_{t'}$$

As previously mentioned, our model includes writing and reading offsets. The non-positive latencies of the introduced anti-dependences generate a specific problem. Indeed, some circuits C in the constructed DDG may have non-positive distance $\lambda(C) \leq 0$. Even if such circuits do not prevent a DDG from being scheduled, it may be so in the presence of resource constraints. Thus, we prohibit such circuits. More details can be found in [21]. Note that this problem does not occur for superscalar (sequential) codes, because the introduced anti-dependences have positive latencies.

The unrolling degree is left free and over any control in SIRA formulation. This factor may theoretically grow exponentially. Minimizing the unrolling degree is to minimize $lcm(\mu_i)$, the least common multiple of the anti-dependence distances of reuse circuits. This non linear problem is very difficult an remains an open problem : as far as we know, there is not a satisfactory solution for it. Fortunately, there exists a hardware feature that allow to avoid loop unrolling. We study it in the next section.

5 Rotating Register Files

A rotating register file [3, 13, 16] is a hardware feature that moves (shift) implicitly architectural registers in a cyclic way. At every new kernel issue (special branch operation), each architectural register specified by program is mapped by hardware to a new physical register. The mapping function is (R denotes an architectural register and R' a physical register) : $R_i \mapsto R'_{(i+RRB) \text{ mod } s}$ where RRB is a rotating register base and s the total number of physical registers. The number of that physical register is decremented continuously at each new kernel. Consequently, the intrinsic reuse scheme between statements describes a hamiltonian reuse circuit necessarily. The hardware behavior of such register files does not allow other reuse patterns. SIRA in this case must be adapted in order to look only for hamiltonian reuse circuits.

Furthermore, even if no rotating register file exists, looking for only one hamiltonian reuse circuit makes the unrolling degree exactly equal to the number of allocated registers, and thus both are simultaneously minimized by the objective function.

Since a reuse circuit is always elementary (Lemma 1), it is sufficient to state that a hamiltonian reuse circuit with $n = |V_{R,t}|$ nodes is only a reuse circuit of size n. We proceed by forcing an ordering of statements from 1 to n according to the reuse relation. Thus, given a loop $G = (V, E, \delta, \lambda)$ and $G^r = (V_{R,t}, E_r, \mu)$ a valid reuse graph of type $t \in \mathcal{T}$, we define a hamiltonian ordering ho_t as a function :

$$\begin{aligned} ho_t \; &: V_{R,t} \rightarrow \mathbb{N} \\ & \quad u^t \mapsto \quad ho_t(u) \end{aligned}$$

such that $\forall u, v \in V_{R,t}$:

$$(u, v) \in E_r \Longleftrightarrow ho_t(v) = \Big(ho_t(u) + 1\Big) \text{ mod } |V_{R,t}|$$

The existence of a hamiltonian ordering is a sufficient and necessary condition to make the reuse graph hamiltonian, as stated in the following theorem.

Theorem 2. *[21] Let $G = (V, E, \delta, \lambda)$ be a loop and G^r a valid reuse graph. There exists a hamiltonian ordering iff the reuse graph is a hamiltonian graph.*

Hence, the problem of cyclic register allocation with minimal critical circuit on rotating register files can be stated as follows.

Problem 2 (SIRA_HAM). Let $G = (V, E, \delta, \lambda)$ be a loop and $\mathcal{R}_t$ the number of available registers of type t. Find a valid reuse graph with a hamiltonian ordering ho_t such that the

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

in which the critical circuit in G is minimized.

An exact formulation for it is deduced from the intLP model of SIRA. We have only to add some constraints to compute a hamiltonian ordering. We expand the exact SIRA intLP model by at most $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|V|^2)$ linear constraints.

1. for each register type and for each value $u^t \in V_{R,t}$, we define an integer variable $ho_{u^t} \geq 0$ which corresponds to its hamiltonian ordering ;
2. we add the linear constraints of the modulo hamiltonian ordering : $\forall u, v \in V_{R,t}^2$:

$$\theta_{u,v}^t = 1 \Longleftrightarrow ho_{u^t} + 1 = |V_{R,t}| \times \beta_{u,v}^t + ho_{v^t}$$

   where $\beta_{u,v}^t$ is a binary variable that holds to the integer division of $ho_{u^t} + 1$ on $|V_{R,t}|$.

When looking for a hamiltonian reuse circuit, we may need one extra register to construct such a circuit. In fact, this extra register virtually simulates moving values among registers if circular lifetimes intervals do not meet in a hamiltonian pattern.

Proposition 1. *[21] Hamiltonian SIRA needs at most one extra register than SIRA.*

Both SIRA and hamiltonian SIRA are NP-complete. Fortunately, we have some optimistic results. In the next section, we investigate the case in which SIRA can be solved in polynomial time complexity.

6 Fixing Reuse Edges

In [12], Ning and Gao analyzed the problem of minimizing the buffer sizes in software pipelining. In our framework, this problem actually amounts to deciding that each operation reuses the same register, possibly some iterations later. Therefore we consider now the complexity of our minimization problem when fixing reuse edges. This generalizes the Ning-Gao approach. Formally, the problem can be stated as follows.

Problem 3 (Fixed SIRA). Let $G = (V, E, \delta, \lambda)$ be a loop and $\mathcal{R}_t$ the number of available registers of type t. Let $E' \subseteq E$ be the set of already fixed anti-dependences (reuse) edges of a register type t. Find a distance $\mu_{u,v}$ for each anti-dependence $(k_{u^t}, v) \in E'$ such that

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

in which the critical circuit in G is minimized.

In following, we assume that $E' \subseteq E$ is the set of these already fixed anti-dependences (reuse) edges (their distances have to be computed). Deciding (at compile) time for fixed reuse decisions greatly simplifies the intLP system of SIRA. It can be solved by the following intLP, assuming a fixed desired critical circuit II. Here, we write a system for VLIW or EPIC codes. For superscalar, we have to set the anti-dependence latency to 1.

$$\begin{array}{ll} \text{Minimize} & \rho = \sum_{(k_{u^t}, v) \in E'} \mu_{u,v}^t \\ & \\ \text{Subject to:} & \\ II \times \mu_{u,v}^t + \sigma_v - \sigma_{k_{u^t}} \geq -\delta_w(v) & \forall (k_{u^t}, v) \in E' \\ \sigma_v - \sigma_u \geq \delta(e) - II \times \lambda(e) & \forall e = (u, v) \in E - E' \end{array} \qquad (1)$$

Since II is a constant, we do the variable substitution $\mu'_u = II \times \mu^t_{u,v}$ and System 1 becomes :

$$\begin{array}{ll} \text{Minimize} & (II.\rho =) \sum_{u \in V_{R,t}} \mu'_u \\ & \\ \text{Subject to:} & \\ \mu'_u + \sigma_v - \sigma_{k_{u^t}} \geq -\delta_w(v) \quad \forall (k_{u^t}, v) \in E' & \\ \sigma_v - \sigma_u \geq \delta(e) - II \times \lambda(e) \; \forall e = (u,v) \in E - E' & \end{array} \tag{2}$$

There are $\mathcal{O}(|V|)$ variables and $\mathcal{O}(|E|))$ linear constraints in this system.

Theorem 3. *[21] The constraint matrix of the integer programming model in System 2 is totally unimodular, i.e., the determinant of each square sub-matrix is equal to 0 or to $\pm$ 1.*

Consequently, we can use polynomial algorithms to solve this problem of finding the minimal value for the product $II.\rho$.

We must be aware that the back substitution in $\mu = \frac{\mu'}{II}$ may produce a non integral value for the distance μ. If we ceil it by setting $\mu = \lceil \frac{\mu'}{II} \rceil$, a sub-optimal solution may result[3]. It is easy to see that the loss in terms of number of registers is not greater than the number of loop statements that write into a register ($|V_{R,t}|$). This algorithm generalizes the heuristics proposed in [12]. We think that we can avoid ceiling μ by considering the already computed σ variables, as done in [12].

Furthermore, solving System 2 has two interesting follow-ups. First, it gives a polynomially computable lower bound for $MII_{rc}(\rho)$ as defined in the introduction, for this reuse configuration rc. Let us denote as m the minimal value of the objective function. Then

$$MII_{rc}(\rho) \geq \frac{m}{\rho}$$

This lower bound could be used in a heuristics such that the reuse scheme and the number of available registers ρ are fixed. Second, if II is fixed, then we obtain a lower bound on the number of registers ρ required in this reuse scheme rc.

$$\rho_{rc} \geq \frac{m}{II}$$

There are numerous choices for fixing reuse edges that can be used in practical compilers.

1. For each value $u \in V_{R,t}$, we can decide that $reuse_t(u) = u$. This means that each statement reuses the register freed by itself (no sharing of registers between different statements). This is similar to buffer minimization problem as described in [12].
2. We can fix reuse edges according to the anti-dependences present in the original code : if there is an anti-dependence between two statement u and v in the original code, then fix $reuse_t(u') = v$ with the property that u kills u'. This decision is a generalization to the problem of reducing the register requirement as studied in [22].

[3] Of course, if we have $MII = II = 1$ (case of parallel loops for instance), the solution remains optimal.

3. If a rotating register file is present, we can fix an arbitrary (or with a cleverer method) hamiltonian reuse circuit among statements.

The next section summarizes our experimental results.

7 Experiments

All the techniques described in this paper have been implemented and tested on various numerical loops extracted from different benchmarks (Spec95, whetstone, livermore, lin-ddot). This section presents a summary.

Optimal and Hamiltonian SIRA In almost all the cases, both of the two techniques need the same number of registers according to the same II. However, as proved by Prop.1, hamiltonian SIRA may need one extra register, but in very few cases (about 5% of experiments). Regarding the resulted unrolling degrees, even if it may grow exponentially with SIRA (from the theoretical perspective), experiments show that it is mostly lower than the number of allocated registers, i.e., better than hamiltonian SIRA. However, some few cases exhibit critical unrolling degrees which are not acceptable if code size expansion is a critical factor.

Optimal SIRA versus Fixed SIRA In a second step of experiments, we investigate the fixed SIRA strategies (Sect. 6) to compare their results versus the optimal ones (optimal SIRA). We checked the efficiency of two strategies: self reuse strategy (no register sharing), and fixing an arbitrary hamiltonian reuse circuit. Resolving the intLP systems of these strategies become very fast compared to optimal solutions, as can be seen the first part of Fig. 4. We couldn't explore optimal solutions for loops larger than 10 nodes because the computation time became intractable.

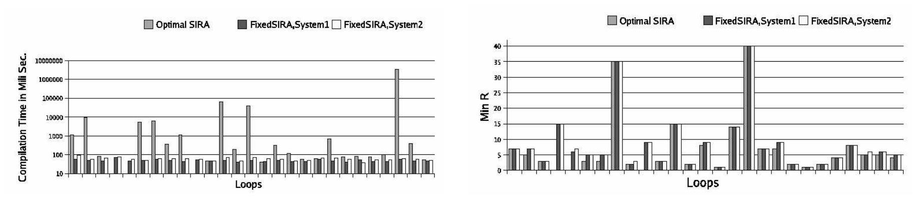


Fig. 4. Optimal versus Fixed SIRA with $II = MII$

For $II = MII$, some experiments do not exhibit a substantial difference. But if we vary II from MII to an upper-bound L, the difference is highlighted as follows.

– Regarding the register requirement, the self reuse strategy is, in most cases, far from the optimal. Disabling register sharing needs a high number of registers, since each statement needs at least one register. However, enabling sharing with an arbitrary hamiltonian reuse circuit is much more beneficial.

– Regarding the unrolling degree, the self reuse strategy exhibit the lowest ones, except in very few cases.

Fixed SIRA : System 1 versus System 2 The compilation time for optimal SIRA becomes intractable when the size of the loop exceeds 10 nodes. Hence, for larger loops, we advice to use our fixed SIRA strategies that are faster but allow sub-optimal results. We investigated the scalability (in terms of compilation time[4] versus the size of DDGs) for fixed SIRA when solving System 1 (non totally unimodular matrix) or System 2 (totally unimodular matrix). Fig. 5 plots the compilation times for larger loops (buffers and fixed hamiltonian). For loops larger than 300 nodes, the compilation time of System 1 becomes more considerable. The error ratio, induced by ceiling the μ variable as solved by System 2 compared to System 1, is depicted in Fig. 6. As can be seen, such error ratio asks us to improve the results of System 2 by re-optimizing the μ variables in a cleverer method as done in [12].

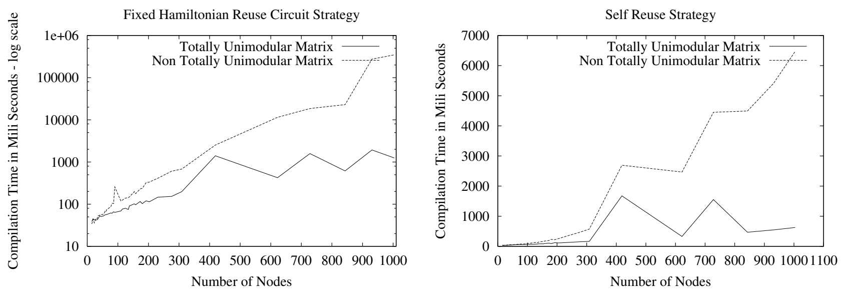


Fig. 5. Compilation Time versus the Size of the DDGs

8 Conclusion

This article presents a new approach consisting in virtually building an early cyclic register allocation before code scheduling, with multiple register types and delays in reading/writing. This allocation is expressed in terms of reuse edges and reuse distances to model the fact that two statements use the same register as storage location. An intLP model gives optimal solution with reduced constraint matrix size, and enables us to make a tradeoff between ILP loss (increase of MII) and number of required registers.

The spilling problem is left for future work. We believe that it is important to take it in consideration *before* instruction scheduling, and our framework should be very convenient for that.

When considering VLIW and EPIC/IA64 processors with reading/writing delays, we are faced to some difficulties because of the possible non-positive distance circuits

[4] counted as the time for generating and solving the intLP systems, and building the allocated DDGs.

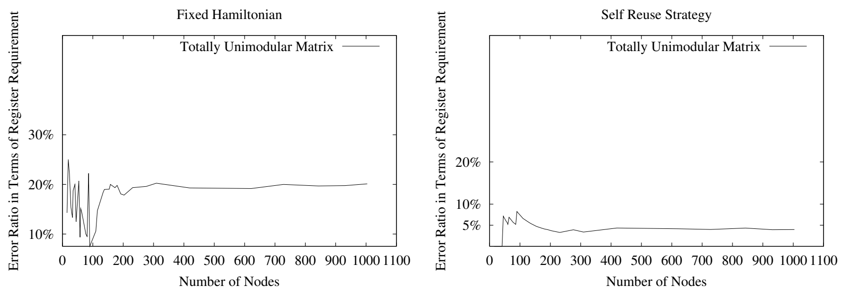


Fig. 6. Error Ratio in Terms of Register Requirement, Induced by System 2, versus the Size of the DDGs

that we prohibit. However, we allow anti-dependences to have non-positive latencies, because this amounts to consider that the destination register is not alive during the execution of the instruction and can be used for other variables. Since pipelined execution time is increasing, this feature becomes crucial in VLIW and EPIC codes to reduce the register requirement.

Each reuse decision implies loop unrolling with a factor depending on reuse circuits for each register type. The unrolling transformation can be applied before the software pipelining pass (the inserted anti-dependences restrict the scheduler and satisfy register constraints) or after it during code generation step. It is better to unroll the loop after software pipelining in order to do not increase the scheduling complexity under resources constraints. Optimizing the unrolling factor is a hard problem and no satisfactory solution exists until now. However, we do not need loop unrolling in the presence of a rotating register file. We only need to seek a unique hamiltonian reuse circuit. The penalty for this constraint is at most one extra register than the optimal for the same II.

Since computing an optimal cyclic register allocation is intractable in real loops, we have identified one polynomial subproblem by fixing reuse edges. With this polynomial approach, we can compute $MII(\rho)$ for a given reuse configuration and a given register count ρ. We can also heuristically find a register usage for one given II.

Our experiments show that disabling sharing of registers with a self reuse strategy isn't a good reuse decision in terms of register requirement. We think that how registers are shared between different statements is one of the most important issues, and preventing this sharing by self reuse strategy consumes much more registers than needed by other reuse decisions.

References

1. D. Berson, R. Gupta, and M. Soffa. URSA: A Unified ReSource Allocator for Registers and Functional Units in VLIW Architectures. In *Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 243–254, Orlando, Florida, Jan. 1993.
2. D. de Werra, C. Eisenbeis, S. Lelait, and B. Marmol. On a Graph-Theoretical Model for Cyclic Register Allocation. *Discrete Applied Mathematics*, 93(2-3):191–203, July 1999.

3. J. C. Dehnert, P. Y.-T. Hsu, and J. P. Bratt. Overlapped Loop Support in the Cydra 5. In *Proceedings of Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–38, New York, Apr. 1989. ACM Press.
4. A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Minimizing Register Requirements of a Modulo Schedule via Optimum Stage Scheduling. *International Journal of Parallel Programming*, 24(2):103–132, Apr. 1996.
5. W. fen Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM Latencies with an Integrated Memory Hierarchy Design. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, Nuevo Leone, Mexico, Jan. 2001.
6. D. Fimmel and J. Muller. Optimal Software Pipelining Under Resource Constraints. *International Journal of Foundations of Computer Science (IJFCS)*, 12(6):697–718, 2001.
7. L. J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji. A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs. *Lecture Notes in Computer Science*, 641:176–??, 1992.
8. R. Huff. Lifetime-Sensitive Modulo Scheduling. In *PLDI 93*, pages 258–267, Albuquerque, New Mexico, June 1993.
9. J. Janssen. *Compilers Strategies for Transport Triggered Architectures*. PhD thesis, Delft University, Netherlands, 2001.
10. C. E. Leiserson and J. B. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, 6:5–35, 1991.
11. J. Llosa. *Reducing the Impact of Register Pressure on Software Pipelined Loops*. PhD thesis, Universitat Politecnica de Catalunya (Spain), 1996.
12. Q. Ning and G. 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, Jan. 1993. ACM Press.
13. 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.
14. F. Sanchez and J. 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.
15. A. Sawaya. *Pipeline Logiciel: Découplage et Contraintes de Registres*. PhD thesis, Université de Versailles Saint-Quentin-En-Yvelines, Apr. 1997.
16. Schlansker, B. Rau, and S. Mahlke. Achieving High Levels of instruction-Level Parallelism with Reduced Hardware Complexity. Technical Report HPL-96-120, Hewlet Packard, 1994.
17. M. M. Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-Independent Storage Mapping for Loops. *ACM SIG-PLAN Notices*, 33(11):24–33, Nov. 1998.
18. W. Thies, F. Vivien, J. Sheldon, and S. Amarasinghe. A Unified Framework for Schedule and Storage Optimization. *ACM SIGPLAN Notices*, 36(5):232–242, May 2001.
19. S.-A.-A. Touati. EquiMax: A New Formulation of Acyclic Scheduling Problem for ILP Processors. In *Interaction between Compilers and Computer Architectures*. Kluwer Academic Publishers, 2001. ISBN 0-7923-7370-7.
20. S.-A.-A. 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, Apr. 2001. ACM.
21. S.-A.-A. Touati. *Register Pressure in Instruction Level Parallelisme*. PhD thesis, Université de Versailles, France, June 2002. ftp.inria.fr/INRIA/Projects/a3/touati/thesis.
22. J. Wang, A. Krall, and M. A. 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.