Parallel Processing Letters
© World Scientific Publishing Company

Early Periodic Register Allocation on ILP Processors

Sid-Ahmed-Ali TOUATI
PRiSM, University of Versailles, France

and

Christine EISENBEIS
INRIA-Futurs, Orsay Parc Club, France

Received December 2003
Revised April 2004
Communicated by Jean-Luc GAUDIOT

ABSTRACT

Register allocation in loops is generally performed after or during the software pipelining process. This is because doing a conventional register allocation as a first step without assuming a schedule lacks the information of interferences between values live ranges. Thus, the register allocator may introduce an excessive amount of false dependences that dramatically reduce the ILP (Instruction Level Parallelism). We present a new theoretical framework for controlling the register pressure before software pipelining. This is based on inserting some anti-dependence edges (*register reuse* edges) labeled with *reuse distances*, directly on the data dependence graph. In this new graph, we are able to fix the register pressure, measured as the number of simultaneously alive variables in any schedule. The determination of register and distance reuse is parameterized by the desired minimum initiation interval (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 on conventional register sets or on rotating register files. We give an optimal exact model, and an approximation that generalizes the Ning-Gao [22] buffer optimization method. We provide experimental results which show good improvement compared to [22]. Our theoretical model considers superscalar, VLIW and EPIC/IA64 processors.

Keywords: Instruction Level Parallelism, Register Allocation, Register Requirement, Software Pipelining, Integer Linear Programming, Code Optimization.

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 minimum initiation interval MII of the DDG : we refer here to MII_{dep} or MII_{rec} since we will not consider any resource constraint.

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 level of ILP, even if the scheduler has extracted it at compile time. Of course, someone could argue that spill operations exhibit high locality, and hence likely produce cache hits, but still we cannot assert it at compile time, unless the compiler makes very optimistic assertions on data locality and cache behavior. The hardware does not always react as it should do, especially when memory hierarchy is involved.

Even if we make optimistic predictions on cache behavior, and let us suppose that the spilled data always remains in the caches, we still have to solve many problems to guarantee that spill operations do not cause damages : our previous studies [16,17] about performing memory requests on modern microprocessors showed that any load and store operations (array references) exhibit high potential conflicts that make them to execute serially even if they are data independent. This is not because of FUs limitations, but because of micro-architectural restrictions and simplifications in the memory disambiguation mechanisms (load/store queues) [16] and possible banking structure in cache levels [17]. Such fine-grain micro-architectural characteristics prevent independent memory requests from being satisfied in parallel are not taken into account by current ILP schedulers. Thus, these possible conflicts between independent loads and stores may cause severe performance degradation even if enough ILP and FUs exist, and even if the data is located in the cache [17]. In other words, memory requests are a serious source of performance troubles that are difficult to fix at compile time. The authors in [11] related that about 66% of application execution times are spent to satisfying memory requests. Thus, we should avoid requesting data from memory if possible.

Another reason for handling register constraints prior to ILP scheduling is that register constraints are much more complex (from the theoretical perspective) than resource constraints. Scheduling under 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 us to iteratively apply scheduling followed by spilling until reaching a solution. Even if we can experimentally reduce the backtracking as in [32], this iterative aspect adds a high algorithmic complexity factor to the pass integrating both register allocation and scheduling.

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 (periodic) register allocation with R available registers [6,23]. We can use either loop unrolling [6,20], inserting move operations [14], or a hardware rotating register file when available [23]. Therefore, a great amount of work tries to schedule a loop such that does not use more than R values simultaneously alive. Usually, a schedule that minimizes the register need under a fixed II. [15,31,22,21,8,12,18]. In this paper we directly work on the loop DDG and modify it in order to fix the register requirement of any further subsequent software pipelining pass. This idea is already present in [2,29] for DAGs and use the concept of *reuse* edge or vector developed in [27,28].

Our article is organized as follows. Sect. 2 defines our loop model and a generic ILP processor. Sect. 3 starts the study with a motivating example. The problem of periodic 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. Sect.7 solves a typical problem for VLIW and EPIC/IA64 codes. A network flow solution for the problem of cyclic register allocation is described in Sect. 8. Finally, we synthesize our experiments in Sect. 9 before concluding. For fluidity of the reading, only the most important formal proofs are presented in this paper. The complete theoretical proofs are provided in the cited references.

2. Loop Model

We consider a simple innermost loop (without branches, with possible recurrences). It is represented by a graph $G = (V, E, \delta, \lambda)$, such that :

- V is the set of the statements in the loop body. The instance of the statement u (an operation) of the iteration i is noted $u(i)$. By default, the operation u denotes the operation $u(i)$;
- E is the set of precedence constraints (flow dependences, or other serial constraints), any edge e has the form $e = (u, v)$, where $\delta(e)$ is the latency of the edge e in terms of processor clock cycles and $\lambda(e)$ is the distance of the edge e in terms of number of iterations.
- 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 whether they refer to values to be stored in registers or not :

1. $V_{R,t}$ is the set of values to be stored in registers of type $t \in \mathcal{T}$. We assume that each statement $u \in V$ writes into at most one register of a type $t \in \mathcal{T}$. Statements which define multiple values with different types are accepted in our model if they do not define more than one value of a certain type. For instance, statements that write into a floating point register and update a conditional register are taken into account in our model. We denote by u^t the value of type t defined by the statement u ;
2. $E_{R,t}$ 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 :

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

Our model requires that each statement writes into at most one register of each type because this characteristics enable us to make some formal proofs using graph theory (shown later). If we target a processor that allows the opportunity to write multiple results of the same type, we can easily relax this restriction by node splitting : a statement u writing multiple results $u_1, ..., u_k$ can be splitted into k dummy nodes, each one writing a single result of the considered type. These k nodes must be connected by a null circuit in order to enforce them to be scheduled at the same clock cycle. Also, we must add any additional data dependence arcs from/to each dummy node to preserve code semantics.

To consider static issue processors (VLIW or IA64) 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 (architecturally visible). We define two delay (offset) functions $\delta_{r,t}$ and $\delta_{w,t}$ in which :

$$\begin{array}{ll} \delta_{w,t}: & V_{R,t} \rightarrow \mathbb{N} \\ & u \mapsto \delta_{w,t}(u) /\ 0 \leq \delta_{w,t}(u) \\ & \text{the write cycle of } u^t \text{ into a register of type } t \text{ is } \sigma(u) + \delta_{w,t}(u) \\ \delta_{r,t}: & V \rightarrow \mathbb{N} \\ & u \mapsto \delta_{r,t}(u) /\ 0 \leq \delta_{r,t}(u) \leq \delta_{w,t}(u) \\ & \text{the read cycle of } u^t \text{ from a register of type } t \text{ is } \sigma(u) + \delta_{r,t}(u) \end{array}$$

For superscalar or EPIC codes, $\delta_{r,t}$ and $\delta_{w,t}$ are equal to zero. A software pipelining is a function σ that assigns to each statement u a scheduling time (in terms of clock cycle) that satisfies the precedence constraints. It is defined by an initiation interval, noted II, and the scheduling time σ_u for the operations of the first iteration. Iteration i of operation u is scheduled at time $\sigma_u + (i-1).II$. For all edge $e = (u, v) \in E$, this 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 will denote in the sequel as MII.

We consider now a register pressure ρ (number of available registers) and all the schedules that have no more than ρ simultaneously alive variables. Any following register allocation will induce new dependencies 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. Starting Example

3.1. Basic Idea

The heart of our method is based on the following observation: there exist some DDGs for which we can guarantee a bounded or fixed register requirement for any valid schedule. The basic case is when the DDG of some loop is a simple single circuit, with dependence distance λ, then any software pipelining of this loop will have *exactly* $R = \lambda$ simultaneously alive variables. On the other hand, circuits in the data dependence graphs are responsible for throughput limitation: a data dependence circuit induces a constraint on the expected instruction level parallelism since any initiation interval II must be greater than the value of its critical ratio $MII = \frac{\delta}{\lambda}$, where δ is the sum of delays of the circuit edges. In this simple case, we get the inequality $II \geq MII = \frac{\delta}{\lambda} = \frac{\delta}{R}$, which express the trade-off between loop parallelism and register pressure.

In this paper we generalize this property from a single circuit to general graphs. In the DDG, we carefully add artificial "reuse" edges such that any such edge is constrained in a circuit, and such that we are able to measure the resulting register pressure, and the resulting MII as well.

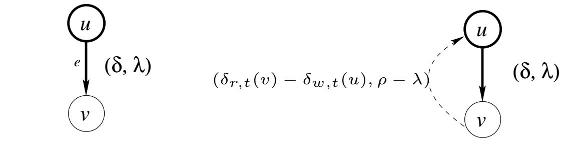
3.2. Data Dependences and Reuse Edges

We give now more intuitions to the new edges that we add between two statements. These edges represent possible reuse by the second operation of the register location released by the first operation. This can be viewed as a variant of [2] or [27,28].

Let us consider the following loop :

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

The DDG of this loop contains only one flow dependence, i.e., 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



(a) Simple DDG (b) Antidependence

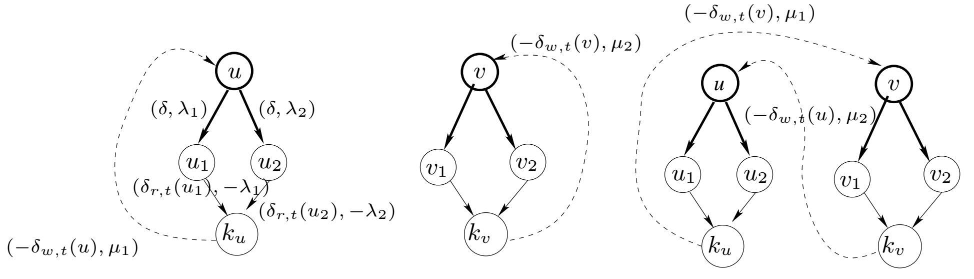
Figure 1: Simple Example

we only have $\rho = 5$ available registers (R_1, ... , R_5). The different instances of u can use only $\rho = 5$ registers to periodically 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 $\delta_{r,t}(v) - \delta_{w,t}(u)$, except for superscalar codes where the latency is 1 (static sequential semantics, i.e., straight line code). 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 [27], must in turn be counted when computing the new minimum initiation interval since a new circuit is created :

$$MII \geq \frac{\delta(e) + \delta_{r,t}(v) - \delta_{w,t}(u)}{\rho}$$

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 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 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 [19], 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 $-\delta_{w,t}(v)$, except for superscalar codes where the latency is 1 (straight line code). The distance of this anti-dependence is $\mu_{u,v}$ to be defined. We will see in a further section that the register requirement can be expressed in terms of $\mu_{u,v}$.


$(-\delta_{w,t}(v), \mu_1)$
$(-\delta_{w,t}(v), \mu_2)$
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
$(-\delta_{w,t}(u), \mu_1)$
v
v_1
v_2
k_v
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

Figure 2: Killing Tasks

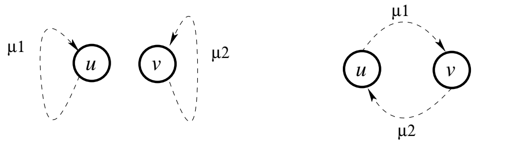
Hence, controlling register pressure means, first, determining which operation should reuse the register killed by another operation (*where should anti-dependences be added?*). Secondly, we have to determine variable lifetimes, or equivalently register requirement (*how many iterations later (μ) should reuse occur*)? As defined by the exact algebraic formulas of MII and ρ, 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 pressure 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. For this simple example with two values, we have only two choices for reuse decisions. However, a general loop with n statements has an exponential number of possible reuse graphs.

There are three main constraints that the resulting DDG must meet. First, it must be schedulable by software pipelining, and the sum of distances along each circuit must be positive. Note that there is no reason why the μ coefficients should be non negative : this means that we are able to allow an operation $u(i)$ to reuse a register freed by an operation $v(i+k)$ since a pipelined execution may schedule $v(i+k)$ to be killed before the issue of $u(i)$ even if $v(i+k)$ belongs to the k^{th} iteration later. Second, the number of registers used by any allocation scheme must be lower or equal to the number of available registers. Third and last, the critical ratio (MII) must be kept as lower as possible in order to save ILP. The next section gives a formal definition of the problem and provides an exact formulation.

4. Problem Description

The reuse relation between the values (variables) is described by defining a new graph called *a reuse graph*. 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. The resulting 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). In the next section, we give a formal definition and model of the register allocation problem based on reuse graphs. We denote by $G_{\to r}$ the DDG associated to a reuse decision r.



(a) First Reuse Graph

(b) Second Reuse Graph

Figure 3: Reuse Graphs

4.1. Reuse Graphs

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

Definition 1 (Reuse Graph) *Let $G = (V, E, \delta, \lambda)$ be a DDG. The reuse graph $G^r = (V_{R,t}, E_r, \mu)$ of type t is defined by the set of values of type t, the set of edges representing reuse choices, and the distances. Two values are connected in G^r by an edge $e = (u^t, v^t)$ iff $v^t\,(i + \mu(e))$ reuses the register freed by $u^t(i)$.*

We call E_r the set of reuse edges and μ a reuse distance. Given $G^r = (V_{R,t}, E_r, \mu)$ a reuse graph of type t, we report the register reuse decision to the DDG $G = (V, E, \delta, \lambda)$ by adding an anti-dependence from k_{u^t} to v iff $e = (u, v)$ is a reuse edge. The distance of this anti-dependence is $\mu(e)$.

Our reuse graph may seem somewhat similar to the interference graph proposed by Chaitin. In particular, the fact that two values are connected in G^r by an edge if the share the same register seems akin to edges in Chaitin's interference graph if register lifetimes do not overlap. However, two fundamental aspects make our approaches radically different : first, the interference graph models all possible reuse decisions (all interferences are reported), while the reuse graph models a fixed reuse choice ; second, the interference graph does not take into account the iteration distances, so the possible reuse decisions can be expressed for only two consecutive iterations. This latter aspect is important since it allows our reuse graph to capture the pipelined execution of a loop where multiple and distant iterations can be executed in parallel.

A reuse graph must obey some constraints to be valid :

1. the resulting DDG must be schedulable, and all circuits must have positive distances;
2. each statement must reuse only one freed register, and each register must be reused by only one statement.

Note that a schedulable DDG does not mean that all its circuits have positive distances. This is because our model admits explicit reading/writing offsets, thus some edges may have a non-positive latency.

The second constraint means that the reuse scheme is the same at each iteration. This condition results in the following lemma.

Lemma 1 *[30] 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).

First, let us assume a reuse graph with a single circuit. If such reuse graph is valid, we can build a periodic 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 *[30] Let $G = (V, E, \delta, \lambda)$ be a DDG and $G^r = (V_{R,t}, E_r, \mu)$ be a valid reuse graph of type t associated with it. If only one reuse circuit C in G^r exists, then the reuse graph defines a periodic register allocation in G for values of type t with exactly $\mu(C)$ registers, if we unroll the loop $\rho = \mu(C)$ times.*

Proof. Let us unroll $G_{\rightarrow r}$ $\rho = \mu_t(C)$ times : each statement $u \in V$ has now ρ copies in the unrolled loop. We note u_i the i^{th} copy of the statement $u \in V_{R,t}$. To prove this theorem, we explicitly express the periodic register allocation, directly on $G_{\rightarrow r}$ after loop unrolling, i.e. we assign registers to the statements of the new loop body (after unrolling). We consider two cases, as follows.

Case 1 : all the μ distances are non-negative For the clarity of this proof, we illustrate it by the example of Fig. 4 which builds a periodic register allocation with 3 registers for Fig. 2.(b) in which we set $\mu_1 = 2$ and $\mu_2 = 1$: we have unrolled this loop 3 times. We allocate $\mu_t(C) = 3$ registers in the unrolled loop as described in Algorithm 1.

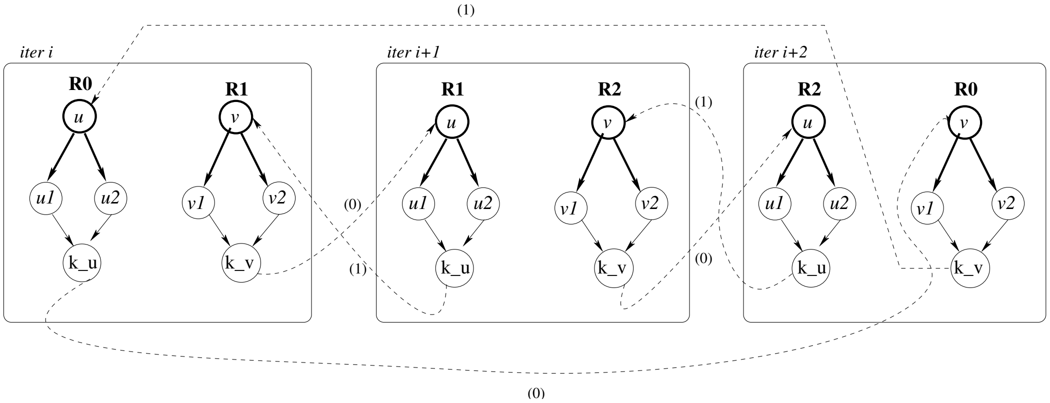


Figure 4: Periodic Register Allocation with One Reuse Circuit

1. We choose an arbitrary value u^t in $V_{R,t}$. It has ρ distinct copies in the unrolled loop. So, we allocate ρ distinct registers to these copies. We are sure that such values exist in the unrolled loop body because $\rho > 0$.

2. Since the reuse relation is valid, we are sure that for each reuse edge (u, v), the killing time of each value $u^t(i)$ is scheduled before the definition time 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 $u^t(i)$. We are sure that $v^t\big((i+\mu^t_{u,v}) \bmod \rho\big)$ exists in the unrolled loop body because $\mu^t_{u,v} \geq 0$. For instance in Fig. 4, we allocate the same register R_1 to $u(1)$ and $v((1 + 2) \bmod 3) = v(0)$. Also, we allocate the register R_0 to $v(2)$ and to $u((2 + 1) \bmod 3) = u(0)$. Finally, we allocate R_2 to both $v(1)$ and $u((1 + 1) \bmod 3) = u(2)$.

3. We follow the other reuse edges to allocate the same register to the two values $v(i)$ and $v'\big((i + \mu^t_{u,v}) \bmod \rho\big)$ iff $reuse(v) = v'$. We continue in the reuse circuit image until all values in the loop body are allocated.

Since the original reuse circuit image is duplicated ρ times in the unrolled loop, and since each reuse circuit image in the unrolled loop consumes one register, we use in total $\rho = \mu_t(C)$ registers. Dashed lines in Fig. 4 represent anti-dependences with their corresponding distances after the unrolling.

Case 2 : there exists a non-positive μ distance In that case, it is always possible to come back to the previous case by a retiming technique [19,30], since loop retiming can make all the distances non-negative. □.

As a consequence to the previous theorem, we deduce how to build a periodic register allocation for an arbitrary number of reuse circuits.

Theorem 2 *[30] 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 periodic register allocation for G with exactly $\mu_t(G^r)$ registers of type t if we unroll the loop α times where*

Algorithm 1 Periodic Register Allocation with a Single Reuse Circuit

```
Require: a DDG $G_{\rightarrow r}$ associated to a valid reuse relation $reuse_t$
  unroll it $\rho = \mu_t(C)$ times {this create $\rho$ copies for each statement}
  for all $w$ a node in the unrolled DDG do
    alloc($w$)$\leftarrow \perp$ {initialization}
  end for
  choose $u \in V_{R,t}$ {an original node}
  for all $u_i$ in the unrolled DDG do {each copy of $u$}
    alloc($u_i$) $\leftarrow$ ListOfAvailableRegisters.pop()
    $n \leftarrow u_i$
    $n' \leftarrow v_{(i+\mu^t_{u,v}) mod\ \rho}$ {where $reuse(u) = v$}
    while alloc($n'$)=$\perp$ do
      alloc($n'$)$\leftarrow$alloc($n$)
      $n \leftarrow n'$
      $n' \leftarrow n''$ {where $(k_{n'}, n'')$ is an anti-dependence in the unrolled loop}
    end while
  end for
```

:

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

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

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

Corollary 1 *[30] 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*

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

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

We should make an important remark regarding loop unrolling. Indeed, we can avoid loop unrolling before the scheduling step in order to not increase the DDG size, and hence to not exhibit more statements to the scheduler. Since we allocate registers directly into the DDG by inserting loop carried anti-dependencies, the DDG can be scheduled without unrolling it (but the inserted anti-dependence edges restrict the scheduler). In other words, loop unrolling can be applied at the code generation step (after code scheduling) in order to apply the register allocation computed before scheduling.

The fact that the unrolling factor may theoretically be 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 that we can reuse. In that case, the meeting graph framework [10] can help to control or reduce this unrolling factor.

From all above, we deduce a formal definition of the problem of optimal periodic 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 [30]. The following section gives an exact formulation of SIRA.

4.2. Exact Formulation for SIRA

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.

Our SIRA exact model uses the linear formulation of the logical implication ($\Longrightarrow$) by introducing binary variables, as previously explained in [30]. We want to express the following system by linear constraints :

$$g(x) \geq 0 \Longrightarrow h(x) \geq 0$$

in which g and h are two linear functions of a variable x. If the domain set of x is bounded, the system is linearized by introducing a binary variable α as follows :

$$\begin{cases} -g(x) - 1 \geq \alpha \underline{g} \\ h(x) \geq (1-\alpha)\underline{h} \\ \alpha \in \{0, 1\} \end{cases}$$

where $\underline{g}$ and $\underline{h}$ are two known finite lower bounds for $(-g-1)$ and h respectively. It is easy to deduce the same formalization for the equivalence ($\Longleftrightarrow$). Now, we are ready to provide our exact formulation.

We first write constraints to compute reuse edges with their distances so that the associated DDG is schedulable. Therefore we look for the existence of at least one software pipelining schedule for a fixed desired critical circuit II.

Basic Variables

- a schedule variable $\sigma_u \geq 0$ for each operation $u \in V$, including one for each killing node k_{u^t}. Note that these schedule variables do not represent the final schedule under resource constraints (that will be computed after our SIRA pass), but they only represent intermediate variables for our SIRA formulation ;

- 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

- bound the scheduling variables by assuming a constant L as a worst schedule time of one iteration : $\forall u \in V \ : \ \sigma_u \leq L$
- data dependences (the existence of at least one valid software pipelining schedule)
$$\forall e = (u, v) \in E : \ \sigma_u + \delta(e) \leq \sigma_v + II \times \lambda(e)$$
- schedule killing nodes for consumed values : $\forall u^t \in V_{R,t}, \ \forall v \in Cons(u^t) \, | e = (u, v) \in E_{R,t} : \sigma_{k_{u^t}} \geq \sigma_v + \delta_{r,t}(v) + \lambda(e) \times II$
- there is an anti-dependence between k_{u^t} and v if (u, v) is a reuse edge :
$$\forall t \in \mathcal{T}, \ \forall (u, v) \in V^2_{R,t} \ : \theta^t_{u,v} = 1 \Longrightarrow \sigma_{k_{u^t}} - \delta_{w,t}(v) \leq \sigma_v + II \times \mu_{u,v}$$
- if there is no register reuse between two values $(reuse_t(u) \neq v)$, then $\theta^t_{u,v} = 0$. The anti-dependence distance $\mu^t_{u,v}$ must be set to 0 in order to not be accumulated in the objective function. $\forall t \in \mathcal{T}, \ \forall (u, v) \in V^2_{R,t} : \theta^t_{u,v} = 0 \Longrightarrow \mu^t_{u,v} = 0$

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

- a register can be reused by one operation : $\forall t \in \mathcal{T}, \forall u \in V_{R,t} : \sum_{v \in V_{R,t}} \theta^t_{u,v} = 1$
- a statement can reuse one released register : $\forall t \in \mathcal{T}, \forall u \in V_{R,t} : \sum_{v \in V_{R,t}} \theta^t_{v,u} = 1$

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

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

The size of this system is bounded by $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|E| + |V|^2)$ linear constraints.

As previously mentioned, our model includes writing and reading offsets. The non-positive latencies of the introduced anti-dependences generate a specific problem. Indeed, the existence of a valid periodic schedule does not prevent some circuits C in the constructed DDG from having a non-positive distance $\lambda(C) \leq 0$. Note that this problem does not occur for superscalar (sequential) codes, because the introduced anti-dependences have positive latencies (sequential semantics). We will discuss this problem further.

In the previous formulation, we have fixed the II (desired critical circuit) and looked for a schedule that minimizes the register pressure. But we can also do the reverse, this means just formulate the register constraints given by the processor and look for the minimal II for which the system is satisfiable. This can not be simply done by adding “min II” in the formulation because some inequalities are not linear in II. But alternatively we can perform a binary search on II. Such binary search can be used because we have formally proved in [30] that if a schedule exists at initiation interval II, then another schedule exists at initiation interval $II + 1$ that requires at most the same number of registers. This result is conditioned by the fact that the parameter L, which is the total schedule time of one iteration, must be non constrained, i.e., we must be able to extend L with a bounded factor when incrementing II.

The unrolling degree is left free and over any control in SIRA formulation. This factor may theoretically grow exponentially because of the lcm function. 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 in discrete mathematics : as far as we know, there is not a satisfactory solution for it.

Software solutions such as the meeting graph have already been mentioned [10]. Alternatively, a hardware solution exists too, namely rotating register files, that do not imply loop unrolling for performing periodic register allocation. This feature is studied in the next section.

5. Rotating Register Files

A rotating register file [7,23,25] is a hardware feature that moves (shifts) implicitly architectural registers in a periodic 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) : $R_i \mapsto R'_{(i+RRB) \bmod s}$ 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 (as defined by the exact algebraic formula of the unrolling factor), 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.

Definition 2 (Hamiltonian Ordering) *Let $G = (V, E, \delta, \lambda)$ be a loop and $G^r = (V_{R,t}, E_r, \mu)$ a valid reuse graph of type $t \in \mathcal{T}$. A hamiltonian ordering ho_t of this loop according to its reuse graph is a function defined by :*

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

such that $\forall u, v \in V_{R,t} : (u, v) \in E_r \iff ho_t(v) = \Big(ho_t(u) + 1\Big)$ *mod* $|V_{R,t}|$

Fig. 5 is an example of a hamiltonian ordering of a reuse graph with 5 values. The existence

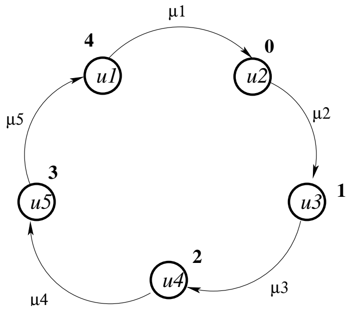


Figure 5: Hamiltonian Ordering

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

Theorem 3 *[30] 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 periodic 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*

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

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.

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 include in the model the bounding constraints of the hamiltonian ordering variables :
$$\forall u^t \in V_{R,t} \ : \qquad ho_{u^t} < |V_{R,t}|$$

3. we add the linear constraints of the modulo hamiltonian ordering : $\forall u, v \in V_{R,t}^2$:
$$\theta^t_{u,v} = 1 \Longleftrightarrow ho_{u^t} + 1 = |V_{R,t}| \times \beta^t_{u,v} + ho_{v^t}$$
where $\beta^t_{u,v}$ is a binary variable that holds to the integer division of $ho_{u^t} + 1$ on $|V_{R,t}|$.

We have expanded the exact SIRA intLP model by at most $\mathcal{O}(|V|^2)$ variables and $\mathcal{O}(|V|^2)$ linear constraints.

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 *[30] 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.

6. Fixing Reuse Edges

In [22], 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 the 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.

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

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

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

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

Theorem 4 *[30] 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 ± 1.* Consequently, we can use polynomial algorithms to solve this problem [26] of finding the minimal value for the product $II \times \rho$.

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.[*] 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 ($|V_{R,t}|$). We think that we can avoid

[*] Of course, if we have $MII = II = 1$ (case of parallel loops for instance), the solution becomes optimal since the constraints matrix becomes identical to Theorem 4.

ceiling μ by considering the already computed σ variables, as done in [22]. These authors proposed a method for buffers, which is difficult to generalize to other reuse decisions. A better method that recomputes the original μ in a cleverer way (instead of ceiling them) is described in [22].

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 register pressure 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 equivalent to buffer minimization problem as described in [22].

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 [31].

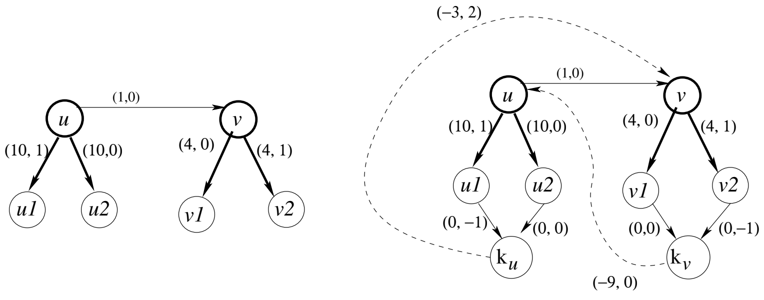
3. If a rotating register file is present, we can fix an arbitrary (or with a cleverer method) hamiltonian reuse circuit among statements.

As explained before, our model includes writing and reading offsets. The non-positive latencies of the introduced anti-dependences generate a specific problem for VLIW and EPIC codes. The next section solves this problem.

7. Eliminating Non-Positive Circuits

The non-positive latencies of the introduced anti-dependences allow us to have more opportunities to optimize registers. This is because we would be able to access a register during the whole execution period of the statement writing in it : in other words, a register would not be busy during the complete execution period of the producer. Furthermore, we would be able to assign to an operation $u(i)$ the register freed by another operation $v(i + k)$ belonging to the k^{th} iteration later. This is possible in software pipelining since the execution of the successive iterations is overlapped. Such reuse choices are exploited by the fact that the reuse distances can be non-positive, leading to circuits with possible non-positive distances.

From the scheduling theory, circuits with non-positive distances do not prevent a DDG from being scheduled (if the latencies are non-positive too). But such circuits impose hard scheduling constraints that may not be satisfiable by resource constraints in the subsequent



(a) Original Loop (b) DDG associated with a Valid Reuse Graph

Figure 6: Nonpositive Circuits

pass of instruction scheduling. This is because circuits with non-positive distances impose scheduling constraints of type “not later than” that are similar to real time constraints. The scheduling theory cannot guarantee the existence of at least a schedule under a limited number of execution resources. Therefore these circuits have to be forbidden.

As an illustration, look at Fig. 6, where flow dependences are in bold edges and statements writing into registers are in bold circles. In the original loop shown in Part (a), there exists a dependence path from u to v with a distance equal to zero (the path is in the loop body). A reuse decision as shown in Part (b) may assign the same register to $u(i)$ and $v(i)$. This creates an anti-dependence from $v(i)$'s killer to $u(i)$. Since the latency of the reuse edge (k_v, u) is negative (-9) and the latency of the path $u \rightsquigarrow k_v$ is 5, the circuit (v, k_v, u, v) with a distance equal to zero does not prevent the associated DDG from being modulo scheduled (since the precedence constraints can be easily satisfied), but may do so in the presence of resource constraints.

Alain Darte [5] provides a solution. We add a quadratic number of retiming constraints to avoid non-positive circuits. We define a retiming r_e for each edge $e \in E$. We have then a shift $r_e(u)$ for each node $u \in V$. We declare then an integer $r_{e,u}$ for all $(e, u) \in (E \times V)$. Any retiming r_e must satisfy the following constraints:

$$\begin{array}{ll} \forall e' = (u', v') \neq e, & r_{e,v'} - r_{e,u'} + \lambda(e') \geq 0 \\ \text{for the edge } e = (u, v), & r_{e,v} - r_{e,u} + \lambda(e) \geq 1 \end{array} \tag{3}$$

Note that an edge $e = (k_{u^t}, v) \in E'$ is an anti-dependence, i.e., its distance is $\lambda(e) = \mu^t_{u,t}$, to be computed. Since we have $|E|$ distinct retiming functions, we add $|E| \times |V|$ variables and $|E| \times |E|$ constraints. The constraint matrix is totally unimodular, and it does not alter the total unimodularity of System 2. The following lemma proves that satisfying System 3 is a necessary and sufficient condition for building a DDG $G_{\rightarrow r}$ with positive circuits distances.

Lemma 2 *[30] Let $G_{\rightarrow r}$ the solution graph of System 1 or System 2. Then: System 3 is satisfied $\Longleftrightarrow$ any circuit in $G_{\rightarrow r}$ has a positive distance $\lambda(C) > 0$.*

If we do not require to include an integer solver inside a compiler, we propose the following network flow formalization for System 2.

8. A Network Flow Solution for Fixed SIRA

In this section, we give a network flow algorithm for solving the totally unimodular System 2. Let forget the problem of non-positive circuits for the moment (we show further how to fix it). The matrix form of System 2 is :

$$\left\{\begin{array}{ll} \text{Minimize} & (1\,0)(\mu'\;\sigma) \\ & \\ \text{Subject to:} & \\ \left[\begin{array}{c|c} I & U \\ 0 & \end{array}\right]\begin{pmatrix}\mu \\ \sigma\end{pmatrix} \geq & \begin{pmatrix} -\delta_w \\ \delta - II \times \lambda \end{pmatrix} \\ & \\ \mu', \sigma & \in \mathbb{Z} \end{array}\right. \tag{4}$$

where μ' is the set of $II \times \mu$ variables, σ is the set of scheduling variables, and U the incidence matrix of the DDG (including anti-dependences and killing nodes). In order to transform this system into a network flow problem, we take its dual form :

$$\left\{\begin{array}{ll} \text{Maximize} & (-\delta_w \;\; \delta - II \times \lambda)(f) \\ & \\ \text{Subject to:} & \\ \left[\begin{array}{cc} I & 0 \\ \hline \multicolumn{2}{c}{U^T} \end{array}\right]\left(\; f \;\right) = & \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ & \\ f \in & \mathbb{N} \end{array}\right. \tag{5}$$

where U^T is the transpose of the incidence matrix U and f is the set of dual variables. Then, the constraints of this system are (after converting the objective function to minimization) :

$$\left\{\begin{array}{l} \text{Minimize } \sum_{e \in E'} \delta_w(e) \times f(e) \\ \qquad + \sum_{e \in E - E'} (II \times \lambda(e) - \delta(e)) \times f(e) \\ \\ \text{Subject to:} \\ f(e) = 1,\ \forall e \in E' \text{ (i.e, an antidependence arc)} \\ \sum_{? \xrightarrow{e} u} f(e) - \sum_{u \xrightarrow{e} ?} f(e) = 0,\ \forall u \in V \text{ (flow constraints)} \\ \\ f(e) \in \mathbb{N},\ \forall e \in E \end{array}\right. \tag{6}$$

System 6 is indeed a min cost flow problem. The network is the graph $G = (V, E)$, where the anti-dependences must have a flow equal to one,[†] and the other arcs have unbounded capacities. The cost of the flow is $\delta_w(v)$ for each anti-dependence arc $(k_u, v) \in E'$, and $II \times \lambda(e) - \delta(e)$ for the other arcs $\in E - E'$. There exist a lot of polynomial time algorithms for computing optimal flows with minimal costs [24,13,3].

8.1. Back Substitution From Network Flow Solution

After computing f^* an optimal min cost flow solution for System 6, we must come back to the original μ' variables ($= II \times \mu$). For this purpose, we use the complementary

[†] This is done by setting a lower and upper capacity equal to one.

slackness theorem, which gives the relationship between the optimal solutions of the dual system (i.e., System 6) and those of the primal system (System 2) :

- $\forall e \in E$, if $f^*(e) > 0$ then the corresponding constraints in System 2 is an equality constraints (the slack variable is zero) ;

- $\forall e \in E$, if $f^*(e) = 0$ then the corresponding constraints in System 2 is an inequality (the slack variable may be non zero).

In our case, we know that, for each anti-dependence $e = (k_u, v)$, the flow $f^*(e) = 1$. Then, the corresponding constraint is ;

$$\mu'_u + \sigma_v - \sigma_{k_u} = -\delta_w(v) \Longrightarrow \mu'_u = -\delta_w(v) - (\sigma_v - \sigma_{k_u}) \tag{7}$$

It remains to compute the σ variables. Since the optimal flow f^* has been already computed, we must satisfy a set of equality and inequality constraints, depending on the value of the computed flow. For this purpose, we define a graph $G_f = (V, E_f, \delta_f)$ that contains the original set of nodes V. The set of arcs E_f is defined as follows :

- $\forall e = (u, v) \in E - E'$, if $f^*(e) = 0$ then (inequality constraint) add an arc (u, v) in E_f with a cost equal to $\delta_f = \delta(e) - II \times \lambda(e)$;

- $\forall e = (u, v) \in E - E'$, if $f^*(e) > 0$ then (equality constraint) add two arcs (u, v) and (v, u) to E_f, with the costs $\delta_f = \delta(e) - II \times \lambda(e)$ and $\delta_f = -\delta(e) + II \times \lambda(e)$ respectively.

It is easy to see that any potential for the graph G_f is a solution that satisfies the set of our constraints. Then, the σ variables are the potentials of the graph G_f, which we use for computing $\mu' = II \times \mu$ by considering Equation 7.

Now, let us examine the problem on non-positive circuits in VLIW/EPIC codes.

8.2. Eliminating Non-Positive Circuits

As shown in Sect.6, we need to define a retiming r_e for each arc $e \in E$. When we consider the variable substitution $\mu' = II \times \mu$, System 3 is transformed to ;

$$\begin{array}{ll} \forall e' = (u', v') \neq e, & r'_{e,v'} - r'_{e,u'} + \lambda'(e') \geq 0 \\ \text{for the considered arc } e = (u, v), & r'_{e,v} - r'_{e,u} + \lambda'(e) \geq II \end{array} \tag{8}$$

where $r' = II \times r$ and $\lambda' = II \times \lambda$. Note that $\lambda' = \mu'$ if the arc is an anti-dependence.

The dual problem of System 8 asks for seeking a distinct flow f_e for each arc in the DDG G (including anti-dependences). Thus, we have to compute $|E|$ distinct feasible flows on the same network (integer multi-flow problem). The costs of each flow f_e is $-II$ for the considered arc e, and 0 for the other arcs.[‡] Hence, the general formulation of the fixed SIRA problem, using a min cost integer multi-flow formulation, is :

[‡] The cost of the flow f_e is $-II$ for the considered arc e because we transform the problem from maximization to minimization.

$$\left\{\begin{array}{l}
\text{Minimize } \sum_{e\in E'} \delta_w(e)\times f(e) \\
\qquad +\sum_{e\in E-E'}(II\times\lambda(e)-\delta(e))\times f(e) \\
\qquad -II\times\sum_{e\in E} f_e(e) \\
\\
\text{Subject to:} \\
f(e)+\sum_{e'\in E} f_{e'}(e)=1, \forall e\in E' \text{ (anti-dependence)} \\
\\
\sum_{?\xrightarrow{e}u} f(e)-\sum_{u\xrightarrow{e}?} f(e)=0, \forall u\in V \\
\\
\sum_{?\xrightarrow{e}u} f_{e'}(e)-\sum_{u\xrightarrow{e}?} f_{e'}(e)=0, \forall u\in V, \forall e'\in E-E' \\
\\
f(e)\in\mathbb{N}, \forall e\in E \\
f_e(e')\in\mathbb{N}, \forall e,e'\in E
\end{array}\right. \tag{9}$$

Unfortunately, solving exact integer multi-flow problems with algorithmic solutions is not as trivial as in the single flow case, since the complexity of a general min cost integer multi-flow problem is strongly NP-hard [4]. As far as we know, there is not a (combinatorial) polynomial algorithm that would compute an exact solution our integer multi-flow problem, except those algorithms that use integer linear resolution techniques ; they would transform our multi-flow problem to an intLP program, and then they solve it. Since the constraints matrix of our problem is totally unimodular, the exact solution can be found in polynomial time.

9. Experiments

We have developed six tools that perform periodic register allocation as explained in this article. Two optimal ones for SIRA and hamiltonian SIRA (Sect. 4.2 and Sect. 5), and four tools for fixed SIRA (Sect. 6): two of these four tools correspond to the optimal fixed SIRA solutions with System 1 when we fix self reuse edges (Ning and Gao method) and an arbitrary hamiltonian reuse circuit. The other two tools for fixed SIRA correspond to solving the polynomial systems (System 2) with a self reuse and hamiltonian strategy too. We use CPLEX to solve our intLP models. We used a PC under linux, equipped with a PIV 1.2 Ghz processor, and 256 Mo of memory. We did thousands of experiments on several numerical loops extracted from different benchmarks (Spec95, whetstone, livermore, linddot). The data dependence graphs of all these loops are present in [30]. This section presents a summary.

9.1. Optimal and Hamiltonian SIRA

The first set of experiments investigates optimal SIRA versus optimal hamiltonian SIRA (Sect. 4.2 versus Sect. 5). We compare the optimal register requirement of all loops versus varying II (this yield to hundreds of experiments). In most of cases, both 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 (within 5%). This remark has been previously stated in [23]. Regarding the resulted unrolling degrees, we get the following results.

- The unrolling degree is left free from any control in SIRA intLP systems. Even if it may grow exponentially (from the theoretical perspective), experiments show that it is acceptable in most of cases. 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. Here, we advise to use hamiltonian SIRA so that the minimal register need is exactly the unrolling degree, both minimized by the objective function of hamiltonian SIRA. Of course, we do not require loop unrolling in the presence of a rotating register set.

As previously cited, it should be noted that the fact that the unrolling factor may be significantly high 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 [10] can help to control or reduce this unrolling factor.

9.2. Fixed SIRA versus Optimal SIRA

To check the efficiency of our simplified method (Fixed SIRA), we prefer to compare its results against the optimal ones instead of performing a comparison to all the existing techniques in the literature. This is because of three main reasons. First, our method is performed at the DDG level, while the existing register minimization techniques are carried out during loop scheduling. So, we are not considering exactly the same problem. Second, our method is more generic since it takes into account superscalar, VLIW and EPIC/IA64 codes. Third and last, we think that comparing the efficiency of our methods to the optimal results is an acceptable experimental methodology.

We checked the efficiency of two strategies : self reuse strategy as described in [22], and fixing an arbitrary hamiltonian reuse circuit. We choose the former approach as a base for comparison because our work already generalizes their framework.

Resolving the intLP systems of these two strategies become very fast compared to optimal solutions, as can be seen the first part of Fig. 7, while the difference in terms of register requirement is presented in the second part. Note that we couldn't explore the optimal solutions of SIRA and hamiltonian SIRA in loops with more than 10 nodes because the integer optimization ran out of time. However, as we will see, the fixed SIRA systems allows to treat larger loops. For $II = MII$, some experiments do not exhibit a substantial difference between SIRA and fixed SIRA. But if we vary II from MII to an upper-bound L, the difference is highlighted. We summarize our results as follows.

- Regarding the register requirement, the Ning and Gao 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. Hence, even if we increase the II, the minimal register requirement is always lower bounded by the number of statements in the loop. However, enabling sharing with an arbitrary hamiltonian reuse circuit is much more beneficial. In many cases, it results in nearly optimal register need. The maximal experimental difference with the optimum that we get with this technique is 4 registers.

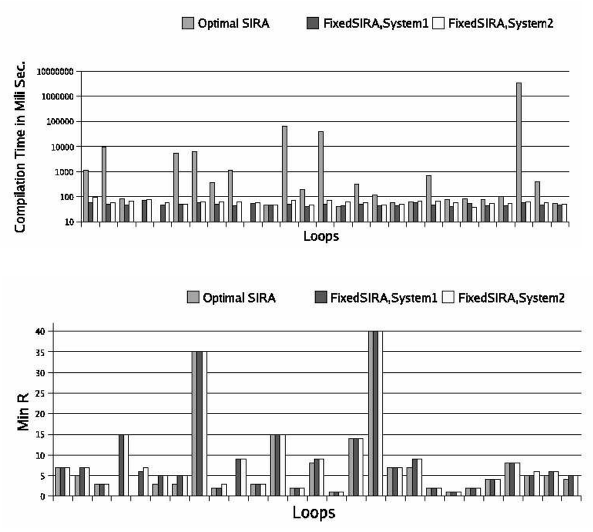


Figure 7: Optimal versus Fixed SIRA with $II = MII$

- Regarding the unrolling degrees, the Ning and Gao strategy exhibit the lowest ones, except in very few cases. This technique may be more beneficial if code size expansion is a critical factor. Arbitrary hamiltonian reuse circuits, if no rotating register set exists, require to unroll the loops with the same number of allocated registers.

9.3. Fixed SIRA : System 1 versus System 2

Performing optimal SIRA solutions involves solving the exact intLP models of Sect. 4.2 or Sect. 5. The compilation time becomes intractable when the size of the loop exceeds 10 nodes. Hence, for larger loops, we advice use of our fixed SIRA strategies that are faster but allow sub-optimal results.

We investigated the scalability (in terms of compilation time versus the size of DDGs) for fixed SIRA when solving System 1 (non totally unimodular matrix) or System 2 (totally unimodular matrix). Fig. 8 plots the compilation times for larger loops (buffers and fixed hamiltonian). The difference is negligible till 300 nodes. For loops larger than 300 nodes, the compilation time of System 1 becomes more considerable (multiple seconds for Ning and Gao method, multiple minutes for fixed hamiltonian).

The error ratio, induced by ceiling the μ variable as solved by System 2 compared to the optimal ones solved by System 1, is depicted in Fig. 9. While the hamiltonian strategy exhibits an error of 20% after 300 nodes, the Ning and Gao strategy has an error ratio less than 5%. As can be seen, the error introduced in a fixed hamiltonian reuse strategy is greater than the one introduced with a self reuse strategy. The cumulative distribution of the error in all our experiments is depicted in Fig.10 : while all sub-optimal experiments have an error ratio lower than 20% with a self reuse strategy, a fixed (arbitrary) hamiltonian reuse technique exhibits an error lower than 50% for all sub-optimal experiments. We deduce

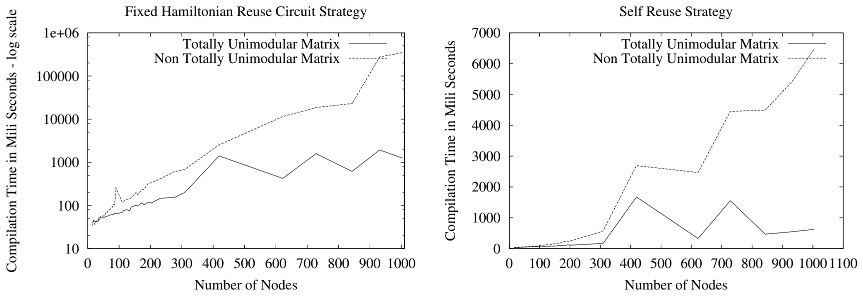


Figure 8: Compilation Time versus the Size of the DDGs

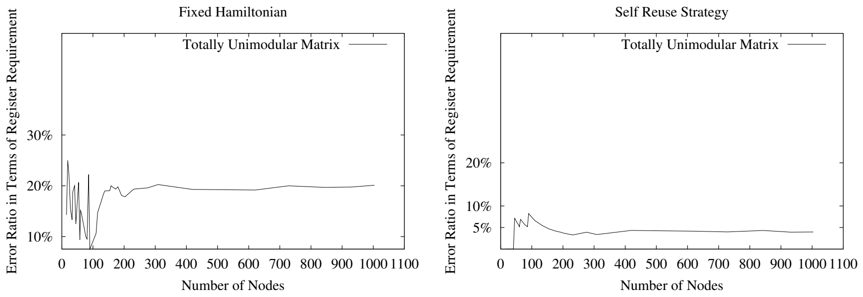


Figure 9: Error Ratio in Terms of Register Requirement, Induced by System 2, versus the Size of the DDGs

that ceiling the μ variables is not a good choice in terms of register requirement. Thus, we should recompute the μ variables with a cleverer method as previously explained in [31]. These authors gave a heuristics to recompute previously substituted integer variables without ceiling them by considering the the already computed σ variables. The result is not necessarily optimal, but still may optimize the back substitution.

The previous plots show that the error ratio induced by ceiling the μ variables if we use the fixed hamiltonian approach is more important than the Ning and Gao buffer minimization case. However, the fixed hamiltonian approach is still better than buffer minimization in terms of register requirement, as can be seen in Fig. 11, while the compilation times for both methods are in the same order of magnitude (check the totally unimodular plots of the two parts in Fig. 8).

We must be aware that solving a fixed SIRA problem with System 1 may be very time consuming in some critical cases. The left side of Fig. 12 plots the compilation time of a complex loop with 309 nodes when we vary the desired critical circuit in a fixed hamiltonian strategy. As can be seen, System 1 becomes very time consuming at the value $II = 94$, while System 2 exhibits a stable compilation time if we vary II, since its constraints matrix does not contain II. Also, the error ratio of the register requirement as solved by System 2 when compared to the optimal one as produced by System 1 may vary in function of the desired critical circuit II (see the right hand side of Fig. 12). Using Ning and Gao method is less critical than the fixed hamiltonian technique. We could solve all intLP problems

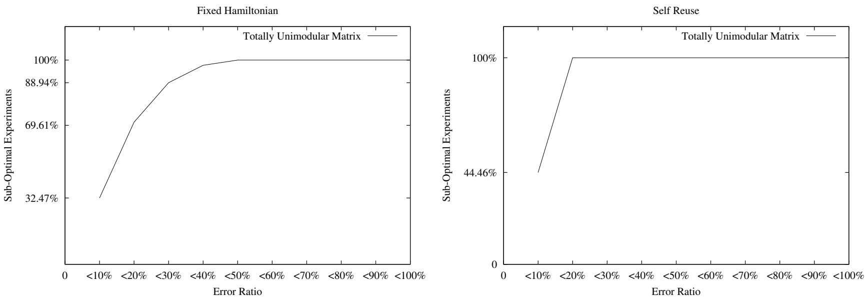


Figure 10: Cumulative Distribution of the Error Ratio, in Terms of Register Requirement, of System 2

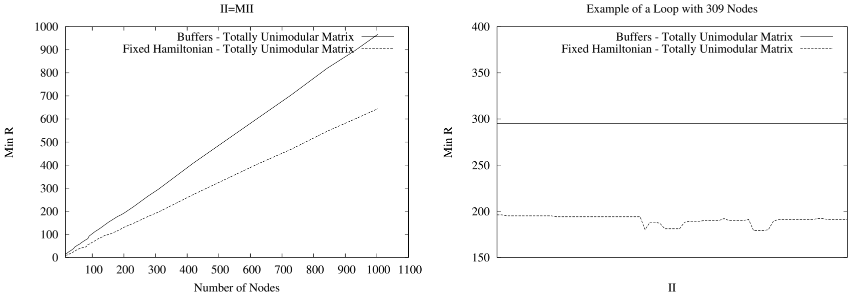


Figure 11: Ning and Gao Method versus Fixed Hamiltonian in terms of Register Requirement (System 2)

(with varying II and the size of DDGs), even for large loops (see Fig. 13). As can be seen, the error ratio is constant. But still, solving the Ning Gao problem with System 1 has not been proved to be polynomial, unless we transform it to System 2, which induces potential error ratio after back substitution. And in this case, the fixed hamiltonian method still needs less registers, even with a higher error ratio.

10. Discussion and Conclusion

This article presents a new theoretical approach consisting in virtually building an early periodic register allocation before code scheduling, with multiple register types and delays in reading/writing. Thus, our theoretical framework is more generic than the exiting ones.

Register 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. Indeed, the size complexity of our intLP formulations depends only the size of the input DAG (quadratic on the number of edges and nodes). This is better than the size complexity of the existing techniques in the literature that model register constraints [1,9]. These exact intLP systems have a size complexity that depends on a worst total schedule time factor, and this latter

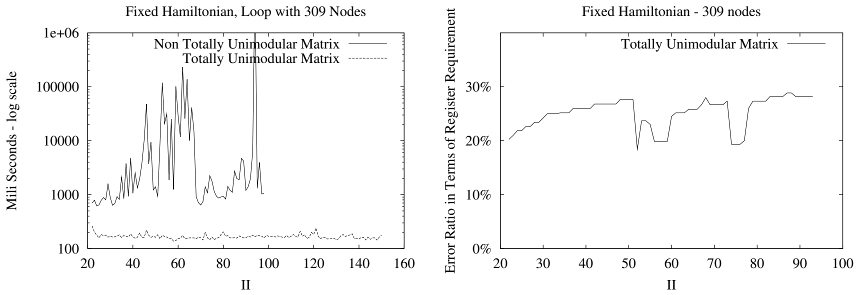


Figure 12: Minimizing the Register Requirement with a Fixed Hamiltonian Circuit (309 nodes)

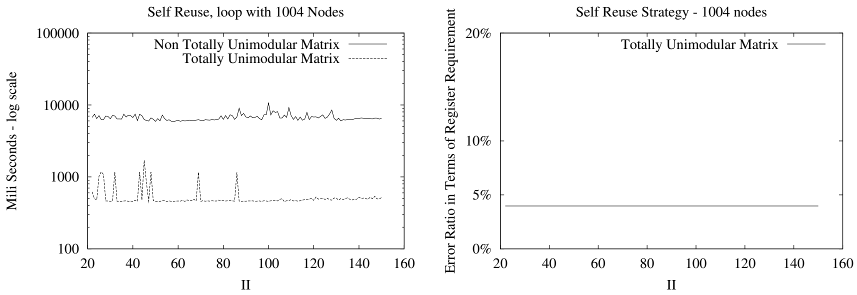


Figure 13: Minimizing the Register Requirement with Ning and Gao Method (1004 nodes)

does not depend on the size of the input DAG. Thus, such size complexity is pseudo-polynomial, and not polynomial as in our intLP system.

Since computing an optimal periodic register allocation is intractable in large loops (larger than 15 nodes for instance), we have identified one polynomial subproblem by fixing reuse edges. With this polynomial algorithms, we can compute $MII(\rho)$ for a given reuse configuration and a given register pressure ρ. We can also heuristically find a register usage for one given II.

We can use this result in different ways, as setting self-reuse edges [22] or fixing arbitrary (or with a cleverer algorithm) hamiltonian circuits. Experiments show that fixing an arbitrary hamiltonian reuse circuit needs much less registers than [22], whether we compute optimal solutions or not. However, unrolling degrees with Ning and Gao method may be better if no rotating register file exists.

Our experiments show that disabling sharing of registers with a self reuse strategy as done in [22] 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.

When considering VLIW/IA64 processors and reading/writing delays, we are faced with some difficulties because of the possible non-positive distance circuits that we prohibit, without losing the ability of considering arcs with non-positive latencies. Thus, our framework can consider the fact 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 codes to reduce the register requirement.

Each reuse decision implies loop unrolling with a factor depending on reuse circuits for each register type. Optimizing this 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 MII. Experimental results show that only very few cases need this extra register.

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.

Finally, another future work will look for algorithms that fix "good" reuse decisions. Our first attention will be oriented to hamiltonian reuse circuits since they experimentally exhibit reduced register requirement.

References

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

[2] 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.

[3] T. Cormen, C. E. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, McGraw-Hill, Cambridge, Massachusetts, 1990.

[4] M.-C. Costa, L. Ltocart, and F. Roupin. Minimal Multicut and Maximal Integer Multiflow: a Survey . In *Proceedings of the European Chapter on Combinatorial Optimization, ECCO XIV*, Bonn, Germany, May 2001.

[5] A. Darte, G.-A. Silber, and F. Vivien. Combining Retiming and Scheduling Techniques for Loop Parallelization and Loop Tiling. *Parallel Processing Letters*, 4(7):379–392, 1998.

[6] 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.

[7] 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.

[8] 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.

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

[10] C. Eisenbeis, S. Lelait, and B. Marmol. The Meeting Graph: A New Model for Loop Cyclic Register Allocation. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 264–267, Limassol, Cyprus, June 1995. ACM Press.

[11] 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.

[12] D. Fimmel and J. Muller. Optimal Software Pipelining Under Resource Constraints. *International Journal of Foundations of Computer Science (IJFCS)*, 12(6):697–718, 2001.

[13] M. Gondran and M. Minoux. *Graphes et algorithmes*. Eyrolles, Paris, 3rd edition, 1995.

[14] 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.

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

[16] W. Jalby and C. Lemuet. WBTK: A New Set of Microbenchmarks to Explore Memory System Performance. In *Los Alamos Computer Science Institute (LACSI) Symposium*, Oct. 2002.

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

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

[19] C. E. Leiserson and J. B. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, 6:5–35, 1991.

[20] S. Lelait. *Contribution l'Allocation de Registres dans les Boucles*. PhD thesis, Universit d'Orlans, France, Jan. 1996.

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

[22] 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.

[23] 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.

[24] K. H. Rosen, J. G. Michaels, J. L. Gross, J. W. Grossman, and D. R. Shier, editors. *Handbook of Discrete and Combinatorial Mathematics*. CRC, Boca Raton FL, 2000.

[25] 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.

[26] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, New York, 1987.

[27] 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.

[28] 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.

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

[30] S.-A.-A. Touati. *Register Pressure in Instruction Level Parallelisme*. PhD thesis, Universit de Versailles, France, June 2002.

[31] 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.

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