Post-Pass Periodic Register Allocation to Minimise Loop Unrolling Degree

Mounira Bachir, Sid Touati, Albert Cohen

Post-Pass Periodic Register Allocation to Minimise Loop Unrolling Degree

Mounira BACHIR, Sid-Ahmed-Ali TOUATI, Albert COHEN

April 10, 2009

Abstract

This paper solves an open problem regarding loop unrolling after periodic register allocation. Although software pipelining is a powerful technique to extract fine-grain parallelism, it generates reuse circuits spanning multiple loop iterations. These circuits require periodic register allocation, which in turn yield a code generation challenge, generally addressed through: (1) hardware support — rotating register files — deemed too expensive for embedded processors, (2) insertion of register `move`s with a high risk of reducing the computation throughput — initiation interval (II) — of software pipelining, and (3) post-pass loop unrolling that does not compromise throughput but often leads to unpractical code growth. The latter approach relies on the proof that MAXLIVE registers are sufficient for periodic register allocation [2, 3, 5]; yet the only heuristic to control the amount of post-pass loop unrolling does not achieve this bound and leads to undesired register spills [4, 7].

We propose a periodic register allocation technique allowing a software-only code generation that does not trade the optimality of the II for compactness of the generated code. Our idea is based on using the remaining registers: calling R_{arch} the number of architectural registers of the target processor, then the number of remaining registers that can be used for minimising the unrolling degree is equal to $R_{\text{arch}} - \text{MAXLIVE}$.

We provide a complete formalisation of the problem and algorithm, followed by extensive experiments. We achieve practical loop unrolling degrees in most cases — with no increase of the II — while state-of-the-art techniques would either induce register spilling, degrade the II or lead to unacceptable code growth.

keywords Periodic Register Allocation, Software Pipelining, Loop Unrolling, Embedded Code Optimisation

1 Introduction

Our focus is on the exploitation of instruction-level parallelism (ILP) in embedded VLIW processors [13]. Increased ILP translates into higher register pressure and stresses the register allocation phase(s) and the design of the register files. In the case of software-pipelined loops, variables can stay alive across more than one kernel iteration, which is challenging for code generation. The classical software solution that does not alter the computation throughput consists in unrolling the loop a posteriori [4, 13]. We investigate ways to keep the size of the generated code compatible with embedded constraints without compromising the throughput benefits of software pipelining. Namely, we want to minimise the unrolling degree resulting from periodic register allocation of a software-pipelined loop, *without altering the initiation interval* (II).

Lam introduced *maximal variable extension* [4, 13] to minimally unroll a software pipelined kernel for a software-only solution to code generation. Having MAXLIVE variable simultaneously alive [12], maximal variable expansion requires at least MAXLIVE registers to generate the code. Unfortunately, maximal variable expansion does not provide any guarantee on the final number of registers: this limitation has been highlighted in [1, 3]. That is, it is possible to allocate more than MAXLIVE registers, and there is no known precise upper bound. In practice, it means that maximal variable expansion may generate spill code even if MAXLIVE is below the number of architectural registers of the target processor. This means maximal variable expansion is not a robust solution to post-pass periodic register allocation.

On the other hand, formal guarantees have been achieved using a graph-theoretical framework called *the meeting graph* [1]. Given a software-pipelined loop, meeting graphs guarantee a periodic register allocation with exactly MAXLIVE registers. The same framework provides multiple ways to generate code achieving this lower bound.

1. If the target processor contains rotating register files, loop unrolling is not required. However, such hardware support may find its place in high performance processors (Intel's Itanium) but never made its way to embedded processors.

2. Alternatively, loop unrolling can be avoided thanks to periodic register renaming in software. This is achieved through the insertion of `move` operations. However, inserting those operations may decrease the computation throughput, i.e., increase the initiation interval (II). This cost often nullifies the benefits of software pipelining itself.

3. The only remaining choice is to unroll the loop up to the least common multiple of the weights (total distance) of the reuse circuits. The resulting unrolling degree is often unacceptable and may reach absurd levels.

To make things worse, practical applications of software pipelining require a pre-pass of schedule-independent periodic register allocation, or face an uncontrolled impact on MAXLIVE. We thus need to resort another graph theoretical framework called *reuse graphs* [2]. Reuse graphs are used inside a framework called SIRA. This framework generalises previous research result on periodic register allocation [1, 3] by considering both scheduled and unscheduled loops. As a result, reuse graphs can be used both for a pre-pass periodic register allocation to control the aggressiveness of scheduling algorithms, or as a post-pass periodic register allocation to generate code.

Reuse graphs have another usefulness. Indeed, they are used to compute the *sufficient unrolling* degree that we should apply to the loop so that it is always possible to allocate exactly $R_{\text{min}} = \text{MAXLIVE}$ registers, *independently of the actual scheduling* [2]. The drawback of this allocation is that the unrolling factor is equal to the least common multiple of the weight of all reuse circuits. This paper presents a new method to reduce the loop unrolling degree without altering the II. The heart of our method is based on the following observation. Let R_{arch} be the number of available architectural registers in the processor; when a periodic register allocation is performed, we may allocate $R_{\text{min}} = \text{MAXLIVE} \leq R_{\text{arch}}$ registers. Hence it remains $R = R_{\text{arch}} - R_{\text{min}}$ free registers. Our goal is to exploit these remaining registers to minimise the loop unrolling degree α. That is, our loop compaction method is based *on using extra free registers if they exist* to reduce the unrolling degree, without adding extra `move` operations, and without altering the II of the software pipelined schedule.

This paper is organised as follows. Section 2 presents the most relevant related work. Section 3 is a brief introduction to reuse graphs describing how to perform periodic register allocation. Section 4 formalises the problem of minimisng the loop unrolling degree, then describes a solution. Section 5 details our main algorithms. Section 6 presents extensive experimental results, showing that our optimal solution is fast and efficient in practice. Finally, we summarise our results and discuss some perspectives.

2 Related Work

We review the main issues and approaches to code generation for periodic register allocation.

2.1 Rotating Register File

A *rotating register file* (RRF) [6] is a hardware mechanism to prevent successive lifetime intervals from being assigned to the same physical registers. Consider the following example:

LOOP
  $a[i+2] = b[i] + 1$
  $b[i+2] = a[i] + 2$
ENDLOOP

In this example, variable $a[i]$ spans three iterations (defined in iteration $i-2$ and used in iteration i). Hence, at least 3 physical registers are needed to carry simultaneously $a[i]$, $a[i+1]$ and $a[i+2]$. A rotating register file R automatically performs the `move` operation at each iteration. R acts as a FIFO buffer. The major advantage is that instructions in the generated code see all live values of a given variable through a single operand, avoiding explicit register copying. Below $R[k]$ denotes a register with offset k from R.

Iteration i	Iteration $i+2$
$R = b[i] + 1$	$R[+2] = b[i] + 1$
$b[i+2] = R[-2] + 2$	$b[i+2] = R + 2$

Using a RRF avoids increasing code size due to loop unrolling, or to decrease the computation throughput due to the insertion of `move` operations.

2.2 Move Operations

This method is also calledd *register renaming*. Considering the previous example, we use 3 registers to allocate $a[i]$ and perform `move` operations at the end of each iteration [10, 11]: $a[i]$ in register $R1$, $a[i+1]$ in register $R2$ and $a[i+2]$ in register $R3$. Then we use `move` operations to shift registers across the register file at every iteration:

LOOP
  $R3 = b[i] + 1$
  $b[i+2] = R1 + 2$
  $R1 = R2$
  $R2 = R3$
ENDLOOP

However, it is easy to see that if variable v spans d iterations, we have to insert $d-1$ extra `move` operations *at each iteration*. In addition, this may increase the II and may require rescheduling the code if these `move` operations do not fit into the kernel. This is generally unacceptable as it negates most of the benefits of software pipelining.

2.3 Loop Unrolling

Another method, *loop unrolling*, is more suitable for embedded processors. The loop body itself is bigger but no extra operations are executed in comparison with the original code. Lam designed a general loop unrolling scheme called *modulo variable expansion* [4]. In fact, the major criterion of this method is to minimize the loop unrolling degree because the size of the i-WARP processor is low [4]. the *modulo variable expansion* method guarantees the minimal unrolling degree to enable code generation after a given periodic register allocation. This unrolling degree is obtained by dividing the length of the longest live range ($\max_v LT_v$) by the number of cycles of the kernel $\frac{\max_v LT_v}{II}$. In practice, many works propose a simple way to implement a generalized form of modulo expansion [9]. However, this method does not guarantee a register allocation with MAXLIVE registers [1, 3], and in general it may lead to unnecessary spills breaking the benefits of software pipelining. A concrete example of this limitation can be found in [7].

Several algorithms have been proposed to achieve an allocation with a minimum number of registers equal to MAXLIVE [2, 3, 5]. The algorithm of Eisenbeis et al.[5] achieves this bound, thanks to a dedicated graph representation called the *meeting graph*. This graph describes how to find a periodic register allocation with MAXLIVE registers if we sufficiently unroll the pipelined loop. They proceed by decomposing the meeting graph into elementary circuits labelled with their weights (w_i), in which each circuit correspond to a reuse pattern. The drawback is that the unrolling factor α is associated with the least common multiple of the (w_i), and that it is difficult to extract a circuit decomposition that minimises α. Another relevant approach based on *reuse graphs* is described in the next section.

3 Reuse Graphs

The literature brought many fundamental results on periodic register allocation for software pipelining using loop unrolling. For instance, the methods [1, 3, 4, 13] can be used for such purpose on *already scheduled* (software

pipelined) loops. If periodic register allocation should be done before software pipelining (for any reason), then such methods cannot be applied. Nevertheless, pre-pass periodic register allocation is critical to control aggressiveness of software pipelining algorithms, keeping register pressure (MAXLIVE) within acceptable levels. Many ad-hoc heuristics are used in production compilers for this purpose, yet there exists a graph theoretical framework called SIRA [2] designed to perform periodic register allocation *either before or after software pipelining*. In this article, we rely on this general theoretical framework as a register allocator, because we want to be able to optimise the loop unrolling degree either before or after software pipelining.

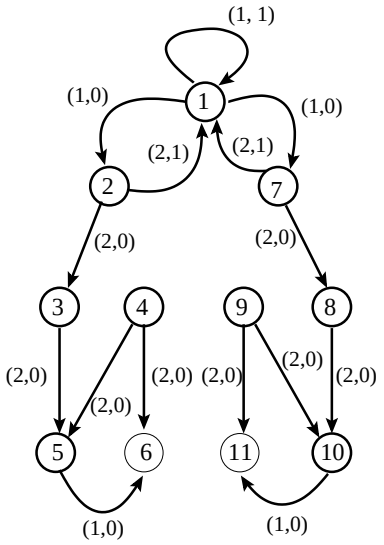
The problem with SIRA is that it sometimes induces huge loop unrolling factors, reducing its practical applicability. Our work improves SIRA by *adding a post-pass of loop unrolling minimisation*. This section is an overview on the main results of the SIRA framework [2].

Figure 1 (a) is an illustration of a data dependence graph (DDG) of an innermost loop devoted to a software pipelining schedule. The nodes (loop statements) writing into a register are in bold circle. The flow data dependences through registers are in bold arcs. The nodes that are not in bold circles are the set of statements that do not write into registers (such as nodes 6 and 11). Each arc is labelled by a pair of values representing the latency of the dependence (in processor clock cycle) and the distance of the dependence (in terms of loop iterations). Doing a periodic register allocation for this loop means to decide how to share the available registers between the set of loop operations (statements instances). The difficulty here resides in the fact that the variables lifetime intervals become periodic when considering software pipelining[3].

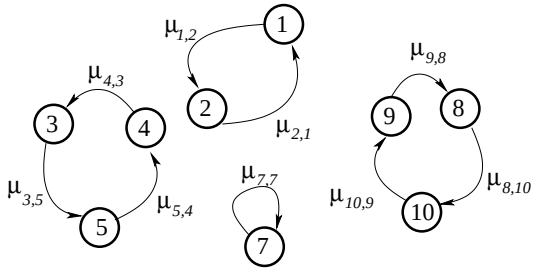
Figure 1 (b) is an illustration of a reuse graph associated to the loop. The set of nodes in the reuse graph is the set of nodes writing into a register: here, nodes 6 and 11 are excluded because they do not require registers. Each arc in the reuse graph represent a *reuse arc*. It models the register sharing between statements. A reuse arc $e = (u, v)$ with a label $\mu_{u_v} \in \mathbb{N}$ means that the operation u of iteration i and the operation v of iterations $i + \mu_{u_v}$ shares the same destination register. That is, the lifetime interval of the variable u of iteration i is necessarily before the lifetime interval of the variable v of iteration $i + \mu_{u_v}$.

Reuse graphs have many formal proved characteristics making them a good formal solution for periodic register allocation [2, 8]. First, a reuse graph is composed of a set of elementary and disjoint circuits, called *reuse circuits*. Second, each node writing into a register belongs to one and only one reuse circuit. Third, the weights of reuse graphs describe precisely the number of allocated registers and the unrolling degree. Let us give more details on this last important property. Let be $\mu_i = \sum_e \in C_i$ be the weight of a reuse circuit C_i (it is the sum of all the μ labels of its arcs). Let $\{C_1, \cdots, C_k\}$ be the set of reuse circuits. Let be $R = \sum_i \mu_i$ the total sum of all the labels, and let be $\alpha = \text{lcm}(\mu_1, \cdots, \mu_k)$ the least common multiple of the reuse circuits weights. Then, given a loop DDG and valid reuse graph associated with it, the following assertion is proved correct [2, 8]: if the loop is unrolled α times, then the reuse graph describes a periodic register allocation with exactly $R = \sum_i \mu_i$ registers. For instance, the reuse graph of Figure 1 (b) has four reuse circuits: $C_1 = \{1, 2\}$, $C_2 = \{7\}$, $C_3 = \{3, 4, 5\}$ and $C_4 = \{8, 9, 10\}$. Each reuse circuit C_i has a weight μ_i: $\mu_1 = \mu_{1,2} + \mu_{2,1}$, $\mu_2 = \mu_{7,7}$. $\mu_3 = \mu_{4,3} + \mu_{3,5} + \mu_{5,4}$ and $\mu_4 = \mu_{9,8} + \mu_{8,10} + \mu_{10,9}$. If the loop is unrolled $\alpha = \text{lcm}(\mu_1, \mu_2, \mu_3, \mu_4)$ times, then we can build a periodic register allocation with $R = \sum_i \mu_i$ registers.

Note that the problem of building a valid reuse graph with $\sum_i \mu \leq R_{\text{arch}}$ has been solved in[2, 8]. Thanks to reuse graphs, we are able to formally guarantee the number of allocated registers with a sufficient unrolling degree. This periodic register allocation can be done after software pipelining (without altering the II), or before software pipelining (without altering the critical cycle if possible). We recall that we can avoid loop unrolling in the presence of a hardware rotating register file: unfortunately, embedded VLIW processors do not have such architectural support. We can however insert `move` operations to simulate a rotating register file. Such extra `move` operations may alter the II. So, in the absence of hardware support, the unique formal solution for periodic register allocation with exactly MAXLIVE registers without altering the II seems to be proposed by the SIRA framework. The problem with SIRA is that it provides a satisfactory solution from the computer science perspective, but the solution is not satisfactory in practice. This is because the unrolling degree α computed by SIRA may be large yielding to considerable code size expansion. In this paper, we propose a post-pass optimisation method that minimises the unrolling degree to its lowest possible value. The following section is devoted to this important problem.



(a) DDG Example



(b) Reuse Graph Example

Figure 1: Reuse Graphs

4 Loop Unrolling Problem

The fact that the unrolling factor α may theoretically be high would happen only if we actually want to allocate the variables on this minimal number of registers $R_{\text{min}} = \sum_C \mu(C)$ with the computed reuse scheme. However, there may be other reuse schemes for the same number of registers, or there may remain some registers after the register allocation step in the architecture that we can use $R = R_{\text{arch}} - R_{\text{min}}$ (R_{arch} is the number of architectural registers). In that case, we develop a method using these remaining registers in order to reduce this unrolling factor. This method is applied after the periodic register allocation step performed by the framework SIRA. This post-pass minimisation consists in adding some registers between the remaining registers to each reuse circuit in order to minimise the least common multiple denoted α^*. This idea is described in *LCM-MIN Problem* detailed in the next section. Figure 2 illustrates the global steps for minimising loop unrolling degree. As can be seen, our method performs after a conventional periodic register allocation.

4.1 LCM-MIN Problem

The *LCM-MIN Problem* can be formally modelled as follow:

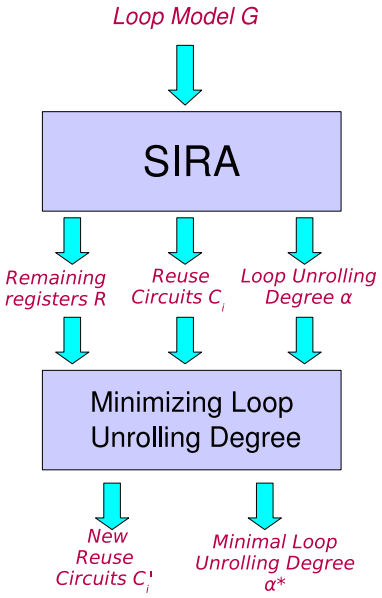


Figure 2: Loop Unrolling Minimisation

Problem 1 (LCM-MIN)**.** *Let $R \in \mathbb{N}$ be the number of remaining registers. Let $\mu_1, \ldots, \mu_k \in \mathbb{N}$ be the weights of the reuse circuits. Compute the added registers $r_1, \ldots, r_k \in \mathbb{N}$ such that:*

1. $\sum_{i=1}^{k} r_i \leq R$
2. *lcm*$(\mu_1 + r_1, \ldots, \mu_k + r_k)$ *is minimal.*

Before stating our solution for Problem 1, we propose to find a solution for a sub-problem that we call *LCM-Problem*. The solution of this sub-problem constitutes the basis of the solution of Problem 1. *LCM-Problem* proposes to find for a fixed loop unrolling degree β, the different added registers $r_1, \ldots, r_k$ among the remaining registers R to the different reuse circuits such as: $\sum_{i=1}^{k} r_i \leq R$ and lcm$(\mu_1 + r_1, \ldots, \mu_k + r_k) = \beta$. A formal description is given in the next section.

4.2 LCM Problem

We formulate the *LCM Problem* as follow:

Problem 2 (LCM Problem)**.** *Let $R \in \mathbb{N}$ be the number of remaining registers. Let $\mu_1, \ldots, \mu_k \in \mathbb{N}$ be the weights of the reuse circuits. Given a positive integer β, compute the different added registers $r_1, \ldots, r_k \in \mathbb{N}$ such that:*

1. $\sum_{i=1}^{k} r_i \leq R$
2. *lcm*$(\mu_1 + r_1, \ldots, \mu_k + r_k) = \beta$.

Before describing our solution for *LCM Problem*, we state Lemma 1 and Theorem 1 that we need to use afterwards. We prove only Theorem 1 because the proof of Lemma 1 seems obvious.

Lemma 1. *Let assume that we find a list of the added registers $r_1, \ldots, r_k$ among the remaining registers R with a minimal number of registers ($\sum_{i=1}^{k} r_i$ is minimal). Let assume that this minimal list of the added registers satisfies the second condition of Problem 2 (lcm$(\mu_1 + r_1, \ldots, \mu_k + r_k) = \beta$). If the first condition is not fulfilled ($\sum_{i=1}^{k} r_i$ minimal $> R$) then* LCM Problem *cannot be resolved.*

Theorem 1. *Let β be a positive integer and D_β be the set of its divisors. Let $\mu_1, \ldots, \mu_k \in \mathbb{N}$ be the weights of the reuse circuits. If we find a list of the added registers $r_1, \ldots, r_k \in \mathbb{N}$ for Problem 2, thus we have the following results:*

1. $\beta = lcm(\mu_1 + r_1, \ldots, \mu_k + r_k) \Rightarrow \forall i = 1, k : \ \beta \geq \mu_i$
2. $\beta = lcm(\mu_1 + r_1, \ldots, \mu_k + r_k) \Rightarrow \forall i = 1, k :$
   $\exists d_i, \ r_i = d_i - \mu_i$ *with* $d_i \in D_\beta \wedge d_i \geq \mu_i$.

Proof. The first issue can be proved as follows:

$$\beta = \text{lcm}(\mu_1 + r_1, \ldots, \mu_k + r_k) \Rightarrow \forall i = 1, k \ \beta \geq \mu_i + r_i \tag{1}$$

From (1) we have:

$$\forall i = 1, k : \ \beta \geq \mu_i + r_i \Rightarrow \forall i = 1, k : \ \beta - \mu_i \geq r_i \tag{2}$$

From (2) we have:

$\forall i = 1, k : \ \beta \geq \mu_i$ because $\forall i = 1, k : \ r_i \geq 0$ (each $r_i \in \mathbb{N}$)

The first issue is proved.

The second issue can be proved by using the definition of the least common multiple of a set of positive integers. Hence, we have:

$\beta = \text{lcm}(\mu_1 + r_1, \ldots, \mu_k + r_k)$

$$\Rightarrow \forall i = 1, k : \ \mu_i + r_i \text{ is a divisor of } \beta \tag{3}$$

From (3) we have:

$\forall i = 1, k \ : \ \mu_i + r_i$ is a divisor of β

$$\Rightarrow \ \forall \, i = 1, k \ : \ \exists \, d_i \ \in \ D_\beta \mid \mu_i + r_i = d_i \tag{4}$$

From (4) we find:

$$\begin{cases} \forall \, i = 1, k \ : \ r_i \geq 0 \\ \exists \, d_i \ \in \ D_\beta \mid \mu_i + r_i = d_i \end{cases} \Rightarrow \begin{cases} \forall \, i = 1, k \\ \exists \, d_i \ \in \ D_\beta \ : \\ r_i = d_i - \mu_i \\ \text{with } d_i \geq \mu_i \end{cases}$$

The second issue of Theorem 1 is proved. □

After proving the Theorem 1 and by using Lemma 1, we describe our solution for *LCM Problem* in the next section.

4.3 Solution for LCM Problem

Proposition 1. *Let β be a positive integer and D_β be the set of its divisors. Let R be the number of remaining register. Let $\mu_1, \ldots, \mu_k \in \mathbb{N}$ be the weights of the reuse circuits. A minimal list of the added registers ($r_1, \ldots, r_k \in \mathbb{N}$ with $\sum_{i=1}^{k} r_i$ is minimal) can be found by adding to each reuse circuit μ_i a minimal value r_i such as $r_i = d_i - \mu_i$ with $d_i = \min\{d \in D_\beta \mid d \geq \mu_i\}$. Hence, we have the following issues:*

1. $\beta = lcm(\mu_1 + r_1, \ldots, \mu_k + r_k) \wedge \sum_{i=1}^{k} r_i \leq R \Rightarrow$ *we find one solution for Problem 2;*
2. $\beta = lcm(\mu_1 + r_1, \ldots, \mu_k + r_k) \wedge \sum_{i=1}^{k} r_i > R \Rightarrow$ *Problem 2 cannot be resolve.*

Proof. In Theorem 1, we have proved that:

$\beta = \text{lcm}(\mu_1 + r_1, \ldots, \mu_k + r_k) \Rightarrow$

$$\forall \, i = 1, k \, \exists \, d_i \ \in \ D_\beta \mid r_i = d_i - \mu_i \ \wedge \ d_i \geq \mu_i \tag{5}$$

From (5) we have:

$$r_i \text{ is minimal } \Rightarrow d_i \text{ is the smallest divisor of } \beta \ \geq \ \mu_i \tag{6}$$

Algorithm 1 LCM Problem

Require: k the number of reuse circuits, the different weights of reuse circuits μ_i, the remaining register R and β
Ensure: the different added registers $r_1, \ldots, r_k$ with $\sum_{i=1}^{k} r_i$ $minimal$ if it exists and a boolean $success$
  $sum \leftarrow 0$ {initialisation}
  $success \leftarrow true$ {defines if we find the different added registers or not}
  $i \leftarrow 1$ {represents the number of reuse circuit}
  calculate the different divisors of β
  while $i \leq k \wedge success$ **do**
    $d_i \leftarrow DIV\_NEAR(\beta, \mu_i)$ {DIV_NEAR returns the smallest divisors of β greater or equal to μ_i}
    $r_i \leftarrow d_i - \mu_i$
    $sum \leftarrow sum + r_i$
    if $sum > R$ **then**
      $success \leftarrow false$
    else
      $i++$
    end if
  end while

From (6) a minimal list of the added registers $r_1, \ldots, r_k$ with $\sum_{i=1}^{k} r_i$ is minimal can be found as follows:

$\forall i = 1, k:\ r_i$ is minimal $\Rightarrow\ \forall i = 1, k:\ r_i = d_i - \mu_i \wedge d_i = \min\{d \in D_\beta \mid d \geq \mu_i\}$

According to Lemma 1,if we find a list of the added registers (the different values of r_i) among the remaining registers such as $\sum_{i=1}^{k} r_i$ is minimal $\leq R$ then these different values of r_i can be a solution for *LCM Problem*. Otherwise, if $\sum_{i=1}^{k} r_i$ is minimal $> R$ then we are sure that there are no solution for Problem 2.

Figure 3 represents a graphical solution for *LCM Problem*. For the fluidity of the reading, we assume that the different weights and the different divisors of β are sorted on the same axis in an ascending order.

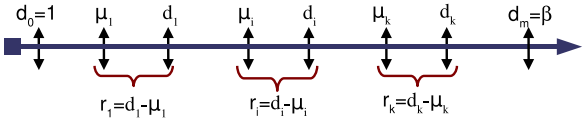


Figure 3: Graphical solution for the LCM Problem

Algorithm 1 implements our proposition for *LCM Problem*. In this algorithm, we minimise the least common multiple of k integers (the different weights of reuse circuits μ_i) using the remaining registers R. It checks if β can become the new loop unrolling degree. This algorithms finds out the list of added registers among the remaining registers R between the reuse circuits (the different values of $r_i\ \forall i = 1, k$), if such list of added registers exists. It returns also a boolean $success$ which takes the following values:

$$success = \begin{cases} true & \text{if } \sum_{i=1}^{k} r_i \leq R \\ false & \text{otherwise} \end{cases}$$

□

The solution of *LCM Problem* constitutes the basis of a solution for *LCM-MIN Problem* explained in the next section.

5 Solution for LCM-MIN Problem

For the resolution of *LCM-MIN Problem* we have to use the solution of the *LCM Problem* and the result of Theorem 1.

According to Theorem 1, the research space S for α^* (the solution of *LCM-MIN Problem*) is bounded.

$$\begin{cases} \forall\, i = 1, k\ \alpha^* \geq \mu_i \text{ (From Th 1)} \\ \alpha^* \leq \alpha \text{ (our objective)} \end{cases} \Rightarrow \max_{1 \leq i \leq k} \mu_i \leq \alpha^* \leq \alpha$$

In addition, α^* is a multiple of each $\mu_i + r_i$ with $0 \leq r_i \leq R$. If we assume that $\mu_k = \max_{1 \leq i \leq k} \mu_i$ then α^* is a multiple of $\mu_k + r_k$ with $0 \leq r_k \leq R$. Furthermore, the research space S can be stated as follows:

$S = \{\beta \in \mathbb{N} \mid \beta \text{ is multiple of } (\mu_k + r_k)\ \forall r_k = 0, R\ \wedge\ \mu_k \leq \beta \leq \alpha\}$

After describing the set S of all possible value of α^*. The minimal α^* the solution for Problem 1 is defined as follow:

$\alpha^* = \min\{\beta \in S | \exists (r_1, \ldots, r_k) \in \mathbb{N}^k \wedge \text{lcm}(\mu_1 + r_1, \ldots, \mu_k + r_k) = \beta \wedge \sum_{i=1}^{k} r_i \leq R\}$

Figure 4 portrays all values of the set S. An arrow between two nodes means that the value in the first node is less than the value of the second node: $a \rightarrow b \Rightarrow a < b$. The value μ_k represents the value of the reuse circuit number k. By assumption, it is also the greatest value of all reuse circuits. α is the initial loop unrolling value. Each node is a potential solution (β) which can be considered as the minimal loop unrolling degree. A dashed node can not be a potential candidate because its value is greater than α. Let $\tau = \alpha\ div\ \mu_k$ be the number of total lines. Each line describes a set of multiples. For example, the line j describes a set of multiples $S_j = \{\beta | \exists r_k, 0 \leq r_k \leq R, \beta = j \times (\mu_k + r_k) \wedge \beta \leq \alpha\}$

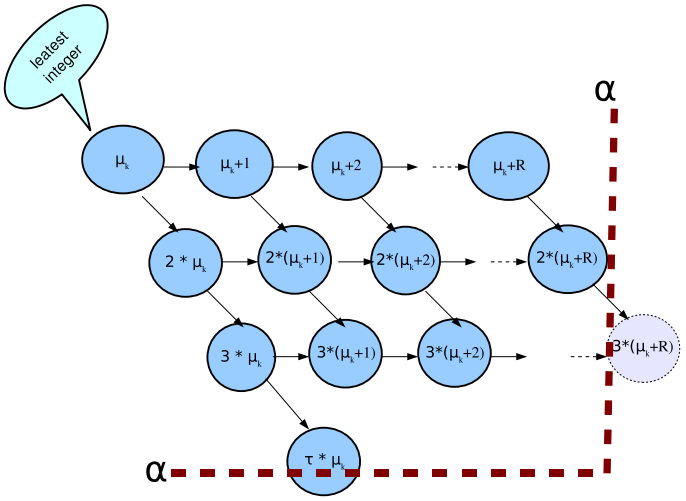


Figure 4: The set S of all possibles values

In order to compute α^*, our solution consists in checking if each node of S can be a solution for *LCM Problem*: at last we are sure that the minimum of all these values is the minimal loop unrolling degree.

Despite traversing all the nodes of S, we describe in Figure 5 an efficient way to find the minimal α^*. We proceed line by line. In each line, we apply Algorithm 1 to each node until the value of the predicate $success$ returned by Algorithm 1 is $true$ or until we arrive at the last line when $\beta = \alpha$. If the value β of the node i of the line j verifies the predicate ($success = true$), then we have two cases:

1. If the value of this node is less than the value of the first node of the next line then we are sure that this value is optimal ($\alpha^* = \beta$). This is because all the remaining nodes are greater than β (by construction of the set S).

2. Else we have found a new value of unrolling degree less than the original α. We note this new value α' and we try once again to optimise it until we find the optimal (the first case). The set of research becomes smaller ($S' = \{\beta \in \mathbb{N} | \forall r_k = 0..R : \beta \text{ is multiple of } (\mu_k + r_k) \wedge (j+1) \times \mu_k \leq \beta \leq \alpha'\}$)

Algorithm 2 implements our solution for *LCM-MIN Problem*. This latter minimises the loop unrolling degree α which is the least common multiple of k reuse circuits whose weights are $\mu_1, \ldots, \mu_k$. Our method is based on using

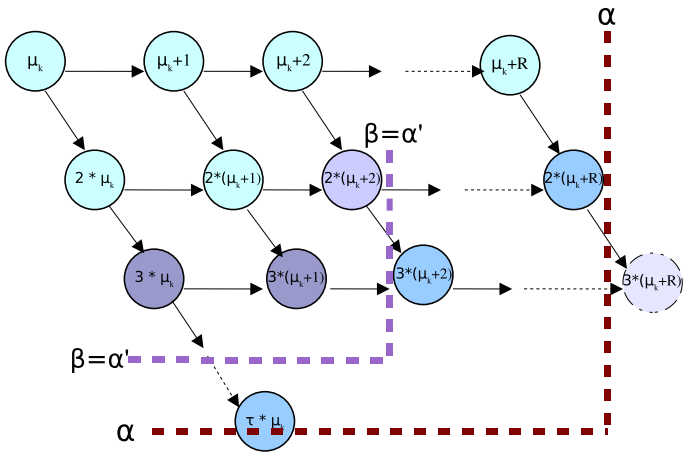


Figure 5: How to Traverse the Set S

the remaining registers R. This algorithm computes α^* the minimal value of loop unrolling degree and the minimal list $r_1, \ldots, r_k$ of the added registers to the different reuse circuits.

Fig 6 illustrates a concrete example. We want to minimise the initial loop unrolling degree generated by the following reuse circuits: $\mu_1 = 3, \mu_2 = 4, \mu_3 = 5, \mu_4 = 7, \mu_5 = 8$. The loop unrolling degree α is their least common multiple ($\alpha = 840$). In this configuration we assume that we have 32 registers in the machine. So hence we have $R = 5$ remaining registers. The new loop unrolling degree (minimal least common multiple) found thanks to our method is $\alpha^* = 8$. The minimal number of registers added to each reuse circuits are $r_1 = 1, r_2 = 0, r_3 = 3, r_4 = 1, r_5 = 0$. The ratio $ratio = \frac{\alpha}{\alpha^*} = 105$ and the execution time is about 41 micro second.

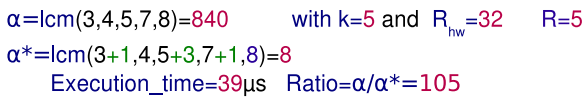


Figure 6: Example of Loop Unrolling Minimisation

6 Experimental Results

To study the efficiency of our approach, we developed a tool to generate many thousands of random data dependence graphs (DDG) containing multiple hundreds of reuse circuits. We then integrated the results into a backend optimiser, considering real DDG of classical benchmarks. We use both random DDG and real DDG to investigate the efficiency of our solution. We detail these tools and the experimental results in the following sections.

6.1 Results on Randomly Generated DDG

At first, our software generates k the number of distinct reuse circuits and their weights $(\mu_1, \ldots, \mu_k)$. Afterwards, we calculate the number of remaining registers $R = R_{\text{arch}} - \sum_{i=1}^{k} \mu_i$ and the loop unrolling degree $\alpha = \text{lcm}(\mu_1, \ldots, \mu_k)$. Finally, we apply our method for minimising α.

We did extensive random generations on many configurations: we varied the number of available registers R_{arch} from 4 to 256, and we considered many thousands of random graphs containing multiple hundreds of reuse circuits.

Each reuse circuit can be arbitrarily large. That is, our experiments are done on random data dependence graphs with unbounded number of nodes (as large as someone wants). Only the number of reuse circuits is bounded.

Figure 7 is a 2-D plot representing the code size compaction ratio obtained thanks to our method. The code size compaction is counted as the ratio between the initial unrolling degree and the minimised one ($ratio = \frac{\alpha}{\alpha^*}$). The X-axis is the number of available hardware registers (going from 4 to 256), the Y-axis is the code compaction ratio. As can be seen, our method allows to have a code size reduction going from 1 to more than 10000! In addition, we note also in Figure 7 that the ratio is very important when the R_{arch} is greater. For example, the ratio of some minimisation exceeds 10000 when $R_{\text{arch}} = 256$.

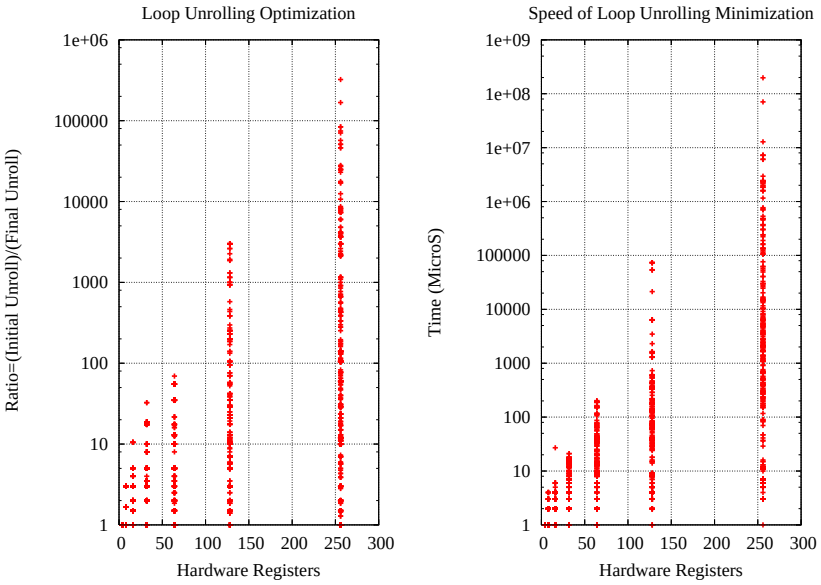


Figure 7: Loop Unrolling Minimisation Experiments

Furthermore, our method is very fast. Figure 7 plots the speed of our method on a dual-core 2 GHz Linux PC, ranging from 1 micro-second to 10 seconds. This speed is satisfactory for optimising compilers devoted to embedded systems (not to interactive compilers like gcc or icc). We remark also the speed of extremely rare minimisation (when $R_{\text{arch}} = 256$) can reach 1000 seconds.

We show in Figure 8, the harmonic mean for all the code compaction ratios in each configuration. These very high numbers confirm that our approach significantly decreases the initial loop unrolling degree.

6.2 Results on Concrete DDG

Initially, each periodic register allocation in our backend compilation framework implies loop unrolling with a factor α depending on reuse circuits weights. In a new version of our backend, we integrate our loop unrolling minimisation method as a post-pass of the periodic register allocation.

At first, we have applied our code optimisation method (periodic register allocation followed by unroll factor minimisation) on many DDG extracted from various real benchmarks, either from the embedded domain and from the high performance computing domain: DSP-filters, Spec, Lin-ddot, Livermore, Whetstone, etc. The total number of experimented DDG is 310, their sizes go from 2 nodes and 2 edges up to 360 nodes and 590 edges. Afterwards, we have performed experiments on these DDG, depending on the considered number of registers. We considered three configurations as follow:

1. machine with unbounded number of registers;

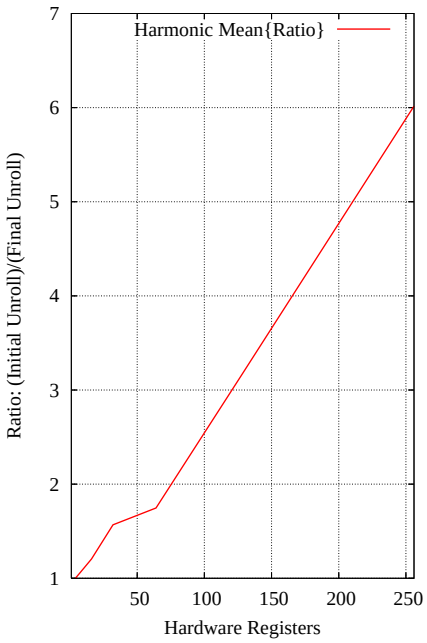


Figure 8: Statistics on Randomly Generated DDG

2. machine with bounded number of registers varied from 4 to 256;
3. machine with bounded number of registers varied from 4 to 256 with the option `continue` (described later).

6.2.1 Machine with Unbounded Number of Registers

Theoretically, the best result for the *LCM-MIN Problem* is $\alpha^* = \mu_k$ the greatest value of $\mu_i,\ \forall i = 1, k$. Hence, we aim with these experiments to calculate the mean of the added registers ($\sum_{i=1}^{k} r_i$) required to obtain μ_k.

In order to interpret all the data resulted from the application of our method to all DDG, we decide to make some statistics. Indeed, we have looked for arithmetic mean to represent the average of the added registers ($AVR_{ar}(\sum_{i=1}^{k} r_i)$) needed to obtain μ_k. Moreover, we calculate the harmonic mean of all the ratio ($AVR_{har}(\frac{\alpha}{\mu_k})$).

Our experiments show that using 12.1544 additional registers in average are sufficient to obtain a minimal loop unrolling degree with $\alpha^* = \mu_k$. We note also that we have a high harmonic mean for the ratio ($AVR_{har}(\frac{\alpha}{\mu_k}) = 2.10023$). That is, our loop unrolling minimisation pass is very efficient regarding code size compaction.

6.2.2 Machine with Bounded Number of Registers

We consider a machine with a bounded number of architectural registers R_{arch}. We varied R_{arch} from 4 to 256 and we apply our code optimisation method on all DDG. Afterwards, we made statistics on the resulting data. For each configuration, we looked for an arithmetic mean to represent the average of number of added registers ($AVR_{ar}(\sum_{i=1}^{k} r_i)$). Moreover, we calculate the harmonic mean of all the ratio described as $AVR_{har}(\frac{\alpha}{\alpha^*})$. Finally, we also calculate the arithmetic mean of the remaining registers ($AVR_{ar}(R)$) after the register allocation step given by our backend compilation framework.

Table 1 shows that our solution find the minimal in all configurations except when $R_{\text{arch}} = 4$. In average, a small number of added registers are sufficient to have a minimal loop unrolling degree (α^*). For example: in the configuration with 32 registers, we find the minimal loop unrolling degree, if we add in average 1.07806 registers

among 9.72285 remaining registers. We note also that we have in many configuration, a high harmonic mean for the ratio ($AVR_{har}(ratio)$). For example, in the machine with 256 registers, $AVR_{har}(ratio) = 2.72581$.

R_{arch}	$AVR_{ar}(\sum_{i=1}^{k} r_i)$	$AVR_{har}(ratio)$	$AVR_{ar}(R)$
4	0	1	0.293562
8	0.0151163	1.00729	0.818314
16	0.250158	1.10463	2.72361
32	1.07806	1.4149	9.72285
64	3.07058	1.96319	29.0559
128	14.0731	2.71566	79.6419
256	15.2288	2.72581	207.118

Table 1: Machine with Bounded Number of Registers

Figure 9 shows the initial loop unrolling degree and the final loop unrolling degree of three benchmarks. We note that the final loop unrolling degree is very small compared to the initial loop unrolling degree.

Figure 10 shows the harmonic mean of the minimised (final) loop unrolling weighted by the number of nodes of different DDG. We calculate this weighted harmonic mean on different configurations. We give a realistic VLIW processor with an issue width of 4 instructions per cycle, where all the DDG are pipelined with $II = MII = \max(MII_{\text{ress}}, MII_{\text{dept}})$. In all configurations, the average of the final unrolling degree of pipelined loops is below 8, a significant improvement over the initial unrolling degree. E.g., in the configuration where $R_{\text{arch}} = 64$, the minimised loop unrolling is in average equal to 7.78.

6.2.3 Machine with Bounded Number of Registers and Option `continue`

In these experiments we use option `continue` of our backend. Without this option, our backend computes the first periodic register allocation which verifies $\sum \mu_i \leq R$ (not necessarily minimal). If we use the option `continue`, our backend generates the periodic register allocation that minimises $\sum \mu_i$. In order to compare these two configurations (Machine with Bounded Number of Registers versus Machine with Bounded Number of Registers using option `continue`), we reproduce the statistics of the previous experiments using this additional option. The results are described in Table 2.

R_{arch}	$AVR_{ar}(\sum_{i=1}^{k} r_i)$	$AVR_{har}(ratio)$	$AVR_{ar}(R)$
4	0	1	0.33412
8	0.015841	1.00774	0.885657
16	0.253726	1.10477	2.79591
32	1.09681	1.42146	9.96854
64	3.25124	2.02749	31.1405
128	9.40373	2.28922	81.7739
256	15.1959	2.71729	207.394

Table 2: Machine with Bounded Registers with Option `continue`

Comparing Table 1 and Table 2, notice that some configurations yield a better harmonic mean for the code compaction ratio with option `continue`, when $R_{\text{arch}} \leq 64$. Conversely, the ratio without option `continue` is better when $R_{\text{arch}} \geq 128$. These strange results are side-effects of the reuse circuits generated by SIRA, which differ depending on the number of architectural register. In addition, the complex mathematical structure of the *LCM-MIN Problem* does not allow to say that, the number of remaining registers R, the lower the unrolling degree would be. I.e., increasing the number of remaining registers (by performing minimal periodic register allocation) does not necessarily mean a maximal reduction of loop unrolling degree.

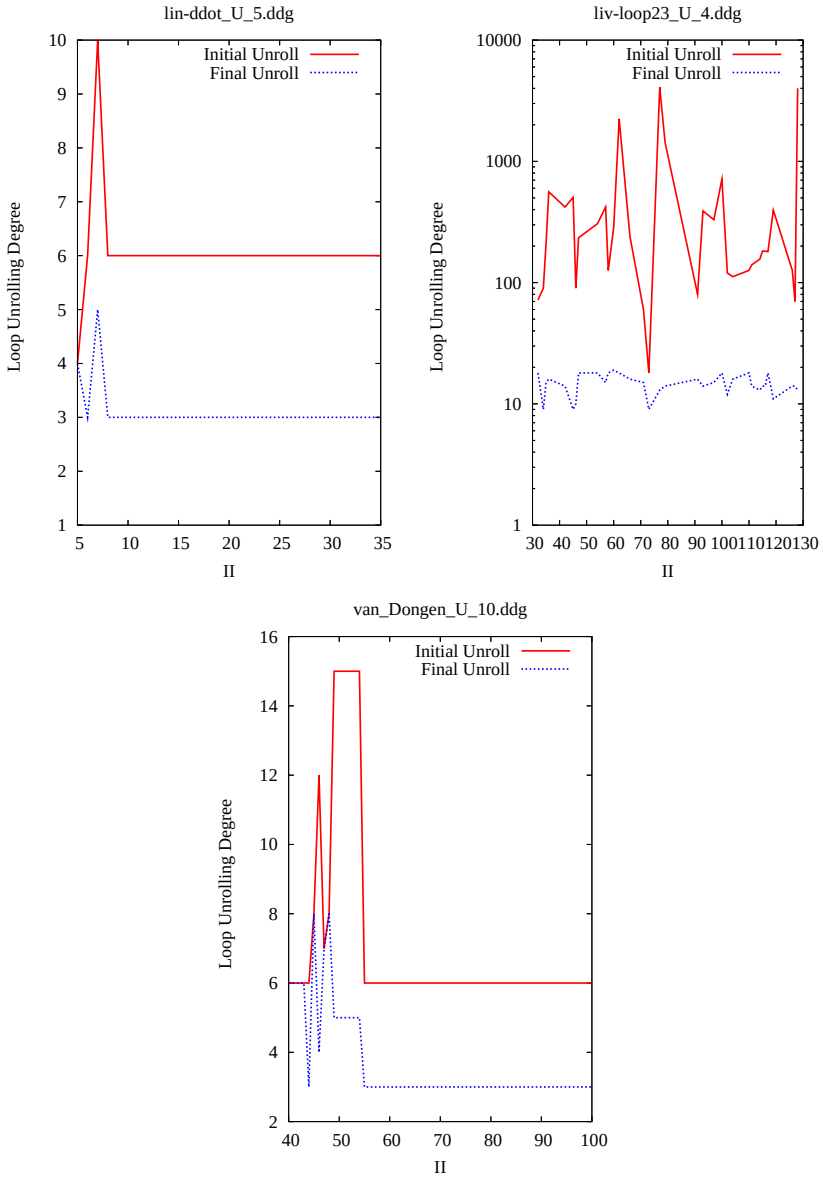


Figure 9: Loop Unrolling Minimisation for Some Benchmarks

7 Conclusion

Contrary to maximal variable expansion [4, 13], periodic register allocation for software pipelined loops require exactly MAXLIVE registers, as proven in [1, 2, 3]. In case of embedded VLIW processors, we cannot rely of rotating register files. So, periodic register allocation implies either loop unrolling or inserting extra `move` operations. Inserting extra operations is not a satisfactory solution for us, because it may alter the II. In this paper, we present a solution for an open problem: how to minimise the loop unrolling degree associated with a given periodic register allocation? Our loop unrolling degree minimisation is implemented as a post-pass of register allocation, and exploits the remaining free registers. Our method guarantees that the final number of available registers is still below R_{arch} while resulting in a minimal unroll degree.

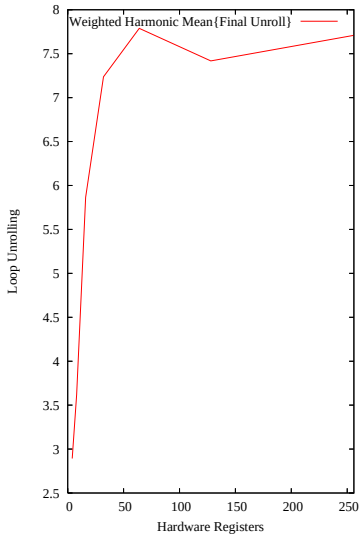


Figure 10: Weighted Harmonic Mean For Minimised Loop Unrolling Degree

We provide a formal setting for this problem, together with algorithms to solve it and with extensive experimental evidence of the practicality of the approach. For instance, considering an embedded VLIW processor family such as the ST2xx with $R_{\text{arch}} = 64$ architectural registers, consuming 3.07 remaining registers on average brings the unrolling degree of pipelined loops below 8, with smaller degrees for larger loops in general.

As a side-result of this work, we notice that the presence of rotating registers files is not really necessary, as loop unrolling seems to be a satisfactory solution to generate code after periodic register allocation. Nevertheless, we noticed that some loops still require high unrolling degrees even after our optimisation. Our future work is twofold: (1) how to insert `move` operations without altering the II while minimising the unroll degree, and (2) how to combine loop unroll degree minimisation with periodic register allocation. The result might be that optimal exploitation of remaining registers plus the insertion of II-preserving `move`s are not sufficient to bring the unrolling degree down to acceptable levels for all loops. This would lead to an interesting limit study of software pipelining without hardware support for rotating register files.

8 Acknowledgments

This work has been supported by the ANR MOPUCE project (ANR number 05-JCJC-0039).

References

[1] 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.

[2] S.A.A. Touati and C. Eisenbeis. Early Periodic Register Allocation on ILP Processors. Parallel Processing Letters, Vol. 14, No. 2, June 2004. World Scientific.

[3] L. J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji. A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs. In Proceedings of the International Conference on Compiler Construction (CC'02). *Lecture Notes in Computer Science*, 641:176–191, 1992.

[4] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines, In Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation, pages 318-328, Atlanta, Georgia, June 22-24, 1988.

[5] 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, Limasol,Cyprus, June 1995. ACM Press.

[6] J.C. Dehnert, P.Y Hsu, and J.P. Bratt. Overlapped Loop Support in the Cydra 5. In proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, pages 26-38, Boston, Massachusetts, 1989.

[7] Sylvain Lelait. *Contribution à l'Allocation de Registres dans les Boucles*. PhD thesis, Université d'Orléans, France, January 1996.

[8] S.A.A. Touati. *Register Pressure in Instruction Level Parallelism*. PhD thesis, Université de Versailles, France, June 2002.

[9] B.D de Dinechin. *A Unified Software Pipeline Construction Scheme For Modulo Scheduled Loops*. Proceedings of the 4th International Conference on Parallel Computing Technologies, pages 189-200, Yaroslavl, Russia, August 7?9, 1997.

[10] R. Cytron and J. Ferrante. What's in a Name? or the Value of Renaming for Parallelism Detection and Storage Allocation, Proceedings of the 1987 International Conference on Parallel Processing, pages 19-27, Pennsylvanie, August 1987.

[11] A. Nicolau, R. Potasman and H. Wang. Register Allocation, Renaming and Their Impact on Fine-Grain Parallelism, Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing, Santa Clara, California, August 1991.

[12] Richard A. Huff. Lifetime-Sensitive Modulo Scheduling, In Proceedings of the ACM SIGPLAN 93 Conference on Programming Language Design and Implementation, pages 258-267, Albuquerque, New Mexico, June 23-25, 1993.

[13] J. A. Fisher, P. Faraboschi and C. Young. Embedded Computing: a VLIW Approach to Architecture, Compilers and Tools, Book, Morgan Kaufmann Publishers, 2005

Algorithm 2 LCM-MIN Problem

Require: k number of reuse circuits, different weights of reuse circuits μ_i, the remaining register R and the loop unrolling degree α

Ensure: the minimal loop unrolling degree α^* and a list $r_1, \ldots, r_k$ of added registers with $\sum_{i=1}^{k} r_i$ minimal

```
α* ← μ_k {minimal value of loop unrolling α*}
if α = α* ∨ R = 0 then
  if R = 0 then
    α* ← α {α cannot be minimised,no remaining registers}
  end if
else
  r_k ← 0 {number of registers added to the reuse circuit μ_k}
  β ← μ_k {value of the first node in the set S}
  j ← 1 {line number j in the set S}
  τ ← α div μ_k {total number of lines in the set S}
  stop ← false {stop = true if the minimal is found}
  success ← false {predicate returned by Algorithm 1}
  while β ≤ α ∧ ⌉(stop) do
    success ← LCM_Problem(β, μ_i, R) {LCM Problem solved by Algorithm 1}
    if ⌉(success) then
      if r_k < R then
        r_k + +
      else
        r_k ← 0 {we go to the first node of the next line}
        j + +
      end if
      β ← j × (μ_k + r_k)
      if β > α ∧ j < τ then
        r_k ← 0 {dashed node, we go to the first node of the next line}
        j + +
        β ← j × μ_k
      end if
    else
      α* ← β
      if α* ≤ (j + 1) × μ_k then
        stop ← true {we are sure that α* is the minimal loop unrolling degree}
      else
        α ← α* {we find a new value of α to minimise}
        τ ← α div μ_k
        r_k ← 0
        j + +
        β ← j × μ_k
      end if
    end if
  end while
end if
```