Decomposing Meeting Graph Circuits to Minimise Kernel Loop Unrolling

Mounira Bachir
INRIA Saclay – Ile-de-France
Mounira.Bachir@inria.fr

Sid-Ahmed-Ali Touati
University of Versailles
Sid.Touati@uvsq.fr

Albert Cohen
INRIA Saclay – Ile-de-France
Albert.Cohen@inria.fr

Abstract

This article studies an important open problem in backend compilation regarding loop unrolling after periodic register allocation. Although software pipelining is a powerful technique to extract fine-grain parallelism, 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 (13; 12). However, the resulting unrolling degree is often unacceptable and may reach absurd levels. Alternatively, loop unrolling can be avoided thanks to software register renaming. This is achieved through the insertion of `move` operations. However, inserting those operations may increase the initiation interval (II) and nullifies the benefits of software pipelining itself.

We propose in this article a new technique to minimise the loop unrolling degree generated after periodic register allocation. In fact, this technique consists on decomposing the generated meeting graph circuits by inserting move instructions without compromising the throughput benefits of software pipelining.

The different experiments showed that the execution time is acceptable and good results can be produced when we have many functional units which can execute move operations.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation, Compilers, Optimization

General Terms Algorithms, Performance

Keywords Periodic Register Allocation, Software Pipelining, Loop Unrolling, Register Move Instructions, Code Optimisation

1. Introduction

Our focus is on the exploitation of instruction-level parallelism (ILP) in embedded VLIW processors (12). 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 and generally addressed through: (1) hardware support — rotating register files — deemed too expensive for almost 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.

We investigate ways to keep the size of the generated code compatible with embedded system 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).

Having a minimal unroll factor reduces code size, which is an important performance measure for embedded systems because they have a limited memory size. Regarding high performance computing (desktop and supercomputers), loop code size may not be important for memory size, but may be so for I-cache performance. In addition to the minimal unroll factors, it is necessary that the code generation scheme for periodic register allocation does not generate additional spill; the number of required registers must not exceed `MAXLIVE` (11) (the maximum number of values simultaneously alive). Prohibiting spill code aims to maintain II and to save performance.

When the instruction schedule is fixed then the circular lifetime intervals (CLI) and `MAXLIVE` are known. In this situation, known methods exist for computing unroll factors. These are:

- modulo variable expansion (MVE) (12; 13) which computes a minimal unroll factor but may introduce spill (since MVE may need more than `MAXLIVE` registers without proving an appropriate upper-bound);
- Hendren's heuristic (10) which computes a sufficient unroll factor without introducing spill, but with no guarantee in terms of minimal register usage or unrolling degree; and
- the meeting graph framework (6) which is based on mathematical proofs which guarantee that the unroll de-

Listing 1. Loop Program Example

```
FOR i=0, i<N, i=i+1
 a[i+2] = b[i]+1
 b[i+2] = c[i]+2
 c[i+2] = a[i]+3
ENDFOR
```

gree will be sufficient to reach register minimality (*i.e.* `MAXLIVE`), but not that the unroll degree itself will be minimal.

Bachir et al. (2; 3) claim that the loop unrolling minimisation (LUM) using extra remaining registers is an efficient method to bring loop unrolling as low as possible — with no increase of the II. However, some kernel loops may still require high unrolling degrees. These occasional high unrolling degrees suggest that it may be worthwhile to consider combining the insertion of `move` operations with kernel loop unrolling.

In this work, we study the loop unrolling problem after a given periodic register allocation by decomposing the different generated meeting graph circuits (MGC) by inserting move instructions without compromising the throughput benefits of software pipelining and still using a minimal number of registers equal to `MAXLIVE` to allocate the different variables. We argue that the approach based on decomposing meeting graph circuits can overcome the shortcomings of loop unrolling minimisation.

The rest of this article is organised as follows. Section 2 presents the most relevant related work for code generation. In Section 3, we present the problem we are dealing by displaying a motivating example. In section 4, we define the main notions, present our method and the algorithm we designed to decompose the different meeting graph circuits in order to minimise loop unrolling degree. In Section 5, we present our experimental results, which demonstrate that our method is efficient in practice, then we conclude in Section 6.

Throughout the paper, the different loops are already software pipelined and we rely on the meeting graph framework as the periodic register allocator.

2. Code Generation: Background and Challenges

We review the main issues and approaches to code generation for periodic register allocation using the loop example described in Listing 1.

There are two ways to deal with periodic register allocation: using special architecture support such as *rotating register files*, or without using such support. This latter may require the insertion of `move` operations or loop unrolling

Listing 2. Example of Inserting Move Operations (Register Renaming)

```
FOR i=0, i<N, i=i+1
  R3 = b[i]+1
  b[i+2] = c[i]+2
  c[i+2] = R1+3
  R1 = R2
  R2 = R3
ENDFOR
```

2.1 Rotating Register File

A *rotating register file* (RRF) (8) is a hardware mechanism to prevent successive lifetime intervals from being assigned to the same physical registers.

In Listing 1, 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]=c[i]+2        b[i+2]=c[i]+2
 c[i+2]=R[-2]+3       c[i+2]=R+3
```

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 called *register renaming*. Considering the example of Listing 1 for allocating $a[i]$, we use 3 registers and perform `move` operations at the end of each iteration (5; 14): $a[i]$ 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 as shown in Listing 2. 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 to maintain II without requiring hardware support such as RRF. The resulted loop body itself is bigger but no extra operations are executed in comparison with the original code. Here different registers are used for different instances of the variable a of Listing 1. In Listing 3, the loop is unrolled three

Listing 3. Example of Loop Unrolling

```
FOR i=0, i<N, i=i+3
  R1 = b[i]+1
  b[i+2] = c[i]+2
  c[i+2] = R2+3
  R2 = b[i+1]+1
  b[i+3] = c[i+1]+2
  c[i+3] = R3+3
  R3 = b[i+2]+1
  b[i+4] = c[i+2]+2
  c[i+4] = R1+3
ENDFOR
```

Listing 4. Example of MVE

```
FOR i=0, i<N, i=i+3
  R1 = R5+1
  R4 = R8+2
  R7 = R2+3
  R2 = R6+1
  R5 = R9+2
  R8 = R3+3
  R3 = R4+1
  R6 = R7+2
  R9 = R1+3
ENDFOR
```

times. $a[i+2]$ is stored in $R1$, $a[i+3]$ in $R2$, $a[i+4]$ in $R3$, $a[i+5]$ in $R1$, and so on.

By unrolling the loop, we avoid inserting extra `move` operations. The drawback is that the code size will be multiplied by 3 in this case, and by the unrolling degree in the general case. This can have a dramatic impact by causing unnessary instruction cache misses when the code size of the loop happens to be larger than the size of the instruction cache. For simplicity, we did not expand the code to assign registers for b and c. In addition, brute force searching for the best solution using loop unrolling has a prohibitive cost, existing solutions may either sacrifice the register optimality (10; 13; 15) or incur large unrolling overhead (6; 16).

2.3.1 Modulo Variable Expansion

Lam designed a general loop unrolling scheme called *modulo variable expansion* (MVE) (13). In fact, the major criterion of this method is to minimize the loop unrolling degree because the memory size of the i-WARP processor is low (13). The MVE method defines a 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 $\alpha = \lceil \frac{\max_v LT_v}{II} \rceil$. Once the loop is unrolled, MVE uses the interference graph for allocation.

Having `MAXLIVE` the maximum number of values simultaneously alive, the problem with MVE is that it does not guarantee a register allocation with minimal number of register equal to `MAXLIVE` (6; 10), and in general it may lead to unnecessary spills breaking the benefits of software pipelining. A concrete examples of this limitation can be found in (1).

In Listing 1, the longest live range lasts 8 cycles and the number of cycles of the loop is 3 cycles, so $\alpha = \lceil \frac{8}{3} \rceil$, and we should unroll the loop 3 times. Then we can assign to each variable a number of registers equal to the least integer greater than the span of the variable that divides II. In Listing 4, each variable $a,\ b,\ c$ is assigned 3 registers using MVE: R1, R2, R3 for a, R4,R5,R6 for b, R7, R8, R9 for c, and the loop is unrolled 3 times.

One can verify that it is not possible to allocate the different variables on less than 9 registers when unrolling the loop 3 times. But MVE does not ensure a register allocation with a minimal number of registers, and hence is not optimal. As we will see in the next section, we need 8 registers to allocate the different variables. In MVE, the round up to the nearest integer for choosing the unrolling degree may miss opportunities for achieving an optimal register allocation.

2.3.2 Meeting Graphs

The algorithm of Eisenbeis et al.(7; 9) can generate a periodic register allocation using a minimal number of registers equal to `MAXLIVE` if the kernel is unrolled, thanks to a dedicated graph representation called the *meeting graph*. It is a more accurate graph than the usual interference graph, as it has information on the number of clock cycles of each variable lifetime and on the succession of the lifetimes all along the loop. It is based on circular lifetime intervals (CLI). A preliminary remark is that without loss of generality, we can consider that the width of the interval representation is constant at each clock cycle. If not, it is always possible to add unit-time intervals in each clock cycle where the width is less than `MAXLIVE` (9).

The formal definition of the meeting graph is as follows.

Definition 1 (Meeting Graph)**.** *Let F be a set of circular lifetime intervals graph with constant width `MAXLIVE`. The meeting graph related to F is the directed weighted graph $G = (V, E)$. V is the set of circular intervals. Each edge $e \in E$ represents the relation of meeting. In fact, there is an edge between two nodes v_i and v_j iff the interval v_i ends when the interval v_j begins. Each $v \in V$ is weighted by its lifetime length in terms of processor clock cycles.*

The meeting graph (MG) allows us to compute an unrolling degree which enables an allocation of the loops with RC=`MAXLIVE` registers. It can have several connected components of weight $\mu_1, \ldots, \mu_k$ (if there is only one con-

Listing 5. Example of Meeting Graph

```
FOR i=0, i<N, i=i+8
  R1 = R2+1
  R4 = R5+2
  R7 = R7+3
  R2 = R3+1
  R5 = R6+2
  R8 = R8+3
  R3 = R4+1
  R6 = R7+2
  R1 = R1+3
  R4 = R5+1
  R7 = R8+2
  R2 = R2+3
  R5 = R6+1
  R8 = R1+2
  R3 = R3+3
  R6 = R7+1
  R1 = R2+2
  R4 = R4+3
  R7 = R8+1
  R2 = R3+2
  R5 = R5+3
  R8 = R1+1
  R3 = R4+2
  R6 = R6+3
ENDFOR
```

nected component, its weight is $\mu_1 = RC$), this leads to the upper bound of unrolling $\alpha = lcm(\mu_1, ..., \mu_k)$ (RC if there is only one connected component). Moreover a possible lower bound of loop unrolling is computed by decomposing the graph into as many circuits as possible and then computing the least common multiple (lcm) of their weights (7; 9). The circuits are then used to compute the final allocation. This method can handle variables that are alive during several iterations. This allocation always finds an allocation with a minimal number of registers (`MAXLIVE`).

Figure 1(a) displays the circular lifetime intervals representing the different variables (a, b, c) of the loop example described in Listing 1, the maximum number of variables simultaneously alive `MAXLIVE` $= 8$. As shown in Figure 1(b), the meeting graph is able to use 8 registers to allocate the different variables instead of 9 with Modulo Variable Expansion by unrolling the loop 8 times. For the loop described in Listing 1, the meeting graph generates the code shown in Listing 5.

The main drawback of the meeting graph method is that the loop unrolling degree can be high in practice although the number of registers used is minimal. That may cause spurious instruction cache misses or even be impracticable due to the memory constraints, like in embedded processors. In or-

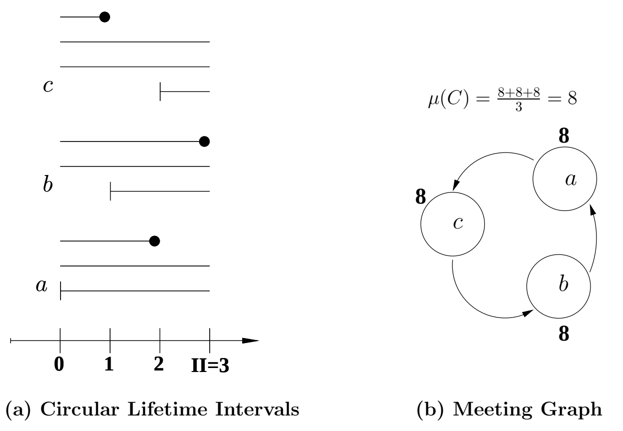
c
b
a
0
1
2
II=3
(a) Circular Lifetime Intervals
$\mu(C) = \frac{8+8+8}{3} = 8$
8
a
8
c
b
8
(b) Meeting Graph

Figure 1. Meeting graph of the loop example in Listing 1

der to minimise loop unrolling, Bachir et al. (2; 3) propose a method called loop unrolling minimisation (LUM method) that minimises the loop unrolling degree by using extra remaining registers for a given periodic register allocation. The LUM method brings loop unrolling as low as possible — with no increase of the II. However, some kernel loops may still require high unrolling degrees. This occasional cases suggest that it may be worthwhile to consider combining the insertion of `move` operations with kernel loop unrolling.

3. Circuit Decomposition: a Motivating Example

In this section, we introduce our method which consists in decomposing the different circuits generated after the meeting graph periodic register allocation, to minimise loop unrolling degree and show how it works on an example. Our study uses the meeting graph framework since it is most challenging to minimal register allocation when parallelism between loop iterations is exploited.

Our goal is to avoid a large unrolling degree while still achieving the use of minimal number of registers. Our approch is based on the following observation: In the previous work that propose a periodic register allocation with a minimal number of register (7; 16), the high generated loop unrolling degree comes from the least common multiple of the different weights of the generated circuits.

Let us study the loop described in Listing 6. This loop is software pipelined by DESP (4). The initiation interval $II = 4$ and the circular lifetime intervals family is described in Figure 2(a). This circular lifetime intervals family has a maximum width `MAXLIVE` $= 3$. As shown in Figure 2(b), the meeting graph is decomposed into two strongly connected components $C_1 = \{v_1, v_2, v_4\}$ and $C_2 = \{v_3\}$ with the following weights $\mu(C_1) = 2$ and $\mu(C_2) = 1$. Moreover, the meeting graph achieves a periodic register allocation with a minimal number of register $R_{min} =$ `MAXLIVE` $= 3$ if we unroll the loop twice ($lcm(2, 1) = 2$).

Listing 6. Whetstone cycle4-2 Loop example

```
FOR i=0, i<10, i=i+1
 a[i] = d[i-2]
 b[i] = a[i]
 c[i] = b[i]
 d[i] = c[i]
ENDFOR
```

Our objective with decomposing the meeting graph circuits is to find a periodic register allocation with a small circuit weights which leads to a small *lcm*. Our exploratory method consists on decomposing the different circuits by adding m extra move instructions which can be performed in parallel with the other operations.

Let us assume that we can add one move operation at each clock cycles without increasing II. By the way, if we split the variables v_4 at clock cycle $II = 4$ into two variables v_{4_1} and v_{4_2} as shown in Figure 3(a), then the circuit C_1 is decomposed into two circuits $C_{1_1} = \{v_1, v_{4_1}\}$ and $C_{1_2} = \{v_2, v_{4_2}\}$ with the following weights $\mu(C_{1_1}) = 1$ and $\mu(C_{1_2}) = 1$, as shown in Figure 3(b). It results a periodic register allocation with a minimal number of registers $R_{min} = \texttt{MAXLIVE} = 3$ without unrolling the loop by adding one move operation.

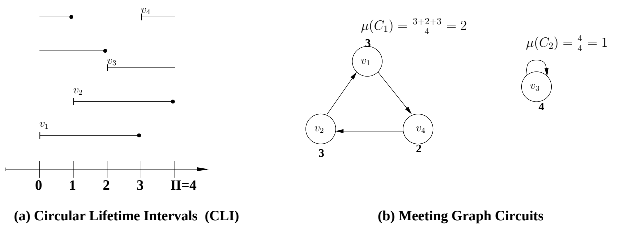
v_4
v_3
v_2
v_1
0
1
2
3
II=4
(a) Circular Lifetime Intervals (CLI)
$\mu(C_1) = \frac{3+2+3}{4} = 2$
$\mu(C_2) = \frac{4}{4} = 1$
v_1
v_2
v_4
v_3
3
3
2
4
(b) Meeting Graph Circuits

Figure 2. Circular lifetime intervals and meeting graph for the loop of Listing 6

We will now describe more precisely the circuit decomposition algorithm

4. Circuit Decomposition Method

In this section, we describe how to decompose the different circuits in order to minimise loop unrolling. In fact, given the different meeting graph circuits $C_1, \ldots, C_k$, we look for their best decomposition into many new circuits $C_{1_1}, \ldots, C_{k_n}$ such that the least common multiple of their new weights is the minimal loop unrolling degree. In fact, decomposing a given circuit C into two circuits C_1 and C_2 means that $\mu(C) = \mu(C_1) + \mu(C_2)$. Lemma 1 demonstrates that each new circuit respect the relation of meeting (9).

Lemma 1. *Let C be a meeting graph circuit with a weight $\mu(C)$. Let $v_1, \ldots, v_n$ be the different variables composing the circuit C. If $\mu(C) > 1$ then there exists at least one decomposition C into two circuits C_1 and C_2 which respect the relation of meeting and $\mu(C) = \mu(C_1) + \mu(C_2)$*

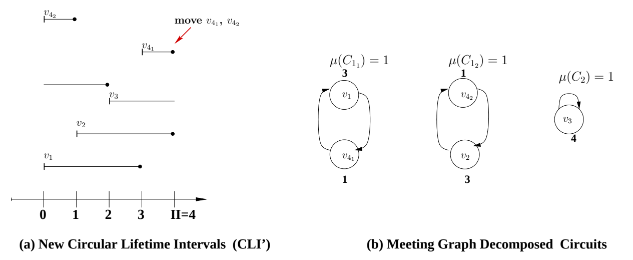

Figure 3. Circuit decomposition method

Proof. Let $C = \{v_1, \ldots, v_n\}$ be a meeting graph circuit. Let S and E be two functions defined as $S(v)$ is the clock cycle when the variable v starts and $E(v)$ is the clock cycle when the variable v ends.

C respects the relation of meeting because $\forall i = 1, n : v_i$ ends when v_{i+1} begins. In addition, the sum of variables lifetime (*LT*) is multiple of II which means $\sum_{i=1}^{n} LT(v_i) = \mu(C) \times II$.

Furthermore, if $\mu(C) > 1$ then we can decompose the circuit C into two circuits C_1 and C_2 such that $\mu(C_1) + \mu(C_2) = \mu(C)$ and $\mu(C_1), \mu(C_2) \in \mathbb{N}^*$. For instance, $\mu(C_1) = 1 > 0$ and $\mu(C_2) = \mu(C) - 1 > 0$.

In order to prove the relation of meeting for both circuits C_1 and C_2, we look for the variables composing each circuit. In fact, we look for a variable $v \in C$ which is alive at a chosen clock cycle denoted *cycle* such as: *cycle* $= S(v_1) + \mu(C_1) \times II$. Let v_i be this variable. Two possible cases are then araised:

1. if $S(v_i) =$ *cycle* then the circuit C is decomposed into two circuits $C_1,\ C_2$ without adding any move operation. The decomposition of the initial circuit C is as follows: $C_1 = \{v_1, \ldots, v_{i-1}\}$ and $C_2 = \{v_i, \ldots, v_n\}$
2. if $S(v_i) <$ *cycle* $< E(v_i)$ then the variable v is splitted at time *cycle* into two variables $v_{i_1},\ v_{i_2}$ which decompose the circuit C into two circuits by adding one move operation. The decomposition of the circuit C is as follows: $C_1 = \{v_1, \ldots, v_{i_1}\}$ and $C_2 = \{v_{i_2}, \ldots, v_n\}$

Circuit decomposition without increasing II proceeds as follows.

1. First of all, we collect the following information: the circular lifetime intervals family (CLI), the different meeting graph circuits and the number m_{cycle} of free move instruction at each clock cycle (*cycle* $= 0, II - 1$). Indeed, these move instructions, if added, can be executed in parallel with other operations without altering the II.
2. Secondly, we give to each circuit C an arbitrary start point denoted $S'(C)$ and an arbitrary end point de-

noted $E'(C)$. For example if $C = \{v_1, \ldots, v_n\}$ then $S'(C) = S(v_1)$ and $E'(C) = E(v_n)$. In addition, C is decomposed into two circuits if its weight is: $\mu(C) > 1$. By the way, the circuit C can be splitted only at some clock cycles described in the following set:

$$\textit{SPLIT}_C = \{\textit{cycle} \mid \forall m \in \mathbb{N}^* \text{ such that:}$$
$$\textit{cycle} = S'(C) + m \times II \text{ and } S'(C) < \textit{cycle} < E'(C)$$
$$\text{and } m_{(\textit{cycle} \mod II)} > 0\}$$

In fact, we need this information to exactly know at each clock cycle we split the circuit. For instance, if we want to split C at II then the exact split cycle is *cycle* $=$ *S'(C)* $+ II$. From the definition of the set $\textit{SPLIT}_C$, we have the following inequalities:

$$S'(C) < \textit{cycle} < E'(C) \Rightarrow$$
$$S'(C) < (S'(C){+}m \times II) < E'(C) \Rightarrow 0 < m < \mu(C)$$

Furthermore, decomposing the circuit C means to look for a variable v in this circuit which is alive at this clock cycle $(S'(C) + m \times II)$.

3. In order to know the total number of possible circuit decomposition at clock cycle *cycle*, we look for the set $CIR_{\textit{cycle}}$ which contains the different circuits starting at this clock cycle *cycle*. In addition, each circuit $C \in CIR_{\textit{cycle}}$ can be decomposed at a given clock cycle *cycle* $= S'(C) + m \times II$ such as $m = 1, \mu(C) - 1$. That is, we can deduce the number $wc_{\textit{cycle}}$ of possible decomposed circuits at time *cycle* described as follows:

$$wc_{\textit{cycle}} = \sum_{C \in CIR_{\textit{cycle}}} (\mu(C) - 1)$$

So hence, the total number of circuit decompositions at clock cycle *cycle* is the sum of the different combinations

$$\sum_{i=0}^{m_{\textit{cycle}}} C^i_{wc_{\textit{cycle}}}$$

Consequently, the total number of generated circuits at clock cycle $II - 1$ is

$$\prod_{\textit{cycle}=0}^{II-1} \sum_{i=0}^{m_{\textit{cycle}}} C^i_{wc_{\textit{cycle}}}$$

4. Finally, compute the least common multiple of the generated circuits and choose the best decomposition with the minimal loop unrolling degree.

Algorithm 1 implements our solution for decomposing the differents circuits following the possible added move intructions without altering II. In this algorithm, we require the initial meeting graph circuits denoted *MGC*, the initiation interval II and the information about free units at each clock cycle *cycle* denoted as $m_{\textit{cycle}}$. This algorithm returns the best circuit decomposition *Circuits* which provides the minimal loop unrolling degree. In addition, because we are looking for all possible circuit decompositions which do not increase II, we need to use three sets $G,\ G_t$ and G_c to store the different circuits after decomposition.

As we can see in Algorithm 1, the principal algorithm calls the sub-algorithm called *Decompose-Circuit* with the following parameters: the initial meeting graph circuits *Circuits*, the circuit *circuit* which we want to decompose into two circuits, the cycle *window* where exactly we want to decompose the circuit *circuit*. This sub-algorithm returns the new circuit decomposition *Result* and a boolean equal to *TRUE* if the circuit is decomposed, *FALSE* otherwise. We also use the following algorithm MINIMIMAL-LCM-CIRCUITS which provides the best circuit decomposition where the least common multiple (loop unrolling) is minimal.

Algorithm 1 Circuits decomposition with free move instructions

```
Require: $MGC$ the set of the meeting graph circuits, $II$ the Initiation Interval and at each clock cycle cycle the number of possible added move instructions $m_{cycle}$ without atering $II$
Ensure: Circuits the best circuit decomposition which provides the minimal loop unrolling degree
  $G \leftarrow \{MGC\}$ {at the beginning the set $G$ contains the initial meeting graph circuits}
  for $cycle = 0$ to $II - 1$ do
    $G_t \leftarrow G$ {initially the set $G_t$ contains all the generated circuits decomposition at clock time $cycle - 1$}
    while $m_{cycle} <> 0$ do
      $G_c \leftarrow \emptyset$ {initialisation of the set $G_c$ which contains the generated circuits decomposition combination}
      for each $Circuits \in G_t$ do
        for each $circuit \in Circuits$ do
          if $S'(circuit) = cycle$ then
            $window \leftarrow cycle + II$
            while $window < E'(circuit)$ do
              if Decompose-Circuit($Circuits, circuit, window, Result$) then
                $G_c \leftarrow G_c \cup \{Result\}$
              end if
              $window \leftarrow window + II$
            end while
          end if
        end for
      end for
      $G \leftarrow G \cup G_c$
      $G_t \leftarrow G_c$
      $m_{cycle} \leftarrow m_{cycle} - 1$
    end while
  end for
  Circuits ← MINIMIMAL-LCM-CIRCUITS($Inter$)
  return Circuits
```

Algorithm 1 delivers all possible decomposing circuits in finite time because it is dominated by the number of generated circuits: $\prod_{cycle=0}^{II-1} \sum_{i=0}^{m_{cycle}} C^{i}_{wc_{cycle}}$

5. Experiments

Measurements are taken to study the effectiveness of minimising the loop unrolling degree by decomposing the initial meeting graph circuits by adding free move operations without increasing II and without increasing the number of allocated registers (the allocation is done with a minimal number of register). In fact, we implemented Algorithm 1 which generates all the possible new meeting graph circuits decompositions (MGCs) following the number of free move operations at each clock cycle and then choose the best circuits decomposition with a minimal loop unrolling degree.

We did extensive experiments on more than 1900 DDGs extracted from various known benchmarks, namely Spec92fp, Spec92int, Livermore loops, Linpack and Nas. In addition, we studied many theoretical cases depending on the number of units which can execute the move operation per clock cycle. In fact, we increase the number of functional units which can execute the move-instruction from one unit to four units (only for theoretical study). Let us notice that the number of units which execute the move operations per clock cycles is a parameter of our program. For each case, we varied the number of architectural registers (R^{arch}) from 16 to 128 registers.

The different experiments show that the technique of decomposing the meeting graph circuits is fast in practice. On average, the execution time is less than 1 seconds at all cases. For instance, the average of the runtime is about 3 milliseconds when we add at most 2 move operations per cycle.

In order to display the main statistics of initial loop unrolling degree and final loop unrolling degree resulted by decomposing circuits method, we show in Table 1 for each machine configuration, the number of DDGs where using the circuits decomposition method (CDM) improves loop unrolling. As we can see, the number of DDGs decomposition increases with the number of units which execute move operations. In order to highlight the main statistics for initial loop

R^{arch}	One move	Two moves	Three moves	Four moves
16	16	75	143	646
32	22	99	179	717
64	29	113	201	759
128	29	113	201	763

Table 1. Number of DDGs where loop unrolling improves thanks to circuit decomposition

unrolling degree and final loop unrolling degree, we display the following numbers: the smallest loop unrolling degree ($\min$), lower quartile ($Q1 = 25\%$), median ($Q2 = 50\%$), the arithmetic mean of loop unrolling degree (average), upper quartile ($Q3 = 75\%$), and largest loop unrolling degree (MAX). The different statistics are shown in the following tables:

- Table 2 shows observations when at most one move instruction can be added per clock cycle.
- Table 3 shows observations when at most two move instructions can be added per clock cycle.
- Table 4 shows observations when at most three move instructions can be added per clock cycle.
- Table 5 shows observations when at most four move instructions can be added per clock cycle.

As we can see, thanks to circuits decomposition method (CDM), we do not unroll 25% of DDGs in all configurations. In addition, we have better improvement when we have four functional units which can execute more operations. In that case, we do not unroll 50% of DDGs and we unroll 75% of DDGs only twice. However, we have a small gain for the maximun loop unrolling in each configuration. Notice that meeting graph proposes to perform a periodic register allocations by unrolling the loop `MAXLIVE` or `MAXLIVE`$+1$ as described in (9; 2), if final loop unrolling is greater then `MAXLIVE`.

R^{arch}	Loop Unrolling	MIN	Q1	Median	Q3	MAX
16	Initial unrolling	2	2	2	5	6
	CDM	1	1	1	2	5
32	Initial unrolling	2	2	4	12	210
	CDM	1	1	2	5	30
64	Initial unrolling	2	2	6	66	2040
	CDM	1	1	3	30	1008
128	Initial unrolling	2	2	6	66	2040
	CDM	1	1	3	30	1008

Table 2. Configuration where at most one move can be added per clock cycle

R^{arch}	Loop Unrolling	MIN	Q1	Median	Q3	MAX
16	Initial unrolling	2	2	3	6	12
	CDM	1	1	2	3	7
32	Initial unrolling	2	2	4.5	6.75	210
	CDM	1	1	2	4	60
64	Initial unrolling	2	2	6	12	420
	CDM	1	1	2	6	210
128	Initial unrolling	2	2	6	12	420
	CDM	1	1	2	6	210

Table 3. Configuration where at most two move operations can be added per clock cycle

6. Conclusion

We presented a new technique to minimise the degree of loop unrolling after periodic register allocation. This technique is based on the meeting graph. It searches for the best

R^{arch}	Loop Unrolling	MIN	Q1	Median	Q3	MAX
16	Initial unrolling	2	2	3	5	20
	CDM	1	1	1	2	10
32	Initial unrolling	2	2	4	6	210
	CDM	1	1	2	3	30
64	Initial unrolling	2	2	4	12	840
	CDM	1	1	2	4.25	420
128	Initial unrolling	2	2	4	12	840
	CDM	1	1	2	4.25	420

Table 4. Configuration where at most three move operations can be added per clock cycle

R^{arch}	Loop Unrolling	MIN	Q1	Median	Q3	MAX
16	Initial unrolling	2	2	2	6	30
	CDM	1	1	1	2	15
32	Initial unrolling	2	2	2	6	210
	CDM	1	1	1	2	30
64	Initial unrolling	2	2	2	6	3696
	CDM	1	1	1	2	2310
128	Initial unrolling	2	2	2	6	6864
	CDM	1	1	1	2	4290

Table 5. Configuration where at most four move operations can be added per clock cycle

decomposition of the meeting graph circuits, through the insertion of register move instructions in free scheduling slots left after software pipelining. The throughput of the software pipeline is guaranteed to be preserved through the decomposition.

Our experiments showed that the running time of the algorithm is acceptable, andthat good results can be produced when multiple functional units are capable of executing move instructions. In fact, when the architecture has four functional units which can perform move operations, we do not unroll 50% of the DDGs and we unroll only twice 75% of the DDGs.

References

[1] Mounira Bachir. *Loop Unrolling Minimisation for Periodic Register Allocation*. PhD thesis, Université de Versailles Saint-Quentin-En-Yvelines, France, 2010.

[2] Mounira Bachir, David Gregg, and Sid-Ahmed-Ali Touati. Using the meeting graph framework to minimise kernel loop unrolling for scheduled loops. In *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing*, Delaware, USA, 2009.

[3] Mounira Bachir, Sid-Ahmed-Ali Touati, and Albert Cohen. Post-pass periodic register allocation to minimise loop unrolling degree. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 141–150, New York, NY, USA, 2008. ACM.

[4] Antoine Sawaya Christine Eisenbeis. Optimal loop parallelization under register constraints. Technical report, INRIA, France, January 1996.

[5] Ron Cytron and Jeanne Ferrante. What's in a name? -or- the value of renaming for parallelism detection and storage allocation. In *Proceedings of the 1987 International Conference on Parallel Processing (ICPP)*, pages 19–27, 1987.

[6] D. de Werra, Ch. Eisenbeis, S. Lelait, and B. Marmol. On a graph-theoretical model for cyclic register allocation. *Discrete Applied Mathematics*, 93(2-3):191–203, 1999.

[7] Dominique de Werra, Christine Eisenbeis, Sylvain Lelait, and Elena Stohr. Circular-arc graph coloring: On chords and circuits in the meeting graph. *European Journal of Operational Research*, 136(3):483–500, February 2002.

[8] James C. Dehnert, Peter Y.-T. Hsu, and Joseph P. Bratt. Overlapped loop support in the cydra 5. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 26–38, New York, NY, USA, 1989. ACM.

[9] Christine Eisenbeis, Sylvain Lelait, and Bruno Marmol. The meeting graph: a new model for loop cyclic register allocation. In *PACT '95: Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques*, pages 264–267, Manchester, UK, 1995. IFIP Working Group on Algol.

[10] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. In *CC '92: Proceedings of the 4th International Conference on Compiler Construction*, pages 176–191, London, UK, 1992. Springer-Verlag.

[11] Richard A. Huff. Lifetime-sensitive modulo scheduling. *SIGPLAN Not.*, 28(6):258–267, 1993.

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

[13] M. Lam. Software pipelining: an effective scheduling technique for vliw machines. *SIGPLAN Not.*, 23(7):318–328, 1988.

[14] Alexandru Nicolau, Roni Potasman, and Haigeng Wang. Register allocation, renaming and their impact on fine-grain parallelism. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 218–235, London, UK, 1992. Springer-Verlag.

[15] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. *SIGPLAN Not.*, 27(7):283–299, 1992.

[16] Sid-Ahmed-Ali Touati and C. Eisenbeis. Early Periodic Register Allocation on ILP Processors. *Parallel Processing Letters*, 14(2):287–313, 2004.