# Using The Meeting Graph Framework to Minimise Kernel Loop Unrolling for Scheduled Loops

Mounira BACHIR[1], David GREGG[2], and Sid-Ahmed-Ali TOUATI[3]

[1] INRIA Saclay – Ile-de-France
Mounira.Bachir@inria.fr
[2] Trinity College Dublin, Dublin 2, Ireland
David.Gregg@cs.tcd.ie
[3] Université de Versailles Saint-Quentin en Yvelines, France,
Sid.Touati@uvsq.fr

**Abstract.** This paper improves our previous research effort [1] by providing an efficient method for kernel loop unrolling minimisation in the case of already scheduled loops, where circular lifetime intervals are known. When loops are software pipelined, the number of values simultaneously alive becomes exactly known giving better opportunities for kernel loop unrolling. Furthermore, fixing circular lifetime intervals allows us to reduce the algorithmic complexity of our method compared to [1] by computing a new research space for minimal kernel loop unrolling. The meeting graph (MG) [3] is one of the frameworks proposed in the literature which models loop unrolling and register allocation together in a common formal framework for software pipelined loops. Although MG significantly improves loop register allocation, the computed loop unrolling may lead to unpractical code growth.
This work proposes to minimise the loop unrolling degree in the meeting graph by making an adaptation of the approach described in [1]. We explain how to reduce the research space for minimal kernel loop unrolling in the context of MG, yielding to a reduced algorithmic complexity. Furthermore, our experiments on SPEC2000, SPEC2006, MEDIABENCH and FFMPEG show that in concrete cases the loop unrolling minimisation is very fast and the minimal loop unrolling degree for $75\%$ of the optimised loops is equal to 1 (*i.e.* no unroll), while it is equal to 7 when the software pipelining (SWP) schedule is not fixed.

## 1 Introduction

In production compilers, register allocation involves keeping as many variables as possible in registers, thereby avoiding the need to introduce spill code which is particular danger in software-pipelined loops [5, 7, 9]. When no hardware support is available, kernel loop unrolling is currently *the only* method of code generation that avoids introducing unnecessary move and spill operations and does not alter the initiation interval after software pipelining. However, the degree of unrolling should be minimised to control code size and hence I-cache performance.

Our objective in this research effort is the same as in [1]: we are interested in the minimal value of kernel loop unrolling which allows a periodic register allocation for

software pipelined loops without exceeding the number of architectural registers. Prohibitive unrolling degrees decrease the benefit of software pipelining because:

1. The code size increase is not appreciate for embedded systems;
2. The performance of large loops may suffer from I-cache effects;

When a loop is already scheduled, the meeting graph (MG) [5] is a formal framework which proposes to achieve a periodic register allocation with a minimum number of registers equal to MAXLIVE, the number of values simultaneously alive. This graph describes how to find this allocation if we sufficiently unroll the pipelined loop. The meeting graph is decomposed into its elementary circuits labelled with their weights ($w_i$), where each circuit corresponds to a reuse pattern. The drawbacks are that the unrolling factor $\alpha$ is the least common multiple of the weights $(w_i)$, and that it is difficult to extract a circuit decomposition that minimises $\alpha$.

The current article adapts the loop unrolling minimisation method described in [1] to reduce the kernel loop unrolling generated by the MG and proposes a new research space $S$ for minimal kernel loop unrolling. In addition, the asymptotic complexity of our method for already software pipelined loops is reduced compared to [1].

Compared to our previous result [1], the current study handles already scheduled loops (after SWP). In contrast, our previous result deals with unscheduled loops (before SWP), which means that the computed unrolling factor was conservative and valid for all subsequent SWP schedules. The difference resides in the fact that, when a loop is already scheduled in the MG, circular lifetime intervals are fixed and hence we have (in theory) better opportunities to minimise the unrolling degree. Actually, having an upper bound for kernel loop unrolling (MAXLIVE or MAXLIVE $+ 1$) depending on whether the MG has one or more strong connected components, we reduce the research space $S$ described in [1] by computing all the possible new kernel loop unrolling less or equal to MAXLIVE or MAXLIVE $+ 1$.

This article is organised as follows. Section 2 presents relevant related work. Section 3 provides a brief recap of the meeting graph describing how to perform periodic register allocation for a software pipelined loop. Section 4 defines the problem of minimising the kernel loop unrolling, then we describe our solution for loop unrolling minimisation. Section 5 presents some experimental results, showing that our loop unrolling minimisation method is fast and efficient in practice. Finally, we summarise our results and discuss some perspectives.

## 2 Code Generation Approaches for Cyclic Register Allocation

When a loop is software pipelined, we have three schemes for handling the overlapping lifetimes :

1. Dynamic remapping via the use of rotating register files [4, 12]. Such hardware support may find its place in high performance processors (such as Intel's Itanium) but is hardly ever found in embedded processors.
2. Software register renaming. This is achieved through the insertion of `move` operations [11]. However, inserting these operations may decrease the computation throughput.

3. Kernel loop unrolling, which is more suitable for embedded processors. However, brute force searching for the best solution using loop unrolling has a prohibitive cost, existing solutions may either sacrifice the register optimality [6, 9, 12] or incur large unrolling overhead [3, 14].

When no hardware support is available, the classical software solution that is guaranteed never to alter the computation throughput is loop unrolling. The following section presents the best known techniques using loop unrolling.

### 2.1 Modulo Variable Expansion

Lam designed a general loop unrolling scheme called *modulo variable expansion* (MVE) [9, 12]. This method defines a minimal unrolling degree to permit code generation after a given periodic register allocation. This unrolling degree is obtained by dividing the length of the longest lifetime ($LT$) ($(\max_v LT_v)$) by the initiation interval $\frac{(\max_v LT_v)}{II}$. However, this method does not guarantee a register allocation with a minimal number of registers equal to MAXLIVE and in general it may lead to unnecessary spills or move operations negating the benefits of SWP. These extra spill or move operations may increase the initiation interval of the SWP. A concrete example of this limitation can be found in [13].

### 2.2 SIRA Reuse Graphs

Usually the register allocation is performed after or during the software pipelining process[3, 6]. This is because doing a conventional register allocation as a first step without assuming a schedule lacks the information of interferences between values live range. However, there exists a theoretical approach, called Schedule Independent Register Allocation (SIRA) [14] for controlling the register pressure before software pipelining. This new method is based on inserting some anti dependence arcs (register reuse arcs) labeled with reuse distances $\mu$, directly into the data dependence graph $G$. Such periodic register allocation is modeled by *reuse graphs*. Reuse graphs are decomposed into reuse circuits which model register reuse decisions between the instructions of the loop: an arc $e = (u, v)$ in a reuse circuit labeled with a distance $\mu(e)$ means that the instruction $u$ of iteration $i$ and the instruction $v$ of iteration $i + \mu(e)$ share the same destination register. The sum of these labels along the circuit defines its weight.

Reuse graphs have another use. They may be used to compute the *sufficient unrolling* degree that we should apply to the loop so that it is always possible to allocate exactly $R_{\text{min}} =$ MAXLIVE registers, *independently of the actual scheduling* [14]. The drawback of this allocation is that the unrolling factor is equal to the least common multiple of the weight of all reuse circuits.

Another relevant approach using loop unrolling for already software pipelined loops is described in the next section.

## 3 Meeting Graph

Several algorithms have been proposed to achieve a periodic register allocation with a minimum number of registers [5, 6]. The *meeting graph* (MG) [5, 10] describes how

to find a periodic register allocation with MAXLIVE registers if we sufficiently unroll the pipelined loop. It is a more accurate graph than the usual interference graph, as it has information on the number of cycles of each variable lifetime and on the succession of the lifetimes all along the loop. It allows us to compute an unrolling degree which enables an allocation of the loops with $R_{min}$ = MAXLIVE registers. A MG can have several connected components of weight $\mu_1, \ldots, \mu_k$ (if there is only one connected component, its weight is $\mu_1 = R_{min}$), this leads to the upper bound of unrolling $\alpha = \text{lcm}(\mu_1, ..., \mu_k)$ ($R_{min}$ if there is only one connected component). Moreover a possible lower bound is computed by decomposing the graph into as many circuits as possible and then computing the lcm of their weights. 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 an optimal number of registers (MAXLIVE). The main drawback of this method is that the loop unrolling degree can be high in practice although the number of registers used is optimal.

An interval family representing variable lifetime is shown in Figure 1. From these lifetimes, the corresponding meeting graph is drawn. The interval family has a width equal to $R_{min} = 4$. This width means that we need at least 4 registers (colors) to be allocated successfully. On each node, a weight equal to the number of clock cycles (time steps) of the lifetime is added. The weight of the connected component is $\frac{20}{5} = 4$, as $II = 5$ and the sum of nodes' weights is $20$.

Moreover it has 4 chords, so there are two ways to decompose it. One way leads to an unrolling of $\text{lcm}(1, 3) = 3$ iterations and the second to an unrolling of $\text{lcm}(2, 2) = 2$ iterations. Here the maximal loop unrolling is $R_{min} = 4$, we have only one connected component. Lelait's solution [5] is to find the decomposition by searching for the greatest number of chords which *do not intersect* inside the graph. This search is equivalent to looking for the maximum stable set in the circle graph induced by the chords.

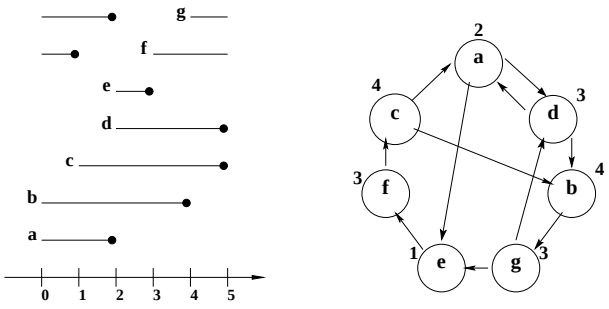


**Fig. 1.** Meeting Graph Example

## 4 Loop Unrolling Problem in the Meeting Graph

Code generation methods such as meeting graph [5] and SIRA [14] rely on the proof that MAXLIVE registers are sufficient for periodic register allocation if we unroll the loop up to the least common multiple of the weights (total distance) of the reuse circuits. Unfortunately, the resulting unrolling degree is often unacceptable.

We recently proposed a solution to dramatically reduce the cost of the unrolling degree [1] inside the framework SIRA [14]. The results showed that the final loop unrolling degree is greatly reduced. The loop unrolling minimisation algorithm (LUM) has exponential time complexity in the worst case but in practice the solution is very fast (about 2 minutes in extreme cases) [1]. We aim in the current work to improve the complexity of the algorithm and to show the effectiveness of this work on already software pipelined loops using the meeting graph technique for the periodic register allocation.

Figure 2 illustrates an example. We want to minimise the loop unrolling degree of the five circuits of the meeting graph in Figure 2(b). Initially the meeting graph has only one connected component with $\mu_1 = \text{MAXLIVE} = 27$ (see Fig 2(a)). Then, the meeting graph is decomposed by following the different chords, which results $5$ circuits with the following weights $\mu_1 = 3, \mu_2 = 4, \mu_3 = 5, \mu_4 = 7, \mu_5 = 8$ as shown in Fig 2(b). The kernel loop unrolling degree resulting from this decomposition is $\alpha = 840$, the LCM of the weights of the different circuits. However, this bound is very large and we cannot allow the loop to be unrolled $840$ times. The meeting graph proposes in this case to unroll the loop MAXLIVE times. This bound is also large. In order to minimise it, we apply the loop unrolling minimisation for the meeting graph circuits. Let us assume that we have $R_{hw} = 32$ architectural registers. Hence we have $R = R_{hw} - R_{min} = 32 - 27 = 5$ remaining registers.

Our work aims to compute the minimal loop unrolling degree $\alpha^*$ for the software pipelined loop using the meeting graph framework. We are willing to exploit the remaining registers, looking for a good distribution of these registers over all the different strongly connected components. In Figure 2(b), the final loop unrolling degree found with this method is $\alpha^* = 8$. The minimal number of registers added to each circuit of the meeting graph are: $r_1 = 1$, $r_2 = 0$, $r_3 = 3$, $r_4 = 1$, $r_5 = 0$. Note that $r_i$ is the number of registers added to the $i^{th}$ circuit of meeting graph. LUM guarantees that the new number of allocated registers will not exceed the number of architectural registers; $R_{alloc} \leq R_{hw}$.

In the next section, we briefly recall the formal description for minimising loop unrolling in the meeting graph technique.

### 4.1 Adaptation of Loop Unrolling Minimisation Problem in Meeting Graph

The heart of our loop unrolling minimisation (LUM Problem) is based on the following observation. Let $R_{\text{arch}}$ 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 there are $R = R_{\text{arch}} - R_{\text{min}}$ free registers remaining. Our goal is to exploit these remaining registers to minimise the loop unrolling degree $\alpha$. That is, our loop compaction method is based *on using extra free registers if they exist* to reduce

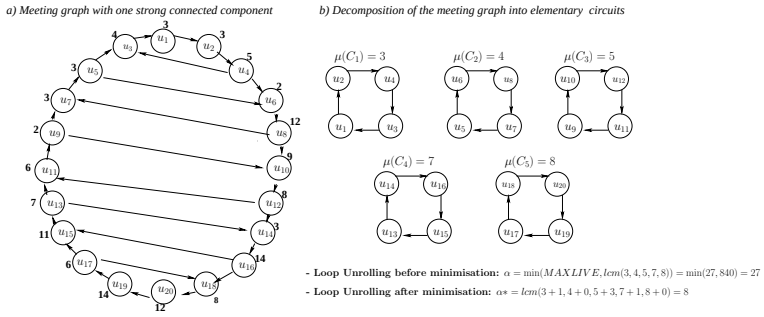


**Fig. 2.** Example of Loop Unrolling Minimisation in Meeting Graph

the unrolling degree, without adding extra `move` operations, and without altering the $II$ of the software pipelined schedule.

The formal description of LUM Problem in the meeting graph is as follows:

**Problem 1 (LUM)** *Let $R \in \mathbb{N}$ be the number of remaining registers after the periodic register allocation performed by the meeting graph technique. Let $\mu_1, \ldots, \mu_k \in \mathbb{N}$ be the weights of the different circuits generated by decomposing the meeting graph following the different chords. Compute the added registers $r_1, \ldots, r_k \in \mathbb{N}$ to the different circuits such that:*

- $\sum_{i=1}^{k} r_i \leq R$ *(validity constraint).*
- $\alpha^* = lcm(\mu_1 + r_1, \ldots, \mu_k + r_k)$ *is minimal (optimisation objective).*

The following section recalls the solution for LUM Problem described in [1].

### 4.2 Solution for Loop Unrolling Minimisation [1]

The solution of *LUM Problem* is detailed in [1]. The procedure to compute the minimal kernel loop unrolling $\alpha^*$, using our solution consists of:

- constructing the set $S$ of all possible value of $\beta$ which can be a potential new loop unrolling degree. The construction of the set is described in [1].
- checking if each value $\beta$ in the search space $S$ can be a solution for the *LCM-Problem* [1]: it is guaranteed that the minimum of all these values is the minimal loop unrolling degree.

In general, the *LCM-Problem* determines if a fixed loop unrolling degree $\beta$ can be the new loop unrolling. This is done by adding to each circuit weight $\mu_i$ of the meeting graph, a minimal number of registers $r_i$ from the remaining $R$ registers such that $\mu_i + r_i$ is the smallest divisor of the fixed loop unrolling $\beta$ greater or equal to $\mu_i$. If the additional registers do not exceed the number of remaining registers $\sum_{i=1}^{k} r_i \leq R$, then

$\beta$ can be the new loop unrolling degree. In this case, the algorithm returns a predicate $success$ with the value $true$.

Instead of computing all values $\beta$ of $S$ which satisfy the *LCM-Problem* and finally taking the minimal one, we describe in [1] an efficient way to traverse the set $S$ in order to find the minimal kernel loop unrolling. The solution is exponential in the worst case but in practice it runs very fast [1]. This is due to reducing the research space $S$ each time a new loop unrolling degree $\alpha'$ less than the original $\alpha$ is found.

Figure 3 illustrates the search space solution (a partial order between nodes). The value of each node represents a potential new loop unrolling degree and an arc between two nodes $a,\ b$ $(a \rightarrow b)$ means that $a < b$ and the absence of an arc (ot a path) between two nodes means that the order is unknown. If we assume that $\mu = \mu_k$ (maximum weight of all the different circuits in the meeting graph) and $R$ are remaining registers after the periodic register allocation then we traverse the set $S$ by proceeding line by line in the figure. In each line, we apply Algorithm *LCM-Problem* [1] to each node in turn until the value of the predicate $success$ returned by this algorithm is $true$ or until we arrive at the last line where $\beta = \alpha$. If the value $\beta$ of the node $i$ of the line $j$ verifies the predicate ($success = true$), then we have two cases:

- case a: 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 $\beta$ (by construction of the set $S$).
- case b: otherwise we have found a new value of unrolling degree which is less than the original $\alpha$. We note this new value $\alpha$" and we try once again to minimise it until we find the minimal (case a). The search space becomes smaller ($S' = \{\beta \in \mathbb{N} | \forall r = 0..R : \beta$ is multiple of $(\mu + r) \wedge (j + 1) \times \mu \leq \beta \leq \alpha'\}$)

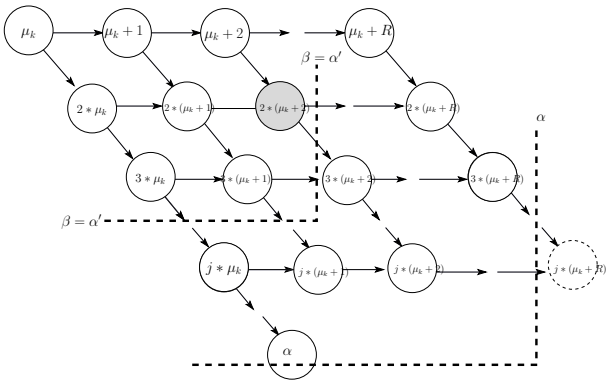


**Fig. 3.** Loop Unrolling Values in the Search Space $S$ [1]

We compute in the following sections the complexity of LUM Problem called LCM-MIN algorithm in [1] and then explain how to improve it in the meeting graph.

### 4.3 Complexity of the LCM-MIN Algorithm [1]

The definition of the *LCM-MIN Problem* is:

- Input: $\mu_1, \ldots, \mu_k$ (natural numbers) and $R$ (natural number).
- Output: $r_1, ..., r_k$ (natural numbers). Such that :
  - $r_1 + \ldots + r_k \leq R$ and
  - $LCM(\mu_1 + r_1, \ldots, \mu_k + r_k)$ is minimal

Let us say that we can solve the *LCM-Problem* [1] in an amount of time equal to $T$. In the worst case, we visit every node in the set $S$ (see fig 3).
The set $S$ has $\frac{R \times \alpha}{\mu_k}$ nodes (assuming that $\mu_k = \max \mu_i$ and $\alpha = LCM(\mu_1, ..., \mu_k)$).
Hence, the algorithmic complexity of *LCM-MIN Algorithm* is $T_{total} = O(\frac{T \times R \times \alpha}{\mu_k})$.
Assuming that $\mu_1, ..., \mu_k$ are relatively prime, then $\alpha$ has its maximum value $\alpha \leq \mu_1 \times \mu_2 \times \ldots \times \mu_k \leq {\mu_k}^k$. Thus, the algorithlmic complexity of *LCM-MIN Algorithm* is $T_{total} = O(\frac{T \times R \times {\mu_k}^k}{\mu_k}) = O(T \times R \times {\mu_k}^{(k-1)})$. Since $R \leq R_{arch}$ and $k \leq n$ ($n$ is the number of loop statements), we have $T_{total} = O(T \times R_{arch} \times {\mu_k}^{(n-1)})$. If we fix a processor architecture, $R_{arch}$ becomes a constant and the worst case complexity becomes equal to $T_{total} = O(T \times {\mu_k}^{(n-1)})$. We conclude that the worst case complexity of *LCM-MIN Algorithm* [1] is exponential on the value of $\mu_k$.

### 4.4 Improving LCM-MIN Algorithm for the Meeting Graph Framework

Meeting Graph can have several strongly connected components of weight $\mu_1, \ldots, \mu_k$ (if there is only one connected component, its weight is $\mu_1 = R_{min}$). This leads to the upper bound of unrolling $\alpha = \text{lcm}(\mu_1, ..., \mu_k)$ (MAXLIVE $= R_{min}$ if there is only one connected component). In addition, if $\alpha >$ MAXLIVE, MG proposes a new upper bound of loop unrolling degree $u_{max}$ equal to MAXLIVE or MAXLIVE $+ 1$. In fact, if MG has one strongly connected component then the maximum loop unrolling degree is $u_{max} = R_{min} =$ MAXLIVE. Otherwise, if it has several strongly connected components, [5] proposes to create one strongly connected component by adding a complete turn of unitary dummy intervals in the MG. One extra register is needed to achieve this, which yields to allocate MAXLIVE $+ 1$ registers by unrolling the loop $u_{max} =$ MAXLIVE $+ 1$. Consequently, we may need one extra register to cyclically permute all the values in registers.

Moreover a possible lower bound is computed by decomposing the MG into as many circuits as possible and then computing the lcm of their weights. However, in practice, the maximal loop unrolling degree can be high even though the number of registers used is minimal.

Our research result finds the minimal loop unrolling degree $\alpha^*$ regarding a fixed schedule using the MG technique. We use the LCM-MIN algorithm, looking for a good distribution of the remaining registers over all the different MG circuits. Having an upper bound for loop unrolling degree (MAXLIVE or MAXLIVE $+ 1$), we reduce the search space $S$ by computing all the possible new loop unrolling degree $\beta$ less or equal to MAXLIVE or MAXLIVE $+ 1$ depending if the MG has one or more strongly connected components. Figure 4 describes the new search space $S$ in the MG.

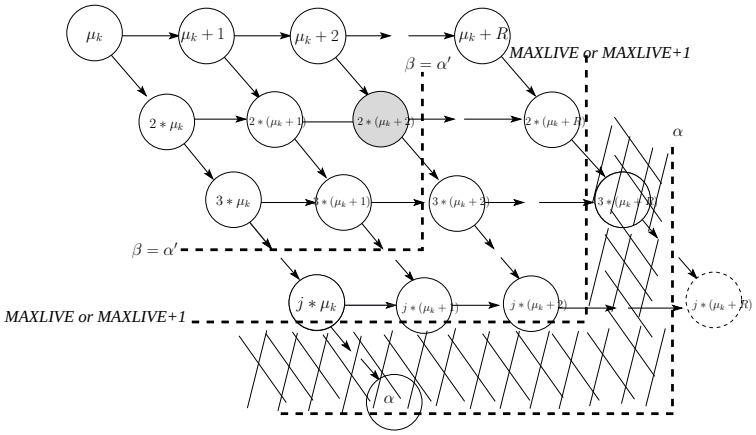


**Fig. 4.** The new Search Space $S$ in the Meeting Graph

Contrary to [1], we are faced here to a fixed loop schedule. This reduces the cardinal of the research space $S$ (see Figure 4), the efficiency of solving *LCM-MIN Problem* is improved. Actually, in the worst case, we visit every node in the new set $S$. In this case, the set $S$ has $R \times \frac{\text{MAXLIVE}}{\mu_k}$ nodes (respectively $R \times \frac{\text{MAXLIVE+1}}{\mu_k}$). So hence, the total resolution time is $T_{total} = O(T \times R \times \frac{\text{MAXLIVE}}{\mu_k})$.

In addition, the loop variables must be allocated with a number of register less or equal to the number of architectural registers $R_{arch}$. This amounts to the following inequality: $T \times R \times \frac{\text{MAXLIVE}}{\mu_k} \leq T \times R \times \frac{R_{arch}}{\mu_k} \leq T \times R \times R_{arch}$. Since $R \leq R_{arch}$, we can write $T \times R \times \frac{\text{MAXLIVE}}{\mu_k} \leq T \times R_{arch}^2$. By fixing a processor architecture, $R_{arch}$ becomes a constant and the complexity of solving *LCM-MIN Problem* becomes $O(T)$. From this complexity analysis, we conclude that in the meeting graph *LCM-MIN Algorithm* can be solved in a better time than in the previous approach [1].

## 5 Experimental Results

We integrated our loop unrolling minimisation method as a post-pass of the meeting graph technique in LoRA [2] framework (Loop optimal Register Allocation). LoRA implements the meeting graph technique and several heuristics (Lam's heuristic [9] and those of Hendren et al. [6]), for combining register allocation and loop unrolling for SWP loops.

The following section presents our experimental results. We consider a machine with a bounded number of architectural registers $R_{\text{arch}}$. We varied $R_{\text{arch}}$ from 16 to 256 and we apply the meeting graph technique followed by our code optimisation method on all data dependence graphs (DDGs). Afterwards, we made statistics on the resulting data: the initial loop unrolling degree $\alpha$, the final loop unrolling degree $\alpha^*$ and the ratio $\frac{\alpha}{\alpha^*}$. For each configuration, we chose to graph the different results as boxplots, which are a convenient way of graphically depicting groups of numerical data through their

five-number summaries: the smallest observations ($\min$), lower quartile ($Q1 = 25\%$), median ($Q2 = 50\%$), upper quartile ($Q3 = 75\%$), and largest observations ($\max$). In addition, we looked for an arithmetic mean to represent the average of $\alpha$, $\alpha^*$ and the ratio.

### 5.1 Unroll Degree Minimisation with Fixed SWP Schedules

In this section, we start the study with smaller loops from various known benchmarks, namely Spec92fp, Spec92int, Livermore loops, Linpack and Nas. The number of optimised loops for these small benchmarks is 1935. All the loops are scheduled with DESP [8]. The next section will present experiences on loops of bigger applications.

Loop unrolling minimisation method is applied when meeting graph finds a periodic register allocation less than or equal to the number of architectural registers. Otherwise, MG does not unroll the loop and proposes an heuristic to introduce spill code.

Table 1 shows the number of DDGs when MG finds periodic register allocation without spilling among 1935 DDGs and the number of DDGs where spill codes are introduced.

| $R_{\text{arch}}$ | Unrolled Loop with MG | Spilled Loops with MG |
|---|---|---|
| 16 | 1602 | 333 |
| 32 | 1804 | 131 |
| 64 | 1900 | 35 |
| 128 | 1929 | 6 |
| 256 | 1935 | 0 |

**Table 1.** Number of Unrolled Loops compared to the number of Spilled Loops resulting from Meeting Graph technique

To highlight the improvements of our loop unrolling minimisation method on DDGs where MG found a solution (no spill), we show in Figure 5 a boxplot for each processor configuration. We remark that the final (minimised) loop unrolling of half of the DDGs is under 2 and that the minimised loop unrolling of $75\%$ of applications is less than or equal to 3, while the upper quartile of initial loop unrolling is less than or equal to 6. We note also that the maximum loop unrolling degree is improved in each processor configuration. For example, in the machine with $128$ registers, the maximum loop unrolling degree is reduced from $21840$ to $41$. In addition, we looked for an arithmetic mean to represent the average of the initial loop unrolling $\alpha$, the final loop unrolling $\alpha^*$ and $ratio = \frac{\sum \alpha}{\sum \alpha^*}$. Table 2 shows that on average the final loop unrolling degree is greatly reduced compared to the initial loop unrolling degree.

For each configuration we also computed the number of loops where the minimised loop unrolling degree is less than MAXLIVE. We draw in Table 3 the different results. It shows that in each configuration, the minimal loop unrolling degree obtained using

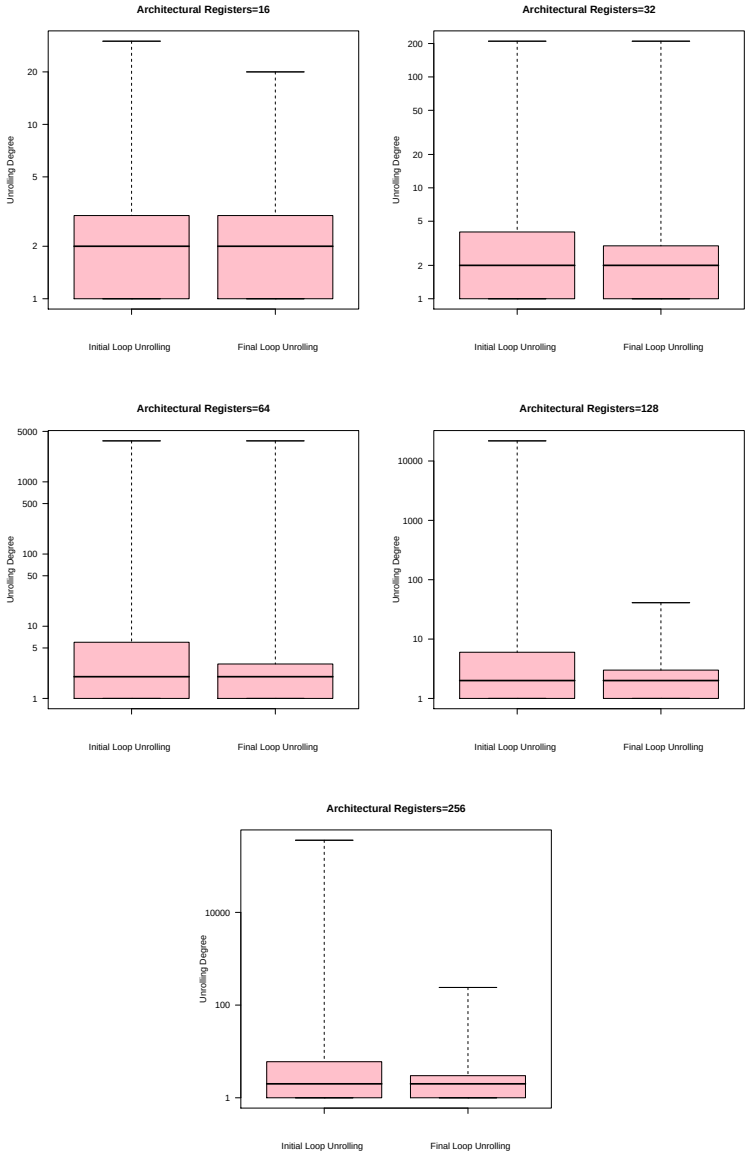


**Fig. 5.** Initial vs. Final Loop Unrolling in each Configuration (Small Benchmarks)

our method is greatly less than MAXLIVE. Only a very small number of loops are unrolled MAXLIVE times.

| $R_{\text{arch}}$ | Average Initial Loop Unrolling Factors | Average Minimised Loop Unrolling Factors | Average Arithmetic Ratio |
|---|---|---|---|
| 16 | 2.743 | 2.207 | 1.242 |
| 32 | 4.81 | 2.569 | 1.872 |
| 64 | 25.86 | 11.02 | 2.346 |
| 128 | 236.6 | 2.852 | 82.959 |
| 256 | 525.7 | 3.044 | 172.7 |

**Table 2.** Arithmetic Mean of Initial Loop Unrolling, Final Loop Unrolling and Ratio

| $R_{\text{arch}}$ | Minimal loop unrolling $<$ MAXLIVE | number of loops unrolled MAXLIVE times | Total number of loops |
|---|---|---|---|
| 16 | 1601 | 1 | 1602 |
| 32 | 1801 | 3 | 1804 |
| 64 | 1893 | 7 | 1900 |
| 128 | 1929 | 0 | 1929 |
| 256 | 1935 | 0 | 1935 |

**Table 3.** Comparison between Final Loop Unrolling Factors and MAXLIVE

We made also statistics in order to check the timing of our approach. The results shows that on average the execution time of loop unrolling minimisation in the meeting graph is about $5$ microseconds. The maximum run-time is about $600$ microseconds. These timing observations have been conducted on a regular Linux workstation (Intel core Duo 2.4 GHZ).

The following section explains when should either use the meeting graph framework or SIRA framework.

### 5.2 Comparison between the Meeting Graph Framework and SIRA

Our loop unrolling minimisation method is independent of the technique used for periodic register allocation. Consequently, it can be performed either before software pipelining (where the method is implemented inside the SIRA framework as in [1]) or after software pipelining (where the method is implemented inside LoRA as described in this article).

In order to compare the final loop unrolling in both LoRA (MG) and SIRA, we conducted other experiments on larger applications from both high performance and embedded benchmarks: SPEC2000, SPEC2006, MEDIABENCH and FFMPEG. The number of experimented loops is 9027. We consider a machine with a bounded number of architectural registers $R_{\text{arch}}$. We varied $R_{\text{arch}}$ from 16 to 256.

The experiments show that final loop unrolling degrees computed by LoRA (MG) are lower than those computed by SIRA. In addition, the minimal loop unrolling degree for 75% of SIRA optimised loop is less or equal to 7 meanwhile LoRA does not unroll (unroll degree equal to 1).

We highlight in Table 4 some of the different results. We report the arithmetic mean of final loop unrolling and the maximum final loop unrolling. It shows that in each configuration, the average of minimal loop unrolling degree obtained thanks to our method is small when using MG compared with the average of final loop unrolling in SIRA. We also show that the maximum final loop unrolling degrees are low in MG compared to those in SIRA except in the case of FFMPEG. In the first line of Tab 4, we see that the value 30 exceeds MAXLIVE $+ 1$, while our method should results in an unrolling factor equal to at most $MAXLIVE + 1$, if enough remaining registers exist. This extreme case is due here to the fact that there is no register left to apply our loop unrolling minimisation method.

The choice between the two techniques depends if the loop is already software pipelined or not. If periodic register allocation should be done for any reason before software pipelining then SIRA is more appropriate; otherwise LoRA followed by loop unrolling minimisation provides lower loop unrolling degrees.

| $R_{\text{arch}}$ | Benchmarks | Average Final Loop Unrolling | | Maximum Final Loop Unroll | |
|---|---|---|---|---|---|
| | | MG | SIRA | MG | SIRA |
| 16 | FFMPEG | 1.127 | 2.479 | 30 | 28 |
| | MEDIABENCH | 1.175 | 2.782 | 12 | 26 |
| | SPEC2000 | 1.113 | 2.629 | 9 | 28 |
| | SPEC2006 | 1.085 | 2.758 | 9 | 16 |
| 32 | FFMPEG | 1.219 | 3.662 | 9 | 57 |
| | MEDIABENCH | 1.185 | 3.032 | 9 | 84 |
| | SPEC2000 | 1.118 | 2.823 | 9 | 28 |
| | SPEC2006 | 1.09 | 2.966 | 9 | 26 |
| 64 | FFMPEG | 1.3 | 6.476 | 9 | 72 |
| | MEDIABENCH | 1.426 | 3.225 | 63 | 84 |
| | SPEC2000 | 1.119 | 2.881 | 9 | 45 |
| | SPEC2006 | 1.09 | 3.001 | 9 | 26 |
| 128 | FFMPEG | 1.345 | 9.651 | 9 | 88 |
| | MEDIABENCH | 1.215 | 3.338 | 14 | 84 |
| | SPEC2000 | 1.119 | 2.916 | 9 | 45 |
| | SPEC2006 | 1.09 | 3.063 | 9 | 275 |
| 256 | FFMPEG | 1.345 | 9.733 | 9 | 88 |
| | MEDIABENCH | 1.214 | 3.384 | 14 | 84 |
| | SPEC2000 | 1.119 | 2.946 | 9 | 45 |
| | SPEC2006 | 1.09 | 3.256 | 9 | 27 |

**Table 4.** Optimised Loop Unrolling Factors: MG vs. SIRA (Large Benchmarks)

### 5.3 Comparison between the Meeting Graph Framework and MVE

Comparing MG and modulo variable expansion (MVE) is an old problem already explained in the literature. While MVE produces less unrolling in practice (in almost all experiments, MVE does not unroll the loop), MVE does not have a mathematical guarantee that a periodic register allocation with MAXLIVE registers is possible, and no upper bound is known. In practice, this means MVE may introduce spill code even if MAXLIVE is lower than or equal to the number of architectural registers. In some compiler construction contexts, such uncertainty is not acceptable. In our experiments, this problem occurs in 86 loops of FFMPEG, 100 loops of MEDIABENCH, 240 loops of SPEC2000 and 111 loops of SPEC2006. In contrast to MVE, MG followed by loop unrolling minimisation has the formal guarantee that at most $R_{arch}$ registers are allocated. Experimentally, this solution is good since no unrolling is required in 75% of the loops. In some extreme cases where the unrolling degree is still prohibitive with MG, the compiler can choose to use MVE with a risk of spilling, or even not apply software pipelining at all.

## 6 Conclusion

Contrary to maximal variable expansion [7, 9], periodic register allocation for software pipelined loops requires exactly MAXLIVE registers, as proved for the Meeting Graph [5]. This graph describes how to find a periodic register allocation with MAXLIVE registers if we sufficiently unroll the pipelined loop. Although MG improves significantly loop register allocation, the loop unrolling factor is equal to the least common multiple of the weight of the different reuse circuits which can be high in the practice.

This paper proposes to minimise the loop unrolling degree computed by MG by adapting the approach described in [1]. We explained how to adapt loop unrolling minimisation in the MG framework. Considering fixed circular lifetime intervals allows to have lower loop unrolling factors, while the algorithmic complexity of the optimisation method is greatly reduced compared to [1]. We showed experimental results on a large set of benchmark loops (FFMPEG, MEDIABENCH, SPEC2000, SPEC2006): in concrete cases the minimal loop unrolling degree for $75\%$ of scheduled loops is equal to 1 (i.e. no unroll), while it is equal to 7 when the SWP schedule is not fixed.

As a side-result of this work, we notice that our loop unrolling minimisation method is independent of the technique used for periodic register allocation. Consequently, loop unrolling minimisation [1] can be performed before or after any periodic register allocation technique and the final loop unrolling seems to be a satisfactory solution to generate code after periodic register allocation.

## 7 Acknowledgments

This research result was supported by the EU Network of Excellence HIPEAC. I would like to thank David Gregg and all research group in Trinity College (Dublin) for their

warm welcome and valuable advice. This work has also been supported by the MOP-UCE project (ANR number 05-JCJC-0039). This work was partly supported by the HiPEAC IST-217068 and ACOTES IST-34869 european projects.

## References

1. 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.
2. Sylvain Lelait Christine Eisenbeis. Lora, a package for loop optimal register allocation. Technical report, INRIA, France, June 1999.
3. 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.
4. 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.
5. 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.
6. 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.
7. P. Faraboschi J. A. Fisher and C. Young. *Embedded Computing: a VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann Publishers, 2005.
8. M. Jourdan J. Wang, C. Eisenbeis and B. Su. Decomposed software pipelining: a new perspective and a new approach. *International Journal Parallel Programming*, 22(3):351–373, 1994.
9. M. Lam. Software pipelining: an effective scheduling technique for vliw machines. *SIGPLAN Not.*, 23(7):318–328, 1988.
10. Sylvain lelait. *Contribution á l'allocation de registres dans les boucles*. PhD thesis, Université d'Orléans, January 1996.
11. 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.
12. 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.
13. Bruno Marmol Sylvain Lelait. Allocation cyclique de registres et déroulage de boucles. *Technique et Science Informatiques*, 16(5):583–608, May 1997.
14. Sid-Ahmed-Ali Touati and C. Eisenbeis. Early Periodic Register Allocation on ILP Processors. *Parallel Processing Letters*, 14(2):287–313, 2004.