Minimal Unroll Factor for Code Generation of Software Pipelining

Mounira Bachir, Sid Touati, Brault Frédéric, David Gregg, Albert Cohen

Minimal Unroll Factor for Code Generation of Software Pipelining

Mounira BACHIR, Sid-Ahmed-Ali TOUATI*, Frederic BRAULT, David GREGG, Albert COHEN

June 18, 2012

Abstract

We address the problem of generating compact code from software pipelined loops. Although software pipelining is a powerful technique to extract fine-grain parallelism, it generates lifetime intervals spanning multiple loop iterations. These intervals require periodic register allocation (also called variable expansion), which in turn yields a code generation challenge. We are looking for the minimal unrolling factor enabling the periodic register allocation of software pipelined kernels. This challenge is generally addressed through one of: (1) hardware support in the form of rotating register files, which solve the unrolling problem but are expensive in hardware; (2) register renaming by inserting register `move`s, which increase the number of operations in the loop, and may damage the schedule of the software pipeline and reduce throughput; (3) post-pass loop unrolling that does not compromise throughput but often leads to impractical code growth. The latter approach relies on the proof that MAXLIVE registers (maximal number of values simultaneously alive) are sufficient for periodic register allocation [10, 13]. However, the best existing heuristic for controlling this code growth — modulo variable expansion [16] — may not apply the correct amount of loop unrolling to guarantee that MAXLIVE registers are enough, which may result in register spills [10].

This paper presents our research results on the open problem of minimal loop unrolling, allowing a software-only code generation that does not trade the optimality of the initiation interval (II) for the compactness of the generated code. Our novel idea is to use the remaining free registers after periodic register allocation to relax the constraints on register reuse.

The problem of minimal loop unrolling arises either before or after software pipelining, either with a single or with multiple register types (classes). We provide a formal problem definition for each scenario, and we propose and study a dedicated algorithm for each problem.

Our solutions are implemented within an industrial-strength compiler for a VLIW embedded processor from STMicroelectronics, and validated on multiple benchmarks suites.

Keywords: Periodic register allocation, software pipelining, code generation, instruction level parallelism, embedded systems, compilation.

1 Introduction

Most high performance numerical applications exhibit intensive computations in loops. Software Pipelining (SWP) is an important instruction scheduling technique for improving the execution rate of inner loops. It combines multiple iterations of the loop body into a compact pipelined kernel to facilitate the exploitation of instruction level parallelism (ILP) [18, 16, 19]. The number of cycles between two successive iterations of the kernel loop is called the *initiation interval*.

When a loop is software pipelined, live ranges of variables may extend beyond a single iteration of the loop. As a result, multiple live ranges of the same variable may be in flight at any program point. One may not use regular register allocation algorithms because these different live range instances would create self-interferences in the interference graph [10, 12, 16]. In compiler construction, when no hardware support is available, kernel loop unrolling avoids introducing unnecessary move and spill operations by duplicating the kernel loop body a sufficient number of times

*Professor, University of Nice Sophia-Antipolis. Email: Sid.Touati@inria.fr

to remove live range's self-interference. Computing an adequate unroll factor and allocating registers to the separated live range instances is called *periodic register allocation*.

In this research we are interested in the minimal loop unrolling factor which allows a periodic register allocation for software pipelined loops (without inserting spill or move operations). Having a minimal unroll factor reduces code size, which is an important performance measure for embedded systems because they have a limited memory size. On larger machines, such as desktop, server and supercomputers, total memory size is typically much less limited, but code size is nonetheless important for I-cache performance. In addition to minimal unroll factors, it is necessary that the code generation scheme for periodic register allocation does not generate additional spill; the number of registers required must not exceed MAXLIVE (the number of values simultaneously alive). Spill code can increase the initiation interval and thus reduce performance in the following ways:

1. Adding spill code may increase the length of data dependence chains, which may increase the achievable initiation interval.
2. Spill code consumes execution resources which restrains the opportunity for achieving a high degree of instruction-level parallelism.
3. Memory requests consume more power than accessing the same values in registers.
4. Memory operations (except with scratch-pad local memories) have unknown static latencies. Compilers usually assume short latencies for memory operations, despite the fact that the memory access may miss the cache. Without good estimates of the performance of a piece of code, the compiler may be guided to bad optimisation decisions

When the schedule of a pipelined loop is known, there are a number of known methods for computing unroll factors and performing periodic register allocation, but none of them is fully satisfactory:

- Modulo Variable Expansion (MVE) [12, 16] computes a minimal unroll factor but may introduce spill code because it does not provide an upper bound on register usage.
- Hendren's heuristic [13] computes a sufficient unroll factor to avoid spilling, but with no guarantee in terms of minimal register usage or unrolling degree.
- The meeting graph framework [6] which guarantees that the unroll factor will be sufficient to minimise register usage (reaching MAXLIVE), but not that the unroll factor will itself be minimal.

In addition, periodic register allocation can be performed before SWP or after SWP, depending on the compiler construction strategy, as shown in Figure 1. Our article will not debate the best phase order; each compiler has its own characteristic and design implications. Instead we wish to improve the state of the art in loop unrolling minimisation for both possible phase orderings. If periodic register allocation is done before SWP as in Figure 1 (a), the instruction schedule is not fixed, and none of the above periodic register allocation techniques apply.

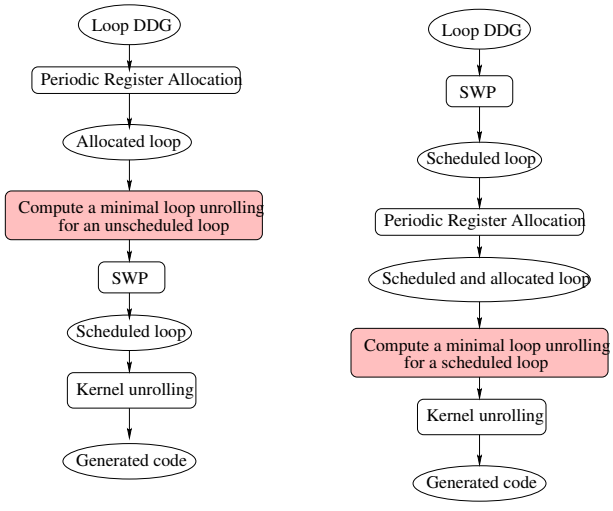
Contributions. This article advances the state of the art in the following directions:

1. We improve the meeting graph method, achieving *significantly smaller unroll factors* while *preserving an optimal register usage* on already scheduled loops. The key idea of our method is to exploit unused registers beyond the minimal number required for periodic register allocation (MAXLIVE [15])
2. The existing work in the field of kernel unrolling for periodic register allocation deals with already scheduled loops [6, 13, 12, 16]. As mentioned earlier, we also wish to handle not-yet-scheduled loops, on which the cyclic lifetime intervals are not known by the compiler. This article proposes a method for minimal kernel unrolling when SWP has not yet been carried out, by *computing a minimal unroll factor that is valid for the family of all valid cyclic schedules* of the data dependence graph (DDG). On the other hand, if register allocation is performed after SWP as in Figure 1 (b), the instruction schedule is fixed and hence the cyclic lifetime intervals and MAXLIVE are known.

3. We also extend the model of periodic register allocation to handle processor architectures with multiple register types (a.k.a. classes). On such architectures, state-of-the-art algorithms [10, 24] compute the *sufficient unrolling degree*, i.e., the unrolling degree that should be applied to a loop so that it is always possible to allocate the variables of each register type with a minimal number of registers. This article demonstrates that minimising the unroll factor on each register type separately does not define a global minimal unroll factor, and we provide an appropriate problem definition and an algorithmic solution in this context.

4. We contribute to the enlightenment of a poorly understood dilemma in back-end compiler construction. First, as mentioned earlier and as shown in Figure 1, we offer the compiler designer more choices to control the register pressure and the unroll factor for periodic register allocation at different epochs of the compilation flow. Second, we greatly simplify the phase ordering problem induced by the interplay of modulo scheduling, periodic register allocation, and post-pass unrolling. We achieve this by providing strong guarantees, not only in terms of register usage (the absence of spills induced by insufficient unrolling), but also in terms of reduction of the unroll factor.

5. Our methods are implemented within an industrial-strength compiler for STMicroelectronics' ST2xx VLIW embedded processor family. Our experiments on multiple benchmarks suites LAO, FFMPEG, MEDIABENCH, SPEC CPU2000, and SPEC CPU2006, are unprecedented in scale. They demonstrate the maturity of the techniques and contribute valuable empirical data never published in research papers on software pipelining and periodic register allocation. They also demonstrate the effectiveness of the proposed unroll degree minimisation, both in terms of code size and in terms of initiation intervals (throughput), along with satisfactory compilation times. Better, our techniques outperform the existing compiler for the ST2xx processor family which allocates registers after software pipelining and unrolls using MVE. Our techniques generate less spill code, fewer move operations, and yield a lower initiation interval on average, with a satisfactory code size (loops fitting within the instruction cache). These experiments are also teachful in their more negative results. As expected, they show that achieving strong guarantees on spill-free periodic register allocation yields generally higher unroll factors than heuristics providing no such guarantees like MVE [12, 16]. It was more unexpected (and disappointing) to observe that the initiation intervals achieved with MVE remain generally excellent, despite the presence of spills and a higher number of move operations. This can be explained by the presence of numerous empty slots in the cyclic schedules, where spills and move operations can be inserted, and by the rare occurrence of these spurious operations on the critical path of the dependence graph.



(a) Case of Unscheduled Loops

(b) Case of Scheduled Loops

Figure 1: Minimal unroll factor computation depending on phase ordering

Outline. The paper is organised as follows. Section 2 describes existing research results that are necessary to understand the rest of this article. Section 3 formalises the problem of minimising the loop unrolling degree in the presence of multiple register types when the loop is unscheduled. For clarity, we start the explanation of our loop-unrolling minimisation algorithm in Section 4 with the case of a single register type. Then, Section 5 generalises the solution to multiple register types. When the loop is already scheduled, an adapted algorithm is provided in Section 6 based on the meeting graph framework. Section 7 presents detailed experimental results on standard benchmark suites. In Section 8 we discuss related work on code generation for periodic register allocation, and we explain our contribution compared to the previous work. Finally, we conclude in Section 9.

2 Background

2.1 Loop Model and Software Pipelining

A *data dependence graph* (DDG) is a directed multigraph $G = (V, E)$ where V is a set of vertices representing instructions in a loop (also called statements, nodes, operations) and E is a set of edges representing data dependences (both flow and memory-based).

The modeled processor may have several register types: the set of available register *types* is denoted by $\mathcal{T}$. For instance, $\mathcal{T} = \{BR, GR, FP\}$ for branch, general purpose, and floating point registers respectively. Register types are sometimes called register *classes*. The number of available registers of type t is noted $\mathcal{R}^t$; it may be less than the total number of registers of type t as some architectural registers are often reserved for specific purposes.

For a given register type $t \in \mathcal{T}$, we define $V^{R,t} \subseteq V$ to be the set of statements $u \in V$ that produce values to be stored inside registers of type t. We u^t denotes the value of type t defined by an instruction $u \in V^{R,t}$. Indeed, a statement u may produce multiple values of distinct types, but we assume a given statement may not produce multiple values of the same type. The value may also be written u when the type is irrelevant or clear from the context.[1]

Concerning the set of edges E, we distinguish *flow* edges of type t — denoted $E^{R,t}$ — from the remaining edges. A flow edge $e = (u, v)$ of type t represents the producer-consumer relationship between the two statements u and v: u creates a value read by the statement v. Considering a register of type t, the set $E - E^{R,t}$ of non-flow edges are called memory-based edges.

Since we focus an loops and take into account loop-carried dependences, the DDG $G = (V, E)$ may be cyclic. Each edge $e \in E$ becomes labeled by a pair of values $(\delta(e), \lambda(e))$. $\delta : E \to \mathbb{Z}$ defines the latency of edges and $\lambda : E \to \mathbb{Z}$ defines the distance in terms of number of iterations. In order to exploit the parallelism between the instructions belonging to different loop iterations, we rely on periodic scheduling instead of acyclic scheduling, also called *software pipelining* (SWP).

SWP can be modeled by a periodic scheduling function $\sigma : V \to \mathbb{Z}$ and an *initiation interval* II. The operation u of the i^{th} loop iteration is noted $u(i)$, it is scheduled at date $\sigma(u) + i \times II$. Here, $\sigma(u)$ represents the execution (issue) date of $u(0)$, the clock cycle number of u for the first loop iteration. The schedule function σ is valid *if* it satisfies the periodic precedence constraints

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

SWP allows instructions to be scheduled independently of the original loop iterations barriers. The maximal number of values of type t simultaneously alive, noted MAXLIVE^t, defines the minimal number of registers required to allocate periodically the variables of the loop without introducing spill code. However, since some live ranges of variables span multiple iterations, special care must be taken when allocating registers and colouring an interference graph; this is the core focus of the paper and will be detailed later.

Let RC^t be the number of registers of type t to be allocated for the kernel of the pipelined loop. If the register allocation is optimal, we must have $RC^t =$ MAXLIVE^t for all register types $t \in \mathcal{T}$. We will see that there are only two theoretical frameworks that guarantee this optimality for SWP code generation. Other frameworks have $RC^t \geq$ MAXLIVE^t, with no guaranteed upper bound for RC^t.

[1] A few instruction sets allow this, and it can be modeled by node duplication: a node creating multiple results of the same type is split into multiple nodes of the same type.

2.2 Loop Unrolling after SWP with Modulo Variable Expansion

Code generation for SWP has to deal with many issues: prologue/epilogue codes, early exits from the loops, variables spanning multiple kernel iterations, etc. In our article, we focus on the last point: how can we generate a compact kernel for variables spanning multiples iterations when no hardware support exists in the underlying processor architecture? When no hardware support exists, and when prohibiting the insertion of additional move operations (i.e., no additional live range splitting), kernel loop unrolling is the only option. The resulted 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* (MVE) [16]. In fact, the major criterion of this method is to minimise the loop unrolling degree because the memory size of the i-WARP processor is low [16]. 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 of all live ranges LT_v of variables v defined in the pipelined kernel, by the initiation interval, i.e., $\lceil \frac{\max_v LT_v}{II} \rceil$.

MVE is easy to understand and implement, and it is practically effective in limiting code growth. This is why it has been adopted by several SWP frameworks [5, 15], and included in commercial compilers. The problem with MVE is that it does not guarantee a register allocation with MAXLIVE^t registers of type t, and in general it may lead to unnecessary spills breaking the benefits of software pipelining. A concrete example of this limitation is illustrated in Figures 2 and 3; we will use it as a running example in this section. Figure 2 is a SWP example with two variables v_1 and v_2. For simplicity, we consider here a single register type t. New values of v_1 and v_2 are created every iteration. For instance, the first value of v_1 is alive during the time interval $[0, 2]$, the other values (v_1' and v_1'') are created every multiple of II. This figure shows a concrete example with a SWP kernel having two variables v_1 and v_2 spanning multiple kernel iterations. In the SWP kernel, we can see that MAXLIVE$^t = 3$.

To generate a code for this SWP kernel, MVE unrolls it with a factor of $\left\lceil \frac{\max(LT_{v_1}, LT_{v_2})}{II} \right\rceil = \left\lceil \frac{\max(3,3)}{2} \right\rceil = 2$. Figure 3 illustrates the considered unrolled SWP kernel. The values created inside the SWP kernel are v_1, v_2, v_1', and v_2'. Because of the periodic nature of the SWP kernel, the variables v_1' and v_2' are alive as entry and exit values (see the figure of the lifetime intervals in the SWP kernel). Now, the interference graph of the SWP kernel is drawn, and we can see that it cannot be coloured with less than 4 colours (a maximal clique is $\{v_1, v_2, v_1', v_2'\}$). Consequently, it is impossible to generate a code with $RC^t = 3$ registers, except if we add extra copy operations in parallel. If inserting copy operations is not allowed or possible (no free slots, no explicit ILP), then we need $RC^t = 4$ registers to generate a correct code. This example gives a simple case where $RC^t >$ MAXLIVE^t, and it is not known if RC^t is bounded. As consequence, it is possible that the computed SWP schedule has MAXLIVE$^t \leq \mathcal{R}^t$, but the code generation performed with MVE requires $RC^t > \mathcal{R}^t$. This means that spill code has to be inserted even if MAXLIVE$^t \leq \mathcal{R}^t$, which is unfortunate and unsatisfactory.

Fortunately, an algorithm exists that achieves an allocation with a minimal number of registers equal to $RC^t =$ MAXLIVE^t [6, 10]. This algorithm exploints the meeting graph, introduced in the next section.

2.3 Meeting Graphs (MG)

The algorithm of Eisenbeis et al. [6, 10] can generate a periodic register allocation using MAXLIVE^t registers if the kernel is unrolled, thanks to a dedicated graph representation called the *meeting graph* (MG). It is a more accurate graph than the usual interference graph, as it holds information on the number of clock cycles of each live range and on the succession of the live ranges along the loop iterations. It allows us to compute an unrolling degree which enables an allocation of the loops with $RC^t =$ MAXLIVE^t registers.

Intuitively, the meeting graph is useful because it captures information about pairs of values where one value dies on the same clock cycle that another value becomes alive. If we try to allocate such values to the same register, then there is no dead time when the register contains a dead value. By identifying circuits in the meeting graph, we find sets of live values that can be allocated to one or more registers with no dead time.

Let us consider again the running example in Figure 2. The meeting graph that corresponds to that SWP is illustrated in Figure 4: a node is associated to every variable created in the SWP kernel. Hence we have two nodes v_1 and v_2. A node u is labeled with a weight $\omega(u)$ corresponding to the length og its respective live range, here 3 clock cycles for variables v_1 and v_2. There is an edge connecting a source node to a sink node *if and only if* the lifetime

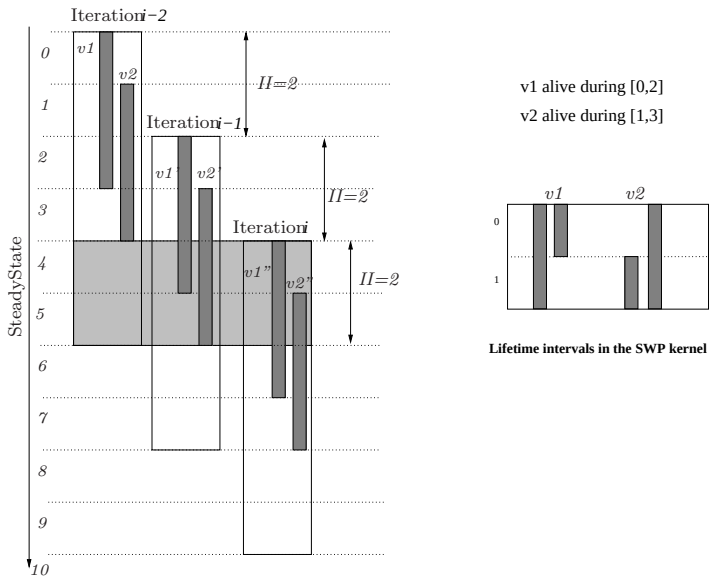


Figure 2: Example to highlight the short-comings of the MVE technique

interval of the first node ends when the sink one starts. By examining the SWP kernel in Figure 2, we see that the copies of v_1 end when those of v_2 start, and vice-versa. Consequently, in the MG of Figure 4, we have an edge from v_1 to v_2 and vice-versa to model an abstraction of the register reuse for this fixed SWP schedule.

Now, the question is: what is the benefit of such graph structure? Using this graph structure we are able to compute a provably sufficient unrolling factor to apply in order to achieve an optimal register allocation, that is $RC^t =$ MAXLIVE^t [6, 10].

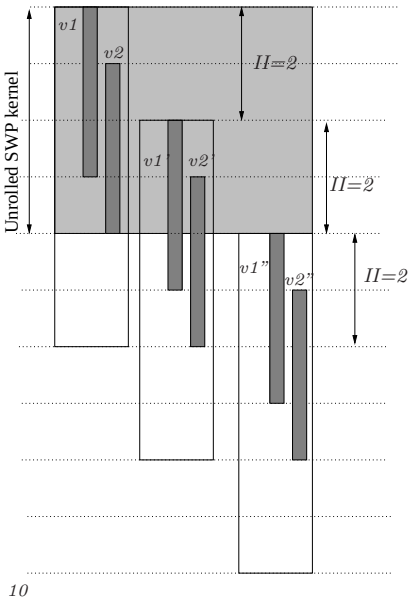
Let us consider the set of the strongly connected components (SCC) of the MG. In our simple example, there is a single SCC. The weight of every SCC numbered k is defined as $\mu_k = \frac{\sum_{v \in SCC_k} \omega(v)}{II}$. Note that one of the properties of the MG is that $\sum_{v \in SCC_k} \omega(v)$ is always a multiple of II, and $\frac{\sum_{\forall v} \omega(v)}{II} =$ MAXLIVE^t. In our simple example with a single SCC, its weight is equal to $\mu_1 = \frac{3+3}{2} = 3$.

Then, the *sufficient* unrolling factor computed using the MG is equal to $\alpha = lcm(\mu_1, ..., \mu_k)$, where lcm denotes the least common multiple [10]. It has been proved that if the SWP kernel is unrolled α times, then we can generate code with $RC^t =$ MAXLIVE^t registers. In the example illustrated in Figure 4, we have a single SCC so $\alpha = \mu_1 = 3$, which means that the kernel has to be unrolled with a factor equal to 3. The interference graph shows that three colours are sufficient, which allows us to generate correct code with only three registers, rather than the four required with modulo variable expansion (compared to Figure 3).

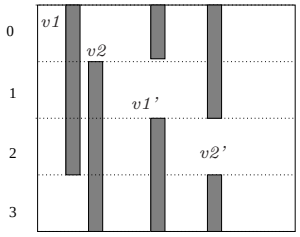
Without formally proving the correctness of the unrolling factor defined above (the interested reader is invited to study [6, 10]), the intuition behind the least common multiple (LCM) formula comes from the following fact: if we successfully generate code for a SCC by unrolling the kernel μ_i times, then we can generate a correct code for the same SCC by unrolling the kernel with any multiple of μ_i. Hence, if we are faced with a set of SCCs, it is sufficient to consider the LCM of the μ_i's to have a correct unrolling factor for all the SCCs.

In addition to the previous unroll factor formula, the MG also allows us to guarantee that MAXLIVE^t or MAXLIVE^t+ 1 are sufficient unrolling factors. In the example of Figure 4, we have the coincidence that $\alpha =$ MAXLIVE^t, but this is not always the case. Indeed, one of the purposes of MG is to have unrolling factors α lower than MAXLIVE^t. This objective is not always reachable if we want to have $RC^t =$ MAXLIVE^t, Eisenbeis et al. [10] try to reach it by decomposing the MG into a maximal number of elementary circuits. In practice, it turns out that α may be very high, reducing the practical benefit of register optimality $RC^t =$ MAXLIVE^t.

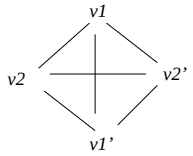
The next section recalls a theoretical framework that applies periodic register allocation before SWP, while allowing the computation of a sufficient unrolling degree for a complete set of possible SWP schedules.



MVE unrolls the SWP kernel with a factor of 2



Lifetime intervals in the SWP kernel



Interference graph of the SWP kernel

```
FOR i=0, i< N; i=i+2
  date 0: R0= v1(i)
  date 1: R1= v2(i)
  date 2: R2= v1'(i)
  date 3: R0= v2'(i)
ENDFOR
```

Impossible correct code with 3 registers

```
FOR i=0, i< N; i=i+2
  date 0: R0= v1(i)
  date 1: R1= v2(i)
  date 2: R2= v1'(i)
  date 3: R3= v2'(i)
ENDFOR
```

Correct code with 4 registers

```
FOR i=0, i< N; i=i+2
  date 0: R0= v1(i)  || R1 = R0
  date 1: R1= v2(i)  || R2 = R1
  date 2: R2= v1'(i) || R0=R2
  date 3: R0= v2'(i)
ENDFOR
```

Correct code with 3 registers and parallel copy operations

Figure 3: SWP kernel unrolled with MVE

2.4 SIRA and Reuse Graphs

Reuse graphs are a generalisation of previously work by de Werra et al. and Hendren et al. [6, 13]. They are used inside a framework called SIRA [23, 24]. Unlike the previous approaches for periodic register allocation, reuse graphs are used before software pipelining to generate a move-free or a spill-free periodic register allocation in the presence of multiple register types. Reuse graphs provide a formalised approach to generating code which requires neither register spills nor move operations. Of course, it is not always possible to avoid spill code, some DDGs are complex enough to always require spilling, the SIRA framework is able to detect such situations before SWP.

A simple way to explain SIRA is to provide an example. All the theory has already been presented in Touati and Eisenbeis [24], and we recently showed that optimising the register requirement for multiple register types in one go is a better approach than optimising for every register type separately [23]. Figure 5(a) provides an initial DDG with two register types t_1 and t_2. Statements producing results of type t_1 are in dashed circles, and those of type t_2 are in bold circles. Statement u_1 writes two results of distinct types. Flow dependence through registers of type t_1 are in dashed edges, and those of type t_2 are in bold edges.

Each edge e in the DDG is labeled with the pair of values $(\delta(e), \lambda(e))$. Now, the question is how to compute a periodic register allocation for the loop in Figure 5(a) without hurting the instruction level parallelism if possible.

Periodic register constraints are modeled using *reuse graphs*. We associate a reuse graph $G^{\text{reuse},t}$ to each register type t, see Figure 5(b). The reuse graph is computed by the SIRA framework, Figure 5(b) is one of the examples that

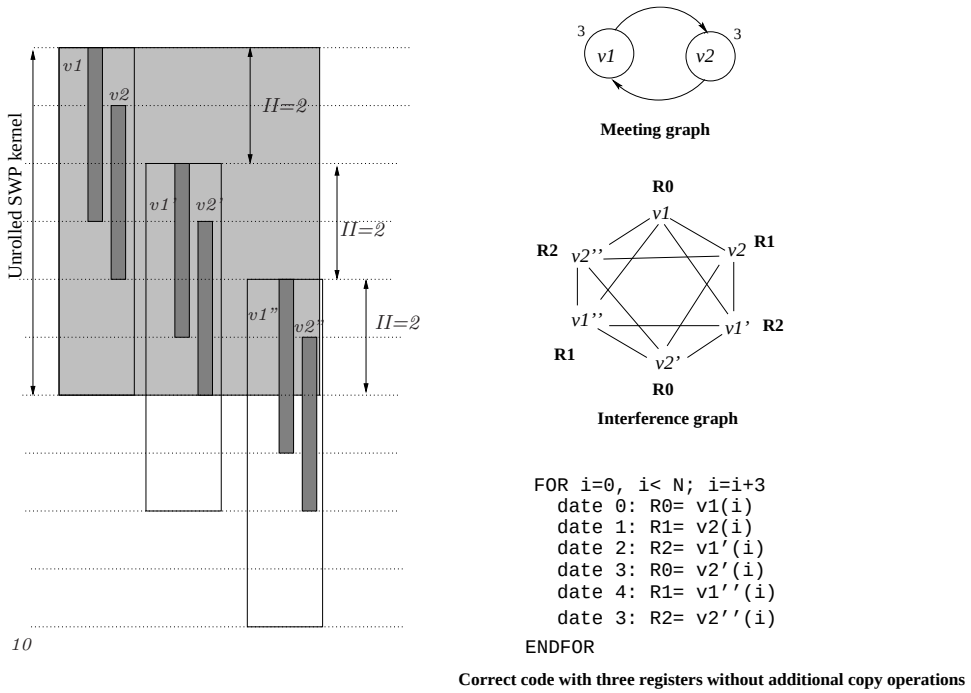
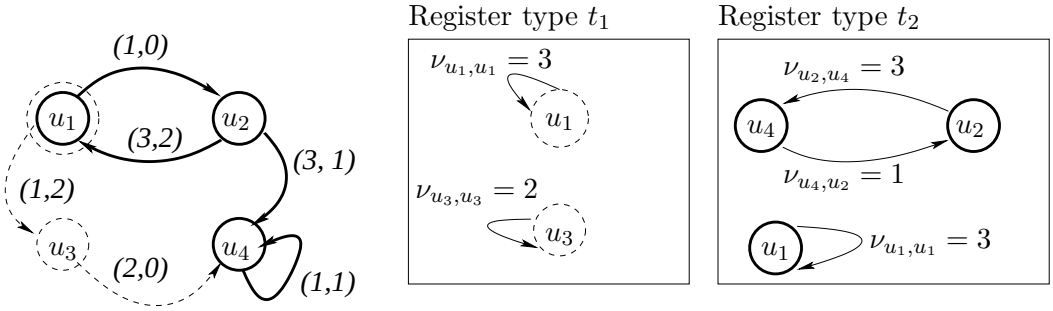


Figure 4: Example to explain the optimality of the meeting graph technique



(a) Initial DDG (b) Reuse Graphs for Register Types t1 and t2

Figure 5: Example for SIRA and reuse graphs

SIRA may produce. Note that the reuse graph is not unique, other valid reuse graphs may exist.

A reuse graph $G^{\text{reuse},t} = (V^{R,t}, E^{\text{reuse},t})$ contains $V^{R,t}$, i.e., only the nodes writing to registers of type t. These nodes are connected by *reuse edges*. For instance, in G^{reuse,t_2} of Figure 5(b), the set of reuse edges is $E^{\text{reuse},t_2} = \{(u_2,u_4),(u_4,u_2),(u_1,u_1)\}$. Also, $E^{\text{reuse},t_1} = \{(u_1,u_3),(u_3,u_1)\}$. Each reuse edge $e_r = (u,v)$ is labeled by an integral distance $\nu^t(e_r)$, that we call *reuse distance*. The existence of a reuse edge $e_r = (u,v)$ of distance $\nu^t(e_r)$ means that $u(i)$ (iteration i of u) and $u(i+\nu^t(e_r))$ (iteration $i+\nu^t(e_r)$ of v) *share the same destination register* of type t. Hence, reuse graphs allow to completely define a periodic register allocation for a given loop. In the example of Figure 5(b) and for register type t_2, we have $\nu^{t_2}((u_2,u_4)) = 3$ and $\nu^{t_2}((u_4,u_2)) = 1$.

Let C be a circuit in the reuse graph $G^{\text{reuse},t}$ of type t; we call C a *reuse circuit*. We note $\mu^t(C) = \sum_{e_r \in C} \nu^t(e_r)$ the weight of the reuse circuit C. The following corollary provides a sufficient unrolling factor for all register types.

Corollary 1 *[24] Let $G = (V, E)$ be a loop DDG with a set of register types $\mathcal{T}$. Each register type $t \in \mathcal{T}$ is associated with a valid reuse graph $G^{reuse,t} = (V^{R,t}, E^{reuse,t})$. The loop can be allocated with $RC^t = \sum_{e_r \in E^{reuse,t}} \nu^t(e_r)$ registers for each type t if we unroll it α times, where:*

$$\alpha = lcm(\alpha_1^t, \cdots, \alpha_n^t)$$

α^{t_i} is the unrolling degree of the reuse graph of type t_i, defined as

$$\alpha^t = lcm(\mu^t(C_1), \cdots, \mu^t(C_n))$$

The above corollary seems to be close to the meeting graph result. This is not exactly true, since here we are generalizing the meeting graph result to unscheduled loops in the presence of multiple registers types. Unlike the meeting graph, the above defined unrolling factor is valid for a whole set of SWP schedules, not for a fixed one. In addition, the reuse graph allows us to guarantee that prior to software pipelining $RC^t = \sum_{e_r \in E^{\text{reuse},t}} \nu^t(e_r) \leq \mathcal{R}^t$ for any register type, while maintaining instruction level parallelism if possible (by taking care not to increase the critical circuit of the loop, known as the MII_{dep}).

Note that when compilation time matters, we can avoid unrolling the loop before the SWP step. This avoids increasing the DDG size, which would result in significantly more work for the scheduler. Because we allocate registers directly on the DDG by inserting loop carried anti-dependencies, the DDG can be scheduled without unrolling it. In other words, loop unrolling can be applied at the code generation step (after SWP) in order to apply the register allocation computed before scheduling.

Example 1 *Let consider as illustration the example of Figure 5. Here $\alpha_{t_1} = lcm(3, 2) = 6$ and $\alpha_{t_2} = lcm(3+1, 3) = 12$. That is, the register type t_1 requires that we unroll the loop 6 times if we want to consume $RC^{t_1} = 3 + 2 = 5$ registers of type t_1. At this compilation step, SWP has not been carried out but SIRA guarantees that the computed unroll factor and register count are valid for any subsequent SWP. As an illustration, a valid sequential trace for the for the register type t_1 is given in Listing 1 (we do not show the trace for register type t_2, and we omit the prologue/epilogue of the trace).*

The reader may check that we have used 5 registers of type t_1. According to the reuse graph, every pair of statements $(u_1(i), u_1(i+3))$ uses exactly the same destination register, because there is a reuse edge (u_1, u_1) with a reuse distance $\nu^{t_1}(u_1, u_1) = 3$; Every pair of statements $(u_3(i), u_3(i+2))$ uses the same destination register too, because there is a reuse edge (u_3, u_3) with a reuse distance $\nu^{t_1}(u_3, u_3) = 2$. We can check in the generated code that the reuse circuit (u_1, u_1), which contains a single reuse edge in this example, uses three registers (R_1, R_2 and R_3); The reuse circuit (u_3, u_3) uses two registers (R_4 and R_5).

Regarding the register type t_2, it requires an unrolling factor equal to 12 if we want to consume $RC^{t_2} = 3+1+3 = 7$ registers of type t_2. Consequently, a common valid unroll factor for both the register types t_1 and t_2 is equal to $\alpha = lcm(6, 12) = 12$. For space reasons, we do not show the full code generation for the loop in Figure 5 with an unrolling factor of 12. However, later in Section 5, we will show how we will minimise the unrolling degree to get a reasonable value equal to 4, it will be then possible to write a reasonably short code for the example.

The main advantage of the meeting graph and reuse graph approaches over MVE is their ability to guarantee spill-free and move-free code generation, before or after SWP. However, they have an important drawback, which is that the unroll factor may be very large. The next section defines the problem of unroll degree minimisation for unscheduled loops. Later, we will extend the problem to scheduled loops.

3 Problem Description of Unroll Factor Minimisation for Unscheduled Loops

The reuse graph method, which guarantees a register allocation with exactly MAXLIVE registers, may result in a large unrolling factor. However, there may be additional unused registers:: each register type t may have some remaining registers $R^t = \mathcal{R}^t - RC^t$ (where $\mathcal{R}^t$ is the number of available architectural registers of type t). We have developed a method to use any remaining registers to reduce the unrolling factor. This method is applied after the periodic register allocation step performed by the SIRA framework. This post-pass minimisation consists in adding zero or more unused registers to each reuse circuit in order to minimise the least common multiple of the size of the circuits (denoted α^*). This idea is described in the next problem.

Listing 1: Example of a sequential kernel code generation for the register type t_1

```
FOR i=0, i<N, i=i+6
 u_1(i)  : R1 = ...
 u_2(i)  :
 u_3(i)  : R4 = R2...
 u_4(i)  : ...= R4...

 u_1(i+1): R2 = ...
 u_2(i+1):
 u_3(i+1): R5 = R3...
 u_4(i+1): ...= R5...

 u_1(i+2): R3 = ...
 u_2(i+2):
 u_3(i+2): R4 = R1...
 u_4(i+2): ...= R4...

 u_1(i+3): R1 = ...
 u_2(i+3):
 u_3(i+3): R5 = R2...
 u_4(i+4): ...= R5...

 u_1(i+4): R2 = ...
 u_2(i+4):
 u_3(i+4): R4 = R3...
 u_4(i+4): ...= R4...

 u_1(i+5): R3 = ...
 u_2(i+5):
 u_3(i+5): R5 = R1...
 u_4(i+5): ...= R5...
ENDFOR
```

Problem 1 (Loop Unroll Minimisation (LUM)) *Let α be the initial loop unrolling degree and let $\mathcal{T} = \{t_1, \ldots, t_n\}$ be the set of register types. For each register type $t_j \in \mathcal{T}$, let $R^{t_j} \in \mathbb{N}$ be the number of remaining registers after a periodic register allocation for this register type. Let k_j be the number of reuse circuits of type t_j. We note $\mu_{i,t_j} \in \mathbb{N}$ as the weight of the i^{th} reuse circuit of the register type t_j. For each reuse circuit i and each register type t_j, we must compute the added registers r_{i,t_j} such that we find a new periodic register allocation with a minimal loop unrolling degree. This is described by the following constraints:*

1. $\alpha^* = lcm(lcm(\mu_{1,t_1} + r_{1,t_1}, \ldots, \mu_{k_1,t_1} + r_{k_1,t_1}), \ldots, lcm(\mu_{1,t_n} + r_{1,t_n}, \ldots, \mu_{k_n,t_n} + r_{k_n,t_n}))$ *is minimal (optimality constraint).*
2. $\forall t_j \in \mathcal{T}, \sum_{i=1}^{k_j} r_{i,t_j} \leq R^{t_j}$ *(validity constraints)*

That is, this formal problem describes the idea of increasing the number of allocated registers without exceeding the number of available ones (to guarantee the absence of spilling), with the goal of minimising the global unroll factor. Increasing the number of allocated registers is done by increasing the weights of the reuse circuit. If a reuse circuit consists of multiple edges, then increasing the weigth of any edge inside this reuse circuit is a valid solution. This solution is valid as proved in the next lemma. Intuively, this lemma says that if we succeed in building a periodic register allocation with RC_1^t registers of type t, then we can build a periodic register allocation with RC_2^t registers of type t, where $RC_1^t \leq RC_2^t \leq \mathcal{R}^t$

Lemma 1 *Let $G = (V, E)$ be a loop data dependence graph. Let $G^{reuse,}$ be a valid reuse graph of each register type $t \in \mathcal{T}$ associated with the loop G. Let $\mathcal{R}$ be the number of available registers of type t. Let (u_i^t, u_j^t) a single arbitrary reuse arc in $G^{reuse,t}$ with its associated reuse distance $\nu_{i,j}^t \in \mathbb{Z}$. Then:*

$$\nu_{i,j}^t \leq \mathcal{R}^t \Longrightarrow \forall x \in [0, \mathcal{R}^t - \nu_{i,j}^t], \nu_{i,j}^t + x \text{ is a valid reuse distance for the reuse arc } (u_i^t, u_j^t)$$

Proof:

The proof comes from the formal linear constraints defining the validity of ν variables. These constraints have been defined in [24, 23], that we summarise here. The proof is organised in three subsections. Subsection 3.1 recalls the integer linear program that defines the validity constraints of a reuse graph. Then, Subsections 3.2 and 3.2 prove that $\nu_{i,j}^t + x$ does not violate these validity conditions.

Our processor model considers both UAL (Unit Assumed Latencies) and NUAL (Non UAL) semantics [21]. Given a register type $t \in \mathcal{T}$, we model possible delays when reading from or writing into registers of type t. We define two delay functions $\delta_{r,t} : V \mapsto \mathbb{N}$ and $\delta_{w,t} : V^{R,t} \mapsto \mathbb{N}$.[2] These delay functions model NUAL semantics. Thus, the statement u reads from a register $\delta_{r,t}(u)$ clock cycles after the schedule date of u. Also, u writes into a register $\delta_{w,t}(u)$ clock cycles after the schedule date of u. In UAL, the code semantics is sequential, these delays are not visible to the compiler, so we have $\delta_{w,t} = \delta_{r,t} = 0$.

In this proof, we consider $G = (V, E)$ to be a loop DDG, and $G^{\text{reuse},t}$ an associated reuse graph to be computed using integer linear constraints as defined below.

3.1 Linear Constraints for ν Variables

This section briefly recalls the construction of the reuse graph. A recent description for multiple register types can be found in Touati et al. [23, 24]. We say that a value is killed when all its consumers have already read it, and consequently, it does not have to occupy a register anymore. Any last reading instruction is called its killer. If the DDG is already scheduled, then it is easy to compute the killing instruction and the killing date of each value. However, if the DDG is not already scheduled as in our case, then the killing instruction is not known. For each value v, we create a virtual killer K, adding edges from all the consumers of v to the killer node K, and we also introduce reuse edges from K to all subsequent iterations of the consumers of v.

[2] w is a write to a register of type t, hence the restriction to $V^{R,t}$ for $\delta_{w,t}$.

3.1.1 Basic variables

- We define a schedule variable $\sigma_i \in \mathbb{N}$ for each statement $u_i \in V$, including σ_{K_i} for each killer K_i. We consider L as a maximal value for σ variables, L is sufficiently large (for instance $L = \sum_{e \in E} \delta(e)$).
  Since our instruction scheduling function is a modulo schedule with initiation interval II, we only consider the integer execution date of the first operation occurrence $\sigma_i = \sigma(u_i(0))$ and the execution date of any other occurrence $u_i(k)$ becomes equal to $\sigma(u_i(k)) = \sigma_i + k \times II$.
- A binary variable $\theta^t_{i,j}$ for each pair of statements $(u^t_i, u^t_j) \in V^{R,t} \times V^{R,t}$. It is set to 1 if and only if (u^t_i, u^t_j) is a reuse arc;
- A reuse distance $\nu^t_{i,j} \in \mathbb{N}$ for each pair of statements $(u^t_i, u^t_j) \in V^{R,t} \times V^{R,t}$ that is a reuse arc.

3.1.2 Linear constraints

- **Data dependences**
  The schedule must at least satisfy the precedence constraints defined by the DDG.

$$\forall e = (u_i, u_j) \in E : \sigma_j - \sigma_i \geq \delta(e) - II \times \lambda(e) \tag{1}$$

- **Flow dependences**
  Each flow dependence $e = (u^t_i, u^t_j) \in E^{R,t}, \forall t \in \mathcal{T}$ means that the statement occurrence $u_j(k + \lambda(e))$ reads the data produced by $u_i(k)$ at time $\sigma_j + \delta_{r,t}(u_j) + (\lambda(e) + k) \times II$. Then, we must schedule the killer K_i of the statement u_i after all u_i's consumers. $\forall t \in \mathcal{T}, \forall u_i \in V^{R,t}$, $\forall u_j \in \{v \mid (u_i, v) \in E^{R,t}\} | e = (u_i, u_j) \in E^{R,t}$:

$$\sigma_{K_i} \geq \sigma_j + \delta_{r,t}(u_j) + II \times \lambda(e) \tag{2}$$

- **Storage dependences**
  There is a storage dependence between K_i and u^t_j if (u^t_i, u^t_j) is a reuse arc of type t. $\forall t \in \mathcal{T}, \forall (u_i, u_j) \in V^{R,t} \times V^{R,t}$:

$$\theta^t_{i,j} = 1 \Longrightarrow \sigma_{K_i} - \delta_{w,t}(u_j) \leq \sigma_j + II \times \nu^t_{i,j}$$

  This involvement can result in the following inequality: $\forall t \in \mathcal{T}, \forall (u^t_i, u^t_j) \in V^{R,t} \times V^{R,t}$,

$$\sigma_j - \sigma_{K_i} + II \times \nu^t_{i,j} + M_1 \times (1 - \theta^t_{i,j}) \geq -\delta_{w,t}(u_j) \tag{3}$$

  where M_1 is an arbitrarily large constant.
  If there is no register reuse between two statements u_i and u_j, then $\theta^t_{i,j} = 0$ and the storage dependence distance $\nu^t_{i,j}$ must be set to 0 in order to not be accumulated in the objective function.

$$\forall (u_i, u_j) \in V^{R,t} \times V^{R,t} : \ \nu^t_{i,j} \leq M_2 \times \theta^t_{i,j} \tag{4}$$

  where M_2 is an arbitrarily large constant.
- **Reuse relations**
  The reuse relation of type t must be a bijection from $V^{R,t}$ to $V^{R,t}$. A register of type t can be reused by one statement and a statement can reuse one released register:

$$\forall u^t_i \in V^{R,t} : \sum_{u^t_j \in V^{R,t}} \theta^t_{i,j} = 1 \tag{5}$$

$$\forall u^t_j \in V^{R,t} : \sum_{u^t_i \in V^{R,t}} \theta^t_{i,j} = 1 \tag{6}$$

  From the above integer linear program, we see that the ν variables are constrained by Inequality 4 and 3. The two following subsections treat them separately.

3.2 Valid Upper Bounds of ν Variables (Inequality 4)

From the above linear constraints, we prove here that increasing the values of ν variables does not violate the upper bounds of ν variables.

The constraints which define an upper bound for ν variables are defined by Inequality 4:

$$\forall (u_i, u_j) \in V^{R,t} \times V^{R,t} : \ \nu^t_{i,j} \leq M_2 \times \theta^t_{i,j}$$

We have two cases regarding $\theta^t_{i,j} \in \{0, 1\}$ value:

1. $\theta^t_{i,j} = 0 \Longrightarrow \nu^t_{i,j} \leq 0$. Since $\nu^t_{i,j} \in \mathbb{N} \Longrightarrow \nu^t_{i,j} = 0$. This means that if (u^t_i, u^t_j) is not a reuse arc, then its reuse distance is equal to zero.
2. $\theta^t_{i,j} = 1 \Longrightarrow \nu^t_{i,j} \leq M_2$. Since M_2 is arbitrarily large, this means that $\nu^t_{i,j}$ can arbitrarily verify this condition. In other words, this means that if (u^t_i, u^t_j) is a reuse arc, then its reuse distance can be arbitrarily large too.

We can decide for a proper finite value for M_2 that verifies Inequality 4:

By assumption $\nu^t_{i,j} \leq \mathcal{R}^t \Longrightarrow \nu^t_{i,j} \leq \max_{t \in \mathcal{T}} \mathcal{R}^t$. We can deduce a finite value for M_2 as $M_2 = \max_{t \in \mathcal{T}} \mathcal{R}^t$. The formal result of our lemma is directly deduced from the assumption that $\nu^t_{i,j} \leq \mathcal{R}^t$, and by setting the natural number $x = \mathcal{R}^t - \nu^t_{i,j}$, we have obviously $\nu^t_{i,j} + x \leq M_2$.

The next section checks that $\nu^t_{i,j} + x$ also verifies the other linear constraints.

3.3 Storage Constraints on ν^t Variables (Inequality 3)

The other constraints on ν variables are those of Inequality 3:

$$\sigma_j - \sigma_{K_i} + II \times \nu^t_{i,j} + M_1 \times (1 - \theta^t_{i,j}) \geq -\delta_{w,t}(u_j) \Longrightarrow \nu^t_{i,j} \geq \frac{-\sigma_j + \sigma_{K_i} - M_1 \times (1 - \theta^t_{i,j}) - \delta_{w,t}(u_j)}{II}$$

Since $x = \mathcal{R}^t - \nu^t_{i,j} \geq 0 \Longrightarrow \nu^t_{i,j} + x \geq \frac{-\sigma_j + \sigma_{K_i} - M_1 \times (1 - \theta^t_{i,j}) - \delta_{w,t}(u_j}{II}$

This means that the value $\nu^t_{i,j} + x$ verifies the storage constraints too. Consequently, it constitutes a valid reuse distance.

⌟

For clarity, we first present a solution to Problem 1 in the case of a single register type.

4 Algorithmic Solution for Unroll Factor Minimisation: Single Register Type

In this section, we solve the problem of minimal unroll degree in the case of a single register type, based on reuse graphs (unscheduled loops). When we consider a single register type, then we have a single reuse graph for the considered register type. The formula for computing the unrolling degree becomes equal to a single LCM of the weights of the reuse circuits of the implicit register type. By replacing the notations of $\mu_{i,t}$ ($r_{i,t}$ and R^t resp.) by μ_i (r_i and R resp.), Problem 1 amounts to the following one.

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

To our knowledge, Problem 2 has no simple, closed-form solution, and its algorithmic complexity is still an open problem[3].

Before stating our solution for Problem 2, we propose to find a solution for a sub-problem that we call *Fixed Loop Unrolling Problem*. The solution of this sub-problem constitutes the basis of the solution of Problem 2. The *Fixed Loop Unrolling Problem* proposes to find, for a fixed unrolling degree β, the number of registers that should be added to each circuit to ensure that the size of each circuit is a divisor of β. That is, we find the number of registers added to each circuit $r_1, ..., r_k$ such that $\sum_{i=1}^{k} r_i \leq R$ and β is a common multiple of the different updated weights $\mu_1 + r_1, ... + \mu_k + r_k$. A formal description is given in the next section.

4.1 Fixed Loop Unrolling Problem

We formulate the *Fixed Loop Unrolling Problem* as follow:

Problem 3 (Fixed Loop Unrolling 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. *β is the common multiple of the news circuits weights $\mu_1 + r_1, \ldots, \mu_k + r_k$*

To improve readability, we use CM to denote common multiple.

Before describing our solution for Problem 3, we state Lemma 2 and Lemma 3 that we need to use afterwards.

Lemma 2 *Let us note some properties of the* Fixed Loop Unrolling Problem*:*

1. $\beta \geq \max_i \mu_i \Longrightarrow \exists (r_1, \ldots, r_k) \in \mathbb{N}^k$ *such that:* $CM(\mu_1 + r_1, \ldots, \mu_k + r_k) = \beta$
2. *Let $r_1, \ldots, r_k$ be the solution of Problem 3 such that $\sum_{i=1}^{k} r_i$ is minimal. If $\sum_{i=1}^{k} r_i > R$ then* Fixed Loop Unrolling Problem *cannot be solved.*

Proof:

- The first issue can be proved by finding an obvious list of added registers

  $$(r_1, \ldots, r_k) \in \mathbb{N}^k \text{ such that: } CM(\mu_1 + r_1, \ldots, \mu_k + r_k) = \beta$$

  Let us assume that $\beta \geq \max_i \mu_i$. If we put $\forall i = 1, k: \ r_i = \beta - \mu_i$ then

  $$\forall i = 1, k: \ r_i \geq 0 \text{ and } CM(\mu_1 + r_1, \ldots, \mu_k + r_k) = \beta \text{ because } \forall i = 1, k: \ \mu_i + r_i = \beta$$

- The second issue can be proved by contradiction.

[3]Indeed, a similar reduced problem exists in cryptography theory: Given two natural numbers a, b, compute $x \leq R^t \in \mathbb{N}$ such that $gcd(a, b+x)$ is maximal (gcd denotes the greatest common divisor, GCD). This GCD maximisation problem is defined for two integers only, it is equivalent to minimising the LCM of two integers because $lcm(a, b) = \frac{a \times b}{gcd(a,b)}$. The GCD maximisation problem of two integers is known to be equivalent to the integer factorisation problem: the decision problem of integer factorisation has unknown complexity class till now. It is currently solved with approximate methods devoted to very large numbers [14]. Problem 2 is a generalisation of the GCD maximisation problem. The heuristic presented in [14] is not appropriate in our case because: 1) The problem tackled in [14] deals with two integers only, that we cannot generalise to minimise the LCM to multiple integers because $LCM(x_0, \cdots, x_k) \neq \frac{x_0 \times \cdots \times x_k}{gcd(x_0, \cdots, x_k)}$ for $k > 2$. 2) We deal with multiple small numbers (in practice, $R \leq 128$), allowing to design optimal methods efficient in practice instead of heuristics.

Let us assume that we find another solution $r'_1, \ldots, r'_k$ for the problem 3 such that $\sum_{i=1}^{k} r'_i \leq R$. However, in the second part of Lemma 2, we find a list of added register $r_1, \ldots, r_k$ such that $\sum_{i=1}^{k} r_i$ is minimal. Consequently,

$$\sum_{i=1}^{k} r_i \leq \sum_{i=1}^{k} r'_i \text{ (by assumption, it is minimal) } \Longrightarrow R < \sum_{i=1}^{k} r_i \leq \sum_{i=1}^{k} r'_i$$

which constitues a contradiction with $\sum_{i=1}^{k} r'_i \leq R$.
Thus, if there exists a list of added registers which fulfill the constraints of the problem 3 such that $\sum_{i=1}^{k} r_i$ minimal $> R$ then *Fixed Loop Unrolling Problem* cannot be resolved.

⌟

Lemma 3 *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 3, then we have the following results:*

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

Proof:

The first issue can be proved as follows:

β is the common multiple (CM) of the news circuits weights $\mu_1 + r_1, \ldots, \mu_k + r_k$

$$\Rightarrow \forall i = 1, k : \ \beta \geq \mu_i + r_i \tag{7}$$

From (7) we have:

$$\forall i = 1, k : \ \beta \geq \mu_i + r_i \Rightarrow \forall i = 1, k : \ \beta - \mu_i \geq r_i \tag{8}$$

From (8) 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 common multiple (CM) of a set of positive integers. Hence, we have:

β is the common multiple (CM) of the news circuits weights $\mu_1 + r_1, \ldots, \mu_k + r_k$

$$\Rightarrow \forall i = 1, k : \ \mu_i + r_i \text{ is a divisor of } \beta \tag{9}$$

From (9) we have:

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

$$\Rightarrow \ \forall \, i = 1, k \ : \ \exists \, d_i \ \in \ D_\beta \mid \mu_i + r_i = d_i \tag{10}$$

From (10) 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 Lemma 3 is proved.

⌟

After proving Lemma 3 and by using Lemma 2, we describe our solution for the *Fixed Loop Unrolling Problem* in the next section.

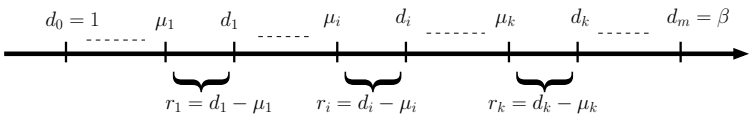


Figure 6: Graphical solution for the fixed loop unrolling problem

4.2 Solution for the Fixed Loop Unrolling 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 registers. 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 $d_i = \min\{d \in D_\beta \mid d \geq \mu_i\}$. If we denote CM as common multiple then the two following implications are true:*

1. $\beta = CM(\mu_1 + r_1, \ldots, \mu_k + r_k) \wedge \sum_{i=1}^{k} r_i \leq R \Rightarrow$ *we find a solution for Problem 3;*
2. $\beta = CM(\mu_1 + r_1, \ldots, \mu_k + r_k) \wedge \sum_{i=1}^{k} r_i > R \Rightarrow$ *Problem 3 has no solution.*

Proof:

In Lemma 3, we have proved that:

$$\beta = CM(\mu_1 + r_1, \ldots, \mu_k + r_k) \Longrightarrow \forall\, i = 1, k :\ \exists\, d_i \ \in\ D_\beta \mid r_i = d_i - \mu_i \ \wedge\ d_i \geq \mu_i \tag{11}$$

From Equation (11) we have:

$$r_i \text{ is minimal } \Rightarrow d_i \text{ is the smallest divisor of } \beta \ \geq\ \mu_i \tag{12}$$

From Equation (12) a list of the added registers $r_1, \ldots, r_k$ with $\sum_{i=1}^{k} r_i$ is minimal can be defined as follows:

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

According to Lemma 2, 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 the *Fixed Loop Unrolling Problem*. Otherwise, if $\sum_{i=1}^{k} r_i$ is minimal $> R$ then we are sure that there are no solution for Problem 3.

⌟

Figure 6 represents a graphical solution for the *Fixed Loop Unrolling Problem*. We assume that the different weights and the different divisors of β are sorted on the same axis in an ascending order.

Algorithm 1 implements our solution for the *Fixed Loop Unrolling Problem*. This algorithm tries to divide R the remaining registers among the circuits to achieve a fixed common multiple of k integers (the different weights of reuse circuits μ_i). It checks if β can become the new loop unrolling degree. For this purpose, Algorithm 1 uses Algorithm 2 that returns the smallest divisor just after an integer value. Algorithm 1 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:

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

The maximal algorithmic complexity of the Fixed Loop Unrolling Problem is then dominated by the while loop: $\mathcal{O}((\mathcal{R}^t)^2)$.

Algorithm 1 Fixed loop unrolling problem

Require: k: the number of reuse circuits; μ_i: the different weights of reuse circuits; $\mathcal{R}^t$: the number of architectural registers, and β: the loop unrolling degree
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
  $R = \mathcal{R}^t - \sum_{1 \leq i \leq k} \mu_i$ {the remaining register}
  sum $\leftarrow 0$
  success $\leftarrow$ **true** {defines if we find a valid solution for the different added registers}
  $i \leftarrow 1$ {represents the number of reuse circuits}
  $D \leftarrow$ DIVISORS$(\beta, \mathcal{R}^t)$ {calculate the sorted list of divisors of β that are $\leq \mathcal{R}^t$ including β}
  while $i \leq k \wedge success$ **do**
    $d_i \leftarrow$ DIV_NEAR(μ_i, D) {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
      $i \leftarrow i + 1$
    end if
  end while
  return $(r_1, \ldots, r_k)$, success

Algorithm 2 DIV_NEAR

Require: μ_i: the weight of the reuse circuits; $D = (d_1, \ldots, d_n)$: the n divisors of β sorted by ascending order
Ensure: d_i the smallest divisors of β greater or equal to μ_i
  $i \leftarrow 1$ {represents the index of the divisor of β}
  while $i \leq n$ **do**
    if $d_i \geq \mu_i$ **then**
      return (d_i)
    end if
    $i \leftarrow i + 1$
  end while

Algorithm 3 DIVISORS

Require: β: the loop unrolling degree; $\mathcal{R}^t$: the number of architectural registers
Ensure: D the list of the divisors of β that are $\leq \mathcal{R}^t$, including β
  bound $\leftarrow \min(\mathcal{R}^t, \beta/2)$
  $D \leftarrow \{1\}$
  for $d = 2$ to bound **do**
    if $\beta \mod d = 0$ **then**
      $D \leftarrow D \cup \{d\}$ {Keep the list ordered in ascending order}
    end if
  end for
  $D = D \cup \{\beta\}$
  return (D)

Analysis of the Complexity of Algorithm 1

- Regarding the DIVISORS algorithm:
  - The maximal number of iterations is bound $\leq \mathcal{R}^t$.
  - Inserting an element inside the list costs at most $\log(\mathcal{R}^t)$.
  - The maximal complexity of DIVISORS algorithm is $O(\mathcal{R}^t \times \log(\mathcal{R}^t))$.
- Regarding the DIV_NEAR algorithm: $O(n) \leq O(\mathcal{R}^t)$.

- Regarding the Fixed Loop Unrolling Problem algorithm:
  - Calling DIVISORS costs $O(\mathcal{R}^t \times \log(\mathcal{R}^t))$.
  - The while loop iterates at most $k \leq \mathcal{R}^t$ times.
  - At each iteration, calling DIV_NEAR costs $O(\mathcal{R}^t)$.
  - The maximal algorithmic complexity of the Fixed Loop Unrolling Problem is then dominated by the while loop: $O((\mathcal{R}^t)^2)$.

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

4.3 Solution for LCM-MIN Problem

For the solution of the *LCM-MIN Problem* (Problem 2) we use the solution of the *Fixed Loop Unrolling Problem* and the result of Lemma 3. According to Lemma 3, the solution space S for α^* (the solution of *LCM-MIN Problem*) is bounded by α, the initial unroll factor.

$$\begin{cases} \forall\, i = 1, k:\ \alpha^* \geq \mu_i \text{ (From Lemma 3)} \\ \alpha^* \leq \alpha \end{cases} \Rightarrow \max_{1 \leq i \leq k} \mu_i \leq \alpha^* \leq \alpha$$

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 solution space S can be defined 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 values of α^*. The minimal α^*, that is the solution for Problem 2, is defined as follows:

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

Figure 7 portrays all values of the set S as a partial lattice. An arrow between two nodes means that the value in the first node is less than the value of the second node: $a \rightarrow b \Longrightarrow a < b$. The value μ_k represents the value of the reuse circuit number k. Because we assumed that μ values are sorted in ascending order, μ_k is the highest weight 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 the lines of the lattice. Each line describes a set of multiples. For example, the line j describes a set of multiples $S_j = \{\beta | \exists r_k, 0 \leq r_k \leq R^t, \beta = j \times (\mu_k + r_k) \wedge \beta \leq \alpha\}$

In order to find α^*, the minimal unroll factor, our solution consists in checking if each node of S can be a solution for the *Fixed Loop Unrolling 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 7 an efficient way to find the minimal α^*. We proceed line by line in the figure. 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 lattice S).
2. Else we have found a unroll factor less than the original α. We note this new value α' and we try once again to minimise it until we find the minimal (the first case). The search space shrinks: $S' = \{\beta \in \mathbb{N} | \forall r_k = 0..R :$ β is multiple of $(\mu_k + r_k) \wedge (j+1) \times \mu_k \leq \beta \leq \alpha'\}$.

Algorithm 4 implements our solution for the *LCM-MIN Problem*. This algorithm 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 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.

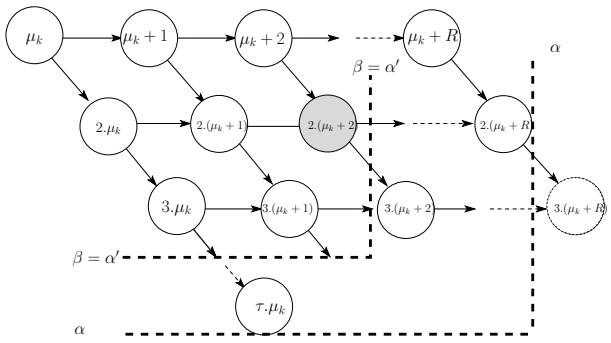


Figure 7: How to traverse the lattice S

Listing 2: Example of a sequential kernel code generation for the register type t_1

```
FOR i=0, i<N, i=i+3
 u_1(i)   : R1 = ...
 u_2(i)   :
 u_3(i)   : R4 = R3...
 u_4(i)   : ...= R4...

 u_1(i+1): R2 = ...
 u_2(i+1):
 u_3(i+1): R5 = R1...
 u_4(i+1): ...= R5...

 u_1(i+2): R3 = ...
 u_2(i+2):
 u_3(i+2): R6 = R2...
 u_4(i+2): ...= R6...
ENDFOR
```

Algorithmic Complexity Analysis of Algorithm 4 In the worst case, Algorithm 1 is processed on all the nodes of the set S in Figure 7. The set S has $\frac{R \times \alpha}{\mu_k}$ nodes ($\mu_k = \max \mu_i$ and $\alpha = lcm(\mu_1, ..., \mu_k)$). We know that $1 \leq \mu_k \leq \mathcal{R}^t$. Consequently, the size of the set S is less or equal to $R^t \times \alpha$. On each node, we process Algorithm 1. Hence, the maximal algorithmic complexity of is $O(R \times \alpha \times (\mathcal{R}^t)^2) = O(R \times (\mathcal{R}^t)^2 \times lcm(\mu_1, ..., \mu_k))$.

Example 2 *Let us come back to the example of Figure 5 on page 8, but we focus on the single register type t_1, and we neglect the other register type t_2. There are initially two reuse circuits with two costs $\mu_1 = \nu^{t_1}(u_1, u_1) = 3$, and $\mu_2 = \nu^{t_1}(u_3, u_3) = 2$. Thus, as shown in Ex. 1, the initial unroll factor is equal to $lcm(3, 2) = 6$. It is easy to see that if we increment the reuse distance $\nu^{t_1}(u_3, u_3)$ from 2 to 3, then the cost of the reuse circuit (u_3, u_3) becomes equal to 3, and hence the unroll factor becomes equal to $lcm(3, 3) = 3$ instead of 6. The new number of allocated registers becomes equal to $RC^{t_1} = 3 + 3$ instead of 5 initially. A valid kernel code generation with 6 registers of type t_1 and an unroll factor equal to 3 is given in Listing 2.*

Example 3 *Let us consider a more complex example with a set of five reuse circuits with the respective weights: $\mu_1 = 3, \mu_2 = 4, \mu_3 = 5, \mu_4 = 7, \mu_5 = 8$. The initial number of allocated registers is equal to $RC = 3 + 4 + 5 + 7 + 8 = 27$. The loop unrolling degree α is their least common multiple ($\alpha = lcm(3, 4, 5, 7, 8) = 840$). Let us assume that we have $\mathcal{R}^t = 32$ architectural registers in the target processor. So hence we have $R = 32 - 27 = 5$ remaining registers. By applying Algorithm 4, we find that the minimal numbers of registers added to each reuse*

Algorithm 4 LCM-MIN algorithm

Require: k: the number of reuse circuits; μ_i: the weights of the reuse circuits; $\mathcal{R}^t$: the number of architectural registers; α: the initial loop unrolling degree

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

```
R = R^t − Σ_{1≤i≤k} μ_i {The remaining registers}
α* ← μ_k {minimal value of loop unrolling α*}
if α = α* ∨ R = 0 then
  if R = 0 then
    α* ← α {α nothing can be done, 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
    {Traversing the set S until we find the minimal loop unrolling factor}
    success ← Fixed_Unrolling_Problem(k, μ_i, R^t, β) {Apply for each node the Algorithm 1}
    if ¬success then
      if r_k < R then
        r_k++ {we go to the next node on the same line}
      else
        r_k ← 0 {we go to the first node of the next line}
        j++
      end if
      β ← j × (μ_k + r_k) {compute the new value of the potential new unrolling factor β}
      if β > α ∧ j < τ then
        {ignore the dashed node}
        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 { β may be the minimal loop unrolling degree}
      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
```

circuits are $r_1 = 1, r_2 = 0, r_3 = 3, r_4 = 1, r_5 = 0$. *The new reuse circuits' weights become* $\mu_1 = 3 + 1 = 4$, $\mu_2 = 4 + 0 = 4$, $\mu_3 = 5 + 3 = 8$,$\mu_4 = 7 + 1 = 8$, $\mu_5 + 0 = 8$. *The new number of allocated registers become equal to* $4 + 4 + 8 + 8 + 8 = 32$. *The new unroll factor becomes equal to* $\alpha^* = lcm(4, 4, 8, 8, 8) = 8$, *which means that we reduced it by a* $ratio = \frac{\alpha}{\alpha^*} = 105$.

The next section extends the algorithm of unroll factor minimisation to the case of multiple register types.

5 Unroll Factor Minimisation in the Presence of Multiple Register Types

In the presence of multiple register types, minimising the loop unrolling degree of each type separately does not lead to the minimal loop unrolling degree for the whole loop, as illustrated in the following example.

Example 4 *Let us return to the example in Figure 5 on page 8. We want to minimise the loop unrolling degree of the initial reuse graph in Figure 5(b), where two register types t_1, t_2 are considered. The initial kernel loop unrolling degree $\alpha = 12$ is the LCM of $\alpha^{t_1} = 6$ and $\alpha^{t_2} = 12$ which are respectively the LCM of the different reuse circuits weights for each register type. In this configuration, let us assume that we have $\mathcal{R}^{t_j} = 8$ available architectural registers in the processor for each register type t_j. Hence we have $R^{t_1} = 8 - 5 = 3$ (resp $R^{t_2} = 1$) remaining registers for register type t_1 (resp t_2). By applying the loop unrolling minimisation for each register type separately as studied in Section 4, the minimal loop unrolling degree for each register type becomes: ${\alpha^{t_1}}^* = 3$ for register type t_1 and ${\alpha^{t_2}}^* = 4$ for register type t_2, see Figure 8(a). However, the global kernel loop unrolling degree is not minimal $\alpha' = lcm({\alpha^{t_1}}^*, {\alpha^{t_2}}^*) = 12$. The minimal global kernel loop unrolling degree is computed below.*

In Figure 8(b), we provide a solution where the minimal loop unrolling degree is $\alpha^* = 4 < \alpha'$. The unroll factor of t_1 is equal to 4, which is not its minimal value (equal to 3 as shown above). However, the global unroll factor that satisfies both t_1 and t_2 is minimal and equal to 4. The minimal number of registers added to each reuse circuit of each type are: $r_{1,t_1} = 1$, $r_{2,t_1} = 0$, $r_{1,t_2} = 1$, $r_{2,t_2} = 0$. Note that r_{i,t_j} is the number of registers added to the i^{th} reuse circuit of the type t_j. Our method explained in the following sections guarantees that the new number of allocated registers will not exceed the number of architectural registers for each register type t_j.

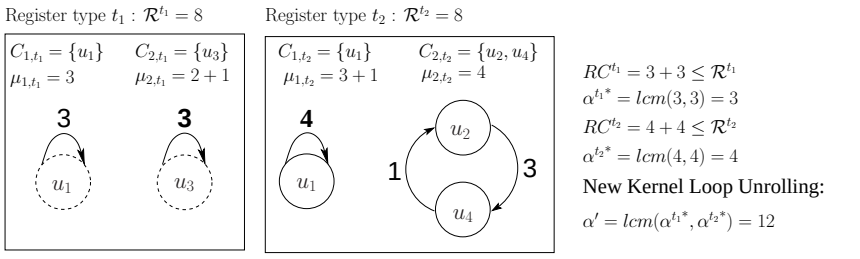
Now let us examine an example of a valid code generation associated to the reuse graphs of Figure 8(b), even though at this stage of compilation, the loop is not yet scheduled. Listing 3 shows a kernel code generation for the register type t_1 only: registers of type t_1 are named with the prefix `R`. The number of allocated registers is $RC^{t_1} = 4 + 2 = 6$ and the unroll factor is equal to 4. Listing 4 shows a kernel code generation for the register type t_2 only: registers of type t_2 are named with the prefix `S`. The number of allocated registers is $RC^{t_2} = 4 + (1 + 3) = 8$ and the unroll factor is equal to 4. The kernel code generation that is correct for both t_1 and t_2 is given in Listing 5 and the unroll factor is minimal and equal to 4: note that the statement u_1 has two destination registers of two distinct types, as previously illustrated in the DDG of Figure 5(a). As can be seen, the initial unroll factor was equal to 12, as computed in Example 1 in page 9, we minimise it here to 4, which is the optimal value. We also guarantee that the number of extra used registers does not exceed the number of remaining registers.

Listing 3: Kernel code generation for register type t_1

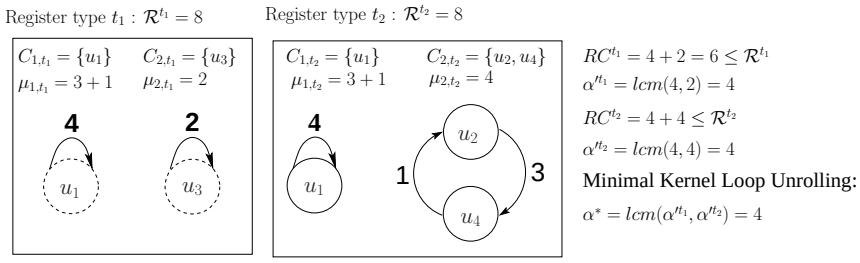
```
FOR i=0, i<N, i=i+4
 u_1(i): R1 =
 u_2(i):
 u_3(i): R5 =  R4 +...
 u_4(i): ...= R5 +...

 u_1(i+1): R2 =
 u_2(i+1):
 u_3(i+1): R6 = R1 +...
 u_4(i+1):...=  R6 +...

 u_1(i+2): R3 =
 u_2(i+2):
 u_3(i+2): R5= R2 + ...
 u_4(i+2):...= R5 +...

 u_1(i+3): R4=
 u_2(i+3):
 u_3(i+3): R6 = R3 +...
 u_4(i+4): ...= R6 +...
ENDFOR
```

Listing 4: Kernel code generation for register type t_2

```
FOR i=0, i<N, i=i+4
 u_1(i): S1 =  S7 +...
 u_2(i): S5 = S1 +...
 u_3(i):
 u_4(i): S6 = S8 + S5

 u_1(i+1): S2 = S8 +...
 u_2(i+1): S6 = S2 +...
 u_3(i+1):
 u_4(i+1): S7= S5 + S6

 u_1(i+2): S3 = S5 +...
 u_2(i+2): S7 = S3 +...
 u_3(i+2):
 u_4(i+2): S8 = S6 + S7

 u_1(i+3): S4 = S6 +...
 u_2(i+3): S8 = S4 +...
 u_3(i+3):
 u_4(i+4): S5 = S7 + S8
ENDFOR
```



(a) Minimising Loop Unrolling for Each Register Type Separately



(b) Minimising Loop Unrolling for all Register Types Conjointly

Figure 8: Modifying reuse graphs to minimise loop unrolling factor

Listing 5: Kernel code generation for the two register types conjointly

```
FOR i=0, i<N, i=i+4
 u_1(i): R1,S1 =  S7
 u_2(i): S5 = S1
 u_3(i): R5 =  R4
 u_4(i): S6 = S8 + S5 + R5

 u_1(i+1): R2,S2 = S8
 u_2(i+1): S6 = S2
 u_3(i+1): R6 = R1
 u_4(i+1): S7= S5 + S6 + R6

 u_1(i+2): R3,S3 = S5
 u_2(i+2): S7 = S3
 u_3(i+2): R5 =  R4
 u_4(i+2): S8 = S6 + S7 + R5

 u_1(i+3): R4,S4 = S6
 u_2(i+3): S8 = S4
 u_3(i+3): R6 = R3
 u_4(i+4): S5 = S7 + S8 + R6
ENDFOR
```

The following section defines the search space S for the minimal kernel loop unrolling α^*.

5.1 Search Space for Minimal Kernel Loop Unrolling

According to the properties of LCM and to the formulation of Problem 1, the search space S for the minimal kernel loop unrolling α^* is bounded. In fact, three cases arise:

Case 1: No remaining registers for all the different register types In this case, the initial loop unrolling degree cannot be minimised $\alpha^* = \alpha$.

Case 2: No remaining registers for some register types Assume that α^j is the loop unrolling degree for the register type $t_j \in \mathcal{T}$. In this way, $\alpha = lcm(\alpha^1, \ldots, \alpha^n)$. We define the subset $\mathcal{T}'$ which contains all the register types such that there are no remaining registers for these register types after periodic register allocation ($\mathcal{T}' \subset \mathcal{T}$ such that $\mathcal{T}' = \{t \in \mathcal{T} \mid R^t = 0\}$). If there are no registers left for these register types, we cannot minimise their loop unrolling degrees, see Section 4. Therefore, the minimal global loop unrolling degree $\alpha^* \geq \alpha^j \; \forall \, t_j \in \mathcal{T}'$. By considering $\alpha' = lcm_{t \in \mathcal{T}'}(\alpha^t)$, we have the following inequality:

$$\alpha' \leq \alpha^* \leq \alpha \tag{13}$$

In addition, from LCM properties:

$$\alpha^* \text{ is multiple of } \alpha' \tag{14}$$

From Equation 13 and Equation 14, the search space S is defined as follows:

$$S = \{\beta \in \mathbb{N} \mid \beta \text{ is multiple of } \alpha' \wedge \alpha' \leq \beta \leq \alpha\}$$

Here, each value β can be a potential final loop unrolling degree.

Case 3: All register types have some remaining registers From the associative property of LCM, we have:

$$\alpha^* = lcm(lcm(\mu_{1,t_1} + r_{1,t_1}, \ldots, \mu_{k_1,t_1} + r_{k_1,t_1}), \ldots, lcm(\mu_{1,t_n} + r_{1,t_n}, \ldots, \mu_{k_n,t_n} + r_{k_n,t_n}))$$

$$\Longrightarrow \alpha^* = lcm(\mu_{1,t_1} + r_{1,t_1}, \ldots, \mu_{k_1,t_1} + r_{k_1,t_1}, \ldots, \mu_{1,t_n} + r_{1,t_n}, \ldots, \mu_{k_n,t_n} + r_{k_n,t_n})$$

The final loop unrolling factor α^* is a multiple of each updated reuse circuit weight ($\mu_{i,t_j} + r_{i,t_j}$) with the number of additional registers (r_{i,t_j}) varied from 0 (no added register for this circuit) to R^{t_j} (all the remaining registers are added to this reuse circuit).

Furthermore, if we assume that μ_{k_n,t_n} is the maximum weight of all the different circuits for all register types ($\mu_{k_n,t_n} = \max\limits_{t_j}(\max\limits_{i} \mu_{i,t_j})$) then α^* is a multiple of this specific updated circuit (α^* is a multiple of $(\mu_{k_n,t_n} + r_{k_n,t_n})$ with $0 \leq r_{k_n,t_n} \leq R^{t_n}$). We notice here that any reuse circuit satisfies this later property, but it is preferable to consider the reuse circuit with a maximal weight because it decreases the cardinality of the search space S. Finally the search space S can be stated as follows:

$$S = \{\beta \in \mathbb{N} \mid \beta \text{ is multiple of } (\mu_{k_n,t_n} + r_{k_n,t_n}) \, , \forall r_{k_n,t_n} = 0, R^{t_n} \wedge \mu_{k_n,t_n} \leq \beta \leq \alpha\}$$

After describing the set S of all possible values of α^* (case 2 and case 3), the minimal kernel loop unrolling α^* is defined as follows:

$$\alpha^* = \min\{\beta \in S | \forall t_j \in \mathcal{T}, \exists (r_{1,t_j}, \ldots, r_{k_j,t_j}) \in \mathbb{N}^{k_j} \text{ such that:}$$

β is the Common Multiple (CM) of the following updated reuse circuits weights:

$$\mu_{1,t_1} + r_{1,t_1}, \ldots, \mu_{k_1,t_1} + r_{k_1,t_1}, \ldots, \mu_{1,t_j} + r_{1,t_j}, \ldots, \mu_{k_j,t_j} + r_{k_j,t_j}, \ldots, \mu_{1,t_n} + r_{1,t_n}, \ldots, \mu_{k_n,t_n} + r_{k_n,t_n}$$

$$\wedge \sum_{i=1}^{k_j} r_{i,t_j} \leq R^{t_j}\}$$

Another problem arises here: how to decide if the value β can be a potential new loop unrolling? Solving this problem is explained in the next section.

5.2 Generalisation of the Fixed Loop Unrolling Problem in the Presence of Multiple Register Types

Problem 4 (General Fixed Loop Unrolling) *Let $\beta \in S$ be a fixed loop unrolling degree and let $\mathcal{T} = \{t_1, \ldots, t_n\}$ be the set of register types. β can be a potential new loop unrolling if and only if we find for each register type $t_j \in \mathcal{T}$, a minimal distribution of the remaining registers R^{t_j} between its reuse circuits (μ_{i,t_j}) such that this new loop unrolling degree β satisfies the following constraints:*

1. $\beta = CM(\mu_{1,t_1} + r_{1,t_1}, \ldots, \mu_{k_1,t_1} + r_{k_1,t_1}, \ldots, \mu_{1,t_n} + r_{1,t_n}, \ldots, \mu_{k_n,t_n} + r_{k_n,t_n})$
2. $\forall t_j \in \mathcal{T} \ \sum_{i=1}^{k_j} r_{i,t_j} \leq R^{t_j}$ *: for each register type, the additional registers does not exceed the number of remaining registers*

In order to determine if β can be the new kernel loop unrolling, we propose to generalise the *Fixed Loop Unrolling Problem* solution to all register types. In fact, the different Constraints in Problem 4 are the generalisation of the *Fixed Loop Unrolling Problem* constraints which must be satisfied for all the register types.

In general, the *Fixed Loop Unrolling Problem* proposes to add to each reuse circuit μ_{i,t_j} of each register type t_j, a minimal number of registers r_{i,t_j} from the remaining R^{t_j} registers such that $\mu_{i,t_j} + r_{i,t_j}$ is the smallest divisor of the fixed loop unrolling β greater or equal to μ_{i,t_j}. In this way, if the additional registers, for each register type, do not exceed the number of remaining registers $\sum_{i=1}^{k_j} r_{i,t_j} \leq R^{t_j}$, then β can be the new loop unrolling degree.

By using the associative property of the common multiple, we have:

$$\beta = CM(\mu_{1,t_1} + r_{1,t_1}, \ldots, \mu_{k_1,t_1} + r_{k_1,t_1}, \ldots, \mu_{1,t_n} + r_{1,t_n}, \ldots, \mu_{k_n,t_n} + r_{k_n,t_n})$$
$$\Rightarrow \forall t_j \in \mathcal{T}\ ,\ \beta \text{ is a Common Multiple of } \mu_{1,t_j} + r_{1,t_j}, \ldots, \mu_{k_j,t_j} + r_{k_j,t_j}$$

Consequently, Algorithm 5 implements our solution for Problem 4 by reusing Algorithm 1 previously defined.

Algorithm 5 General fixed loop unrolling problem

Require: $\beta \in S$ the fixed loop unrolling, $\mathcal{T} = \{t_1, \ldots, t_n\}$ the set of register types. For each register type t_j, we require the number k_j of reuse circuits, the different weights of reuse circuits μ_{i,t_j} the remaining register R^{t_j} and its initial loop unrolling degree α_j

Ensure: The boolean $success$ and for each type t_j, the different added registers $r_{1,t_j}, \ldots, r_{k_j,t_j}$ with $\sum_{i=1}^{k_j} r_{i,t_j}$ $minimal$

```
success ← true {defines if β can be the new kernel loop unrolling}
j ← 1 {represents the type t_j of T}
calculate the different divisors of β
while j ≤ n ∧ success do
  if R^{t_j} = 0 then
    if β mod α_j ≠ 0 then
      success ← false {no optimisation for the type t_j, the new unrolling degree must be a multiple of α_j}
    end if
  else
    success ← Fixed_Unrolling_Problem(β, k_j, R^{t_j}, μ_{1,t_j}, ..., r_{1,t_j}, ...) {we don't calculate the different divisors of β
    inside the function}
  end if
  j ← j + 1
end while
```

The solution of the *General Fixed Loop Unrolling Problem* (Problem 4) constitutes the basis of the solution for *Loop Unrolling Minimisation Problem* (Problem 1) explained in the next section.

5.3 Algorithmic solution for the Loop Unrolling Minimisation (LUM, Problem 1)

In order to compute the minimal kernel loop unrolling α^*, our solution consists in checking if each value β in the search space S can be a solution for the *Fixed Loop Unrolling Problem*: it is guaranteed that the minimum of all these values is the minimal loop unrolling degree.

Instead of computing all values β of S which satisfy the *General Fixed Loop Unrolling Problem* and finally taking the minimal one, we describe in Figure 9 an efficient way to find the minimal α^* depending on the construction of the lattice S. Figure 9 also illustrates the different cases of the construction of the solution space S. 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 $a < b$. The absence of an arc between two nodes means that the order is unknown. The structure of the search space depends on the availability of the different types of registers :

- *Case 1 (no registers left for all the different register types):* no loop unroll minimisation is possible, $\alpha^* = \alpha$.
- *Case 2 (no registers left for some register types):* α^* is multiple of α', we apply Algorithm 5 to each node of Figure 9 until the predicate $success$ returned by this algorithm is **true** or until we reach the last node α
- *Case 3: some registers left for all the different register types:* we traverse the set S in the same way as described in Section 4. If we consider $\mu = \mu_{k_n,t_n}$ (maximum weight of all the different circuits for all register types) and $R = R^{t_n}$ (remaining registers for the register type t_n) then we traverse the set S by proceeding line by line. In each line, we apply Algorithm 5 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 β of the node i of the line j verifies the predicate ($success =$ **true**), then we have two cases:

  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 β (by construction of the set S).

  b) Otherwise we have found a new value of unrolling degree which is less than the original α. We note this new value α" 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 $(\mu + r) \wedge (j + 1) \times \mu \leq \beta \leq \alpha$"$\}$)

After describing our solution for the LUM problem in the case of unscheduled loops, the next section studies the same problem but in the context of scheduled loops.

6 Unroll Factor Reduction for Already Scheduled Loops

When the SWP is fixed, circular lifetime intervals are known, and can be modeled using the meeting graph, as referenced in Section 8.

If we base our unroll factor minimisation method on the meeting graph rather than on the reuse graph, we lose in terms of genericity of the unroll factor obtained, but we will see that we are able to reduce algorithmic complexity and the quality of the solution.

Figure 10 illustrates an example with a single register type. We want to reduce the loop unrolling degree of the five circuits of the meeting graph in Figure 10(b). The loop is already software pipelined with an $II = 5$ (the SWP is not drawn). According to the definition of the meeting graph, every node u corresponds to a variable (or to a dummy node to cover all the II period), and its weight $\omega(u)$ is simply its live range. For instance, $\omega(u_1 9) = 14$ means that it is alive during 14 clock cycles. As can be seen, this meeting graph has only one connected component, and MAXLIVE $= \frac{\sum_{\forall u} \omega(u)}{II} = 27$ (see Fig 10(a)). Then, the meeting graph is decomposed into elementary circuits (by following the different chords). Every elementary circuit C_i has a weight equal to $\mu_i = \frac{\sum_{u \in C_i} \omega(u)}{II}$. In the example of Fig 10(b), we have five elementary circuits with the following weights $\mu_1 = 3, \mu_2 = 4, \mu_3 = 5, \mu_4 = 7, \mu_5 = 8$. If we want to allocate exactly $RC =$ MAXLIVE $= 27$ registers, the kernel loop unrolling degree resulting from this decomposition is $\alpha = lcm(\mu_1, \mu_2, \mu_3, \mu_4, \mu_5) = 840$, the LCM of the weights of the different circuits. However, this unroll factor is very large and it is impractical to allow the loop to be unrolled 840 times. The meeting graph proposes

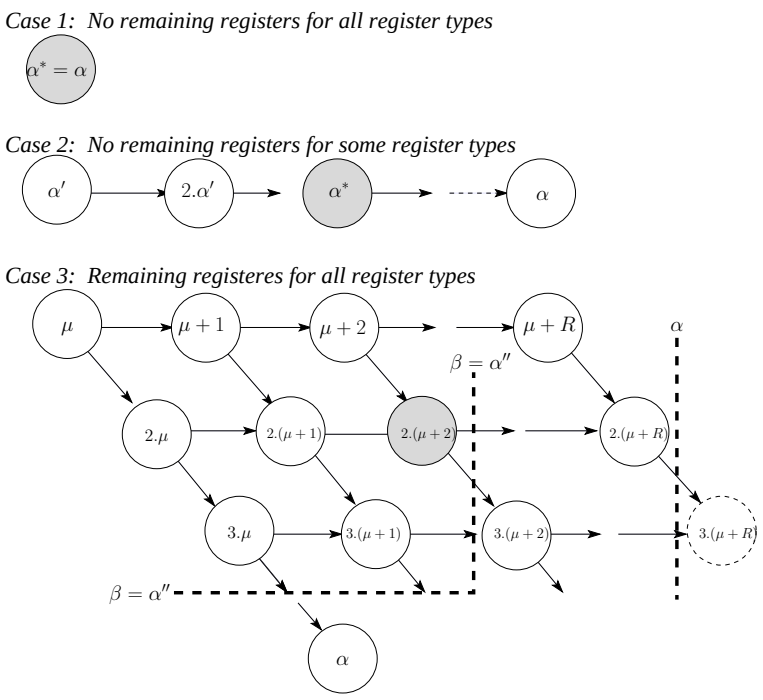


Figure 9: Loop unrolling values in the search space S

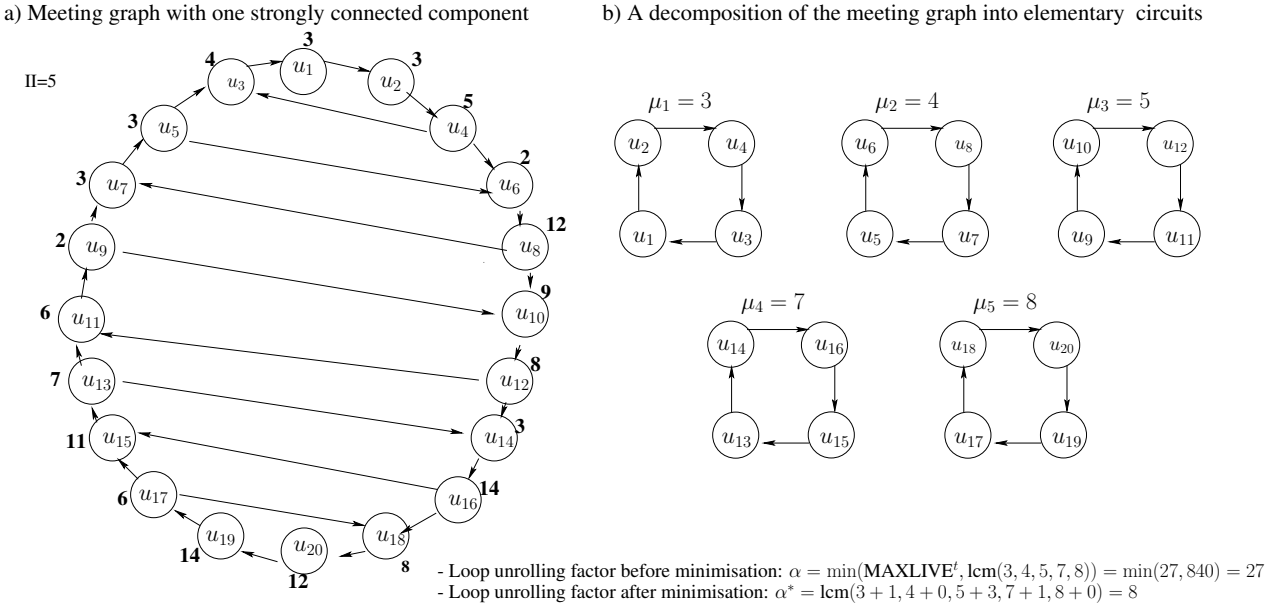


Figure 10: Example of loop unrolling reduction using meeting graph

in this case to unroll the loop MAXLIVE times. This unroll factor is equal to 27, which is lower than 840 but is still too large. In order to reduce it, we apply the loop unrolling reduction for the meeting graph circuits. Let us assume that we have $\mathcal{R}^t = 32$ available architectural registers. Hence we have $R = \mathcal{R}^t - RC = 32 - 27 = 5$ remaining

registers.

Once the meeting graph decomposed into elementary circuits, this section aims to compute the minimal loop unrolling degree α^* for the software pipelined loop using the meeting graph framework. Here, the reader must be aware that this does not guarantee minimality for other possible decompositions of the meeting graph into elementary circuits. Computing the minimal unroll factor for any circuit decomposition of the meeting graph is a combinatorical open problem. So, in the context of this section, we consider a fixed decomposition of the meeting graph, we prefer to use the term *reduction* of unroll degree instead of minimisation to avoid confusion.

As in the previous sections, 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 10(b), the final loop unrolling degree found with this method is $\alpha^* = 8$ instead of 27 or 840. 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.

The formal problem of loop unroll reduction in the context of meeting graph is almost the same as Problem 1 (multiple register types) and Problem 2 (single register types), except that MAXLIVE or MAXLIVE $+ 1$ are known to be valid unroll factors in the case of a single register type. In other words, if we have multiple register types, we are faced with Problem 1 that we studied in Section 5. If we have a single register type, then we have a unique defined MAXLIVE that we can use to improve the solution of Problem 2. Consequently, the problem of unroll factor minimisation in the context of scheduled loops can be stated as follows (for a single register type only).

Problem 5 (LCM-RED in the Context of Meeting Graph) *Let R be the number of remaining registers after a periodic register allocation (PRA) performed by a meeting graph. Let be $\mu_1, \cdots, \mu_k \in \mathbb{N}$ be the weights of the different identified elementary circuits of the meeting graph used for PRA. Compute the added registers $r_1, \cdots, r_k$ such that:*

- $\sum_{i \in [1,k]} r_i \leq R$ *(validity constraint)*
- $\alpha^* = lcm(\mu_1 + r_1, \ldots, \mu_k + r_k)$ *is minimal and*
  - $\alpha^* \leq$ *MAXLIVE*^{t} *if the MG has a unique identified elementary circuits for PRA.*
  - $\alpha^* \leq$ *MAXLIVE*$^t + 1$ *if the MG has multiple identified elementary circuits for PRA.*

The next section explains our solution for Problem 5.

6.1 Improving Algorithm 4 (LCM-MIN) for the Meeting Graph Framework

A meeting graph (MG) can have several strongly connected components of weight $\mu_1, \ldots, \mu_k \in \mathbb{N}$ (if there is only one connected component, its weight is $\mu_1 =$ MAXLIVE). This leads to the upper bound of unrolling $\alpha = lcm(\mu_1, ..., \mu_k)$. In addition, if $\alpha >$ MAXLIVE, the MG framework guarantees an upper bound of loop unrolling degree U_{max} equal to MAXLIVE or MAXLIVE $+ 1$. In fact, if the MG has one strongly connected component then the maximum loop unrolling degree is $U_{max} =$ MAXLIVE. Otherwise, if it has several strongly connected components, Lelait et al. [10] propose to create one strongly connected component by adding a complete cycle of unitary dummy intervals to the MG. One extra register is needed to achieve this, which yields to allocate MAXLIVE $+ 1$ registers by unrolling the loop $U_{max} =$ MAXLIVE $+ 1$. This one extra register is required to cyclically permute all the values in registers.

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

Our result in this section consists in identifying a reduced loop unrolling degree α^* for a fixed periodic schedule using the MG technique. Given a fixed circuit decomposition of the MG, we use Algorithm 4, looking for a good distribution of the remaining registers over all the different MG circuits. Having an upper bound for the loop unrolling degree (MAXLIVE or MAXLIVE$+1$), we reduce the search space S by computing all the possible new loop unrolling degrees β less than or equal to MAXLIVE or MAXLIVE$+1$, depending on whether the MG has one or more strongly connected components. Figure 11 describes the new search space S in the MG.

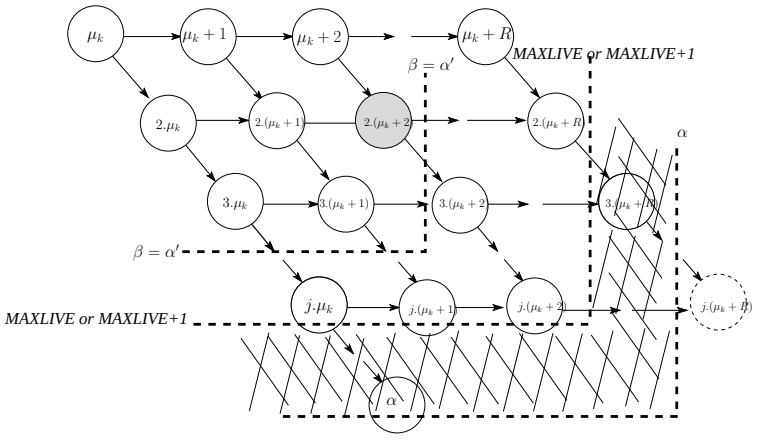


Figure 11: The new search space S in the meeting graph

Algorithmic Complexity Analysis for Solving Problem 5 Our algorithm to solve Problem 5 is very similar to Algorithm 4, so we do not detail it here. The only difference resides in the fact that the solution space S of Figure 11 has shrunk compared to Figure 7. In the worst case, we visit each node in the set S. The set S has $R \times \frac{\text{MAXLIVE}}{\mu_k}$ nodes (respectively $R \times \frac{\text{MAXLIVE}+1}{\mu_k}$). We know that MAXLIVE $\leq \mathcal{R}^t$ and $1 \leq \mu_k$. Thus, the maximal number of nodes is equal to $R \times \mathcal{R}^t$. Since we apply Algorithm 1 on each node at most, the worst case complexity for solving Problem 5 is equal to $O(R \times (\mathcal{R}^t)^3)$, which is better than the worst time complexity of Algorithm 4 described in Section 4.

7 Experiments

Evaluating the performance of a new code optimisation method must be designed carefully. The reason is that any new code optimisation can behave differently inside a compiler, depending on its order in the optimisation flow. The experimental methodology we followed in our work is based on both a standalone evaluation and on an integrated evaluation. A standalone evaluation means that we evaluate the efficiency of loop unrolling minimisation without studying the implication on code quality generated by following compilation passes after loop unrolling minimisation. An integrated evaluation means that we evaluate the final assembly code quality generated by the compiler when we plug our code optimisation method. We target an embedded VLIW architecture (ST231) because software pipelining and tightly controlled loop unrolling is particularly important on such platforms. We also explore multiple configurations regarding the number of available registers in order to understand empirically the relationship between the number of registers and the unrolling factors. All our benchmarks have been cross-compiled on a regular Dell workstation, equipped with Intel Core 2 CPU running at 2.4 GHz and a 64bit Linux kernel.

7.1 Standalone Experiments with Single Register types

This section presents full experiments on a standalone tool by considering a single register type only. Our standalone tool is independent of the compiler and processor architecture. We demonstrate the effectiveness of our loop minimisation method for both unscheduled loops (as studied in Sect 4) and scheduled loops (as studied in Sect 6).

7.2 Experiments with Unscheduled Loops

In this context, our standalone tool takes as input a data dependence graph (DDG) just after a periodic register allocation done by SIRA, and applies a loop unrolling minimisation.

7.3 Results on Randomly Generated Problem Instances

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

We generated a large number of random inputs: we varied the number of available registers $\mathcal{R}^t$ 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 as follows:

1. We generate k a random number of reuse circuits (the number k defined in Problem 2).
2. Then we generate a random value for each μ_i for all $1 \leq i \leq k$. μ_i is the weight of the i^{th} reuse circuit.

Note that the weight μ_i of each euse circuits is independent from the number of nodes inside it: we can have as many nodes we want for a given reuse circuit weight μ_i (since the reuse arcs can be equal to zero).

Figure 12 is a two-dimensional 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 architectural registers (going from 4 to 256), the Y-axis is the code compaction ratio. As can be seen, our methods allows code size reductions between 1 and more than $10,000$. In addition, we note also in Figure 12 that the ratio is very important when the $\mathcal{R}^t$ is greater. For example, the ratio of some minimisation exceeds 10000 when $\mathcal{R}^t = 256$. Figure 13 summarizes all the ratios with their harmonic and geometric means. As can be observed, these average ratios are significant. Note that the averages increase with the number of available registers because we scale the size of our inputs with the number of architectural registers. For a fixed problem size, more available registers will typically lead to a lower loop unrolling factor (because there will be more free registers).

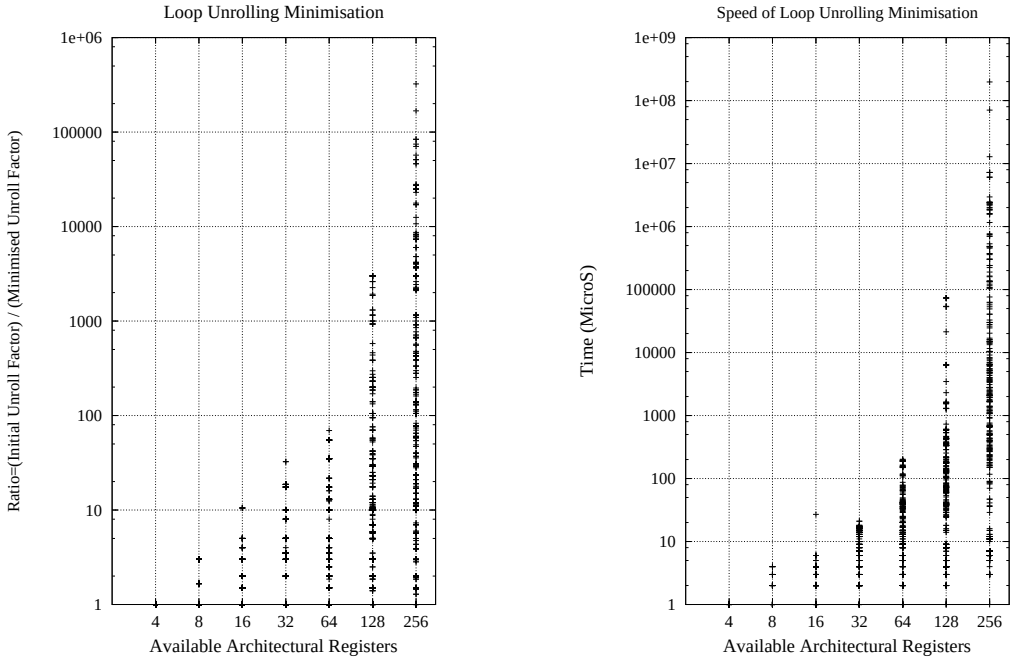


Figure 12: Loop unrolling minimisation experiments (random DDG, single register type)

Furthermore, our method is very fast. Figure 12 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

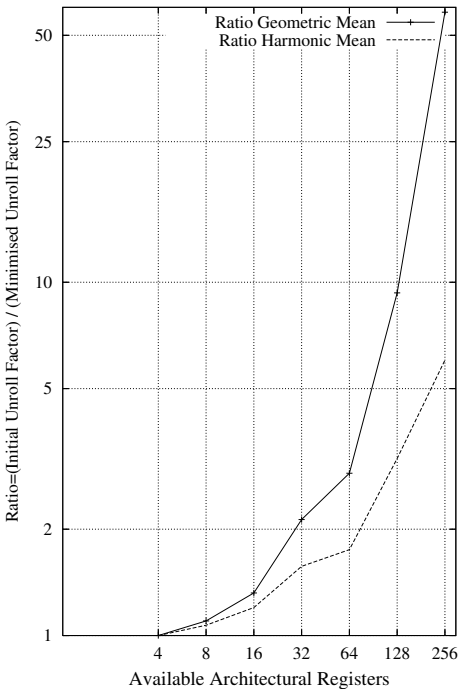


Figure 13: Average code compaction ratio (random DDG, single register type)

systems (not to interactive compilers like gcc or icc). We remark also the speed of extremely rare minimisation (when $\mathcal{R}^t = 256$) can reach 1000 seconds.

7.4 Experiments on Real DDG

The DDG we use here are extracted from various real benchmarks, either from the embedded domain or from the high performance computing domain: DSP-filters, Spec, Lin-ddot, Livermore, Whetstone, etc. In total we use 310 DDGs in our experiments, their sizes go from 2 nodes and 2 arcs up to 360 nodes and 590 arcs. Afterwards, we have performed experiments on these DDGs, depending on the considered number of registers. We considered three configurations as follows:

1. machine with unbounded number of registers;
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).

7.4.1 A machine with an unbounded number of registers

Theoretically, the best result for the *LCM-MIN Problem* (Sect 4) is $\alpha^* = \mu_k$ the greatest value of μ_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 an unrolling degree of μ_k. Recall that μ_k is weight of the largest circuit, so the smallest possible unrolling degree is μ_k.

In order to interpret all the data resulting from the application of our method to all DDG, we present some statistics. We have measured the arithmetic mean of the number of added registers ($AVR^{ar}(\sum_{i=1}^{k} r_i)$) needed to obtain μ_k. We also calculate the harmonic mean of all the ratio ($AVR^{har}(\frac{\alpha}{\mu_k})$).

Our experiments show that using 12.154 additional registers are on average 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 ($AVR^{har}(\frac{\alpha}{\mu_k}) = 2.100$). That is, our loop unrolling minimisation pass is very efficient regarding code size compaction.

7.4.2 Machines with bounded numbers of registers varied between 4 and 256

We consider a machine with a bounded number of architectural registers $\mathcal{R}^t$. We varied $\mathcal{R}^t$ from 4 to 256 and we apply our code optimisation method on all DDGs. For each configuration, we calculate the arithmetic mean of the number of added registers ($AVR^{ar}(\sum_{i=1}^{k} r_i)$), the weighted harmonic mean of all the ratios $AVR^{har}(\frac{\alpha}{\alpha^*})$, and the geometric mean $AVR^{GM}(\frac{\alpha}{\alpha^*})$. 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 finds the minimum unrolling factor in all configurations except when $\mathcal{R}^t = 4$. 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 ion average 1.078 registers among 9.722 remaining registers. We also note that in many configurations we have a high harmonic and geometric mean for the ratio ($AVR^{har}(ratio)$). For example, in the machine with 256 registers, $AVR^{har}(ratio) = 2.725$ and $AVR^{GM}(ratio) = 5.61$. Note that in practice, if we have more architectural registers, then we have more remaining registers. Consequently we can find lower unrolling factors. This explains for instance why the minimum unrolling degree uses more remaining registers when there are 256 architectural registers than when there are 8 (see Table 1), with the advantage of a better loop unroll minimisation ratio in average.

$\mathcal{R}^t$	$AVR^{ar}(\sum_{i=1}^{k} r_i)$	$AVR^{har}(ratio)$	$AVR^{GM}(ratio)$	$AVR^{ar}(R)$
4	0	1	1	0.293562
8	0.015	1.007	1	0.818
16	0.250	1.104	1.16	2.723
32	1.078	1.414	1.73	9.722
64	3.070	1.963	3.34	29.055
128	14.073	2.715	5.54	79.641
256	15.228	2.725	5.61	207.118

Table 1: Machine with bounded number of registers

Figure 14 shows the harmonic mean of the minimised (final) and the initial loop unrolling weighted by the number of nodes of different DDG. We calculate this weighted harmonic mean on different configurations. We assume a VLIW processor with an issue width of 4 instructions per cycle, where all the DDGs are pipelined with $II = MinII = \max(MII_{\text{res}}, MII_{\text{dep}})$. In all configurations, the average of the final unrolling degree of the pipelined loops is below 8, a significant improvement over the initial unrolling degree. E.g., in the configuration where $\mathcal{R}^t = 64$, the minimised loop unrolling is on average equal to 7.78.

7.4.3 Machines with bounded numbers of registers varied between 4 and 256 with the `continue` option

In these experiments we use the `continue` option of our periodic register allocation. Without this option, SIRA computes the first periodic register allocation which verifies $\sum \mu_i \leq \mathcal{R}^t$ (not necessarily minimal). If we use the `continue` option, SIRA generates the periodic register allocation that minimises $\sum \mu_i$, leaving more remaining registers to the loop unrolling minimisation process. In order to compare these two configurations (machine with bounded number of registers versus machine with bounded number of registers using the `continue`) option, we reproduce the statistics of the previous experiments using this additional option. The results are described in Table 2.

By comparing Table 1 and Table 2, we notice that some configurations yield a better harmonic mean for the code compaction ratio with option `continue`, when $\mathcal{R}^t \leq 64$. Conversely, the ratio without option `continue` is better when $\mathcal{R}^t \geq 128$. These strange results are side-effects of the reuse circuits generated by SIRA, which differ depending on the number of architectural registers. In addition, because of the complex mathematical structure of the

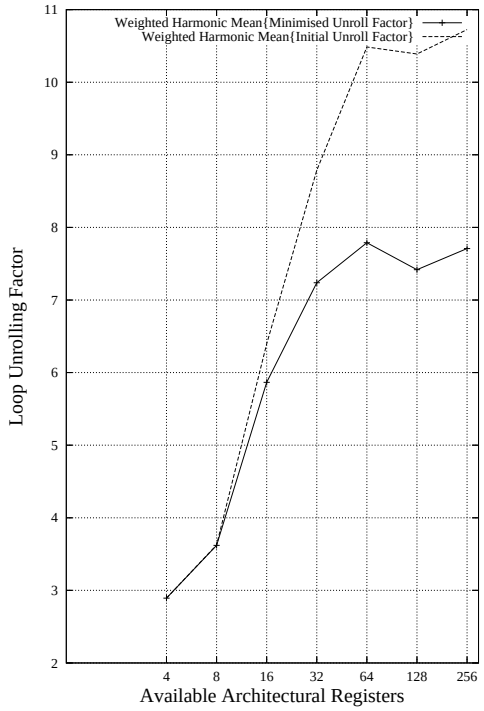


Figure 14: Weighted harmonic mean for minimised loop unrolling degree

$\mathcal{R}^t$	$AVR^{ar}(\sum_{i=1}^{k} r_i)$	$AVR^{har}(ratio)$	$AVR^{GM}(ratio)$	$AVR^{ar}(R^t)$
4	0	1	1	0.33412
8	0.015	1.007	1.01	0.885
16	0.253	1.104	1.16	2.795
32	1.096	1.421	1.74	9.968
64	3.251	2.027	3.59	31.140
128	9.403	2.289	4.32	81.773
256	15.195	2.717	5.58	207.394

Table 2: Machine with bounded registers with option `continue`

LCM-MIN Problem, a larger number of remaining registers R does not imply a lower unrolling degree. *I.e.* increasing the number of remaining registers (by performing minimal periodic register allocation) does not necessarily imply a maximal reduction of the loop unrolling degree.

7.5 Experiments with Scheduled Loops

We integrated our loop unrolling reduction method as a post-pass of the meeting graph technique in LoRA (Loop-optimal Register Allocation) framework [9]. As explained in Section 6, we use the term loop unroll reduction in the context of the MG instead of minimisation. This is because we consider the fixed MG decomposition applied by LoRA, rather than an optimal decomposition. LoRA is performed after SWP and implements the meeting graph technique and several heuristics (Lam's heuristic [16] and those of Hendren et al. [13]), for combining register allocation and

loop unrolling for SWP loops. All the loops are scheduled with DESP [26]. Due to the limitations of the LoRA tool flow, we consider a different benchmark set consisting of 1935 loops extracted from the SPEC FP 92, SPEC INT 92, Livermore loops, Linpack, and NAS benchmarks. In this section, the loops are isolated and optimised as individual program fragments.

The loop unrolling reduction method is applied when meeting graph finds that MAXLIVE $\leq \mathcal{R}^t$. Otherwise, MG does not unroll the loop and uses a heuristic to introduce spill code.

Table 3 shows the performance of the MG approach on the 1935 DDGs in our test set. It shows the number of loops where the MG approach can find a periodic register allocation without spilling, versus the number where spilling is required.

$\mathcal{R}^t$	Unrolled Loop with MG	Spilled Loops with MG
16	1602	333
32	1804	131
64	1900	35
128	1929	6
256	1935	0

Table 3: Number of unrolled loops compared to the number of spilled loops resulted (meeting graph)

In order to highlight the improvements of our loop unrolling reduction method on DDGs where MG found a solution (no spill), we show in Figure 15 a boxplot[4] for each processor configuration. Note that the final (reduced) 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 present the arithmetic mean of the initial loop unrolling α, the final loop unrolling α^* and $ratio = \frac{\sum \alpha}{\sum \alpha^*}$. Table 4 shows that on average the final loop unrolling degree is greatly reduced compared to the initial loop unrolling degree.

$\mathcal{R}^t$	Average Initial Loop Unrolling Factors	Average Reduced 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 4: Arithmetic mean of initial loop unrolling, final loop unrolling and ratio

For each configuration we also computed the number of loops where the reduced loop unrolling degree is less than MAXLIVE. We present the different results in Table 5. It shows that in each configuration, the minimal loop unrolling degree obtained using our method is greatly less than MAXLIVE. Only a very small number of loops are unrolled MAXLIVE times.

We also measure the running time of our approach using instrumentation with the `gettimeofday` function. On average the execution time of loop unrolling reduction method in the meeting graph is about 5 microseconds (average of all MG). The maximum run-time is about 600 microseconds.

[4]Boxplot, also known as *box-and-whisker diagram*, is 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 ($\max$). The min is the first value of the boxplot, and the max is the last value. Sometimes, the extrema values (min or max) are very close to one of the quartiles. This is why we do not distinguish sometimes between the extrema values and some quartiles.

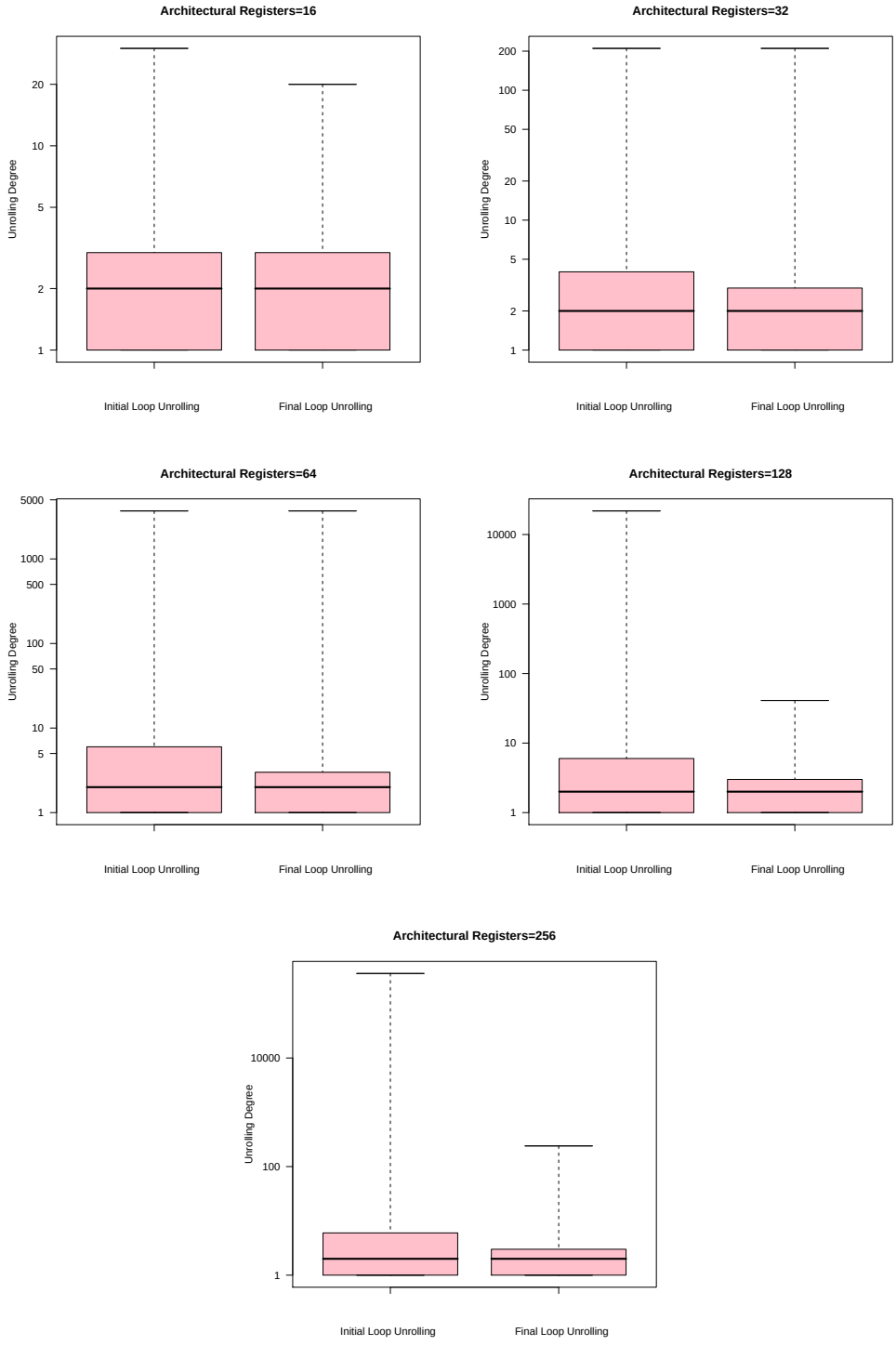


Figure 15: Initial vs. final loop unrolling in each configuration

$\mathcal{R}^t$	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 5: Comparison between final loop unrolling factors and MAXLIVE

7.5.1 Loop Unrolling of Scheduled vs. Unscheduled Loops (Meeting Graph vs. 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 [2]) 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 LoRA (scheduled loops) and SIRA (unscheduled loops), we conducted other experiments on larger DDGs from both high performance and embedded benchmarks: SPEC CPU2000, SPEC CPU2006, MEDIABENCH and LAO (internal ST benchmarks). We applied our algorithm to a total of 9027 loops. We consider a machine with a bounded number of architectural registers $\mathcal{R}^t$. We varied $\mathcal{R}^t$ from 16 to 256.

The experiments show that final loop unrolling degrees computed by LoRA are lower than those computed by SIRA. The minimal unrolling degree for 75% of SIRA optimised loops is less than or equal to 7. In contrast, LoRA does not require any unrolling at all (unroll degree equal to 1) for 75% of loops.

We highlight in Table 6 some of the other 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 due to our method is small when using LoRA compared with the average of final loop unrolling in SIRA. We also show that the maximum final loop unrolling degrees are low in LoRA compared to those in SIRA. The main exception is FFMPPEG where the unrolling degree for the meeting graph on a machine with 16 registers is actually slightly higher. In the first line of Tab 6, 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 are no registers left to apply our loop unrolling reduction method.

The choice between the two techniques depends upon whether 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.

7.5.2 Comparison between the Meeting Graph Framework and MVE

When allocating registers for loops which have already been scheduled using software pipelining, the literature already contains a well-known method: modulo variable expansion (MVE, see Sect 8). Previous research [10] has compared the MG and MVE approaches. In practice, MVE produces very little unrolling (in most cases it does not unroll the loop at all). However, MVE provides no mathematical guarantee that a prediodic 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 contrast to MVE, MG followed by loop unrolling minimisation has the formal guarantee that at most $\mathcal{R}^t$ registers are allocated.

We conducted experiments on 9027 DDGs from both high performance and embedded benchmarks: SPEC CPU2000, SPEC CPU2006, MEDIABENCH, and LAO (internal ST benchmarks). We consider a machine with a bounded number of architectural registers $\mathcal{R}^t \in \{16, 32, 64, 128, 256\}$. While the MVE technique always produces lower unrolling degrees, it turns out that the MG technique is also experimentally 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.

$\mathcal{R}^t$	Benchmarks	Average Final Loop Unrolling		Maximum Final Loop Unroll	
		MG	SIRA	MG	SIRA
16	LAO	1.127	2.479	30	28
	MEDIABENCH	1.175	2.782	12	26
	SPEC CPU2000	1.113	2.629	9	28
	SPEC CPU2006	1.085	2.758	9	16
32	LAO	1.219	3.662	9	57
	MEDIABENCH	1.185	3.032	9	84
	SPEC CPU2000	1.118	2.823	9	28
	SPEC CPU2006	1.09	2.966	9	26
64	LAO	1.3	6.476	9	72
	MEDIABENCH	1.426	3.225	63	84
	SPEC CPU2000	1.119	2.881	9	45
	SPEC CPU2006	1.09	3.001	9	26
128	LAO	1.345	9.651	9	88
	MEDIABENCH	1.215	3.338	14	84
	SPEC CPU2000	1.119	2.916	9	45
	SPEC CPU2006	1.09	3.063	9	275
256	LAO	1.345	9.733	9	88
	MEDIABENCH	1.214	3.384	14	84
	SPEC CPU2000	1.119	2.946	9	45
	SPEC CPU2006	1.09	3.256	9	27

Table 6: Optimised loop unrolling factors of scheduled vs. unscheduled loops

In the following section, we study the efficiency of our method when integrated inside a real industrial compiler.

7.6 Experiments inside Real World Industrial Compiler with Multiple Register Types

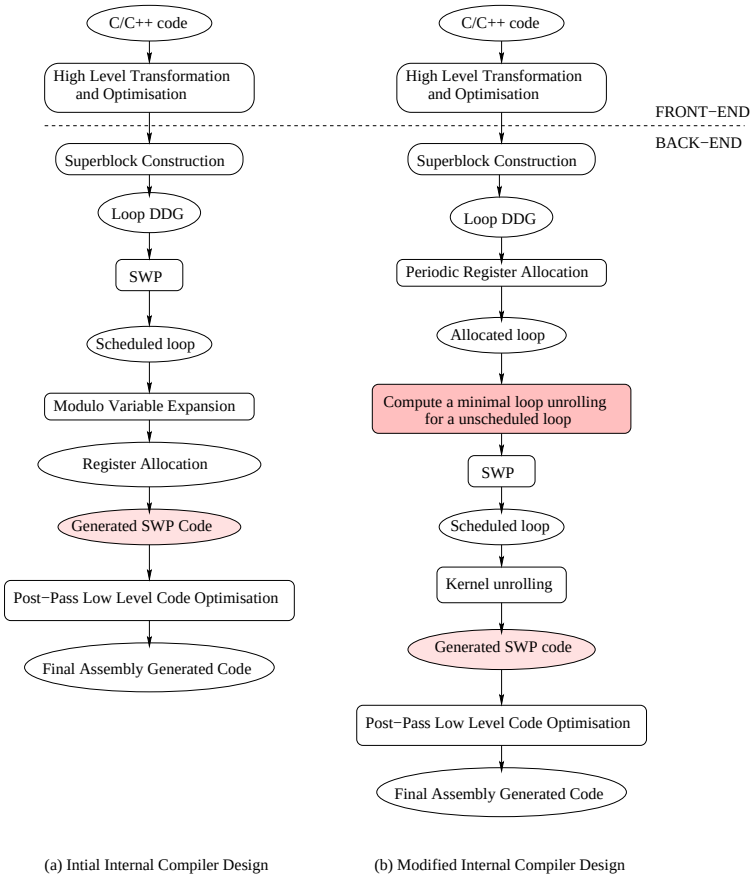
Our experimental setup is based on `st200cc`, a STMicroelectronics production compiler based on the Open64 technology (`www.open64.net`), whose code generator has been extensively rewritten in order to target the STMicroelectronics ST2xx VLIW processor family. These VLIW processors implement a single cluster derivative of the Lx architecture [11], and are used in several successful consumer electronics products, including DVD recorders, set-top boxes, and printers.

The target architecture is the ST231 [22]. The number of architectural registers is configured as 64×32 bits general purpose registers and 4×1 bit branch registers (for predicated execution). We will thus use two register types, GR and BR. We use STMicroelectronic's cycle-accurate simulator. Some architecture parameters such as instruction and bypass latencies influence scheduling, hence periodic register allocation. The issue width is 5 and pipeline depth is 4. The instruction pipeline is fully bypassed and interlocked. The L1 instruction and data caches have a 3 cycle latency for a typical frequency of 400GHz.

Our benchmark set is built of all C and C++ applications from MEDIABENCH, SPEC CPU2000, SPEC CPU2006, FFMPEG, and LAO (internal vendor benchmarks from STMicroelectronics). st200cc does not support Fortran codes. By enabling the `-O3` optimisation level, the number of loops going through SWP and periodic register allocation (and hence optimised with our method) is about 9000. The average size of the DDG is 300 (expressed as the number of nodes plus arcs). The DDG size can go up to $22,830$ in extreme cases.

Figure 16 illustrates the integration of our loop unrolling minimisation method inside a real world compiler. Figure 16(a) shows the initial compiler design, where MVE and register allocation were processed after SWP. After discussion with our industrial partner (STmicroelectronics), we decided to plug the loop unroll minimisation, coupled to SIRA periodic register allocation, all before SWP (Figure 16(b)). The reason is that we want to control spill code early in the instruction scheduling process, while the initial compilation flow does not guarantee to control it.

Hence, the experiments reported in this section demonstrates that the SWP code quality generated by the compilation flow of Figure 16(b) is better than the SWP code generated by Figure 16(b). Other compilation flows are possible of course, experimenting with many of them is outside the scope of this article. Furthermore, changing a method in each compilation pass may change the quality of the generated code. We do not attempt to demonstrate the benefit of our algorithms in any situation with any combination of compilation techniques. Isolating the benefit of loop unrolling minimisation has already been adressed in the last section. This section shows that including it inside an existing compilation flow (Figure 16(b)) improves the quality and size of the generated code. Note that other compilations steps carried out before and after our module may also consume registers and introduce move operations. We are not arguing that our own compilation step will guarantee the absence of spill and move operations for any integrated context: our optimality criteria is for standalone evaluation only. Since integrating an optimal code optimisation step inside an existing compiler does not guarantee, in general, that the generated code will be improved, we show in this section that we really reach this objective and the quality of the generated code is improved in practice.



(a) Intial Internal Compiler Design (b) Modified Internal Compiler Design

Figure 16: The integration of loop unroll minimisation inside a industrial compiler

First, regarding compilation times, our experiments show that the run-time of our SIRA register allocation followed by loop unrolling minimisation (LUM) is less than 1 second per loop on average. So, it is fast enough to be included inside an industrial cross compiler such as st200cc.

7.6.1 Statistics on Minimal Loop Unrolling Factors

Figure 17 shows numerous boxplots representing the initial loop unrolling degree and the final loop unrolling degree of the different loops per benchmark application. In each benchmark family (LAO, MEDIABENCH, SPEC CPU2000, SPE2006), we note that the unroll factor decreases significantly from its initial value to its optimized value.

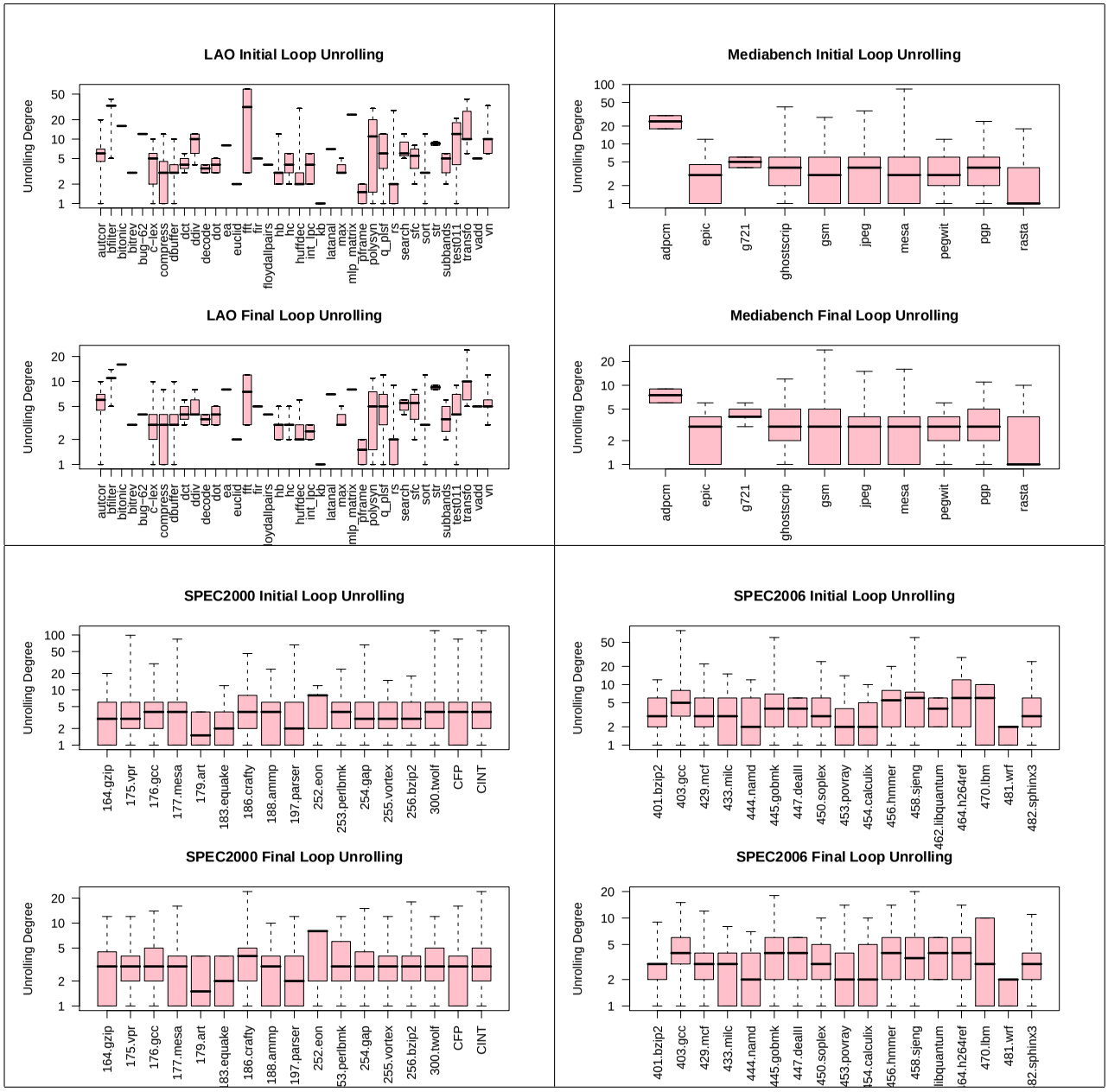


Figure 17: Statistics on loop unrolling minimisation

To highlight the improvements of our loop unrolling minimisation method, we show in Figure 18 a boxplot for each benchmark family (FFMPEG, MEDIABENCH, SPEC CPU2000, SPEC CPU2006). We note that the final loop unrolling of half of the applications is under 3 and that the final loop unrolling of 75% of applications is less than or equal to 5. This compares favourably with the loop unrolling degrees calculated by minimising each register type in isolation. Here, the final loop unrolling degree of half of the applications is under 5 and the final loop unrolling of 75% of the applications is under 7, the final loop unrolling for the remaining loops can reach 50. These numbers demonstrate the advantage of minimising all register types concurrently. Of course, if the code size is a hard constraint, we do not generate the code if the loop unrolling factor is prohibitive and we backtrack from SWP. Otherwise, if the

code size budget is less restrictive, we will demonstrate later that, even if some final unrolling factors seem large, in our case all the loops still fit in the I-cache of ST231.

In order to compare between the codes generated by compilation flow of Figure 16(a) vs. Figure 16(b), we also plot in Figure 18 the loop unrolling degree obtained using MVE. The MVE heuristic always finds a smaller unrolling degree. However, MVE does not guarantee a register allocation with a minimal number of registers and in general it may lead to unnecessary spills breaking the benefits of SWP. In other words, MVE may require spill code insertion even if MAXLIVE$^t \leq \mathcal{R}^t$. This problem occurs in 86 loops of FFMPEG, 100 loops of MEDIABENCH, 240 loops of SPEC CPU2000 and 111 loops of SPEC CPU2006.

With our loop unrolling register allocation method, we formally guarantee that we do not need more registers than the number of architectural registers, for each register type. In addition a suitable quality criterion is to check if the unrolled loops fit in the I-cache and if our method actually decreases the number of spill and move instructions. This is discussed in the next two sections.

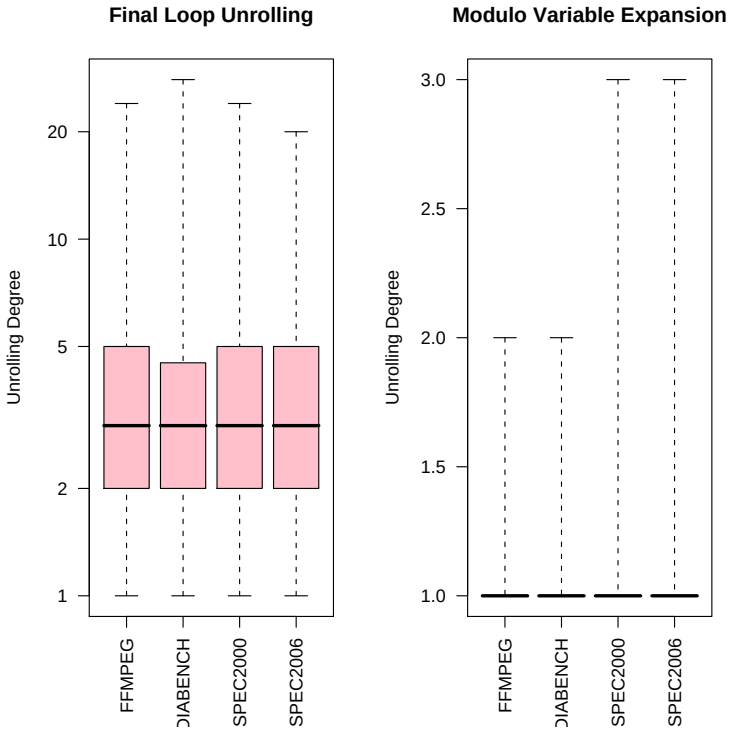


Figure 18: Loop unrolling minimisation versus MVE

7.6.2 Statistics on I-cache Fit

As the MVE heuristic always finds smaller loop unrolling degrees, MVE is better than our loop unrolling in terms of code size. We study whether the resulting unrolled loops fit in the I-cache: that is, we study if the size of the generated loops, in terms of bytes, is less or equal than the size of the I-cache of the ST231 ($32KB$). All the loops generated with MVE fit inside the I-cache. Fortunately our experimental results show that using our techniques, all the unrolled loops fit in the I-cache, even if our unrolling factors are higher than those computed by MVE.

The next section demonstrates that SWP codes generated by Figure 16(b) are better than Figure 16(a) in terms of requiring less spill code and fewer move operations.

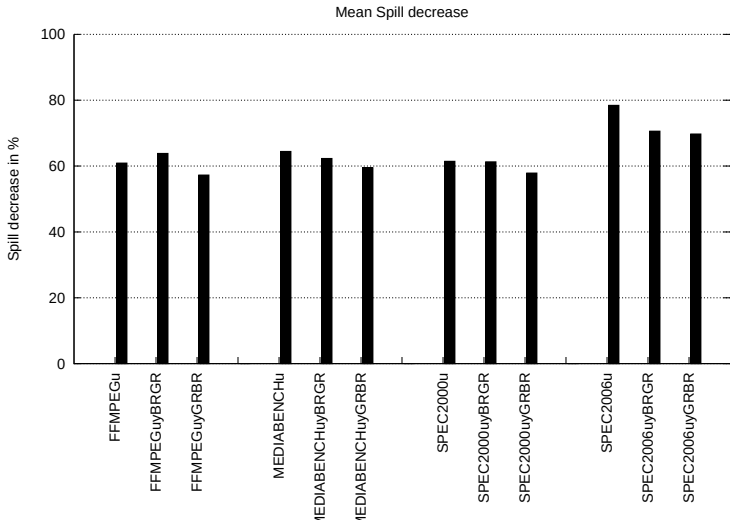


Figure 19: Spill code reduction in %

7.6.3 Statistics on Spill Code and Move Operation Reduction

We measure statically the amount of spill code reduced when we plug our method as described in Figure 16(b). The initial absolute static counts of spills in SWP loops of the whole a applications are (per benchmark family): FFMPEG= 254, MEDIABENCH= 405, SPEC CPU2000= 1270, SPEC CPU2006= 571. The spill code decrease is computed over all SWP loops of the whole applications of each beanchmark family. It is evaluated as the difference between the amount of spill code generated without using SIRA followed by LUM, *i.e.* with MVE, and the amount of spill code generated when using SIRA, all divided by the total amount of spill code generated without using SIRA ($\frac{\sum Spill_{nosira} - \sum Spill_{sira}}{\sum Spill_{nosira}}$). Fig 19 plots the spill code decrease in each benchmark family: the plotted bars correspond to FFMPEG, SPEC CPU2000, SPEC CPU2006 and MEDIABENCH benchmarks. The suffixes BRGR and GRBR on each benchmark name correspond to variants of SIRA where the register allocation is applied in each type separately: the label BRGR means that we first optimise BR registers then GR registers, while GRBR corresponds to the opposite order. Where neither BRGR nor GRBR is specified, it means that both register types have been minimised concurrently. In all variants of SIRA the loop unrolling minimisation is performed concurrently over all types, and not on each type in isolation. As we can see, the compilation flow of Figure 16(b) greatly reduces spill code.

.

We measure the reduction in `move` operations due to using the compilation flow of Figure 16(b). The decrease in `move` operations is calculated as the difference between the number of the `move` operations generated without using SIRA followed by LUM, *i.e.* with MVE, and the number of `move` operations generated with the use of SIRA, all divided by the number of `move` operation generated without using SIRA: $\frac{\sum Move_{nosira} - \sum Move_{sira}}{\sum Move_{nosira}}$. Fig 20 displays the results. As we can see, with all variants of SIRA and in most cases, we reduced the number of `move` operations by $10 - 20\%$. The most significant decrease is obtained when we consider all register types concurrently: we reduce the number of `move` operations by 18.92% for FFMPEG, 9.61% for MEDIABENCH, 14.17% for SPEC CPU2000 and 18.21% for SPEC CPU2006.

It may be argued that performing periodic register allocation (PRA) before SWP reduces the freedom of the scheduler, and thus may prevent the scheduler from finding a good software pipeline. This is a classic phase ordering problem, which has been studied in detail by Touati *et al.* [23]. The next section presents experimental results on this question in our context.

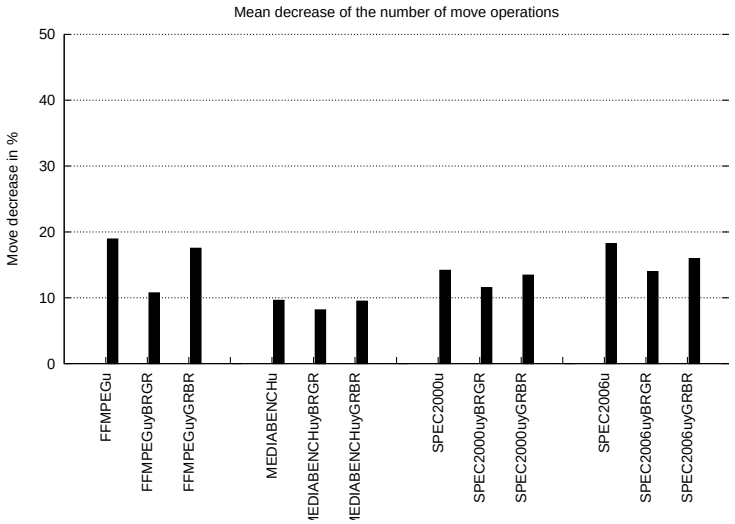


Figure 20: Move operations reduction in %

7.7 Statistics on II Increase

SIRA followed by loop unrolling minimisation are performed before SWP. It may be argued that introducing arcs inside DDG before software pipelining would alter the instruction schedule quality, since extra constraints are added. In practice, this is not the case because the usual software pipelining heuristics are not optimal. SIRA is a sophisticated algorithm that models the periodic data dependence and register constraints on software pipelining very well. Where SIRA introduces extra edges into the DDG, these extra edges will reduce the freedom of the software pipeliner. However, we suspect that these additional edges guide the software pipeliner to good quality schedules by restricting scheduling choices, and the result is that the additional edges only very rarely result in a worse schedule. In Fig 21 we plot the number of DDGs in which applying SIRA (before software pipelining) increases II. As we can see, in most cases, the initiation interval is not altered. The percentage of loops for which the initiation interval increases or for which spilling is necessary is reasonable: in Fig 21, the bar named `II increase` represents the fractions of loops where II increases because of early periodic register allocation, and the bar `No solution` represents the fractions of loops where cyclic register allocations fails and introducing spilling is necessary.

We also measure the mean of the II increase in all our benchmarks as $\frac{\sum_{loops} II_2 - II_1}{\sum_{loops} II_1}$, where II_1 is the value of the initiation interval without using SIRA, and II_2 is the value of the initiation interval when using SIRA. The mean increase of the II is resp. 1.56% for FFMPEG, 0.05% for MEDIABENCH, 1.66% for SPEC CPU2000 and 0.09% for SPEC CPU2006. As can be seen, the average increase of the II is negligible in all benchmarks.

8 Related Work

We review in this section some related work on code generation for periodic register allocation, that we did not deal with as part of the background in Section 2. There is a very large literature on instruction scheduling, software pipelining, and register allocation. For instance, the authors in [**?**] already studied a suitable unrolling factor to produce rate-optimal SWP in data flow programs. That authors showed that the optimum loop unrolling factor is the least commom multiple of the number of registers. In this section we review only research that deals specifically with register allocation and code generation of software pipelined loops.

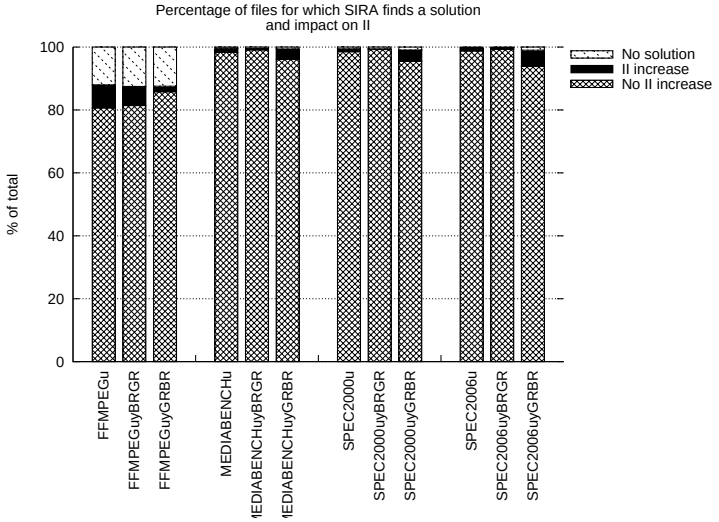


Figure 21: Statistics on II increase and necessary spilling

8.1 Rotating Register Files

In order to generate a code for variables spanning multiple iterations, some processor architectures (Cydra, Itanium) provide hardware support called a *rotating register file* (RRF) [7, 8]. RRF is a hardware mechanism to prevent successive lifetime intervals from being assigned to the same physical registers. Consider the following example:

LOOP
$a[i+2] = a[i] + 1$
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 $i+2$
$R = R[-2] + 1$	$R[+2] = R + 1$

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

8.2 Inserting `move` Operations

This method is also called *register renaming*. Considering the previous example, we use 3 registers to allocate $a[i]$ and perform `move` operations at the end of each iteration [17]: $a[i]$ 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
$R3 = a[i] + 1$
$a[i+2] = R1 + 2$
$R1 = R2$
$R2 = R3$
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.

8.3 Loop Unrolling after SWP

Without RRF and inserting copy operations, loop unrolling is more suitable to maintain II. The resulted loop body itself is bigger but no extra operations are executed in comparison with the original code. MVE has already been explained. Other work proposes to implement a generalized form of modulo expansion in [5, 13], but with the same limitation of MVE, that is $RC^t \geq \text{MAXLIVE}^t$ without proving a formal upper-bound on RC^t. In our work, since we base our approach on reuse graphs and meeting graphs, we are able to guarantee that $\mathcal{R}^t \geq RC^t \geq \text{MAXLIVE}^t$.

8.4 Code Generation for Multidimensional Loops

Instruction level parallelism (ILP) scheduling is a special case of the general k-periodic multidimensional scheduling problem. Indeed, SWP is the special case when the scheduling period is unique and integral. The case of multidimensional memory storage optimisation is also interesting if we target regular loop nests for high performance codes [1]. Using registers instead of memory cells is a special case studied in [20]. Multidimensional approaches are more appropriate for regular computer intensive codes: 1) our target loops are one-dimensional at the back-end level, where the compiler has broken the multidimensional structure of the loop nest 2) The exact mathematical relationship between MAXLIVE^t, II and the unrolling factor is not known yet in the case of multidimensional instruction scheduling. That is, the problem of optimal register allocation in multidimensional loops with minimal unroll factor is still an open problem; a sub-optimal heuristic for this problem is presented in [20]. 3) The code size needed to optimally exploit ILP and registers in multidimensional loop nests is more complex to model with a single unroll factor. Indeed, code generation in this case uses multidimensional unroll factors [4, 25], that may create in theory larger code size than a one-dimensional innermost loop.

9 Conclusion

In the absence of rotating register files, periodic register allocation requires that the SWP kernel is unrolled to generate spill-free or move-free code. Modulo variable expansion is a popular choice for production compilers because, in addition to its simplicity, it generates low unrolling factors but with the risk of introducing unnecessary spill code. On the other hand, the meeting graph approach guarantees that the unrolled loop requires exactly MAXLIVE registers but with the risk of higher unroll factors.

Our work solves this open dilemma. First, we guarantee that the number of required registers in the unrolled SWP kernel does not exceed the number of available registers. Second, we formalise the problem of minimal loop unrolling relying on the remaining registers after register allocation. We provide an algorithm to compute the minimal unroll factor.

Minimizing the unroll factor is a different problem depending on whether we consider a single or multiple register types, or whether we consider scheduled or unscheduled loops. We provided an algorithmic solution for all these variants, and we proved that all are based on a minimisation problem of a least common multiple, called LCM-MIN. If the target architecture contains a single register type, then loop unroll minimisation amounts to minimise a single least common multiple. If the processor contains multiple register types, then we proved than minimising the unroll factor of each register type separately does not lead to global minimum. Consequently we proposed an adapted algorithm based on LCM minimisation that optimise the global unroll factor. If the loops have not yet been scheduled, then our minimisation method is plugged as a post-pass to reuse graphs (SIRA [24]). If the loops have already been scheduled, then our loop reduction method is plugged as a post-pass to meeting graphs. Choosing between the two previous techniques depends on the compiler design flow, but the new theoretical guarantees and algorithms we presented shed a new light on these phase ordering decisions.

The worst case performance of our LCM-MIN algorithm is exponential. However, our solution is very fast in practice, and inputs that result in exponential running time are very rare: indeed, it did not happen at all in the standard benchmarks we experimented with, and seldom with random dependence graphs. However, two open problems remain, despite numerous contacts with number theory experts: the first one is to prove that the problem is (or is not) computationally hard in the worst case; the second problem is to find the average case complexity of our current algorithm.

In addition to this wealth of theoretical and algorithmic results, we spent much effort on a thorough experimental evaluation of the effectivness of our method. Both as a standalone optimisation, and integrated into a production compiler. For a standalone context, independently of the compiler and the architecture, we demonstrated that our unroll factor minimisation is fast and the final resulting unrolling degrees are satisfactory in almost all cases. Nevertheless, we noticed that a few loops still require high unrolling degrees even after our optimisation.

For an integrated context, we plugged our solution inside st200cc compiler for ST231 VLIW processors. We compiled all C and C++ benchmarks from the FFMPEG, LAO, MEDIABENCH, SPEC CPU2000, and SPEC CPU2006 suites. We demonstrated that: (1) Our loop unrolling minimisation is fast enough to be included inside an interactive commercial quality cross compiler; (2) The resulting loop unrolling factors are satisfactory to justify the decision of changing the compilation flow in the backend; (3) Following the latter point in the context of the `st200cc` compiler for the ST231 VLIW processor, we found that our techniques generate better code than the existing compiler, which allocates registers after software pipelining and unrolls using MVE. By better we mean, less spill code, fewer move operations, with satisfactory code size (all loops fitting within I-cache), and with a better initiation interval on average.

This paper presents the most definite theoretical and algorithmic results, and the most complete experimental study to date on periodic register allocation. Nevertheless, the experimental evaluation clearly points to possible improvements and further research. First of all, the occasional high unrolling degrees suggest that it may be worthwhile to consider combining the insertion of move operations with kernel unrolling; preliminary results were recently obtained to confirm this intuition [3]. Second, it is possible that the unexpectedly good initiation intervals achieved with MVE may result from a conjunction of favorable features of the experimental methodology. We would like to replicate these experiments on a clustered VLIW architecture with high contention on register moves, and together with upstream, register-hungry compiler passes like unroll-and-jam. Third, while our study was motivated by compilation challenges of embedded VLIW processors, it may well have an impact on construction of back-end compilers for GPUs. In particular, spilling variables on GPUs is so impractical that compilers may not even implement it (e.g., NVIDIA's nvcc). The direct correlation between register usage and the number of threads a given multiprocessor can handle also pushes for stronger register usage guarantees in the compiler. Finally, GPU cores do not support out-of-order scheduling (AMD GPUs even have VLIW features) and may benefit from software pipelining. We leave these questions and perspectives open for future work.

Acknowledgements

We are highly indebted to Benoît Dupont-de-Dinechin for his help to connect our methods to the st200cc compiler. We are also grateful to the reviewers of earlier versions of this paper who greatly contributed to the improvement of its quality and readability. This research result has been supported by the ANR MOPUCE project (contract number 05-JCJC-0039) and the DIGITEO foundation (contract number 2007-10D).

References

[1] Alain Darte and Robert Schreiber and Gilles Villard . Lattice-Based Memory Allocation . *IEEE Transactions on Computers*, pages 1242–1257, October 2005.

[2] 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.

[3] Mounira Bachir, Sid-Ahmed-Ali Touati, and Albert Cohen. Decomposing meeting graph circuits to minimise kernel loop unrolling. In *9th Workshop on Optimizations for DSP and Embedded Systems (ODES'11, associated with CGO)*, Chamonix, France, April 2011.

[4] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.

[5] Benoît Dupont de Dinechin. A unified software pipeline construction scheme for modulo scheduled loops. In *PaCT '97: Proceedings of the 4th International Conference on Parallel Computing Technologies*, pages 189–200, London, UK, 1997. Springer-Verlag.

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

[7] 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.

[8] James C. Dehnert and Ross A. Towle. Compiling for the Cydra 5. *Journal of Supercomputing*, 7(1-2):181–227, 1993.

[9] Christine Eisenbeis and Sylvain Lelait. LoRA, a Package for Loop Optimal Register Allocation. Technical report, INRIA, France, June 1999.

[10] 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.

[11] Paolo Faraboschi, Geoffrey Brown, Joseph A. Fisher, Giuseppe Desoli, and Fred Homewood. Lx: a technology platform for customizable VLIW embedded processing. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 203–213, New York, NY, USA, 2000. ACM.

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

[13] Laurie 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.

[14] Nick Howgrave-Graham. Approximate Integer Common Divisors. In *Cryptography and Lattices, International Conference (CaLC)*, volume 2146 of *Lecture Notes in Computer Science*, pages 51–66, 2001.

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

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

[17] 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.

[18] K. K. Parhi and D.G. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *Transactions on Computers*, 40(2):178–195, 1991.

[19] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *SIGMICRO Newsl.*, 12:183–198, December 1981.

[20] B. Ramskrishna Rau, Michael S. Schlansker, and P. P. Timmalai. Code generation schema for modulo scheduled loops. In *in Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 158–169, 1992.

[21] Hongbo Rong, Alban Douillet, and Guang R. Gao. Register allocation for software pipelined multidimensional loops. *ACM Trans. Program. Lang. Syst.*, 30(4):1–68, 2008.

[22] Michael Schlansker, Bob Rau, and Scott Mahlke. Achieving High Levels of Instruction-Level Parallelism with Reduced Hardware Complexity. Technical Report HPL-96-120, Hewlett Packard, 1994.

[23] *ST231 Core and Instruction Set Architecture Manual*, 2005.

[24] Sid-Ahmed-Ali Touati, Frederic Brault, Karine Deschinkel, and Benoît Dupont de Dinechin. Efficient Spilling Reduction for Software Pipelined Loops in Presence of Multiple Register Types in Embedded VLIW Processors. *ACM Transactions on Embedded Computing Systems*, 10(4), November 2011.

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

[26] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral code generation in the real world. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'06)*, LNCS 3923, pages 185–201, Vienna, Austria, March 2006. Springer-Verlag.

[27] J. Wang, C. Eisenbeis, M. Jourdan, and B. Su. Decomposed software pipelining: a new perspective and a new approach. *International Journal Parallel Programming*, 22(3):351–373, 1994.