Loop Unrolling Minimisation in the Presence of Multiple Register Types: a Viable Alternative to Modulo Variable Expansion

Mounira Bachir, Frédéric Brault, Sid Touati, Albert Cohen

Loop Unrolling Minimisation in the Presence of Multiple Register Types: a Viable Alternative to Modulo Variable Expansion

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

Frederic Brault
INRIA Saclay – Ile-de-France
Frederic.Brault@inria.fr

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

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

Abstract

Modulo Variable Expansion (MVE) [1] used with software pipelining (SWP) may sacrifice the register optimality (MAXLIVE) and in general may lead to unnecessary spills or move operations negating the benefits of SWP. In contrast, bigger loop unrolling can be performed to meet the MAXLIVE registers requirement [2, 3]. However, the degree of unrolling should be minimised to control code size and hence I-cache performance.

In our previous work, we designed a post-pass unrolling algorithm which minimises the unrolling degree while adjusting the length of reuse circuits through the usage of additional (free) registers [4]. In this paper, we complete our study with an improved algorithm for minimising kernel loop unrolling resulting from cyclic register allocation in the presence of multiple register types showing that considering all register types in conjunction provides a lower unrolling degree than considering each register type in isolation. In addition, we integrate our solution within a real world embedded system compiler: st200cc for the ST2xx family of VLIW embedded processors and compare it to MVE. Our large set of experiments on both high performance and embedded benchmarks (SPEC2000, SPEC2006, MEDIABENCH and FFMPEG) demonstrates the practical applicability and the benefits of our approach.

1. Introduction

Many high performance or soft real-time applications, such as telecom and image processing, exhibit intensive computations in loops. Software pipelining (SWP) is an important instruction scheduling technique for improving the execution rate of these applications, by exploiting the instruction-level parallelism (ILP) in loops [1, 5].

When a loop is software pipelined, we cannot use regular register allocation algorithms because of self-interferences in the usual interference graph [1, 3, 6]. In compiler construction, when no hardware support is available, kernel loop unrolling is *the* method of code generation that does not alter the initiation interval after software pipelining. In fact, unrolling the loop allows us to avoid introducing unnecessary move and spill operations after a periodic register allocation.

Our objective in this research effort is somewhat different: we are interested in the minimal loop unrolling factor which allows a periodic register allocation for software pipelined loops in the presence of multiple register types. We also focus on generating an embedded VLIW loop with reduced spill and code size, without necessarily reducing the execution speed. Unlike high performance and intensive computational fields, we are asked to generate a code with *equivalent* performance but with less memory operations. We believe that an important code quality criterion is to have a reduced amount of memory requests upon the condition of not altering ILP scheduling and execution speed if possible.

In a target architecture with multiple register types (a.k.a. classes), some of state-of-the-art algorithms [2, 3] propose to compute the *sufficient unrolling degree* that we should apply to the loop so that it is always possible to allocate the variables of each register type with a minimal number of registers (MAXLIVE [7]). However, periodic register allocation has long suffered from an important drawback: the resulting unrolling factor can be very high. We recently proposed a solution to dramatically reduce this cost when considering a single register type [4].

In this article, we extend our previous results in two directions:

1. We first provide an efficient algorithm for minimising loop unrolling in the presence of multiple register types, for instance, $T = \{int, float, branch\}$. The minimal loop unrolling degree is reached by exploiting the unused registers to adjust the different weights of reuse circuits generated for each register type, looking for a good distribution of those registers concurrently over all types.

Note that, the mathematical nature of the two problems are not equivalent (single register type vs. multiple register types). Indeed, if we have two or three types of registers, then we have two or three optimisation problems to solve [4]. This paper solves them simultaneously by enumerating the feasible unrolling degrees and searching for the minimal kernel loop unrolling among them.

2. Second, we implemented and integrated our code optimisation method within the industrial compiler of STMicroelectronics, allowing us to make extensive experiments on real world high performance and embedded applications.

This paper is organised as follows. Sect. 2 presents some relevant related work on code generation for periodic register allocation. Sect. 3 formalises the problem of minimising the loop unrolling degree in the presence of multiple register types. Sect. 4 details our algorithmic solution for minimising loop unrolling. Sect. 5 presents detailed experimental results on standard benchmarks (MEDIABENCH, SPEC2000, SPEC2006, STMicroelectronics benchmark), showing that our method is efficient in practice. Finally, we conclude.

2. Related Work

We detail in the following section two techniques that we often refer in this paper. For a comprehensive bibliography on register allocation problems we refer the reader to [8, 3] which contain an extensive list of contributions in the area.

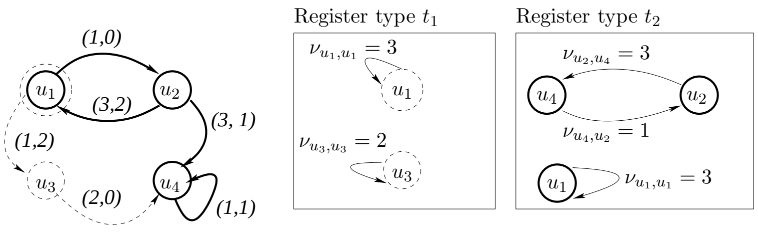
2.1 Modulo Variable Expansion

This method, proposed by Lam [1, 9], defines a minimal unrolling degree to permit code generation after a given periodic register allocation. This unrolling degree is obtained by dividing the length of the longest lifetime (LT) of all register types ($\max_{t\in T}(\max_v LT_{v,t})$) by the initiation interval $\frac{\max_{t\in T}(\max_v LT_{v,t})}{II}$. However, this method does not guarantee for each type t, a register allocation with a minimal number of registers equal to MAXLIVE, the number of values simultaneously alive for this register type. In general, as we will see in Sect. 5, it may lead to unnecessary spills or move operations negating the benefits of SWP. These extra spill or move operations may increase the initiation interval of the SWP.

2.2 SIRA Reuse Graphs

Reuse graphs are a generalisation of the work previously published by [10, 11] and are used inside a framework called SIRA [2]. Unlike the previous approaches for periodic register allocation, reuse graphs can be used before or after software pipelining to generate a move-free or a spill-free periodic register allocation. Reuse graphs give a formal method to generate code without spill and without move operations if loop unrolling is applied. A brief overview of reuse graphs is illustrated in Fig. 1. Fig. 1(a) shows an initial DDG with two register types t_1 and t_2. Statements which involve registers 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 arcs, and those of type t_2 are in bold arcs. We associate a reuse graph G_r^t with each register type t, see Fig. 1(b). A reuse graph G_r^t of register type t contains only the nodes writing inside registers of type t. These nodes are connected by *reuse arcs*. The existence of a reuse arc (u^t, v^t) of distance $\nu^t_{u,v}$ means that the two operations $u^t(i)$ and $v^t(i+\nu^t_{u,v})$ *share the same destination register*. Hence, reuse graphs completely define a periodic register allocation for a given loop, either before SWP (unscheduled loop) or after SWP (already scheduled loop). A formal theorem in [2] proves that the number of allocated registers of type t is equal to $R_{min,t} = \sum_{(u,v)} \nu^t_{u,v}$ if we unroll the loop with a factor equal to α_t. Fig. 1(b) represents a reuse graph that allocates $3+2=5$ registers of type t_1 and $3+1+3=7$ registers of type t_2.

Each register type t requires an unrolling factor α_t. If the reuse graph G_r^t contains multiple reuse circuits $C_1, \cdots, C_k$, then the weight of each reuse circuit is defined by $\mu_{i,t} = \sum_{(u^t,v^t)\in C_i} \nu^t_{u,v}$. The unrolling degree of type t is then equal to $\alpha_t = lcm(\mu_{1,t}, \cdots, \mu_{k,t})$. For instance, the unrolling degree of $G_r^{t_2}$ in Fig. 1(b) is equal to $\alpha_{t_2} = lcm(3+1, 3) = lcm(4,3) = 12$. Similarly, $\alpha_{t_1} = lcm(3,2) = 6$. The global unrolling degree that is valid for all register types concurrently is equal to $\alpha = lcm_{t\in T}(\alpha_t)$. For Fig. 1(b), $\alpha = lcm(\alpha_{t_1}, \alpha_{t_2}) = lcm(6,12) = 12$.



(a) Initial DDG

(b) Reuse Graphs for Register Types t1 and t2

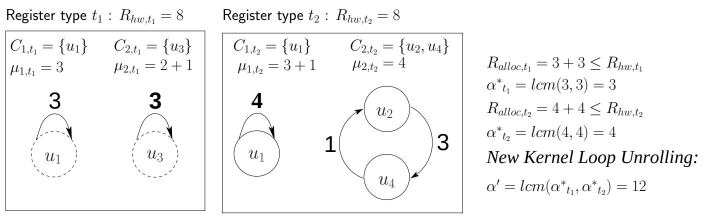
Figure 1. SIRA Reuse Graphs

3. Loop Unrolling Problem

Code generation methods using loop unrolling [2, 3] have long suffered from a drawback: the resulting unrolling factor α can be high. We recently proposed a solution to minimise the unrolling degree when considering a single register type [4]. However, in the presence of multiple register types, minimising the loop unrolling degree of each type seperately does not lead to the minimal loop unrolling degree for the whole loop. Fig. 2 illustrates an example. We want to minimise the loop unrolling degree of the initial reuse graph in Fig 1(b), where two register types t_1, t_2 are considered. The initial kernel loop unrolling degree $\alpha = 12$ is the LCM of

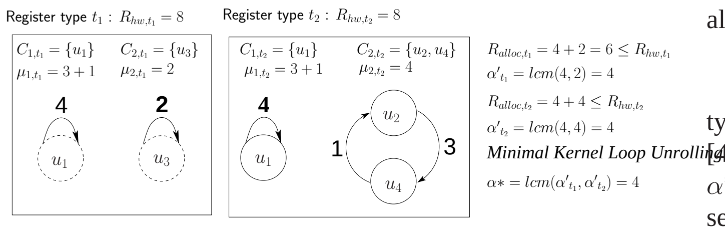
$\alpha_{t_1} = 6$ and $\alpha_{t_2} = 12$ which are respectively the LCM of the different reuse circuits weights for each register type. In this configuration, let assume that we have $R_{hw,t_j} = 8$ architectural registers in the processor for each register type t_j. Hence we have $R_{t_1} = R_{hw,t_1} - R_{min,t_1} = 8 - 5 = 3$ (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 [4] (Fig 2(a)), 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. However, the global kernel loop unrolling degree is not minimal $\alpha' = lcm(\alpha^*_{t_1}, \alpha^*_{t_2}) = 12$.

This paper describes how to find the minimal loop unrolling degree α^* for all register types concurrently. Our goal is to exploit the remaining registers of each register type, looking for a good distribution of those registers over all types. In Fig 2(b), the final loop unrolling degree found with this new method called *Loop Unrolling Minimisation* (LUM) is $\alpha^* = 4 < \alpha'$. 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. LUM guarantees that the new number of allocated registers will not exceed the number of architectural registers for each register type t_j; $R_{alloc,t_j} \leq R_{hw,t_j}$.



Register type t_1 : $R_{hw,t_1} = 8$
Register type t_2 : $R_{hw,t_2} = 8$
$C_{1,t_1} = \{u_1\}$
$\mu_{1,t_1} = 3$
$C_{2,t_1} = \{u_3\}$
$\mu_{2,t_1} = 2 + 1$
3
3
u_1
u_3
$C_{1,t_2} = \{u_1\}$
$\mu_{1,t_2} = 3 + 1$
$C_{2,t_2} = \{u_2, u_4\}$
$\mu_{2,t_2} = 4$
4
u_1
1
u_2
3
u_4
$R_{alloc,t_1} = 3 + 3 \leq R_{hw,t_1}$
$\alpha^*_{t_1} = lcm(3,3) = 3$
$R_{alloc,t_2} = 4 + 4 \leq R_{hw,t_2}$
$\alpha^*_{t_2} = lcm(4,4) = 4$
New Kernel Loop Unrolling:
$\alpha' = lcm(\alpha^*_{t_1}, \alpha^*_{t_2}) = 12$

(a) Minimising Loop Unrolling for Each Register Type Separately



Register type t_1 : $R_{hw,t_1} = 8$
Register type t_2 : $R_{hw,t_2} = 8$
$C_{1,t_1} = \{u_1\}$
$\mu_{1,t_1} = 3 + 1$
$C_{2,t_1} = \{u_3\}$
$\mu_{2,t_1} = 2$
4
2
u_1
u_3
$C_{1,t_2} = \{u_1\}$
$\mu_{1,t_2} = 3 + 1$
$C_{2,t_2} = \{u_2, u_4\}$
$\mu_{2,t_2} = 4$
4
u_1
1
u_2
3
u_4
$R_{alloc,t_1} = 4 + 2 = 6 \leq R_{hw,t_1}$
$\alpha'_{t_1} = lcm(4,2) = 4$
$R_{alloc,t_2} = 4 + 4 \leq R_{hw,t_2}$
$\alpha'_{t_2} = lcm(4,4) = 4$
Minimal Kernel Loop Unrolling:
$\alpha* = lcm(\alpha'_{t_1}, \alpha'_{t_2}) = 4$

(b) Minimising Loop Unrolling for all Register Types Conjointly

Figure 2. Modifying Reuse Graphs to Minimise Loop Unrolling Factor

In the next section, we give a formal description for minimising loop unrolling in the presence of multiple register types.

3.1 Loop Unrolling Minimisation Problem

PROBLEM 1 (LUM). *Let α be the initial loop unrolling degree and let $T = \{t_1, \ldots, t_n\}$ be the set of register types. For each register type $t_j \in T$, 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 generated reuse circuits. We assume $\mu_{i,t_j} \in \mathbb{N}$ is 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 additional registers r_{i,t_j} such that we find a new periodic register allocation with a minimal loop unrolling degree. This can be 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*
2. $\forall t_j \in T\ ,\ \sum_{i=1}^{k_j} r_{i,t_j} \leq R_{t_j}$

Problem 1 proposes to exploit the remaining registers of each type to adjust the weights of the different reuse circuits, looking for an optimal distribution of those registers concurrently over all types in order to find a new periodic register allocation with a minimal loop unrolling degree.

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

3.2 Search Space for Minimal Kernel Loop Unrolling

According to LCM properties 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 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 $t_j \in T$. By the way, $\alpha = lcm(\alpha_1, \ldots, \alpha_n)$. We define the subset T' which contains all the register types such that they have no remaining registers after periodic register allocation ($T' \subset T$ such that $T' = \{t \in T \mid R_t = 0\}$).

In fact, if there are no registers left for these register types, we cannot minimise their loop unrolling degrees [4]. Therefore, the minimal global loop unrolling degree $\alpha^* \geq \alpha_j\ \forall\ t_j \in T'$. By assuming $\alpha' = \text{lcm}_{t \in T'}(\alpha_t)$, 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 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 circuit).

Furthermore, if we assume that μ_{k_n,t_n} is the maximum weight of all the different circuits for all register types

($\mu_{k_n,t_n} = \max_{t_j} (\max_i \mu_{i,t_j})$) 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:

$S = \{\beta \in \mathbb{N} \mid \beta$ 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:

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

$\beta = \text{lcm}(\text{lcm}(\mu_{1,t_1} + r_{1,t_1}, \ldots, \mu_{k_1,t_1} + r_{k_1,t_1}), \ldots, \text{lcm}(\mu_{1,t_j} + r_{1,t_j}, \ldots, \mu_{k_j,t_j} + r_{k_j,t_j}), \ldots, \text{lcm}(\mu_{1,t_n} + r_{1,t_n}, \ldots, \mu_{k_n,t_n} + r_{k_n,t_n})) \wedge \sum_{i=1}^{k_j} r_{i,t_j} \leq R_{t_j}\}$

Here arises another problem: how to decide if the value β can be a potential new loop unrolling. A proposition for solving this problem is explained in the next section.

3.3 Fixed Loop Unrolling Problem

PROBLEM 2 (Fixed Loop Unrolling). *Let $\beta \in S$ be a fixed loop unrolling degree and let $T = \{t_1, \ldots, t_n\}$ be the set of register types. β can be a potential new loop unrolling iff we find for each register type $t_j \in T$, 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. *$\beta = lcm(lcm(\mu_{1,t_1} + r_{1,t_1}, \ldots, \mu_{k_1,t_1} + r_{k_1,t_1}), \ldots, lcm(\mu_{1,t_n} + r_{1,t_n}, \ldots, \mu_{k_n,t_n} + r_{k_n,t_n}))$*
2. *$\forall t_j \in T \quad \sum_{i=1}^{k_j} r_{i,t_j} \leq R_{t_j}$*

In order to determine if β can be the new kernel loop unrolling, we propose to generalise the *LCM-Problem* solution described in our previous work [4] for all register types. The constraints in Problem 2 are *LCM-Problem* constraints which must be checked for all the register types.

In general, the *LCM-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.

However, in the presence of multiple register types, the meaning is slightly different. β is, in fact, the least common multiple of the loop unrolling for all the register types. On the contrary, if we consider each register type separately, β is not necessarily the least commom multiple of its different updated reuse circuits weights, but a multiple of their least common multiple.

The solution of the *Fixed Loop Unrolling Problem* constitutes the basis of the solution for *Loop Unrolling Minimisation Problem* explained in the next section.

4. Solution for Minimal Loop Unrolling

In order to compute the minimal kernel loop unrolling α^*, our solution consists of 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 *Fixed Loop Unrolling Problem* and finally taking the minimal one, we describe in Fig. 3 an efficient way to find the minimal α^* depending on the construction of the set S. Fig. 3 also illustrates the different cases of the construction of the research space S. The value of each node represents a potential new loop unrolling degree and an edge between two nodes a, b ($a \rightarrow b$) means that $a < b$ and a dashed edge between two nodes means the order is unknown. The structure of the search space depends on the availablity of the different types of registers :

- *Case 1 (no registers left for all 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 the algorithm of Fixed Loop Unrolling Problem to each node of Fig. 3 until we find a solution or until we reach the last node α
- *Case 3: some registers left for all register types:* we traverse the set S in the same way as described in [4]. If we assume that $\mu = \mu_{k_n,t_n}$ (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 the solution of Fixed Loop Unrolling Problem to each node in turn until we find a value which satisfies the constraints of Problem 2 or until we arrive at the last line where $\beta = \alpha$. If the value β of the node i of the line j is a solution for the Fixed Loop Unrolling Problem, 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 minimal ($\alpha^* = \beta$). 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 min-

imise 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 $(\mu + r) \wedge (j+1) \times \mu \leq \beta \leq \alpha"\}$)

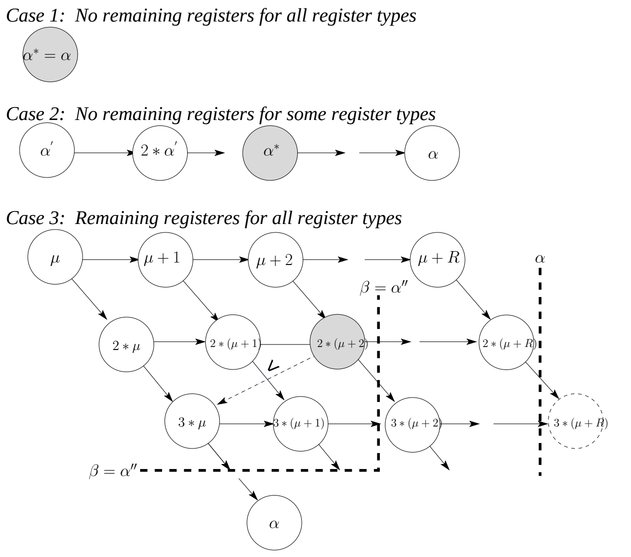


Figure 3. Loop Unrolling Values in the Search Space S

5. Experimental Results

In our study, we take the ST231 embedded VLIW processor as a target. It is currently the latest processor of the ST2xx family from STMicroelectroncis. ST231 is an integer 32 bits VLIW processor [12]. The size of the $L1$ I-cache and D-cache is 32 KB each. For our experiments, we consider the two register types of ST2xx, $T = \{general, branch\}$. The number of architectural registers of each register type is configured as follows: $R_{\text{arch,general}} = 32 \times 32$ bits , $R_{\text{arch,branch}} = 4\times$ 1-bit.

Our loop unrolling minimisation method is independent of the technique used for periodic register allocation. Consequently, it can be performed after any periodic register allocation technique. For the purpose of our study, we chose to develop a new version of SIRA [2] implementing a periodic register optimisation with loop unrolling minimisation in the presence of multiple register types. Then, we integrated our SIRA optimiser inside the st200cc compiler from STMicroelectroncis.

To study the efficiency of our loop unrolling minimisation, we conducted an extensive set of experiments on both high performance and embedded benchmarks: SPEC2000, SPEC2006, MEDIABENCH and ST FFMPEG internal benchmark; FFMPEG is a representative video multimedia application which is used with the ST231. The experiments were performed on close to 9000 software pipelined loops.

First, our experiments show that the run-time of our register allocation followed by loop unrolling minimisation is less than 1 second per loop on average (on a 3.4 GHz Pentium D). So, it is fast enough to be included inside an industrial cross compiler such as st200cc.

The following sections present our experiments. We focus on the benefit of our new method comapred to MVE (in terms of code size, spill reduction, move reduction, and II increase). In order to have an idea of the different results, we chose to graph them as boxplots, which are a convenient way of graphically depicting groups of numerical data through their five-number summaries: the smallest observations, lower quartile ($Q1 = 25\%$), median ($Q2 = 50\%$), upper quartile ($Q3 = 75\%$), and largest observations.

5.1 Statistics on Minimal Loop Unrolling Factors

Fig. 4 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 (FFMPEG, MEDIABENCH, SPEC2000, SPE2006), we note that the loop unrolling degree is reduced significantly from its initial value to its final value.

To highlight the improvements of our loop unrolling minimisation method, we show in Fig. 5 a boxplot for each benchmark family (FFMPEG, SPEC2000, SPEC2006, MEDIABENCH). We remark 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.

In order to compare our method with modulo variable expansion (MVE [1]), we also plot in Fig. 5 the loop unrolling degree obtained thanks to 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 does not exceed the number of architectural registers. This problem occurs in 254 loops of FFMPEG, 405 loops of MEDIABENCH, 1270 loops of SPEC2000 and 571 loops of SPEC2006.

With our loop unrolling register allocation method, we formally guarantee that we do not need, for each register type, more registers than the number of architectural registers. In addition, a suitable quality criterion is to check if the unrolled loops fit in the I-cache. Fortunately, our experimental results show that using loop unrolling technique, all the unrolled loops also fit in the I-cache, even if our unrolling factors are higher than those computed by MVE.

The next section demonstrates that loop unrolling minimisation is a better choice than MVE due to the reduction in spill code and move operations.

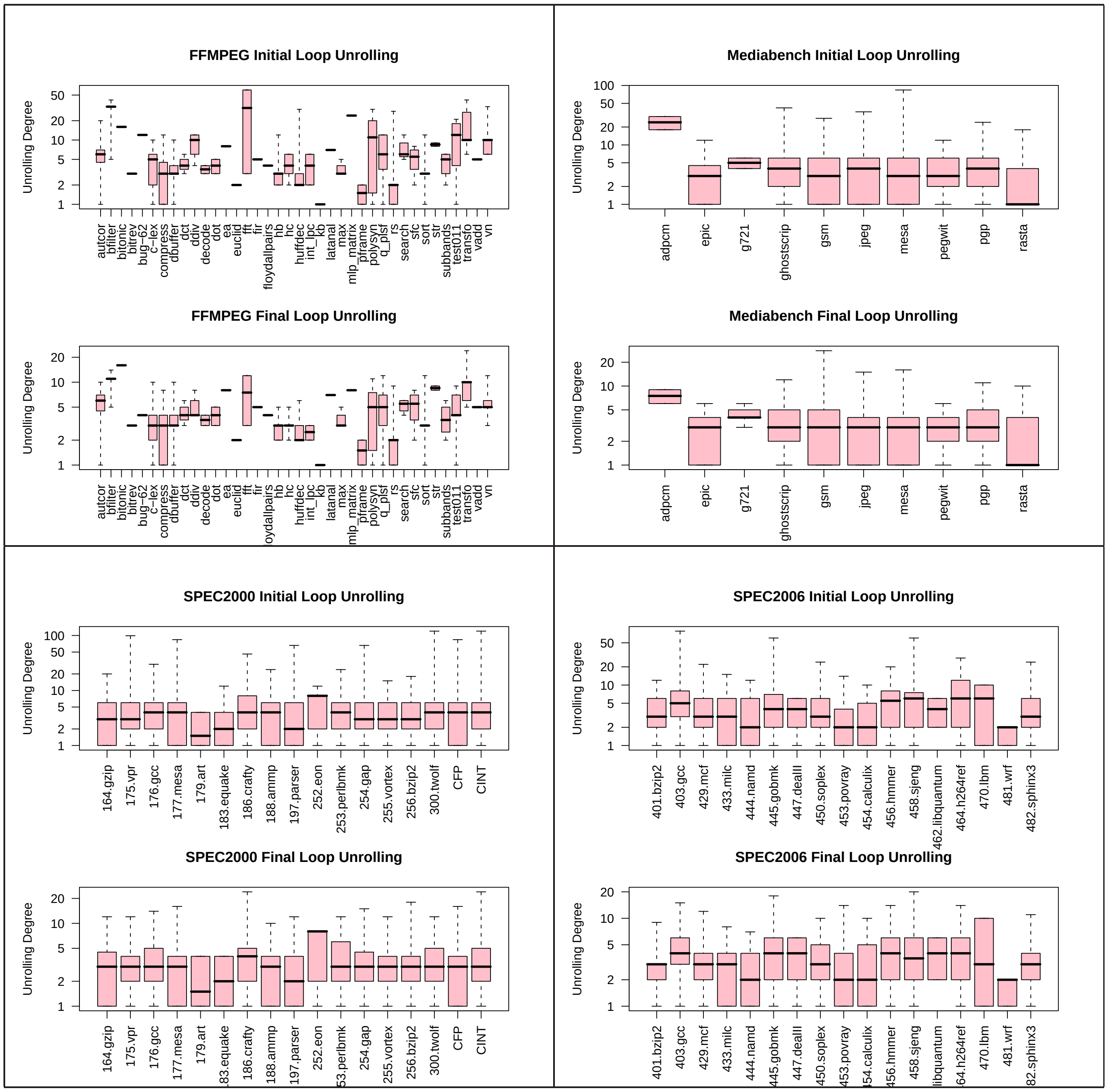


Figure 4. Statistics on Loop Unrolling Minimisation

5.2 Statistics on Spill Code Decrease and Move Operation Decrease

The spill code decrease is computed over all SWP loops. It is evaluated as the difference between the amount of spill code generated without using SIRA, *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 ($\frac{\sum Spill_{nosira} - \sum Spill_{sira}}{\sum Spill_{nosira}}$). The results show that including the new version of SIRA in the st200cc compiler greatly reduces the amount of spill. In fact, the final counts of spills in SWP loops are: FFMPEG=86, MEDIABENCH=100, SPEC2000=240, SPEC2006=111.

In addition, we did statistics on the amount of move operations reduced thanks to our method (on all SWP loops). The decrease in move operations is calculated as the difference between the number of the move operations generated without using SIRA, *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: $\frac{\sum Move_{nosira} - \sum Move_{sira}}{\sum Move_{nosira}}$. The experimental results show that in most cases, we reduced the number of move op-

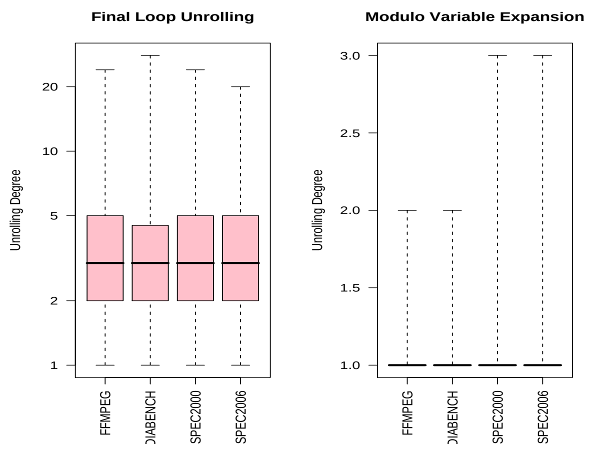


Figure 5. Loop Unrolling Minimisation versus MVE

erations by $10 - 20\%$: we reduce the amount of move operations by 18.92% for FFMPEG, 9.61% for MEDIABENCH, 14.17% for SPEC2000 and 18.21% for SPEC2006.

The way in which SIRA adds arcs to the DDG before software pipelining may in theory increase the critical circuits (minII) and modify ILP scheduling. This is studied in the next section.

5.3 Statistics on II increase

Our cyclic register allocations and loop unrolling minimisation are performed before SWP. It may be argued that introducing arcs inside DDGs before software pipelining would alter the ILP scheduling quality, since extra constraints are added. In practice, this is not the case because the usual software pipelining heuristics are not optimal. This means that even if we introduce additional arcs (either with or without increasing minII), this does not necessarily increase the II. The mean of II increase is resp. 1.56% for FFMPEG, 0.05% for MEDIABENCH, 1.66% for SPEC2000 and 0.09% for SPEC2006. As can be seen, the average of II increase is negligible in all benchmarks.

6. Conclusion

We presented, in this article, an efficient algorithm to minimise kernel loop unrolling resulting from the periodic register allocation in the presence of multiple register types. Our experiments on the embedded VLIW processor ST231 cover a wide range of high-performance and embedded benchmarks (FFMPEG, MEDIABENCH, SPEC2000, SPEC2006). These experiments demonstrate the practical applicability and the benefits of our approach: as our periodic register allocation is applied before software pipelining, it avoids the uncontrolled generation of spill code. Regarding the initiation intervals, the increase is negligible on average, and always less than 1.6% on average. Furthermore, thanks to loop unrolling minimisation, all the loops fit into the I-cache. The total amount of register `move` operations is reduced, yet still over 18% for FFMPEG and SPEC2006, 14% for MEDIABENCH and 9% for SPEC2000.

If code size is more critical, fitting the loops inside the I-cache may not always be satisfactory as a quality criteria. In this case, we have still some opportunities to reduce loop unrolling factors by exploiting `move` operations. This is our current research direction that may combine loop unrolling with `move` insertions.

References

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

[2] Touati, S., Eisenbeis., C.: Early periodic register allocation on ilp processors. Parallel Processing Letters **14**(2) (2004) 287–313

[3] Lelait, S.: Contribution à l'allocation de registres dans les boucles. PhD thesis, Université d'Orléans (January 1996)

[4] Bachir, M., Touati, S.A.A., Cohen, A.: 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, New York, NY, USA, ACM (2008) 141–150

[5] Rau, B.R., Schlansker, M.S., Timmalai, P.P.: Code generation schema for modulo scheduled loops. In: in Proceedings of the 25th Annual International Symposium on Microarchitecture. (1992) 158–169

[6] J. A. Fisher, P.F., Young, C.: Embedded Computing: a VLIW Approach to Architecture, Compilers and Tools. Morgan Kaufmann Publishers (2005)

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

[8] Touati., S.: Register Pressure in Instruction Level Parallelism. PhD thesis, Université de Versailles (2002)

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

[10] de Werra, D., Eisenbeis, C., Lelait, S., Marmol, B.: On a graph-theoretical model for cyclic register allocation. Discrete Appl. Math. **93**(2-3) (1999) 191–203

[11] Hendren, L.J., Gao, G.R., Altman, E.R., Mukerji, C.: A register allocation framework based on hierarchical cyclic interval graphs. In: CC '92: Proceedings of the 4th International Conference on Compiler Construction, London, UK, Springer-Verlag (1992) 176–191

[12] Faraboschi, P., Brown, G., Fisher, J.A., Desoli, G., Homewood, F.: Lx: a technology platform for customizable vliw embedded processing. SIGARCH Comput. Archit. News **28**(2) (2000) 203–213