A New Fast Algorithm for Optimal Register Allocation in Modulo Scheduled Loops

Sylvain Lelait, Guang R. Gao, Christine Eisenbeis

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A New Fast Algorithm for Optimal Register Allocation in Modulo Scheduled Loops

Sylvain Lelait, Guang R. Gao, Christine Eisenbeis

N° 3337

Janvier 1998

THÈME 1

ISSN 0249-6399

INRIA
ROCQUENCOURT

A New Fast Algorithm for Optimal Register Allocation in Modulo Scheduled Loops

Sylvain Lelait*, Guang R. Gao†, Christine Eisenbeis‡

Thème 1 — Réseaux et systèmes
Projet A3

Rapport de recherche n ° 3337 — Janvier 1998 — 26 pages

Abstract: In this paper, we focus on the register allocation phase of software pipelining. We are interested in optimal register allocation. This means that the number of registers used must be equal to the maximum number of simultaneously alive variables of the loop. Usually two different means are used to achieve this, namely register renaming or loop unrolling. As these methods have both drawbacks, we introduce here a solution which is a trade-off between inserting *move* operations and unrolling the loop.

We present a new algorithmic framework of optimal register allocation for modulo scheduled loops. The proposed algorithm, called U&M, is simple and efficient. We have implemented it in MOST (Modulo Scheduling Toolset). An experimental study of our algorithm on more than 1000 loops has been performed and we report a summary of the main results. This new algorithm, that combines loop unrolling and register renaming, performs consistently better than several other existing methods.

Key-words: register allocation, modulo scheduling, loop unrolling

(Résumé : tsvp)

This work was partially supported by a Lise-Meitner Stipendium from the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung).

* Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria, E-mail: sylvain@complang.tuwien.ac.at
† Department of Electrical and Computer Engineering, University of Delaware, 140 Evans Hall, Newark, DE 19716, USA, E-mail: ggao@eecis.udel.edu
‡ Christine.Eisenbeis@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : 01 39 63 55 11 - International : +33 1 39 63 55 11
Télécopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

Un nouvel algorithme rapide et optimal d'allocation de registres pour des boucles ordonnancées modulo

Résumé : Dans ce papier, nous nous concentrons sur la phase d'allocation de registres du pipeline logiciel. Nous nous intéressons à l'allocation de registres optimale. Cela signifie que le nombre de registres doit être égal au nombre maximal de variables en vie simultanément dans la boucle. Habituellement deux moyens sont employés pour atteindre ce but, le renommage de registres ou le déroulage de boucles. Étant donné que ces méthodes ont toutes les deux des désavantages, nous introduisons une solution qui est un compromis entre insérer des opérations de copie de registres et dérouler la boucle.

Nous présentons un nouveau cadre algorithmique pour l'allocation optimale de registres de boucles ordonnancées modulo. L'algorithme proposé, appelé U&M, est simple et efficace. Nous l'avons implémenté dans MOST (Modulo Scheduling Toolset). Une étude expérimentale de notre algorithme sur plus de 1000 boucles a été réalisée et nous en présentons les résultats principaux. Ce nouvel algorithme, qui combine le déroulage de boucles et le renommage de registres, donne de meilleurs résultats que plusieurs méthodes existantes.

Mots-clé : allocation de registres, ordonnancement modulo, déroulage de boucles

1 Introduction

Register allocation is very important for modulo loop scheduling (software pipelining) in high-performance architectures especially when an increasing level of instruction-level parallelism is exploited. Software pipelining is often performed in two phases: (1) first derive a schedule with a maximum computation throughput of a loop (i.e. minimize the initiation interval) under a given resource constraint, (2) then allocate registers for the derived schedule. In production compilers, the register allocation phase is usually performed using heuristics which attempt to minimize the cost of spilling under a given number of registers.

Our objectives in this paper are somewhat different: we are interested in optimal register allocation, i.e. minimize the number of registers required. We argue that this is an important problem for situations where information of the smallest number of registers is required. For example,

- When allocating registers interprocedurally it is beneficial to allocate a minimal number of registers to each procedure using such a solution. This reduces the amount of register saving required at procedure call time, and can also improve interprocedural register allocation [20].
- When performing global register allocation, it is often useful to do the allocation hierarchically, i.e. it is useful to know the minimum register budget needed for a particular code section (i.e. loops) as an input to the overall register allocation decision.

Optimal register allocation for modulo scheduled loops is known to be hard. We have presented and analyzed the difficulty in Section 2 with the discussion of several existing methods, e.g. Lam's Modulo Variable Expansion [12], Eisenbeis' method involving loop unrolling [7] (EJL) and the meeting graph heuristic [8] (MTG). A short discussion on related work is also included at the end of the paper and the readers can find more information in the citations in these sections. In short, the optimal solution to this problem often requires the insertion of "register moves" or loop unrolling. Brute force searching of the best solution has often a prohibitive cost, while existing fast heuristics may either sacrifice the register optimality or incur large unrolling overhead.

In this paper, we present a new method of optimal register allocation for modulo scheduled loops called U&M (for Unroll & Move), as it is a compromise between loop unrolling and the insertion of *move* operations. We note that, for a modulo scheduled loop, the lifetime of a loop variable often spans several iterations, but only at the portion corresponding to the last iteration — called the "fraction-of-an-iteration-interval" or *foai* a term coined in Altman's Ph.D thesis [1] — is there an opportunity of register sharing. The rest of the lifetime can be allocated to a "buffer" — a name coined by Ning and Gao [16] – implemented with a number of registers moves or with unrolling. A very simple and effective heuristic has been proposed to handle the fraction-of-an-iteration-intervals (foais) with a minimum unrolling degree.

We have implemented our algorithm in MOST (Modulo Scheduling Toolset [1]). An experimental study of our algorithm on more than 1000 loops from benchmarks such as the

Livermore loops, Nas, Spec92 or Linpack has been performed and we report a summary of the main results. For the benchmark programs we tested so far, our method performs consistently better than Lam's Modulo Variable Expansion method for the number of registers, and than Eisenbeis' method, and the meeting graph method for the unrolling degree of the loop.

The rest of this paper is organized as follows. In Section 2, we present the problem we are dealing with and the existing methods we mentioned. In Section 3, we define the main notions, present our method, and the algorithms we designed to compute an unrolling degree of the loop. In Section 4, we focus on the complexity of our algorithms and show that our method gives an optimal register allocation. In Section 5, we present experimental results, which show the effectiveness of our method. In Section 6, we mention some other related work, and finally we conclude.

2 A Motivating Example

In this section, we illustrate the problem of loop register allocation using the following example. Our discussion is in the context of modulo scheduling since it is most challenging to register allocation when parallelism between loop iterations is exploited.

```
LOOP
a[i+2] = b[i] + 1
b[i+2] = c[i] + 2
c[i+2] = a[i] + 3
ENDLOOP
```

In Section 2.1, we will first discuss the basic issues and trade-offs of loop register allocation using register moves or loop unrolling techniques on the running example above. In Section 2.2, we briefly compare how the several existing loop register allocation methods perform on the given example and illustrate where these methods may be subject to improvements.

2.1 Basic Issues and Trade-offs

There are two ways to deal with loop register allocation: using special architecture support such as *rotating registers*, or without using such support. The latter may require the insertion of register move instructions or loop unrolling. Although the focus of this paper is not on special architecture support, it may be helpful for understanding the issues and trade-offs to first describe the concept of the rotating register file using the given example.

2.1.1 Rotating Register File

We consider the allocation of variable a in registers. Since $a[i]$[1] spans three iterations (defined in iteration $i-2$ and used in iteration i), at least 3 registers are needed to carry simultaneously $a[i]$, $a[i+1]$ and $a[i+2]$. Furthermore, we should ensure that the generated code stays the same from iteration to iteration. That is, the register allocation should be cyclic over the whole loop – e.g. in this example, the same arithmetic operation on a should see the same register assigned from one iteration to the next. This can be accomplished through hardware mechanism (rotating register files) or through software means: register moving or unrolling.

The hardware mechanism - named rotating register file [4, 5] automatically performs the *move* operations at each iteration. At each iteration one pointer to the register file is automatically moved one location ahead. Below $R[k]$ denotes a register with offset k from R.

- Iteration i

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

- Iteration $i+2$

```
R[+2] = b[i+2]+1
b[i+4] = c[i+2]+2
c[i+4] = R + 3
```

Perhaps the effect of the rotating register is best illustrated by looking at the data dependence graph of this loop shown in Figure 1, where node S_i stands for instruction number i of the loop. Imagine that a rotating register R of size 3 is allocated on the arc between $S1$ and $S2$, carrying the "flow" of subsequent values of array a between the two instructions. R acts as a FIFO "buffer", and the value in R is automatically shifted accordingly. Therefore, the instructions in the generated code will see the same operand R, avoiding the explicit register copying.

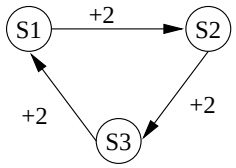

Figure 1: Data dependence graph of the loop

In this case the code size is not increased by unrolling or insertion of *move* operations. But you have to rely on this special hardware mechanism which does not exist in conventional microprocessor architectures.

[1] The notation $a[i]$ should not be understood like an array. It just express the fact that some variable a generated at iteration i is used some iterations later.

2.1.2 Register Move Instructions

One possibility, called register renaming, for allocating $a[i]$ is to use 3 registers and perform *move* operations at the end of each iteration [3, 15]: $a[i]$ is in register R1, $a[i+1]$ in register R2, $a[i+2]$ in the register R3. Then you must use *move* operations to shift the registers at every iteration:

```
LOOP
R3 = b[i]+1
b[i+2] = c[i]+2
c[i+2] = R1+3
R1 = R2
R2 = R3
ENDLOOP
```

Here, only the three registers used for array a are shown. The total registers requirement will be 9 if both b and c are also allocated to registers this way.

It is easy to see that if variable v spans d iterations, then you have to insert $d-1$ *move* operations at each iteration, but sometimes you may need one additional register and d *move*. This is likely to have a bad impact on the instruction schedule. As a matter of fact register *move* operations are usually performed by addition to 0. In addition, if adders are pipelined this results in a very bad code.

2.1.3 Loop Unrolling

Another option is to perform loop unrolling. Here different registers are used for the different instances of the variable. In our example shown below, the loop is unrolled three times, and $a[i+2]$ is stored in R1, $a[i+3]$ in R2, $a[i+4]$ in R3, $a[i+5]$ in R1, and so on. To express this, you have to write different code for each of the original three iterations in the unrolled loop body, since the register assignment scheme changes.

```
LOOP
R1 = b[i]+1
b[i+2] = c[i]+2
c[i+2] = R2+3
R2 = b[i+1]+1
b[i+3] = c[i+1]+2
c[i+3] = R3+3
R3 = b[i+2]+1
b[i+4] = c[i+2]+2
c[i+4] = R1+3
ENDLOOP
```

In this case, we avoid inserting extra *move* operations. The drawback is that the code size will be multiplied by 3 in this case, and by the unrolling degree in the general case. This

can have a dramatic impact on performance by causing unnecessary cache misses when the code size of the loop happens to be larger than the size of the instruction cache. Again, for simplicity, we did not expand the code to assign registers for b and c.

2.1.4 Useful Parameters

The impact of a loop register allocation scheme can be measured by 3 parameters. The first one is the number r of registers used. A inescapable lower bound for r is the maximal number of simultaneously alive variables, denoted as $MaxLive$ [11]. The register allocation is said to be optimal if it uses $MaxLive$ registers. The second one is the unrolling degree u. A large unrolling degree implies large code size and may cause instructions cache misses; u should therefore be as small as possible. The third one is the number m of extra *move* instructions per iteration. The impact of this parameter is hard to measure because it may sometimes be that the *move* instructions can be performed in parallel with the other operations. Analyzing this requires analyzing the loop schedule, which is beyond the scope of this paper.

2.2 Existing Methods: How Do They Perform on This Example ?

In this section we present several existing methods and their performance on the running example in terms of the three parameters just defined. This is in contrast to our approach for the same example as explained in the next section.

2.2.1 Lam's Algorithm for Modulo Variable Expansion: $m = 0$, min u

In her algorithm, also called Modulo Variable Expansion, Lam [12] finds the least unrolling degree that enables coloring. To achieve this purpose she computes the unrolling degree u by dividing the length of the longest live range by the number of cycles of the loop. In this example, the longest live range lasts 8 cycles, and the number of cycles of the loop is 3 cycles, so $u = \lceil \frac{8}{3} \rceil = 3$, and we should unroll three times. Then we can assign to each variable a number of registers equal to the least integer greater than the span of the variable that divides u. For our example, each variable a, b, c is assigned 3 registers - R1, R2, R3 for a, R4, R5, R6 for b, R7, R8, R9 for c, and the loop is unrolled 3 times.

$\boxed{m = 0, r = 9, u = 3}$

```
LOOP                          R8 = R3+3
R1 = R5+1                     R3 = R4+1
R4 = R8+2                     R6 = R7+2
R7 = R2+3                     R9 = R1+3
R2 = R6+1                     ENDLOOP
R5 = R9+2
```

One can verify that it is not possible to allocate on less than 9 registers when unrolling the loop 3 times. But this method does not ensure a register allocation with $MaxLive$

registers, and hence is not optimal. That is, as in this example $MaxLive = 8$, we may be able to use only 8 registers instead of 9. As we will see later, the round up to the nearest integer for choosing the unrolling degree may miss an opportunity for achieving an optimal register allocation.

2.2.2 Algorithms Minimizing the Register Requirements: $m = 0$, min r

There are several algorithms proposed to achieve an allocation with a minimum number of registers equal to $MaxLive$. The algorithm of Eisenbeis et al. [7] successfully allocates the minimal number of registers. Their method, however, does not control the unrolling degree at all. Another relevant approach is by Eisenbeis et al. [8]. This work is based on a new graph representation called "meeting graph" that accounts in the same framework for r and u. They are also able to allocate on $r = MaxLive$ registers, with a better u than EJL in general. The main drawback of that method is its time complexity [13]. For our example the meeting graph method obtains:

```
LOOP                R1 = R1+3           R4 = R4+3
R1 = R5+1           R4 = R5+1           R7 = R8+1
R4 = R8+2           R7 = R8+2           R2 = R3+2
R7 = R2+3           R2 = R2+3           R5 = R5+3
R2 = R6+1           R5 = R6+1           R8 = R1+1
R5 = R1+2           R8 = R1+2           R3 = R4+2
R8 = R3+3           R3 = R4+3           R6 = R6+3
R3 = R4+1           R6 = R7+1           ENDLOOP
R6 = R7+2           R1 = R2+2
```

$\boxed{u = 8, r = 8, m = 0}$

As you can see the loop unrolling degree u is much bigger in this case than the earlier solutions although the number of registers used is optimal. This can lead to instruction cache misses if the unrolled loop body becomes too big. Hence you can have two extreme solutions. The first one is to use *move* operations without loop unrolling. This may have a dramatic impact on the schedule. The other one is to use only loop unrolling. That may cause spurious instruction cache misses or even be impracticable due to some memory constraints, like in embedded processors. Our method combines both alternatives resulting on a lower unrolling degree and generally less *move* operations executed.

3 The U&M Method

This section presents our new method. In Section 3.1, we introduce it intuitively and show how it works on our example. Then in Section 3.3, we describe the algorithms more precisely.

3.1 Intuitive Idea of Our method

Our goal is to avoid a large unrolling degree while still achieving the use of a minimal number of registers. Our approach is based on two observations.

1. In the works presented in Section 2.2.1 that minimize the unrolling degree, the loss of registers comes from an over-approximation of the actual number of necessary registers. For instance, the loop we deal with is scheduled with $II = 3$ cycles and each variable is alive during 8 cycles. Under Lam's method 3 registers are allocated to each variable. But it really needs 2 registers for the 2 full II wrap around plus a fraction of $\frac{3}{8}$ of an iteration which may not actually need to occupy a register during a full iteration. Therefore one very delicate point for saving registers is how to capture and color these fraction-of-an-iteration intervals.

2. In the works that minimize the number of registers, large unrolling degrees are induced by the fact that a least common multiple – "lcm" – is computed. Roughly speaking, if you must unroll p times for one set of variables and q times for another one, then you have to unroll at least $lcm(p, q)$ times. However, it should be obvious that the registers allocated to the non-foai part of live ranges cannot be shared with others. Only the foai parts should be the candidates for coloring.

Based on these observations our register allocation method is performed with three phases:

- Phase 1 : Schedule the loop using a software pipelining algorithm.
- Phase 2: Allocate the remaining foai parts of the lifetimes into registers. Unrolling may be required in this step.
- Phase 3: Allocate the non-foai parts of all live ranges using an existing efficient method, interval graph coloring, without unrolling. Register moves may be used in this step.

Phase 2 aims at coloring the foais with an optimal number of colors. Thus unrolling may be necessary to reduce the number of registers, even if each interval spans less than one "turn" of II cycles, as it is the case in our current example. These intervals are then colored according to u, the computed unrolling degree. For allocating these foai lifetimes we have designed a new heuristic that takes advantage of the fact that no interval spans more than one iteration, and that usually such interval families do not need to be unrolled on more than one iteration to be colored optimally. Our algorithm will aggressively look for such an optimal coloring without using unrolling. Since this is a "common case" under cyclic interval graphs for foais derived in practice, our simple heuristic scores surprisingly well.

The buffers are then allocated to registers in Phase 3 according to the allocation of the foais during the second phase. The assignment of each buffer can be performed as follows. Assume a buffer b of size d, and the last turn is a fraction-of-an-iteration interval. Then, we

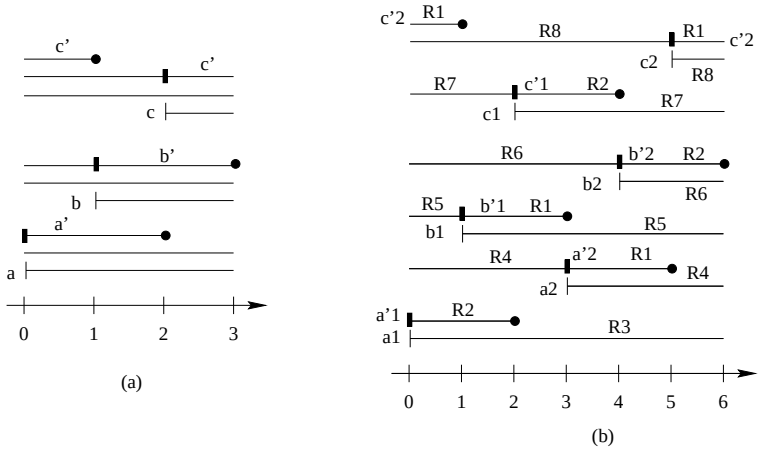

Figure 2: A lifetimes family and its register allocation with our method

allocate $(d-1)$ registers for b and his copies in the u-unrolled loop. The last foai and its instances from other iterations are assigned the registers derived from Phase 2.

So if we apply our method to the same example as the other methods, we obtain the following result. In Figure 2(a), variable lifetimes are depicted by intervals on a circle cut at the origin. Thus we have a line where the last point is equal to the first one. Each variable is alive during 8 cycles, this means 3 iterations as $II = 3$. Each variable is split into one 6 cycles interval and one 2 cycles interval with the latter being the foai. One can see it in Figure 2(a) where lifetime a is cut after 6 cycles and hence gives foai a'. The 3 foais a', b' and c' are allocated on R1 and R2, by unrolling the loop twice. The buffer part of a, (resp. b and c) are allocated on R3 and R4 (resp. R5 and R6, and R7 and R8). In theory 6 *move* per iteration should be inserted. However since each buffer is two iterations long and the loop is unrolled twice, this means that each of the corresponding lifetimes do not overlap with themselves and can be assigned to only one register. Thus we can alternatively assign the buffer parts to each register and avoid the extra *move* instructions. This way we obtain 3 *move* per iteration. The final register allocation is shown in Figure 2(b) and the final code is generated as shown below, where '$S_1; S_2$' denotes that S_1 and S_2 are executed in parallel. $\boxed{m = 3, r = 8, u = 2}$

```
LOOP
R3 = R2 + 1; R2 = R3
R5 = R1 + 2; R1 = R5
R7 = R2 + 3; R2 = R7
R4 = R1 + 1; R1 = R4
R6 = R2 + 2; R2 = R6
R8 = R1 + 3; R1 = R8
ENDLOOP
```

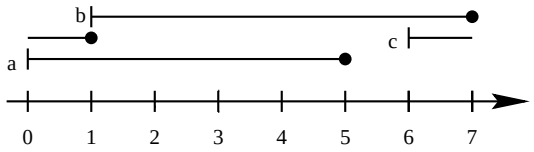

Figure 3: This interval family must be unrolled twice to be colored optimally

We can then summarize the results of the different methods as follows:

- Lam: $m = 0, r = 9, u = 3$.
- EJL and MTG: $m = 0, r = 8, u = 8$.
- U&M: $m = 3, r = 8, u = 2$.

Thus we can see that our algorithm does better than Lam's regarding the number of registers and the unrolling degree, and better than the loop unrolling methods EJL and MTG with regard to the computed unrolling degree. A reasonable number of *move* operations are introduced to achieve this result.

In summary, there are two novel aspects of our approach. First, the 3-phases strategy is new, which permits us the separation of the buffer register allocation from the foais, thus reducing the overall unrolling degree in general. Second, the method of register allocation of foais is itself novel, taking into consideration the features of the circular-arc graphs of foais – a topic of the next subsection.

3.2 Circular-Arc Graph Coloring Problem Is Hard

We deal with cyclic interval families of live ranges generating circular-arc graphs as interference graphs [10] for usual register allocation. In the sequel of this paper, the maximal width of an interval family I will also be noted r_I. It corresponds to the maximum number of lifetimes overlapping a point and is equal to $MaxLive$. Circular-arc graph q-coloring is known to be a polynomial problem, whereas finding the chromatic number of these graphs is an NP-complete problem [9] like a general coloring problem. Fortunately some efficient heuristics exist [10]. Below, we only briefly review with an example.

A typical situation is shown in Figure 3. It is known that although $r_I = 2$, the graph generated by I needs 3 colors without unwinding. Finding this 3-coloring is NP-hard in general, and some heuristic methods have been presented in [10]. This interval family must be unrolled on two iterations to be allocated with 2 registers.

However, we can try to unwind I into a number of u repetitions, and you may get it colored with r_I colors, the optimal you can do. So an interesting question is what is a reasonable value of u, i.e. how much do you need to unwind in order to get a minimum coloring, and how. Furthermore unrolling the loop on r_I iterations does not always ensure a register allocation with r_I registers [8]. Eisenbeis et al. [8] managed to give an upper bound

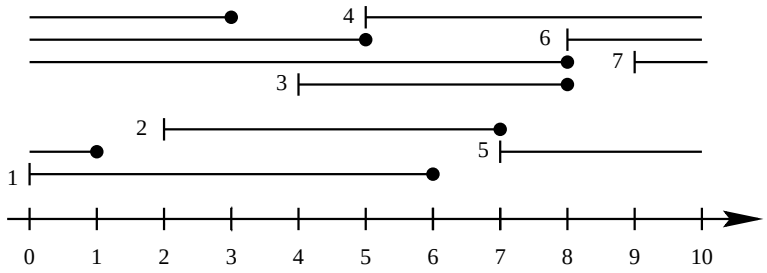

Figure 4: The set of intervals is composed of 2 disjoint interval sets.

of unrolling for any cyclic interval family. This bound is equal to $lcm(r_1, ..r_n)$, where r_i is the number of iterations spanned by a connected component of their meeting graph. Thus, determining the intervals belonging to them is a key in our problem. But we will not use the meeting graph as it can only be used on families of constant width. This feature obliges them to enlarge their graph which can lead to excessive computation time.

3.3 Coloring of FOAI lifetimes: Our Solution

In this section, we discuss our main algorithms for coloring interval graphs derived from foais. A useful concept used in our method is the *tight interval set*. The beginning and the end of an interval i of an interval family I on a circle $\mathcal{C}$ are denoted by $b(i)$ and $e(i)$. We have the set of the points which are not covered by interval i, $P(i) = \{p \in \mathcal{C}, p \notin i\}$ and the set of its endpoints $E(i) = \{b(i), e(i)\}$. So a tight interval set T is defined as $T = \{i \in I,\ \forall j \in T, P(i) \cap P(j) \neq \emptyset \vee E(i) \cap E(j) \neq \emptyset\}$. It contains intervals which either share an endpoint or do not cover at least one common point. Figure 4 shows an interval family composed of two tight interval sets, namely $\{1, 2, 5\}$ and $\{3, 4, 6, 7\}$.

In general, it is useful to decompose the tight interval sets further — as much as possible – in order to reduce the total unrolling degree required to achieve an optimal register allocation. There are many different ways for such decomposing. However, our experiments indicate that for an overwhelming majority of the interval graphs derived from foais in real loops (98.13% of 1394 real loops), there exists a decomposition with unrolling degree equal to 1 that achieves the optimal register allocation. Based on such an observation, we present a heuristic to aggressively decompose an interval family into subsets most of which possibly span at most one iteration.

In the following we assume that I is a set of cyclic intervals, each spanning only a fraction of an iteration. The graph associated to the interval family I will be colored using r_I colors with at most an unwinding factor u equal to the minimum between the *lcm* of the width of the tight interval subsets building I and the *lcm* of the width of the tight interval sets. The number of iterations spanned by a tight interval set T is noted $w(T)$. A tight interval set T can be decomposed in tight interval subsets $t_1, ...t_n$, whose number of iterations spanned are noted $w(t_i)$.

3.3.1 Greedy Heuristic

The greedy heuristic consists of two algorithms, one to find an unrolling degree of the foais, another to color the foais once they are unrolled. The aim of the first one is to find the greatest number of tight interval subsets spanning one iteration. This is in contrast to other algorithms such as MTG [8] which try to find a general optimal solution but do not focus on such special decompositions. The principle of Algorithm 1 is the following. We start with the first interval and take the next interval on the circle. If there is a choice, we take the one which allows to build a tight interval subset with the intervals already visited, or if it is not possible, the smallest one. Then we check if a tight interval subset t can be built with some of the intervals already visited; if several are possible we choose the one which spans the minimum of iterations ($w(t)$ minimum), and we remove t from our list and put its elements aside in the list C. We repeat this process until C contains all the intervals. Then we build the tight interval sets T_i and compute their weight $w(T_i)$. Finally we can compute the unrolling degree $u = \min(lcm(w(T_1), ..., w(T_m)), lcm(w(t_1), ...w(t_n)))$ and apply Algorithm 2 to color the foais.

Sometimes, it is not possible to find an optimal decomposition with unrolling degree equal to 1. In this case, our algorithm will still work and try to find an optimal solution at a higher unrolling degree. Example 1 shows how our algorithms work for a foai family when there are several tight interval subsets and an optimal decomposition exists with an unrolling degree equal to 1.

Example 1 In the case of the example of Figure 4, Algorithm 1 gives the following result: $C = \{1, 5, 2, 6, 4, 3, 7\}$ and:

- $T_1 = \{t_1\} = \{1, 5, 2\}$ with $w(t_1) = 2$
- $T_2 = \{t_2, t_3\} = \{\{7\}, \{6, 4, 3\}\}$ with $w(t_2) = 1$ and $w(t_3) = 2$.

Hence the unrolling degree computed is $u = lcm(2, 1, 2) = 2$. Then we can compute the 5-coloring on the 2 iterations. Below i_j represents lifetime i at iteration j.

- T_1
  - Color 1: $1_1, 5_1, 2_2$
  - Color 2: $1_2, 5_2, 2_1$
- T_2
  - Color 3: $7_1, 7_2$
  - Color 4: $6_1, 4_2, 3_1$
  - Color 5: $6_2, 4_1, 3_2$ $\diamond$

3.3.2 Hybrid Heuristic

In practice, we found that it is often useful to simplify and reduce the interval family before the application of our greedy algorithm. Hence, we have designed the following two-step *hybrid* method based on our greedy heuristic. This method is the U&M method.

Algorithm 1 The Circular Register Relay Algorithm

Require: a set I of fraction of an iteration intervals of maximal width r_I
Ensure: a cyclic register relay road-map C: an ordered sequence of nodes in $|I|$

1: Initialize the coloring sequence $C_p = \emptyset$, $C = \emptyset$.
2: Starting with the smallest leftmost interval x, let $I = I - \{x\}$
3: **while** $(I \neq \emptyset)$ **do** {Main loop which visits each interval once}
4: $x' = Next(x)$
5: $C_p = C_p + \{x'\}$
6: **if** $(end(x) \neq begin(x'))$ **then**
7: Check if $\exists\, y \in C_p$ such that $end(x) < begin(y) < begin(x')$ and such that $w(t(y, ..., x)) = \min_z(t(z, ..., x))$. If so remove $\{y, ..., x\}$ from C_p and add it to C.
8: **else** {Check if x' ends when a visited interval still in C_p begins}
9: **if** $(\exists\, y \in C_p,\ end(x') = begin(y))$ **then**
10: Remove $\{y, ..., x'\}$ from C_p and add it to C.
11: **end if**
12: **end if**
13: $I = I - \{x\}$
14: **end while**
15: **if** $(Cp \neq \emptyset)$ **then**
16: $ii = i$
17: **while** $(Cp \neq \emptyset)$ **do** {Loop which scans the remaining intervals in C_p}
18: **if** $(end(C_p(ii-1)) \neq begin(C_p(ii)))$ **then**
19: Check if $\exists\, y \in C_p$ such that $end(C_p(ii-1)) < begin(y) < begin(C_p(ii))$ and such that $w(t(y, , ..., C_p(ii-1))) = \min_z(t(z, ..., C_p(ii-1)))$. If so remove $\{y, ..., C_p(ii-1))\}$ from C_p and add it to C.
20: **end if**
21: $ii = ii - 1$
22: **end while**
23: **if** $(Cp \neq \emptyset)$ **then**
24: Remove $\{C_p(1), ..., C_p(k)\}$ from C_p and add them to C.
25: **end if**
26: **end if**
27: Build the tight interval sets and their T_i.
28: Return C

Algorithm 2 Coloring algorithm of the unrolled lifetimes

Require: A cyclic interval family I and a relay map C
Ensure: a r_I-coloring of the circular-arc graph G associated to I
1: Unwind I $u = \min(lcm(w(T_1), ..., w(T_m)), lcm(w(t_1), ..., w(t_n)))$ times
2: **for** $i = 1$ to r_I **do**
3: call $Relay(i, C)$ beginning at iteration i
4: **end for**
5:
6: Procedure Relay(i,C)
7: Get color number i
8: Relay i in the successive u iterations i.e. $i, i+1, ..u, 1, ..i-1$) according to C and the t_is (or T_is depending on the way u is computed) and color the intervals along the way.

The hybrid algorithm consists of two steps. First we simplify the interval family by pruning intervals using the first step of the so-called *fat-cover* heuristic of Hendren et al. [10]. A fat cover is a set of non-overlapping intervals covering all the "fat points", i.e. points covered by $MaxLive$ intervals, of the interval graph in one iteration. Hence each fat cover built corresponds to a tight interval subset which spans one iteration. Then we apply the greedy heuristic on the remaining intervals. By reducing the size of the interval family, we reduce the number of choices made by the greedy heuristic, hence leading to a better result. We present in Example 2 how the overall method works on a real loop.

Example 2 This example shows the complete process on a loop where the foai family needs to be unrolled twice. Figure 5 shows the lifetimes produced from the loop *ucbqsort-3* of the benchmark *Nasa7* of Spec92fp.

3 buffers are occupied entirely. They will be allocated to registers R1 for g, R2 for f and R3 for d. The foai family I is composed of the following intervals: a, b, c, d' (a piece of d), e, f' (a piece of f).

Let's describe the way the decomposition is obtained and the unrolling degree is computed according to Algorithm 1. We start with the smallest interval beginning at the origin, that is d', thus $C_p = \{d'\}$. Then we add c which follows immediately d', b which follows c and e which follows b, so $C_p = \{d', c, b, e\}$. As e ends when d' begins, we can build $t_1 = \{d', c, b, e\}$, with $w(t_1) = 2$; we update C, $C = \{d', c, b, e\}$ and C_p, $C_p = \emptyset$. Then we add a to C_p and f', $C_p = \{a, f'\}$. As we had to go over the beginning of a to add f', we can build a tight interval subset $t_2 = \{a\}$, with $w(t_2) = 1$. We update C and C_p, $C = \{d', c, b, e, a\}$ and $C_p = \{f'\}$. Finally we build the last tight interval subset, t_3, and update C and C_p. Hence $t_3 = \{f'\}$, with $w(t_3) = 1$, $C = \{d', c, b, e, a, f'\}$ and $C_p = \emptyset$.

Thus, we obtained only one tight interval set T with $w(T) = 4$, which has been divided in 3 tight interval subsets. We have the following: $t_1 = \{d',\ c,\ b,\ e\}$ with $w(t_1) = 2$, $t_2 = \{a\}$ with $w(t_2) = 1$, $t_3 = \{f'\}$ with $w(t_3) = 1$.

Hence, we have $u = \min(lcm(4), lcm(2,1,1)) = 2$, so we must unroll the foai family on 2 iterations to obtain a coloring with 4 colors using the decomposition with the t_is as it gives the lower *lcm*.

The coloring of the buffers is made according to the coloring of the pieces belonging to them. We check if a buffer and its foai can have the same color in order to lower the number of *move* instructions without changing the coloring of the other foais. We just have to insert *move* instructions to ensure the validity of the live range d. For f it is obvious that we don't need to insert this *move* since we can just allocate the same register for f_1, f_1' and f_2, f_2' to avoid it. This leads to the final allocation shown in Figure 5. The *move* instructions are depicted by thick dashes inside a live range. After scheduling, the loop would require the following *move* operations, number of registers and unrolling degree with the indicated method:

- Buffers and register renaming [16] : $m = 2, r = 9, u = 1$.
- Modulo Variable Expansion [12] : $m = 0, r = 8, u = 2$.
- U&M : $m = 1, r = 7, u = 2$.
- Loop unrolling [7]: $m = 0, r = 7, u = 4$.
- Loop unrolling [8] : $m = 0, r = 7, u = 6$.

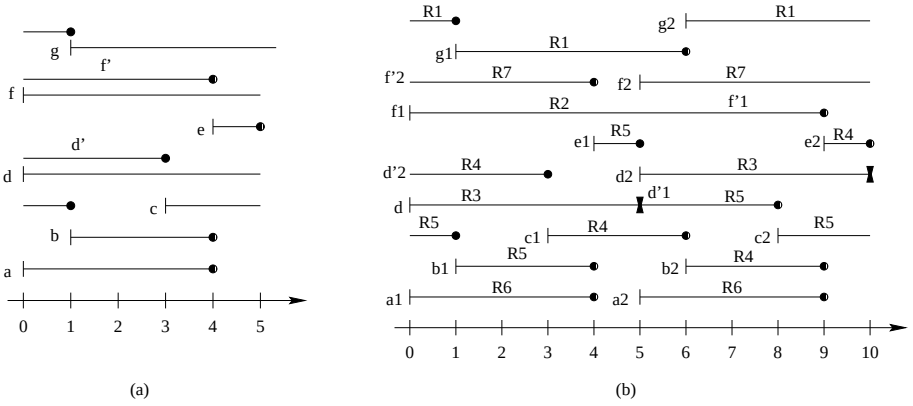

Figure 5: Allocation for the loop *ucbqsort-3* from the *Eqntott* benchmark with U&M

4 Some Theoretical Results

We present in this section some theoretical results about the complexity of our algorithms. These are both polynomial and ensure an optimal allocation for the foai family.

Lemma 1 *Algorithm 1 always terminates and returns a list C where each interval of I appears exactly once.*

Proof: Let the interval family I has a width r_I. We claim that a left-to-right sweep through one iteration of the main loop will reduce the maximal width uniformly at least by one. That is we can consider that the visited intervals have been removed from the graph when the sweep starts from the next iteration. At the end, all intervals will have been added to C_p. The last loop removes the remaining intervals from C_p into C. Hence C contains all the intervals only once. □

Lemma 2 *Algorithm 1 has a complexity $O(\log II(n + \log n))$.*

Proof: The main loop visits each interval only once and at each iteration the circle is partially scanned. This leads to a complexity of $O(\log II\, n)$ for the main loop. The second loop visits each remaining interval in C_p once and scans also partially the circle at each iteration leading to a complexity $O(\log II \log n)$. Hence the overall complexity of Algorithm 1 is $O(\log II(n + \log n))$. □

Now we present some results about the complexity of the coloring algorithm itself. This leads us to conclude that our method, which includes Algorithms 1 and 2, is of polynomial complexity.

Lemma 3 *Algorithm 2 will fully color the circular-arc graph G induced by the u-unrolled family I, which contains $u \times |I|$ intervals, with r_I colors.*

Proof: The interval family is duplicated $u = \min(lcm(w(T_1), ..., w(T_n), lcm(w(t_1), ..., w(t_m))$ times. Thus each t_i, resp. T_i, will require $w(t_i)$, resp. $w(T_i)$ colors for its intervals and their copies. As we have $\sum_{i=1}^{n} w(T_i) = \sum_{i=1}^{m} w(t_i) = r_I$, we will have a coloring with r_I colors. □

Lemma 4 *Algorithm 2 has a complexity of $O(u\, n)$.*

Proof: The unwound interval family has $u \times |I|$ intervals. Since it sweeps the family left-to-right exactly once, and at each point in the sweep, the cost of picking the next one is constant, the total cost is $O(u \times |I|)$. □

This is the best one can do since you have to color $u \times |I|$ intervals. Note that there is no claim that our method will do the minimum unfolding. Hence the optimal coloring of any graph associated with a cyclic interval family I, made only of fraction-of-an-iteration intervals, can be determined in polynomial time. Furthermore we are to able to compute the number of registers which will be used to allocate the whole loop. The following theorem gives the total number of registers used to allocate the whole loop, foais and buffers.

Theorem 1 *The register allocation of any loop without spilling with our method will require a number of registers equal to:*

$$\sum_{i=1}^{n} buffers_i - |I| + r_I$$

This achieves an allocation with an optimal number of registers.

Proof: A buffer-optimal scheduled loop will need $\sum_{i=1}^{n} buffers_i$ for the lifetimes occupying n buffers. In our case, the buffers will occupy only $r_{buffers} = \sum_{i=1}^{n} buffers_i - |I|$ registers since we will reallocate $|I|$ fraction-of-an-iteration intervals. As said before, the foais will occupy r_I registers. Hence the whole loop will need $\sum_{i=1}^{n} buffers_i - |I| + r_I$ registers for a valid allocation. Furthermore this is equal to $MaxLive$ as $MaxLive = r_I + r_{buffers}$. □

Finally the following theorem allows us to compute the maximum number of *move* operations inserted in the u-unrolled loop.

Theorem 2 *The number m' of* move *instructions inserted in the u-unrolled loop for a variable i spanning fully d iterations is at most:*

$$m' = \begin{cases} d - 1 + (f_i \times u) & \text{if } d > u \\ (d-1)(u \bmod d) + (f_i \times u) & \text{otherwise} \end{cases}$$

where $f_i = 1$ if i has a foai part, 0 otherwise.

Proof: We already know that the number of *move* operations inserted for a lifetime spanning d iterations is $d-1$. When the loop is unrolled u times and $d > u$, then the lifetime last longer than one iteration, and each of them requires $\lfloor \frac{d-1}{u} \rfloor$ *move* operations. Hence $d-1$ *move* instructions are necessary for the u lifetimes of the unrolled loop. Then we have u *move* for the u foai parts of the lifetimes if there is any. When $d \leq u$, the lifetimes does not last longer than one iteration, and we must allocate the u lifetimes of the unrolled loop with d registers, so u mod d lifetimes have to be allocated to several registers. For each of these lifetimes, $(d-1)$ *move* operations are necessary. The same as the previous case occurs for the foai parts. □

In this case, we have $m' = m \times u$, where m is the number of *move* operations per iteration of the original loop. For instance, if a variable spans 5 iterations and the unrolling degree we found is 2. Then if the original loop is executed 4 times, with the register renaming method (4×4) 16 *move* instructions will be executed, whereas with the U&M method only (2×4) 8 *move* instructions will be executed. We did not actually make any measurements on the number of *move* operations executed, but in some cases we should execute less and in some cases more instructions than register renaming methods.

5 Experimental Results

This section discusses the main experimental results. In Section 5.1, we present the way we conducted the experiments. In Section 5.2, we described the main results we obtain, and finally the whole results are presented and commented in Section 5.3.

5.1 The Experimental Testbed

We have implemented our new algorithm for loop register allocation in the MOST testbed [19], which was implemented at McGill University. It allows to compare several scheduling heuristics and is able to generate optimal pipelined loops. We also implemented outside MOST the heuristics MTG [8], EJL [7], and of Hendren et al. [10] to use our heuristics for computing an unrolling degree and also to test Lam's heuristic [12].

In our study, we used more than 1000 loops from several benchmarks, namely Spec92fp, Spec92int, Livermore loops, Linpack and Nas. We scheduled these loops with DESP [21].

5.2 Summary of the Main Results

We tested the efficiency of our new approach in terms of unrolling degree of the foai family and in terms of the total number of registers needed to allocate the loops. The main results are summarized as follows:

- Our method to compute an unrolling degree is better than the EJL heuristic [7] in general, and is almost always better than MTG [8] for finding the optimal unrolling degree when it is equal to 1. The unrolling degree found is always lower than if the whole loop had to be unrolled.
- The overall number of registers needed is always as good as, and sometimes better than, Lam's heuristic and achieves the optimal like MTG [8].
- Our heuristic to compute an unrolling degree is much faster than the MTG heuristic [8], and as fast as the EJL heuristic [7].

In summary, at run time our method will improve the overall register usage and introduce less spill code into loops when it is needed. Due to less unrolling the cache behavior will also be improved.

5.3 Detailed Experiments and Analysis

5.3.1 Unrolling degree

We compared the U&M heuristic with two other methods. The first one is the method of Eisenbeis et al. [7], noted EJL, which computes an unrolling degree by looking for the order of a permutation on the intervals necessary to obtain a valid coloring. The second one

developed by Eisenbeis et al. [8], noted MTG, introduces a new kind of graph and solves the problem by looking for a decomposition of this graph.

In Figure 6, greedy denotes our greedy heuristic to compute an unrolling degree in the foais, and U&M our hybrid heuristic. The first column represents the test, the second one represents the number of loops and the last one represents the percentage over the total of loops. The same for the second part of the figure.

MTG better than greedy	62	4.48 %	MTG better than U&M	5	0.36 %
MTG equal to greedy	1323	95.52 %	MTG equal to U&M	1378	99.5 %
MTG worst than greedy	0	0 %	MTG worst than U&M	2	0.14 %
greedy better than EJL	72	5.2 %	U&M better than EJL	83	5.99 %
greedy equal to EJL	1269	91.62 %	U&M equal to EJL	1297	93.65 %
greedy worst than EJL	44	3.18 %	U&M worst than EJL	5	0.36 %
U&M better than greedy	58	4.19 %			
U&M equal to greedy	1327	95.81 %			
U&M worst than greedy	0	0 %			

Figure 6: Comparisons between the heuristics MTG, EJL, our greedy and U&M heuristics

Figure 7 shows the related performance of each heuristic for each benchmark. We indicate the percentage of loops which required to be unrolled once, twice, three times or more.

Some observations:

- From Figure 6, the meeting graph heuristic gave almost always the best result, in only 2 cases over 1385 U&M was better.
- Our heuristic gave a better result than EJL in 5.99% of the cases, the same result in 93.65 % of the cases and a worst result in only 0.36 % of the cases.
- Our heuristic was worse than the meeting graph heuristic in only 5 cases (0.36%), which is a very good result.
- From Figure 7, we can see that between 91.84% and 100% of the loops need only to be unrolled by one iteration, 1.3 % need to be unrolled by 2 iterations. That is, it is always lower than the width of the foai family or the width of the whole interval family.
- Our heuristic is more efficient than the others methods to find the optimal unrolling degree when it is equal to 1. In fact, it gives a worst result than MTG in only one benchmark, *Appsp*.

This shows once more that our U&M heuristic has overall good performances over EJL, and is a bit less efficient in general than the meeting graph heuristic. Moreover we can see that most of the loops do not require to be unrolled (unrolling degree equal to 1). This is an advantage for the U&M heuristic that aggressively tries to find a decomposition which

Benchmark	MTG		U&M				EJL			
	1	2	1	2	3	> 3	1	2	3	> 3
Livermore	96.77%	3.23%	96.77%	3.23%			83.87%	16.13%		
Linpack	100%		100%				100%			
Spec92fp										
Alvinn	100%		100%				94.44%	5.56%		
Doduc	100%		100%				100%			
Ear	100%		100%				95.52%	4.08%		
Fpppp	100%		100%				100%			
Hydro2d	96.91%	3.09%	97.42%	2.58%			93.3%	3.09%	2.58%	1.03%
Mdljdp2	100%		100%				95.12%	4.88%		
Mdljsp2	100%		100%				75.00%	8.33%	8.33%	8.33%
Nasa7	97.87%	2.13%	97.87%	2.13%			85.1%	12.77%	2.13%	
Ora	100%		100%				100%			
Spice2g6	97.09%	2.91%	98.06%	1.94%			91.26%	1.94%	2.91%	3.89%
Su2cor	100%		100%				100%			
Tomcatv	100%		100%				77.78%	22.22%		
Wave	100%		100%				100%			
Spec92int										
Eqntott	91.18%	8.82%	94.12%	5.88%			82.35%	11.76%	5.88%	
Espresso	99.5%	0.5%	99.5%	0.5%			95.46%	3.03%	1.51%	
Gcc	100%		100%				94.78%	4.02%	0.4%	0.8%
Li	100%		100%				100%			
Sc	100%		100%				100%			
Nas										
Applu	94.81%	5.19%	94.8%	3.9%	1.3%		90.9%	3.9%	1.3%	3.9%
Appsp	91.84%	8.16%	90.82%	6.12%	1.02%	2.04%	85.72%	9.18%	1.02%	4.08%
Buk	100%		100%				100%			
Cgm	100%		100%				100%			
Mgrid	100%		100%				92.68%	2.44%	4.88%	

Figure 7: Comparison of performances about computing the unrolling degree of the foais between the heuristics EJL, MTG and our U&M heuristic

leads to an unrolling degree equal to 1. Hence in this case it is faster and more efficient than the meeting graph heuristic which is more "general purpose", it does not try to find an unrolling degree equal to 1, but only a low unrolling degree. Furthermore, as the foais are "sparse", we must enlarge the meeting graph much more than usual, and this degrades its performance. Finally, our method requires a smaller unrolling degree than the heuristics used previously in [7, 8] where they are applied on the whole live ranges of the loop variables.

5.3.2 Total number of registers used

We computed also the number of registers saved by this new method in comparison with the method of Lam [12] and MTG [8]. In Figure 8, we computed the average number of registers found by each heuristic per loop for each benchmark. We can see that our method, U&M, allocates always with the optimal number of registers like the MTG method. We

obtain always as good or better heuristic than Lam's algorithm. The gains are sometimes substantial like for *Fpppp*, *Applu* or *Appsp* where we gain between 1 and 2 registers in average for loops which need 25.72 registers in average.

Benchmark	# loops	average # reg. Lam	average # reg. MTG	average # reg. U&M
Livermore	28	20.61	19.54	19.54
Linpack	27	10.15	9.89	9.89
Spec92fp				
Alvinn	19	10.16	10.16	10.16
Doduc	22	13.50	13.32	13.32
Ear	53	9.47	9.28	9.28
Fpppp	17	23.47	22.47	22.47
Hydro2d	241	9.86	9.55	9.55
Mdljdp2	45	9.04	8.87	8.87
Mdljsp2	11	17.00	16.64	16.64
Nasa7	38	16.03	15.32	15.32
Ora	6	7.00	7.00	7.00
Spice2g6	98	9.41	9.18	9.18
Su2cor	9	5.78	5.78	5.78
Tomcatv	14	13.14	13.00	13.00
Wave	2	13.00	13.00	13.00
Spec92int				
Eqntott	35	8.69	8.46	8.46
Espresso	173	5.72	5.64	5.64
Gcc	256	6.80	6.74	6.74
Li	22	5.64	5.64	5.64
Sc	72	5.82	5.82	5.82
Nas				
Applu	75	27.19	25.49	25.49
Appsp	84	28.24	26.58	26.58
Buk	34	4.74	4.74	4.74
Cgm	20	5.10	5.10	5.10
Mgrid	51	6.76	6.76	6.76

Figure 8: Gains in registers with respect to Lam's heuristic and MTG

5.3.3 Execution Time of Our Method

We made also some execution time comparison in order to verify the timing of our approach. We chose to compare our timing result with EJL and the meeting graph method since these also try to minimize the loop unrolling degree. In Figure 9, the heuristics are labeled the same way as before, the execution times are given in seconds. For each benchmark, we only computed the average execution time for loops where it was measurable with the timing

routine we used on the experimental system. Only the time required for computing the unrolling degree has been measured, as the coloring time itself is the same in the three heuristics. As expected the meeting graph heuristic is more time consuming because of the unit intervals added to have a constant width. In some cases these new intervals slow down the heuristic dramatically. In comparison with EJL, ours is also as fast or even faster, and we are quite encouraged by this result, since hers is known to be very time efficient. The examples presented in Figure 9 are the biggest in term of lifetimes handled by MOST in the various benchmarks, and have between 30 and 55 lifetimes.

Benchmark	greedy	EJL	MTG	Benchmark	greedy	EJL	MTG
Livermore	0.003	0.027	100.2	Nasa7	0.005	0.067	770.54
Linpack	0.01	0.01	25.27	Spice2g6	0.00	0.043	5.18
Spec92fp				Spec92int			
Fpppp	0.005	0.05	7.04	Eqntott	0.00	0.03	19.32
Doduc	0.005	0.015	6.63	Gcc	0.00	0.04	2.26
Ear	0.005	0.015	11.74	Nas			
Hydro2d	0.005	0.04	26.53	Applu	0.005	0.06	0.9
Mdljdp2	0.005	0.075	287.42	Appsp	0.00	0.05	83.84

Figure 9: Execution times of the three heuristic on some examples

6 Related Work

In Section 2 we have already discussed several important contemporary works which are most related to this paper. These are works about Modulo Variable Expansion [12] and methods involving loop unrolling [7, 8].

Ning and Gao [16] only consider buffers to allocate the loop. Hendren et al. [10] can not handle lifetimes which are longer than one iteration.

Mangione-Smith et al. [14], Rau [18], Eichenberger and Davidson [6] presented some work related to register allocation and instruction scheduling, but they do not perform the allocation effectively and only predict the register requirements for a given schedule.

Rau et al. [17]also present some interesting work with some heuristics which work very well, but they mainly use hardware features like predicated execution and rotating register file [5], which are beyond the scope of this paper. Furthermore the only method presented which do not use these features is the Modulo Variable Expansion method. Bodík and Gupta [2] also present a method to do the register allocation for arrays that can also lower the number of *move* instructions inserted.

7 Conclusion

In this paper we proposed a novel way to optimize register allocation in loops, when a buffer optimal schedule has already been found. The original buffers are to greedy in registers, so we coalesce pieces of buffers into the same registers, after a possible step of loop unrolling, to minimize register use. Loop unrolling, another alternative to reduce register requirements, may decrease performance due to instruction cache misses. Our method is a trade-off between unrolling the scheduled loop body and register renaming, which still optimizes the number of registers needed.

We designed a heuristic for this purpose, and compared it with two others heuristics aimed at computing a loop unrolling degree. Our method combines the advantages of both loop unrolling and register renaming. Compared to unrolling the whole loop, the unrolling degree computed is lower, so we will have less problems with instruction cache management. In comparison with register renaming [3, 15], we will use less or as many *move* instructions between live range pieces as the loop will be unrolled to get an allocation with an optimal number of registers. The experimental results we obtained with MOST show that our heuristic is almost as efficient as MTG and is faster than the others heuristics. Furthermore the number of registers used is always equal to $MaxLive$ like other methods dealing with loop unrolling [7, 8]. we showed that our method is effective and gets an optimal result.

We plan to extend the method to compute the unrolling degree for general loops, where live ranges are alive during several iterations, we also intend to study the possibility of minimizing spill cost using our method. In addition we will measure the number of *move* operation executed.

References

[1] Erik R. Altman. *Optimal Software Pipelining with Function Unit Register Constraints.* PhD thesis, McGill University, Montréal, Canada, October 1995.

[2] Rastislav Bodík and Rajiv Gupta. Array Data-Flow Analysis for Load-Store Optimizations in Superscalar Architectures. In *Proceedings of the Eighth Annual Workshop on Languages and Compilers for Parallel Computing*, number 1033 in LNCS, pages 1–15, Columbus, Ohio, August 1995. Springer Verlag.

[3] Ron Cytron and Jeanne Ferrante. What's in a Name? or the Value of Renaming for Parallelism Detection and Storage Allocation. In Sartaj K. Sahni, editor, *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, University Park, Pennsylvanie, August 1987. London: Penn State press.

[4] James C. Dehnert, Peter Y.-T. Hsu, and Joseph P. Bratt. Overlapped Loop Support in the Cydra 5. In Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, pages 26–38, Boston, Massachusetts, 1989.

[5] J.C. Dehnert and R.A. Towle. Compiling for the Cydra 5. *Journal of Supercomputing*, 7(1/2), January 1993.

[6] Alexandre E. Eichenberger, Edward S. Davidson, and Santosh G. Abraham. Minimum Register Requirements for a Modulo Schedule. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 75–84, San Jose, California, November 30–December 2, 1994. ACM SIGMICRO and IEEE Computer Society TC-MICRO.

[7] Christine Eisenbeis, William Jalby, and Alain Lichnewsky. Compiler techniques for optimizing memory and register usage on the Cray-2. *International Journal on High Speed Computing*, 2(2), June 1990.

[8] Christine Eisenbeis, Sylvain Lelait, and Bruno Marmol. The Meeting Graph : a New Model for Loop Cyclic Register Allocation. In Lubomir Bic, Wim Böhm, Paraskevas Evripidou, and Jean-Luc Gaudiot, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT'95*, pages 264–267, Limassol, Cyprus, June 27–29 1995. ACM Press.

[9] M.R. Garey, D.S. Johnson, G.L. Miller, and C.H. Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM J. Alg. Disc. Meth.*, 1(2):216–227, June 1980.

[10] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. *The Journal of Programming Languages*, 1(3):155–185, September 1993.

[11] Richard A. Huff. Lifetime-Sensitive Modulo Scheduling. *SIGPLAN Notices*, 28(6):258–267, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[12] Monica S. Lam. Software Pipelining : An Effective Scheduling Technique for VLIW Machines. *SIGPLAN Notices*, 23(7):318–328, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[13] Sylvain Lelait. *Contribution à l'allocation de registres dans les boucles.* Thèse de Doctorat, Université d'Orléans, January 1996.

[14] W. Mangione-Smith, S. G. Abraham, and E. S. Davidson. Register Requirements of Pipelined Processors. In *Conference proceedings / 1992 International Conference on Supercomputing*, pages 260–271, Washington, DC, USA, July 19–23 1992. ACM Press.

[15] A. Nicolau, R. Potasman, and H. Wang. Register Allocation, Renaming and Their Impact on Fine-Grain Parallelism. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the Fourth international workshop on Languages and Compilers for Parallel Computing*, number 589 in LNCS, Santa Clara, California, August 7-9 1991. Springer Verlag.

[16] Qi Ning and Guang R. Gao. A Novel Framework of Register Allocation for Software Pipelining. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 29–42, Charleston, South Carolina, January 1993.

[17] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register Allocation for Software Pipelined Loops. *SIGPLAN Notices*, 27(7):283–299, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[18] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, San Jose, California, November 30–December 2, 1994. ACM SIGMICRO and IEEE Computer Society TC-MICRO.

[19] John Ruttenberg, G. R. Gao, A. Stouchinin, and W. Lichtenstein. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 1–11, Philadelphia, Pennsylvania, May 22–24, 1996.

[20] Peter A. Steenkiste and John L. Hennessy. A simple interprocedural register allocation algorithm and its effectiveness for Lisp. *ACM Transactions on Programming Languages and Systems*, 11(1):1–32, January 1989.

[21] Jian Wang, Christine Eisenbeis, Martin Jourdan, and Bogong Su. DEcomposed Software Pipelining: a New Perspective and a New Approach. *International Journal on Parallel Processing*, 22(3):357–379, 1994. Special Issue on Compilers and Architectures for Instruction Level Parallel Processing.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399