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

Sylvain Lelait[1], Guang R. Gao[2], and Christine Eisenbeis[3]

[1] Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8,A-1040 Wien, E-mail: sylvain@complang.tuwien.ac.at
[2] Department of Electrical and Computer Engineering, University of Delaware, 140 Evans Hall, Newark, DE 19716, USA, E-mail: ggao@capsl.udel.edu
[3] INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France, E-mail: Christine.Eisenbeis@inria.fr

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 modulo-scheduled loop body.
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. 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 performs consistently better than several other existing methods.

1 Introduction

Register allocation is very important for modulo loop scheduling in high-performance architectures especially when an increasing level of instruction-level parallelism is exploited. Software pipelining is often performed in two phases: first derive a schedule with a maximum computation throughput of a loop (i.e. minimize the initiation interval) under a given resource constraint, 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 on 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 [19]. 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 [11], Eisenbeis' method involving loop unrolling [6] (EJL) and the meeting graph heuristic [7] (MTG). In short, the optimal solution to this problem often requires the help of the insertion of "register moves" or unrolling the modulo-scheduled loop body. 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 unrolling the scheduled loop body 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 by Altman [1] – there is an opportunity of register sharing. The rest of the lifetime can be allocated to a "buffer" – a name coined by Ning and Gao [15] – implemented with a number of register moves or with unrolling.

We have implemented our algorithm in MOST (Modulo Scheduling Toolset [1]). An experimental study of our algorithm on more than 1000 loops from several benchmarks 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 EJL, and MTG 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 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 U&M 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 example shown in Figure 1(a). Our discussion is in the context of modulo scheduling in order to exploit parallelism between loop iterations. 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. 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.

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

    (a)                     (b)                          (c)
```

Fig. 1. (a) Original loop, (b) allocated using moves, (c) allocated using unrolling

2.1 Basic Issues and Trade-Offs

We focus here on software means to deal with loop register allocation. In the loop of Figure 1(a), $a[i]$ is alive during 3 iterations. 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]: $a[i]$ is in register R1, $a[i+1]$ in R2, $a[i+2]$ in R3. Then you must use *move* operations to shift the registers at every iteration, as shown in Figure 1(b). 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, especially in the sequential case when you need to store temporary variables like in Figure 1(b), you may need one additional register and d moves. This is likely to have a bad impact on the instruction schedule.

Another option is to perform loop unrolling. Here different registers are used for the different instances of the variable. In our example shown in Figure 1(c), 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. 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.

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 $MaxLive$ [10]. 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 ?

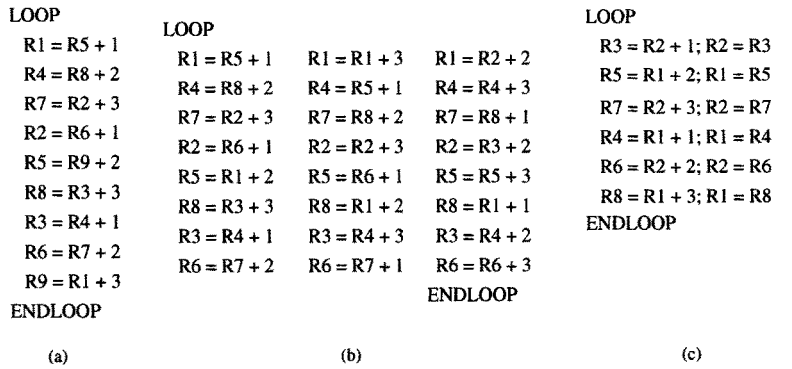

Fig. 2. Loop allocated following (a) Lam's heuristic, (b) meeting graph heuristic, (c) our U&M method

We present here several existing methods and their performance on the running example in terms of the three parameters just defined.

In her algorithm, also called Modulo Variable Expansion, Lam [11] 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 $u = \lceil \frac{8}{3} \rceil = 3$. 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. The allocation can be seen in Figure 2(a). $\boxed{m = 0, r = 9, u = 3}$

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.

There are several algorithms proposed to achieve an allocation with a minimum number of registers equal to $MaxLive$. The algorithm of Eisenbeis et al. [6] successfully allocates the minimal number of registers, that is $MaxLive$. Their method, however, does not control the unrolling degree at all. Another relevant work is by Eisenbeis et al. [7]. This approach is based on a new graph presentation called "meeting graph" that accounts in the same framework for r and u. The graph represents the succession of the intervals along the circle, its decomposition into circuits gives a bound of unrolling. 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 [12]. For our example the MTG heuristic obtains the allocation shown in Figure 2(b). $\boxed{m = 0, r = 8, u = 8}$

We can see that 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 loops 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 3.2 we describe the algorithm 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 optimal number of registers. Our approach is based on two observations. First, in the works presented in Section 2.2 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 foais. Second, in the works that minimize the number of registers, large unrolling degrees are induced by the fact that a least common multiple is computed.

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. 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.

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

If we apply our method to the same example as the other methods, we obtain the following result. In all our figures, 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. The end of a lifetime is depicted by a small circle. In Figure 3(a), 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. Therefore 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. But a simple optimization by permuting registers allows us to use only 3 *move* per iteration. The final register allocation is shown in Figure 3(b) and the final code is generated as shown in Figure 2(c) , where '$S_1; S_2$' denotes that S_1 and S_2 are executed in parallel, which is possible with some processors. $\boxed{m = 3, r = 8, u = 2}$

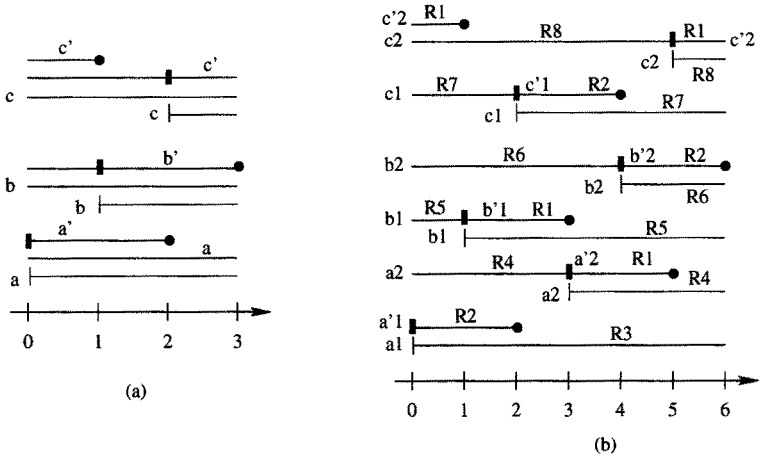

Fig. 3. A lifetimes family and its register allocation with our method

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 in 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 Coloring of FOAI Lifetimes: Our Solution

We deal with cyclic interval families of live ranges generating circular-arc graphs as interference graphs [9] for usual register allocation. In the sequel of this paper, the maximal width of an interval family I will 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 [8] like a general coloring problem. Fortunately some efficient heuristics exist [9].

However, we can try to unwind I into a number of u repetitions, and we may get it colored with r_I colors, the optimal we can do. So an interesting question is how much do we need to unwind in order to get a minimum coloring. Furthermore unrolling the loop on r_I iterations does not always ensure a register allocation with r_I registers [7]. An upper bound of unrolling for any cyclic interval family exists equal to $lcm(r_1, ..r_n)$, where r_i is the weight of a connected component of the meeting graph, has been defined in [7].

Therefore we introduce the concept of *tight interval set*. The beginning and the end of an interval i of an interval family I on a circle C are denoted by $b(i)$ and $e(i)$. We have the set of the points which are not covered by interval i, $P(i) = \{p \in C, p \notin i\}$ 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. If we only consider intervals a and b in Figure 3(a), we have 2 tight interval sets, then if we add c we have only 1 tight interval set.

In general, it is useful to decompose the tight interval sets as much as possible in order to reduce the total unrolling degree required to achieve optimal register allocation. There are many different ways for such decomposing. However, our experiments indicate that for a majority of the circular-arc graphs derived from foais in real loops (98.13% of 1394 loops), there exists a decomposition with an 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 $w(t_i)$.

We have designed the following two-step method. First we simplify the interval family by pruning intervals using the first step of the so-called *fat-cover*

heuristic of Hendren et al. [9]. A fat cover is a set of non-overlapping intervals covering all the "fat points" of the circular-arc graph in one iteration. Hence each fat cover built corresponds to a tight interval subset which spans one iteration. Then we apply a 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.

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 $w(t(y, \ldots, x)) = \min_z(t(z, \ldots, x))$. 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 $w(t(y, \ldots, C_p(ii-1))) = \min_z(w(t(z, \ldots, C_p(ii-1))))$. 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

The greedy heuristic consists of two algorithms, one to find an unrolling degree of the foais, another one 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 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 build a tem-

porary tight interval C_p subset by adding intervals in the order they come on the line. We break C_p in two cases, when an encountered interval ends where an already visited interval begins, and when an already visited interval begins in a gap in C_p. The tight interval subset found is then added to C. We do this until C contains all the intervals, then we build the tight interval sets T_i and compute their weights $w(T_i)$. 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 another simple algorithm [13] to cyclically color the foais. We present in Example 1 how the overall method works on a real loop.

Example 1. This example shows the complete process on a loop where the foai family requires to be unrolled twice. Figure 4 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 4. 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 [15] : $m = 2, r = 9, u = 1$.
- Modulo Variable Expansion [11] : $m = 0, r = 8, u = 2$.
- U&M : $m = 1, r = 7, u = 2$.
- Loop unrolling [6]: $m = 0, r = 7, u = 4$.

– Loop unrolling [7] : $m = 0, r = 7, u = 6$.

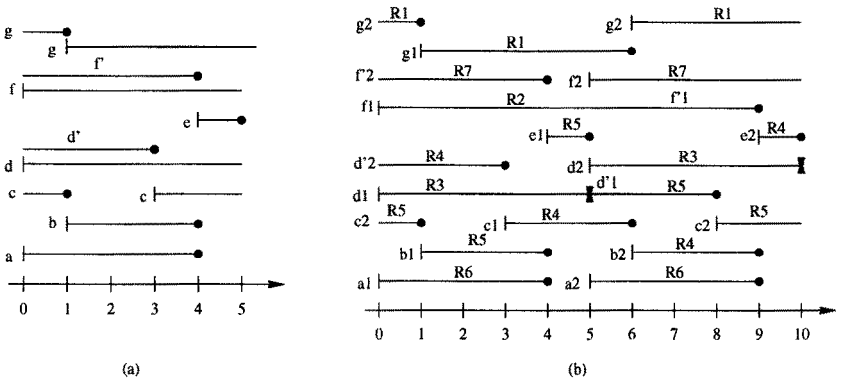

Fig. 4. Allocation for the loop *ucbqsort-3* from the *Eqntott* benchmark with U&M

4 Some Theoretical Results

We present in this section some theoretical results about the complexity of our algorithm and the efficiency of our method.

Algorithm 1 always terminates and returns a list C where each interval of I appears exactly once. Its complexity is polynomial in $O(\log II(n + \log n))$. The algorithm used to finally allocate the colors has also a polynomial complexity [13]. Note that there is no claim that our method will do the minimum unfolding.

Hence the optimal coloring of any graph associated to a cyclic interval family I made only of fraction-of-an-iteration intervals can be determined in polynomial time. Furthermore we are 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. Due to lack of space, the proofs of the theorems can be found in [13].

Theorem 2. *The register allocation of any loop without spilling with the U&M method will require a number of registers r equal to:*

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

This achieves an allocation with an optimal number of registers.

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

Theorem 3. *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.

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 and the main results we obtained. And finally the whole results are presented and commented in Section 5.2.

5.1 Summary of the Experiments

We have implemented our new algorithm for loop register allocation in the MOST testbed [18], 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, EJL, and those of Hendren et al. [9] to use our heuristics for computing an unrolling degree and also to test Lam's heuristic [11]. 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 [20]. In each table, we only present results where the heuristics did find different results, complete results can be found in [13].

We tested the efficiency of our new approach in terms of unrolling degree of the foai family and the total number of registers needed to allocate the loops. Our method to compute an unrolling degree is better than EJL in general, and is almost always better than MTG 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. Our heuristic to compute an unrolling degree is much faster than MTG, and as fast as EJL.

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.

5.2 Detailed Experiments and Analysis

We compared U&M with EJL and MTG. In Figure 5, 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. Figure 6 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.

From Figure 5, we can see that 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 6, 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*.

U&M better than EJL	83	5.99 %	MTG better than U&M	5	0.36 %
U&M equal to EJL	1297	93.65 %	MTG equal to U&M	1378	99.5 %
U&M worst than EJL	5	0.36 %	MTG worst than U&M	2	0.14 %

Fig. 5. Comparisons between the heuristics MTG, EJL and U&M

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%		
Spec92fp										
Alvinn	100%		100%				94.44%	5.56%		
Ear	100%		100%				95.52%	4.08%		
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%	
Spice2g6	97.09%	2.91%	98.06%	1.94%			91.26%	1.94%	2.91%	3.89%
Tomcatv	100%		100%				77.78%	22.22%		
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%
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%
Mgrid	100%		100%				92.68%	2.44%	4.88%	

Fig. 6. Results of the heuristics EJL, MTG and U&M for computing u

This shows once more that U&M has overall good performances over EJL, and is a bit less efficient in general than MTG. 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 U&M that aggressively tries to find a decomposition which leads to an unrolling degree equal to 1. Finally, our foai based method requires a smaller unrolling degree than the heuristics used previously in [6, 7] where they are applied on the whole live ranges of loop variables.

We computed also the number of registers saved by this new method in comparison with the method of Lam [11] and MTG. In Figure 7, we computed the average number of registers found by each heuristic per loop for each benchmark. We can see that U&M allocates always with the optimal number of registers like MTG. We obtain always as good or better results than Lam's algorithm. The gains are sometimes substantial like for *Fpppp*, *Applu* or *Appsp* where we gain between 1 and 2 registers 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				
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
Spice2g6	98	9.41	9.18	9.18
Tomcatv	14	13.14	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
Nas				
Applu	75	27.19	25.49	25.49
Appsp	84	28.24	26.58	26.58

Fig. 7. Gains in registers with respect to Lam's heuristic and MTG

We made also some execution time comparison in order to verify the timing of our approach, these are reported in [13]. The results show that our heuristic is faster than MTG and as fast as the EJL heuristic.

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 [11] and methods involving loop unrolling [6, 7].

Ning and Gao [15] only consider buffers to allocate the loop. Hendren et al. [9] can not handle lifetimes which are longer than one iteration.

Mangione-Smith et al. [14], Rau [17], Eichenberger and Davidson [5] 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. [16] also present interesting work on code generation strategies with register allocation heuristics which work very well, but they mainly use hardware

features like predicated execution and rotating register file [4], 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] present also 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. 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], we use less or as many *move* instructions between live range pieces. The experimental results we obtained with MOST show that our heuristic is almost as efficient as MTG. Furthermore the number of registers used is always equal to $MaxLive$ like other methods dealing with loop unrolling [6, 7].

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.

Acknowledgments

We would like to thank the referees for their valuable comments and remarks, as well as David Gregg, who helped us to improve the readability of this paper. S. Lelait was supported by a grant from the Austrian Science Foundation (FWF).

References

1. Erik R. Altman. *Optimal Software Pipelining with Function Unit Register Constraints.* PhD thesis, McGill University, Montréal, Canada, October 1995.
2. R. Bodík and R. 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. R. Cytron and J. Ferrante. What's in a Name? or the Value of Renaming for Parallelism Detection and Storage Allocation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, University Park, Pennsylvanie, August 1987. London: Penn State press.
4. J.C. Dehnert and R.A. Towle. Compiling for the Cydra 5. *Journal of Supercomputing*, 7(1/2), January 1993.

5. A.E. Eichenberger, E.S. Davidson, and S.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.
6. Ch. Eisenbeis, W. Jalby, and A. Lichnewsky. Compiler techniques for optimizing memory and register usage on the Cray-2. *International Journal on High Speed Computing*, 2(2), June 1990.
7. Ch. Eisenbeis, S. Lelait, and B. Marmol. The Meeting Graph: a New Model for Loop Cyclic Register Allocation. In *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.
8. 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.
9. L.J. Hendren, G.R. Gao, E.R. Altman, and C. Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. *The Journal of Programming Languages*, 1(3):155–185, September 1993.
10. 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.*
11. 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.*
12. Sylvain Lelait. *Contribution à l'allocation de registres dans les boucles.* Thèse de Doctorat, Université d'Orléans, January 1996.
13. S. Lelait, G.R. Gao, and Ch. Eisenbeis. A New Fast Algorithm for Optimal Register Allocation in Modulo Scheduled Loops. Research Report, INRIA, 1998.
14. W. Mangione-Smith, S.G. Abraham, and E.S. Davidson. Register Requirements of Pipelined Processors. In *Proceedings of the 1992 International Conference on Supercomputing*, pages 260–271, Washington, DC, July 19–23 1992. ACM Press.
15. Q. Ning and G.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.
16. 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.*
17. B.R. 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.
18. J. 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.
19. P.A. Steenkiste and J.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.
20. J. Wang, Ch. Eisenbeis, M. Jourdan, and B. Su. DEcomposed Software Pipelining: a New Perspective and a New Approach. *International Journal on Parallel Processing*, 22(3):357–379, 1994.