Fine-grain Scheduling under Resource Constraints

Paul Feautrier*
Laboratoire PRiSM,
Université de Versailles St-Quentin
45 Avenue des Etats-Unis
78035 VERSAILLES CEDEX FRANCE

September 15, 1994

Abstract

Many present-day microprocessors have fine grain parallelism, be it in the form of a pipeline, of multiple functional units, or replicated processors. The efficient use of such architectures depends on the capability of the compiler to schedule the execution of the object code in such a way that most of the available hardware is in use while avoiding so-called *dependences*. In the case of one simple loop, the schedule may be expressed as an affine form in the loop counter. The coefficient of the loop counter in the schedule is the initiation interval, and gives the mean rate at which loop bodies may be executed. The dependence constraints may be converted to linear inequalities in the coefficients of a closed form schedule, and then solved by classical linear programming algorithms. The resource constraints, however, translate to non-linear constraints. These constraints become linear if the initiation interval is known. This leads to a fast searching algorithm, in which the initiation interval is increased until a feasible solution is found.

Résumé

Presque tous les microprocesseurs actuels utilisent du parallélisme à grain fin, que ce soit sous la forme d'un ou plusieurs *pipeline*, de multiples unités fonctionnelles, ou de multiples processeurs. Ces architectures ne peuvent être utilisées efficacement que si le compilateur est capable d'organiser l'exécution du code objet de façon a utiliser au mieux le matériel disponible tout en respectant les dépendances de données. Dans le cas d'une boucle simple, l'ordonnancement peut être spécifié à l'aide d'une fonction affine du compte-tour, qui donne la date de lancement de chaque itération de la boucle. Le coefficient du compte tour dans cette fonction est l'intervalle d'initiation de la boucle. Les contraintes de dépendance peuvent être traduites en contraintes linéaires sur les coefficients de l'ordonnancement. Le problème de minimisation de l'intervalle d'initiation peut être alors résolu par les algorithmes classiques de la programmation linéaire. Les contraintes dues à la pénurie de ressources, elles, se traduisent en contraintes non linéaires, qui deviennent linéaires dès que l'intervalle d'initiation est connu. Ceci permet de définir un algorithme rapide de recherche, à partir d'une borne inférieure évidente, on accroit l'intervalle d'initiation jusqu'à trouver une solution.

*e-mail : `Paul.Feautrier@prism.uvsq.fr`

Contents


```
@INPROCEEDINGS{Feau:94,
     AUTHOR = "Paul Feautrier",
     TITLE = "Fine-grain Scheduling under Resource Constraints",
     BOOKTITLE = "7th Workshop on Language and Compiler
     for Parallel Computers",
     ADDRESS = "Cornell University",
     YEAR = 1994, MONTH = Aug,
     NOTE = "to appear in LNCS"
}
```

1 Introduction

Thinking about parallel programs is a notoriously difficult task. One of the most successful technique for dealing with this problem is *scheduling*, i.e. the construction of a timetable for the operations of the program. Obviously, this method implies the existence of a global clock for the target computer. This is a natural assumption for tightly coupled architectures like SIMD machines and pipelined or superscalar processors. Another technique, data partitioning, is more adapted to the case of asynchronous machines.

Scheduling is a difficult problem. Various special cases have been proved to be NP-hard or NP-complete. Most of the complexity of scheduling can be assigned to the conjunction of two type of constraints:

- dependence constraints, which express the fact that some computations must be executed in a specified order if the meaning of the original program is to be preserved; these constraints are usually expressed as a dependence graph (DG).
- resource constraints, which express the fact that the number of simultaneous operation at any given time is limited by the available resources in the target computer.

While any one of these constraints can be handled easily, it is their simultaneous presence which is at the origin of the difficulty.

Fortunately, in many cases of computer science interest, it is possible to handle the resource constraints in an approximate way. It is customary in this context, to distinguish between *coarse grain*, *medium grain*, and *fine grain* scheduling.

In coarse grain scheduling – e.g., job shop scheduling or macro tasking – the tasks and the resources are few. The schedule is represented in tabular form, and there are approximate techniques, like list scheduling, with precise bounds on the approximation.

In medium grain scheduling, there are many tasks – typically as many as there are operations in an execution of the source programm – and many identical resources – the processors in a massively parallel computers. The schedule must be obtained in closed form. One may ignore the resource constraints in computing the schedule [Fea92a, Fea92b], and then fold the schedule on the available processors. One may prove [Fea89] that this solution is *asymptotically efficient*, provided that the source program has enough intrinsic paralellism.

The situation is different for fine grain scheduling. Here the number of tasks is large. The resources are few and discrete. At the most, resources may be classified into categories, each category having a small number of identical resources.

Fine grain scheduling started some thirty years ago when the first computers with multiple functional units – like the CDC 6600 – were put on the market. It is now a very important technique, due to the advent of many computers with instruction level parallelism, like pipelined computers, VLIW and superscalar processors, etc.

In fine grain scheduling, it is impossible to ignore the resource constraints. Several techniques have been proposed for solving the problem, at least in an approximate way (see [RF93] for a comprehensive review of the subject). Trace scheduling [Fis84] applies list scheduling to basic blocks; it tries to detect critical paths in the program control graph and to enlarge basic blocs by moving code around test instructions.

Software pipelining [RG81] applies to simple loops and aims at executing several instances of the loop body in a staggered way so as to maximize resource usage and minimize the total execution time. A solution to the software pipelining problem for a given loop is characterized by its *initiation interval*, i.e. the time span between two successive iteration of the loop. It is easy to derive bounds for the initiation interval: an upper bound is given by the sequential execution time of the loop body. A lower bound is deduced from an analysis of resource usage, see section 2.2, and another one can be obtained by constructing an unconstrained schedule.

In many algorithms for software pipelining, one assume the iteration interval is given, and applies list scheduling, taking care that each resource allocation is folded modulo the initiation interval when constructing the reservation table (see e.g. [Lam88]). The interval of admissible initiation intervals is explored by binary search until the optimal value is found.

The algorithm of [GS92] applies only if there is only one resource class. The program is first scheduled as if there were no resource constraints. Analysis of the resulting schedule allows one to delete some dependences, and the resulting DG is cycle free. The final schedule is obtained by applying list scheduling with resource constraints to this graph. The resulting schedule is not optimal, but the authors show that the usual bound on the list scheduling approximation applies.

This paper is an attempt to extend the scheduling techniques of [Fea92a], which are based on linear programming, to fine grain scheduling. The next section is a review of these techniques. The main theme of Section 3 is how to convert the resource constraints into bilinear constraints. This is done in two cases. In the first one, there is one unique resource of each type; in the second case, there may be several copies of a resource. In the conclusion, I discuss the complexity of the algorithm and point to some direction for future work.

2 A Review of Scheduling Techniques

We will consider here the problem of scheduling a single loop:

do $i = 1, \ldots$
  S_1
  $\ldots$
  S_n
end do

where the S_k are scalar or array assignments. We have emphasized the fact that the upper bound of the loop is irrelevant for the present problem. The solution we seek must be in the form of the repetition of a uniform pattern, the loop upper bound controlling only the repetition factor.

The schedule we seek is defined by n functions from the iteration counter, i, to an integral time. We will suppose that an appropriate unit of time has been chosen – e.g., the clock cycle – and that all delays and durations are integral multiple of this unit. We will look for schedules in the form:

$$\theta(S, i) = \lfloor ai + b_S \rfloor, \tag{1}$$

where a and the b_k are rational numbers. a is known as the initiation interval of the schedule. Our main objective is its minimization.

There are several reasons for choosing such a form. Firstly, all known methods for computing schedules apply only to affine forms. It is true that a schedule whose values are not integral has no meaning, but it has been shown that the floor of a causal schedule is also causal, and that if the iteration domain is large enough, schedules of the above form are nearly optimal [Qui87].

Before embarking on the solution proper, let us observe that we have some leeway in the selection of b_S in (1). a is necessarily a rational number – if it where not so, the schedule would not be periodic. We have in fact the following

Lemma 1 *Let $a = A/D$ be the representation of the initiation interval in lowest terms. Any schedule of the form (1) is equivalent to a schedule of the form:*

$$\theta(i) = \left\lfloor \frac{Ai + B_S}{D} \right\rfloor \tag{2}$$

where the B_S are integers.

Proof Let i be any value of the iteration counter. We may write:

$$Ai = kD + r, 0 \leq r < D,$$

and

$$b_S = \lfloor b_S \rfloor + \beta_S, 0 \leq \beta_S < 1,$$
$$\theta(i) = k + \lfloor b_S \rfloor + \lfloor r/D + \beta_S \rfloor .$$

From the conditions on r and β_S, we deduce that the last term is either 0 or 1. Since A and D are relatively prime, r takes all integer values from 0 to $D-1$. The value of β_S determines the value r_0 at which the last term switches from 0 to 1:

$$r_0/D + \beta_S < 1 \wedge (r_0 + 1)/D + \beta_S \geq 1$$

which gives:

$$D - D\beta_S - 1 \leq r_0 < D - D\beta_S .$$

We conclude the schedule does not change if we replace β_S by $\lfloor D\beta_S \rfloor / D$. giving $B_S = \lfloor Db_S \rfloor$. ■

Recent research on medium-grain scheduling [MQRS90, Fea92a] favors schedules in which each statement has its own initiation interval. In the case of fine grain parallelism, such a schedule generates very complicated code [1] , hence the insistence on the same initiation interval for all statements.

All schedules must satisfies the so-called *causality condition*: let us write $(S_k, i) \perp (S_l, j)$ if (S_k, i) and (S_l, j) are in dependence, and $(S_k, i) \prec (S_l, j)$ if (S_k, i) is executed before (S_l, j) in the original program. Then the schedule must verify:

$$(S_k, i) \perp (S_l, j) \wedge (S_k, i) \prec (S_l, j) \Rightarrow \theta(S_k, i) + \partial(S_k) \leq \theta(S_l, j), \tag{3}$$

where $\partial(S_k)$ is the duration of S_k.

I have shown in [Fea92b] that in solving (3), one may partition the dependence graph in strongly connected components (scc) and schedule each scc independently. In the case of software pipelining, this deflation method should not be used. The size of the kernel one handles must be decided on other grounds, e.g. by attempting to saturate the available resources. I will say more on this point in the conclusion.

2.1 Dependences

The choice of the dependence relation in (3) is somewhat arbitrary. Ordinary dependences [ZC91], include both the effect of data flow from operation to operation and the constraints generated by the pattern of memory usage in the object program. Value based dependences are much less constraining and are easily computed by Array Dataflow Analysis [Fea91]. There is a value-based dependence between (S, i) and (R, j) iff (S, i) writes into some memory cell `a`, if (R, j) reads `a`, $(S, i) \prec (R, j)$, and there is no write to `a` between (S, i) and (R, j). The result of Array Dataflow

[1] The size of the code grows as the least common multiple of the initiation intervals.

Analysis may be represented by a Dataflow Graph (DFG), whose vertices are associated to statements and edges to dependences. Each edge e from S to R is decorated with a polyhedron $\mathcal{P}_e$ and a transformation h_e such that if $i \in \mathcal{P}_e$ then there is a value-based dependence from $(S, h_e(i))$ to (R, i). One may say that after Array Dataflow Analysis, all values produced by the source code have been given distinct names, and the program has been rewritten using these names. Array Dataflow Analysis may thus be seen as a compile time counterpart of Tomasulo Algorithm.

The shape of the dependence is given by the function h_e. The simplest case is that of *uniform* dependences where h_e is a translation:

$$h_e(i) = i - d_e$$

where d_e is known as the dependence distance. One may encounter more complicated cases, where h_e is an affine function, or even a sublinear function[2]. The scheduling technique of [Fea92a] works whenever the dependence is affine and is not limited to uniform dependences.

Value based dependences will be used throughout this paper. In this context, the causality condition (3) simplifies to:

$$\forall e \in DFG, \forall i \in \mathcal{P}_e : \theta(R, i) \geq \theta(S, h_e(i)) + \partial(S). \tag{4}$$

This condition expresses the fact that since operation (R, i) uses a value which is computed by $(S, h_e(i))$, it cannot start before this operation has terminated.

The solution method starts by substituting the form (1) into (4). In the case of uniform dependences, one may prove that:

Lemma 2 *The causality condition (4) is equivalent to:*

$$ad_e + b_R - b_S \geq \partial(S). \tag{5}$$

Proof That (5) implies (4) is proved in [Fea92a] Theorem 6. To prove the reverse implication, choose for i a multiple of D. Notice that if n is an integer, we have the identity $\lfloor n + x \rfloor = n + \lfloor x \rfloor$. (4) simplifies to:

$$\lfloor B_R/D \rfloor \geq \lfloor (B_R - Ad_e)/D \rfloor - \partial(S).$$

Since the left hand side of this inequality is an integer, we have:

$$\lfloor B_R/D \rfloor \geq (B_R - Ad_e)/D - \partial(S).$$

Now, obviously, $x \geq \lfloor x \rfloor$, hence:

$$B_R/D \geq \lfloor B_R/D \rfloor \geq (B_R - Ad_e)/D - \partial(S),$$

Q.E.D. ∎

By the above lemma, each uniform dependence may be translated to a linear constraint on the a and b's coefficients. For more complicated dependences, one has to resort to the Farkas algorithm [Fea92a], but the result is still a set of linear constraints. One then selects a particular solution according to some objective function.

Of particular interest for fine grain scheduling are the minimum latency schedules, in which one minimizes first the initiation interval a, and then the b_R.

[2] A sublinear function contains integer divisions by constants.

2.2 Resource constraints

In operation research, a *resource* is an entity which may or may not be used by tasks or operations. To each resource is associated a constraint: namely, that the execution intervals of two operations which use the same resource cannot overlap. One may have resource *classes*. In that case, at any given time, the number of active operations which use a given resource cannot exceed the number of resources in the class. We will suppose here that all operations which are instances of the same instruction use the same resource class. For simplicity, we will assume that each operation uses only one resource. This restriction can be easily lifted in case of need. In fact, in this work resource classes will simply be sets of statements. If ρ is a resource class, $S \in \rho$ means that statement S uses a resource from class ρ.

In the case of unique resources, the non overlap constraint may be translated to simple inequalities on schedules. Suppose that S and T use the same resource. If $\langle S, i\rangle$ is scheduled before $\langle T, j\rangle$, then we must have:

$$\theta(T, j) \geq \theta(S, i) + \partial(S),$$

while in the opposite situation the constraint is:

$$\theta(S, i) \geq \theta(T, j) + \partial(T).$$

Since the two situations are exclusive, we may write the resource constraint as:

$$\forall i, j : \theta(T, j) - \theta(S, i) \geq \partial(S) \vee \theta(S, i) - \theta(T, j) \geq \partial(T). \tag{6}$$

Beside that, two operations which are instance of the same instruction necessarily use the same resource and cannot overlap:

$$\forall i, j : i < j \Rightarrow |\theta(S, i) - \theta(S, j)| \geq \partial(S). \tag{7}$$

This condition gives a very simple bound on a. Suppose a large number N of iterations of the loop body are executed in time t. The total usage of resource ρ will be:

$$t_\rho \approx N \sum_{S\in\rho} \partial(S).$$

Suppose there are P_ρ copies of ρ. We have:

$$t \approx Na \leq N \sum_{S\in\rho} \partial(S)/P_\rho,$$

from which we deduce the lower bound for a:

$$a \geq \max_\rho \sum_{S\in\rho} \partial(S)/P_\rho. \tag{8}$$

If the initiation interval satisfies the above constraint, (7) will be automatically satisfied.

In actual processors, resource utilization may be a much more complicated affair than the simplified scheme above. Pipelined resources, for instance, do not appear to be busy for the whole duration of one operation. This is easily taken care of by replacing $\partial(S)$ in (6) by another timing characteristics, the *stalling time* of operation S, noted $\sigma(S)$. The resource constraint is now:

$$\forall i, j : \theta(T, j) - \theta(S, i) \geq \sigma(S) \vee \theta(S, i) - \theta(T, j) \geq \sigma(T). \tag{9}$$

An ordinary functional unit will have $\partial(S) = \sigma(S)$, while a pipelined unit will have $\sigma(S) \ll \partial(S)$.

There may be links between resources, as for instance when one cannot use a functional unit unless there is a free data path to it. That kind of constraint must be handled heuristically.

The problem is more complicated if some resource class has more than one element. A resource is in use at time t if some statement S which uses it has been initiated less than $\sigma(S)$ time units before t. If we identify a resource class with the set of statements which use it, we may write the constraint for resource ρ as:

$$\text{Card } \{(S, i) \mid S \in \rho \wedge t - \sigma(S) < \theta(S, i) \leq t\} \leq P_\rho. \tag{10}$$

3 Two Scheduling Algorithms

Basically, the scheduling method of [Fea92a] works by replacing (4), which represents a potentially infinite system of affine inequalities, by a finite set of constraints on the coefficients a and b_R. Our first problem is to find a similar reduction for (9). We will see that, due to the non-convexity of (9), the result is non linear. Hence, we cannot directly use linear programming to solve the problem. However, the problem lends itself to a simple and efficient solution by searching the space of possible values for a.

3.1 The singular resource case

For schedules of the form (2), we may ignore the floor function in the expression of (6). We have in fact the:

Theorem 3 *Let $\tau(S, i) = \frac{Ai+B_S}{D}$ and $\theta(S, i) = \lfloor \tau(S, i) \rfloor$. Then the two conditions:*

$$\forall i, j : \theta(T, j) - \theta(S, i) \geq \sigma(S) \vee \theta(S, i) - \theta(T, j) \geq \sigma(T). \tag{11}$$

and

$$\forall i, j : \tau(T, j) - \tau(S, i) \geq \sigma(S) \vee \tau(S, i) - \tau(T, j) \geq \sigma(T). \tag{12}$$

are equivalent.

> **Proof** Suppose first that (12) is true. Let us be given two arbitrary integers i and j. We may suppose, without loss of generality, that $\tau(S, i) - \tau(T, j) > 0$. We have, successively:
>
> $$\lfloor \tau(T, j) \rfloor \leq \tau(T, j),$$
>
> $$\lfloor \tau(T, j) \rfloor + \sigma(T) \leq \tau(T, j) + \sigma(T) \leq \tau(S, i),$$
>
> and, since the left hand side is an integer,
>
> $$\lfloor \tau(T, j) \rfloor + \sigma(T) \leq \lfloor \tau(S, i) \rfloor,$$
>
> Q.E.D.
>
> Conversely, suppose that (12) is false for some values of i and j. Set $x = i - j$. We have both:
>
> $$Ax + B_S - B_T \leq \sigma(T)$$
>
> and
>
> $$B_T - B_S - Ax \leq \sigma(S).$$

Set $B = B_T - B_S$ for short. We may suppose that $Ax - B > 0$. The other case is handled in a symmetrical fashion. We have, for all j:

$$\tau(S, j+x) - \tau(T, j) = \frac{Ax - B}{D} \leq \sigma(T).$$

Since A and D are relatively prime, we may select j in such a way that $\tau(S, j+x)$ is an integer. We then have:

$$\lfloor \tau(S, j+x) \rfloor = \tau(S, j+x) \leq \tau(T, j) + \sigma(T),$$

$$\lfloor \tau(S, j+x) \rfloor \leq \lfloor \tau(T, j) \rfloor + \sigma(T),$$

(11) is also false, Q.E.D. ■

From this result, we deduce the resource constraint above may be written in the form:

$$\forall x \in \mathbb{Z} \frac{Ax - B}{D} \geq \sigma(T) \vee \frac{B - Ax}{D} \geq \sigma(S).$$

Now $\frac{Ax-B}{D}$ is an affine function of x whose zero is $x_0 = B/A$. For all values of $x > x_0$, the second inequality is certainly not verified. Hence, the first one must be true, and a necessary and sufficient condition is that:

$$A(\lfloor B/A \rfloor + 1) - B \geq D\sigma(T).$$

The other case is handled similarly and gives:

$$B - A\lfloor B/A \rfloor \geq D\sigma(S).$$

These conditions may even be simplified by observing that, if they are true, then there exists a unique integer q such that:

$$A(q+1) - B \geq D\sigma(T) \wedge B - Aq \geq D\sigma(S). \tag{13}$$

We conclude that the resource constraints in the singular case are given by the following rule:

For all statements S and T which use the same resource:

- Create a new integer variable q_{ST},
- Write the two constraints:

$$\begin{aligned} A(q_{ST} + 1) - B_T + B_S &\geq D\sigma(T), \\ B_T - B_S - Aq_{ST} &\geq D\sigma(S). \end{aligned} \tag{14}$$

These constraints are to be added to the dependence constraints and solved for A and the B_S, A being the objective function to be minimized. Now the constraints generated by (14) are clearly non linear. However, they become linear if we are given the value of A. Remember that we have one upper bound for $a = A/D$ which is simply the sum of the duration of all statements in the loop body – the *sequential* upper bound – and two lower bounds. One of them, the *resource usage* bound, is given by (8), and the other, the *free* bound, is obtained simply by solving the scheduling problem with no resource constraints. The maximum of these two bounds gives the *parallel* lower bound. The problem is that, since a is a rational number, exploring its possible range of values is not a finite process. As has been observed many times, the schedule (2) has period D. D iterations of the loop body are scheduled in A clock cycles, giving a mean activation interval of A/D. When generating code from such a schedule, the loop body has to be replicated D times, which means that D cannot be too large. In the singular resource case, the resource usage bound is an integer. The free bound may be rational, but the actual value of its denominator is no indication, because simplification may occur depending on the values of the statement durations. A better guess may be obtained by observing that when computing the free schedule, one has to solve a linear programming problem by a process analogous to Gaussian elimination. By the familiar Cramer rule, the denominator of the solution is the determinant of a matrix which is extracted from the problem tableau, the basis matrix. The value of this determinant is easily extractable from the linear programming code, and is a good candidate for the unrolling factor.

We have found in practice that the following heuristic gives satisfactory results :

1. Compute the free bound, the resource usage bound and the parallel lower bound, l, which is their maximum.
2. D is set equal to the determinant of the basis matrix or to 1, depending whether the parallel lower bound is the larger bound or not.
3. Set $A = \lceil Dl \rceil$.
4. Solve the complete scheduling problem for A and D.
5. If the problem has no solution, increase A by 1 and start again at step 4.

Let us consider first a very simple example:

```
      program A

      do i = 1,n
1        r1 = a(i)-b(i)
2        c(i) = c(i-2) + r1
      end do
```

Suppose that all operations are executed in unit time. Let $\theta(1, i) = ai + b_1$ and $\theta(2, i) = ai + b_2$ be the prototype schedules. There are two dependences:

- The first one is from $\langle 1, i\rangle$ to $\langle 2, i\rangle$ and gives the constraint:
$$b_2 - b_1 \geq 1.$$
- The second one is from $\langle 2, i-2\rangle$ to $\langle 2, i\rangle$ and gives:
$$2a \geq 1.$$

It is easy to see that the minimum latency solution is:

$$\theta(1, i) = i/2, \qquad \theta(2, i) = i/2 + 1.$$

Suppose now that both statements of the example are executed on the same resource. This gives the following additional constraints:

$$a(q + 1) - b_1 + b_2 \geq 1, \qquad b_1 - b_2 - aq \geq 1.$$

Since there are two statements in the loop and only one resource, we must have $a \geq 2$. An attempt to solve the remaining constraints with $A = 2, D = 1$ succeeds and gives:

$$\theta(1, i) = 2i, \qquad \theta(2, i) = 2i + 1.$$

Since we have an upper and a lower bound for A, it may seem that a binary search for the right value might be a good idea. However, experiment shows that the solution is always near the lower bound. In that case, a simple linear search is sufficient. Let us consider the following example:

```
    program B

    do i = 1,n
1     r0 = a(i-2)/2.0
2     r1 = r0+a(i-3)
3     r2 = r0+a(i-4)
4     a(i) = r1*r2
    end do
```

Suppose that the available resources are an adder, a multiplier and a divider, and that addition takes one cycle, multiplication and division taking two cycles. Analysis of resource usage shows that the minimum initiation interval is two cycles. Dependence analysis shows that statement 1 has to be executed first, that 2 and 3 can be executed in parallel, and that 4 is to be executed last. However, since the cycle is closed by a dependence from 4 at iteration i to 1 at iteration $i+2$, this gives a minimum rate of $5/2$, and this is the parallel lower bound. Hence, we set $D = 2$. The first value of A to be tested is 5, and our integer programming algorithm finds that there is no solution. A is thus increased to 6, and there is a solution. It is easy to see *a posteriori* that this solution is optimal. In fact, since there is only one adder, statements 1 and 2 must be executed sequentially. Hence each iteration of the loop takes 6 cycles. The resulting initiation interval is $6/2 = 3$, indicating that no unrolling is necessary.

Suppose now that the multiplication time is reduced to 1 cycle. The free bound decreases to 2, but the determinant of the basis matrix is still 2. Hence, we set $D = 2$ and $A = 4$. The first solution is found at the second iteration when $A = 5$, giving an initiation interval of $5/2$ with an unrolling factor of two. The schedule is:

$$\theta(1,i) = 5/2i, \qquad \theta(2,i) = 5/2i + 2,$$
$$\theta(i,3) = 5/2i + 3, \qquad \theta(4,i) = 5/2i + 4.$$

To solve this problem, three calls to the integer programming algorithm PIP where needed, which took 0.43 seconds on a low end workstation.

3.2 The many resource case

In the many resource case, the resource constraint is given by (10). In the singular case, we have seen that we have to guess the value of D and to search for the value of A. The many resource case is apparently more complicated. Hence, we will suppose that the algorithm structure is the same, and that our problem is to test whether, A and D being given, there is a possible assignment for the B_S which meets all the constraints of the problem.

Here again, the first step is to get rid of the floor function. Suppose t is given, and that we are trying to count how many instances of S are active at time t. The iteration counter of the active instances is a positive integer such that:

$$t - \sigma(S) < \left\lfloor \frac{Ai + B_S}{D} \right\rfloor \leq t. \tag{15}$$

All terms in these inequalities are integers. Hence, we may rewrite it as:

$$t - \sigma(S) + 1 \leq \left\lfloor \frac{Ai + B_S}{D} \right\rfloor < t + 1.$$

Now $t - \sigma(S) + 1 \leq \lfloor \frac{Ai+B_S}{D} \rfloor$ and $t - \sigma(S) + 1 \leq \frac{Ai+B_S}{D}$ are equivalent. In one direction, this is because $\lfloor x \rfloor \leq x$, and in the other, it results from the monotony of the floor function.

For the other inequality, $\frac{Ai+B_S}{D} < t+1$ clearly imply $\lfloor\frac{Ai+B_S}{D}\rfloor < t+1$. In the reverse direction, $\lfloor\frac{Ai+B_S}{D}\rfloor \leq t$ implies $\frac{Ai+B_S}{D} < t+1$ by definition.

As a consequence, the iterations of S which are active at time t are solutions of:

$$Dt - D\sigma(S) + D \leq Ai + B_S < Dt + D.$$

Our problem is to count the solutions of these inequalities with i as the unknown as a function of t.

Introducing an "excess" variable x, the constraints may be transformed into an equation:

$$Ai + B_S = Dt + D - 1 - x, \tag{16}$$

provided that x satisfies $0 \leq x < D\sigma(S)$. If $N_S(t,x)$ is the count of solutions of (16) for given t and x, then the number of active iterations at time t is:

$$N_S(t) = \sum_{x=0}^{D\sigma(S)-1} N_S(t,x).$$

The first observation is that equation (16) has at most one solution, which is given by:

$$i = \frac{Dt + D - 1 - x - B_S}{A}.$$

To be a legitimate iteration number, this solution has to be a positive integer. i is obviously positive for large enough t. If we ignore the positivity condition, the effect will be to overestimate the resource usage for the *prologue* of the loop nest. It is customary in the field to ignore this factor by considering only very long loops, and this is the best we can do at compile time, since, for most loops, the iteration count is a variable. It may be possible to do better under user guidance: for instance, to inhibit software pipelining when the user knows that the iteration count will be small.

The integrity condition is simply:

$$Dt + D - 1 - x - B_S \equiv 0 \pmod{A}. \tag{17}$$

This has to be evaluated for all values of t. It is clear, however, that the condition depends only on $t \bmod A$. It thus has to be tested for $t \in [0, A-1]$. Another point is that the correspondance from t to $Dt \bmod A$ is bijective, since A and D are relatively prime. As a consequence, we introduce a new variable $t' = Dt \bmod A, 0 \leq t' \leq A-1$. The number of solutions of (16) may be written:

$$N_S(t,x) = \delta((t' + D - 1 - x - B_S) \bmod A),$$

where δ is a variant of the Kronecker symbol:

$$\delta(0) = 1, \ \ \delta(i) = 0, i \neq 0.$$

The total number of solutions is now:

$$N_S(t) = \sum_{x=0}^{D\sigma(S)-1} \delta((t' + D - 1 - x - B_S) \bmod A).$$

All in all, (10) translates to:

$$\sum_{S\in\rho} \sum_{x=0}^{D\sigma(S)-1} \delta((t' + D - 1 - x - B_S) \bmod A) \leq N_\rho. \tag{18}$$

The next step is to "linearize" the Kronecker symbol. This is possible by rewriting B_S as:

$$B_S = AC_S + \sum_{k=0}^{A-1} k y_{S,k}, \tag{19}$$

where the $y_{S,k}$ are integral variables such that:

$$\begin{aligned} 0 \leq \quad y_{S,k} & \quad \leq 1, & (20) \\ \sum_{k=0}^{A-1} y_{S,k} & \quad = \quad 1 & (21) \end{aligned}$$

In fact, we may take $C_S = B_S \div A$. If we then set $y_{S,r} = 1$, all others $y_{S,k}$ being 0, where $r = B_S \bmod A$, we have the required equality.

It is now easy to prove by enumerating cases that:

$$\delta((t' + D - 1 - x - B_S) \bmod A) = y_{S,(t'+D-1-x) \bmod A},$$

and that, as a consequence, the resource constraint (18) takes the form:

$$\forall t' \in [0, A-1] : \sum_{S \in \rho} \sum_{x=0}^{D\sigma(S)-1} y_{S,(t'+D-1-x) \bmod A} \leq N_\rho. \tag{22}$$

This is the required linearization. The solution process may be summarized as follows:

1. Select a value for D, in a manner that will be discussed presently.
2. Set $A = Dl$, where l is the lower bound for the iteration interval.
3. For each statement S in the loop nest, create $A + 1$ new unknowns C_S and $y_{S,k}$, $k = 0, A-1$, write the equality (19) and the constraints (20) and (21).
4. For each resource in the system, write the constraint (22).
5. Express the causality constraint in term of the new unknowns, applying the Farkas algorithm if necessary [Fea92a].
6. If the resulting system is feasible, the problem has been solved. If not, add one to A and start again at step 3.

The unrolling factor D has to be choosen, as above, by heuristic arguments. Since the resource lower bound is no longer an integer, we have two denominators to choose from. Possible suggestions are to take the largest one, or their least common multiple, or the denominator of the largest bound.

Consider the following loop, which is taken from [GS92].

```
   program G
   do i = 0,n
1     a(i) = a(i)+d(i-2)
2     b(i) = a(i)/e(i-2)
3     c(i) = a(i)*e(i-2)
4     d(i) = c(i)+b(i-1)
5     e(i) = e(i)+b(i)
   end do
```

The target computer has three identical processors on which each statement takes unit time, with the exception of 2 which takes two cycles. In our notations, for statements 1, 3, 4 and 5 $\partial(S) = \sigma(S) = 1$, while $\sigma(2) = 2$. As has been observed by Gasperoni et. al., the free schedule is:

$$\begin{aligned} \theta(1,i) &= 3/2i, & \theta(2,i) &= 3/2i+1, \\ \theta(3,i) &= 3/2i+1 & \theta(4,i) &= 3/2i+2, \\ \theta(5,i) &= 3/2i+2. \end{aligned}$$

with an initiation interval of 3/2. This means, in fact, that three iterations can be initiated any two clock cycles. Since each iteration needs 6 cycles, and there are three processors, full utilization of the resources is obtained for an initiation interval of 2. The proposed algorithm succeeds immediately and find the following schedule:

$$\begin{aligned} \theta(1,i) &= 2i, & \theta(2,i) &= 2i+1, \\ \theta(3,i) &= 2i+1 & \theta(4,i) &= 2i+2, \\ \theta(5,i) &= 2i+3. \end{aligned}$$

Two calls to PIP are needed, taking less than 2 seconds on a low end Sparcstation. The present solution is optimal; this is to be compared to the solution obtained by Gasperoni et. al., whose initiation interval is 3.

4 Conclusion and Future Work

In this paper, I have shown how to translate resource constraints into systems of bilinear inequalities. For a given initiation interval, the inequalities become linear. When added to the dependence constraints, they can be tested for feasibility by any integer programming algorithm, in our case, an implementation of the Gomory algorithm. One then has to search for increasing values of the initiation interval until a solution is found.

Extracting the object code from the schedule is a well known problem, which is best explained by an example. Let us consider program G. The first step is to invert the schedule, i.e. to decide who is doing what at any given time. Since the initiation interval is 2, we have to distinguish between even and odd time. Let us suppose that t is even. The solution of $2i = t$ is $i = t/2$. Hence, we know that some processor will be executing iteration $t/2$ of statement 1 at time t. Similarly, $2i + 3 = t$ has no solution, but $2i + 3 = t + 1$ has. Hence, we know that some processor will be executing iteration $\frac{t-2}{2}$ of statement 5 at time t+1. Proceeding in this way, we obtain the following diagram

t	$\langle 1, t/2\rangle$		$\langle 4, \frac{t-2}{2}\rangle$
$t+1$	$\langle 3, t/2\rangle$	$\langle 2, \frac{t-1}{2}\rangle$	$\langle 5, \frac{t-2}{2}\rangle$

The construction of the actual code is now strongly dependent on the machine architecture. On a VLIW processor, for instance, the above diagram directly gives the statements to be packed in two successive instruction words. The problem may be more complicated on a superscalar architecture.

The method has been implemented by extending the scheduler of [Fea92b]. All exemples in this paper have been solved on this implementation. There are many possible improvements on this solution, some of which have already been tested. For instance, in the case of mixed problems, with single resources and resource classes, it is possible to combine the two algorithms, using (13) for single resources and (22) for resource classes.

The following code has been adapted from [DGN92] by randomly replacing additions by multiplications.

```
         program V
         do i = 0,n
1          q(i) = x(i-1)+a(i)
2          r(i) = q(i)*q(i-1)
3          s(i) = r(i)+r(i-1)
4          t(i) = s(i)+s(i-1)
5          u(i) = t(i-1)*t(i-2)
6          v(i) = u(i-1)*u(i-2)
7          w(i) = v(i)+b(i)
8          x(i) = w(i)+c(i)
9          y(i) = t(i)+z(i-1)
10         z(i) = y(i)*d(i)
         end do
```

Let us suppose that all operations have unit duration, and that there are two adders and one multiplier. The free schedule has initiation interval 8/3. The resource bound is 4, since there are 4 multiplications and only one multiplier. Therefore, we try scheduling with $A = 4$ and $D = 1$. For each add statement, we have to introduce 4 "y" unknowns and one "C" unknown, for a total of 40 unknowns. On the other hand, the four multiplications give rise to 6 constraint pairs like (13) in which one has to introduce 6 "q" unknowns. The algorithm immediately succeeds, giving:

$$
\begin{aligned}
\theta(1,i) = 4i \quad &, \quad \theta(2,i) = 4i+1, \\
\theta(3,i) = 4i+2 \quad &, \quad \theta(4,i) = 4i+3, \\
\theta(5,i) = 4i+2 \quad &, \quad \theta(6,i) = 4i, \\
\theta(7,i) = 4i+1 \quad &, \quad \theta(8,i) = 4i+2, \\
\theta(9,i) = 4i+4 \quad &, \quad \theta(10,i) = 4i+7.
\end{aligned}
$$

There are two calls to PIP, totalling about 9.3 seconds. This solution is optimal, since its initiation interval is equal to the above computed lower bound.

One may notice, however, that the dependence graph of program V has two strongly connected components, $\{1,2,3,4,5,6,7,8\}$ and $\{9,10\}$. One may try to schedule these conponents independently. This results in two schedules, one of interval 3 and the other of interval 2, giving an equivalent initiation interval of 5, or 25% more than global scheduling. An interesting observation is that the running time for 4 calls to PIP now *drops down* to about 1.6 seconds.

The controlling factors for the complexity of the algorithm are the size of the initiation interval, A, and the number of statements in the loop body. The algorithm is not sensitive to the number of resources in a class. For instance, the following table gives the initiation interval and the solution time in seconds for program G, for 1 to 4 CPU's, at which time the free schedule is obtained:

CPU	Interval	Time
1	6	1.58
2	3	2
3	2	1.86
4	3/2	0.71

Obviously, the space and time requirements of the algorithms may become prohibitive for very large exemples. The question now is: is there a better way of solving directly the non-linear constraints (13) or (22) than by a combination of integer programming and search?

Acknowledgment

All exemples in this paper have been run using Zbigniew Chamski's multiple precision implementation of PIP.

References

[DGN92] Vincent H. Van Dongen, Guang R. Gao, and Qi Ning. A polynomial time method for optimal software pipelining. In Luc Bougé, Michel Cosnard, Yves Robert, and Denis Trystram, editors, *Parallel Processing: CONPAR 92–VAPP V*, pages 613–624, Springer, LNCS 634, June 1992.

[Fea89] Paul Feautrier. Asymptotically efficent algorithms for parallel architectures. In M. Cosnard and C. Girault, editors, *Decentralized System*, pages 273–284, IFIP WG 10.3, North-Holland, December 1989.

[Fea91] Paul Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, February 1991.

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem, I, one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem, II, multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, December 1992.

[Fis84] J.A. Fisher. The vliw machine: a multiprocessor for compiling scientific code. *Computer*, 45–53, July 1984.

[GS92] Franco Gasperoni and Uwe Schwiegelshohn. Scheduling loops on parallel processors: a simple algorithm with close to optimum performance. In Luc Bougé, Michel Cosnard, Yves Robert, and Denis Trystram, editors, *Parallel Processing: CONPAR 92–VAPP V*, pages 625–636, Springer, LNCS 634, June 1992.

[Lam88] Monica Lam. Software pipelining: an effective scheduling technique for vliw machines. In *Proc. of the SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 318–328, Atlanta, June 1988.

[MQRS90] Christophe Mauras, Patrice Quinton, Sanjay Rajopadhye, and Yannick Saouter. *Scheduling Affine Parameterized Recurrences by means of Variable Dependent Timing Functions.* Technical Report 1204, INRIA, April 1990.

[Qui87] Patrice Quinton. The systematic design of systolic arrays. In F. Fogelman, Y. Robert, and M. Tschuente, editors, *Automata networks in Computer Science*, pages 229–260, Manchester University Press, December 1987.

[RF93] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: history, overview and perspective. *The Journal of Supercomputing*, 7:9–50, 1993.

[RG81] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high-performance scientific computing. In *IEEE/ACM 14th Annual Microprogramming Workshop*, October 1981.

[ZC91] Hans Zima and Barbara Chapman. *Supercompilers for parallel and vector computers*. ACM Press, 1991.