time	i=0	i=1	i=2
0	a_1 a_2		
1	j(L)		
2 or 10	a		
3 or 11	x		
4 or 12	y	a_1 a_2	
5 or 13		j(L)	
6 or 14 or 22		a	
7 or 15 or 23		x	
8 or 16 or 24		y	a_1 a_2
9 or 17 or 25			j(L)
10 or 18 or 26 or 34			a
11 or 19 or 27 or 35			x
12 or 20 or 28 or 36			x

Figure 9: Dynamic schedule when speculation is used.

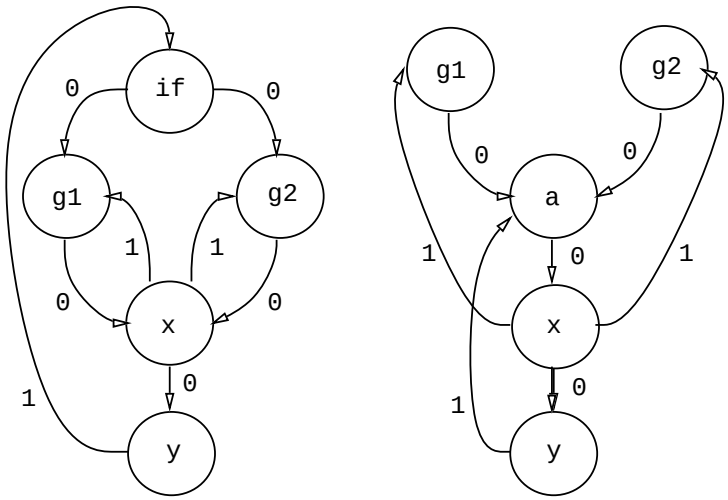

Figure 7: Dependence graphs: (a) with no speculation and (b) with speculation.

time	i=0	i=1	i=2
0	a_1 a_2		
10	a		
11	x		
12	y	a_1 a_2	
22		a	
23		x	
24		y	a_1 a_2
34			a
35			x
36			y

Figure 8: Static schedule when speculation is used.

Example 6.2 This example is taken from [18]. Using the non-speculative model, the loop is written:

$$
\begin{array}{ll}
\textbf{for } i = 0 \textbf{ to } N-1 & \textbf{do} \\
 & \textbf{if } y[i-1] \textbf{ then } a[i] = g_1(x[i-1]); \\
 & \textbf{else } a[i] = g_2(x[i-1]); \\
 & x[i] = f(a[i]); \\
 & y[i] = h(x[i]); \\
 & \textbf{od}
\end{array}
$$

Figure 7(a) shows the dependence graph of the loop. Let's assume that all operations have unit delays (and in particular the branch selection.) with one exception: the one that computes g_1 takes ten units of time. Using the static scheduling technique previously presented in this paper, the best computation rate is 13.

However, we can clearly see that, if the "else" branch is taken, the computation rate can drop from 13 to 4 by simply adding a jump to $x[i]$ so that, as soon as $a[i]$ is computed, $x[i]$ can be computed.

The same idea is applicable when *speculation* is used. This is shown in detail hereafter. If speculation is to be used, different branches must use different variables and a conditional assignment selects the appropriate result. The above loop becomes:

$$
\begin{array}{ll}
\textbf{for } i = 0 \textbf{ to } N-1 & \textbf{do} \\
 & [\ \textbf{if } y[i-1] \textbf{ then }]\ a_1[i] = g_1(x[i-1]); \\
 & [\ \textbf{else }]\ a_2[i] = g_2(x[i-1]); \\
 & a[i] = \ \textbf{if } y[i-1] \textbf{ then } a_1[i] \textbf{ else } a_2[i]; \\
 & x[i] = f(a[i]); \\
 & y[i] = h(x[i]); \\
 & \textbf{od}
\end{array}
$$

Figure 7(b) shows the dependence graph associated to this new loop. Clearly, the maximal computation rate is now equal to 12. Figure 8 shows the execution of the first three iterations, and the pattern.

As explained above, one should use dynamic scheduling instead of static scheduling to improve the average computation rate. The *dynamic schedule* can be simply obtained by adding conditional jumps to the static one. We suggest to add the following conditional jump, noted $j(L)$, to the static schedule:

$$\textbf{if } (\textbf{not } b[i]) \textbf{ then goto } L.$$

The label L is placed at $x[i]$, and the resulting schedule is shown in figure 9.

□

time	i=0	i=1	i=2	i=3
0	b c	b c	b c	b c
1	a	a	a	a

Figure 5: A 1-periodic schedule of infinite computation rate.

If we now fix the value of r, and compute the optimal r-periodic schedule, we still obtain the same one (because $0/1 = 0/r$). This does not help us to find a pattern of different size. On the other hand, if we impose the computation rate to be 1/2 for example, the optimal schedule having that rate is

$$t(i, v) = \lfloor \frac{1 \cdot i + A[v]}{2} \rfloor$$

with $A[b] = A[c] = 0$ and $A[a] = 1$. The resulting schedule of the first four iterations is shown in figure 6. The resulting pattern is now of size $2n$. In this paper, we provide a method for computing the optimal schedule of a given computation rate, which is a direct extension to the one for computing the schedule of optimal computation rate.

time	i=0	i=1	i=2	i=3
0	a b c	b c		
1		a	a b c	b c
2				a

Figure 6: An optimal 2-periodic schedule of fixed computation rate 1/2.

□

Conditional branches

In this section, we illustrate what can be done in the presence of conditional branches. We introduce two models for expressing conditionals, namely the speculative one and the non-speculative. Then, we show how to optimize the execution rate by adding conditional jumps to the code; the resulting schedule is then *dynamic*; i.e., the computation rate is data dependent.

In the non-speculative model, the condition is evaluated first, and then one of the two branches is taken depending on its boolean value. Speculation is more efficient [18]; the two branches are evaluated in parallel, and depending on the boolean value of the condition, the adequate result is selected.

For the speculative model, two specific constructs are provided to express conditionals, namely *conditional blocks* and *conditional assignments*. All variables are different so that the "if" branch and the "else" branch of the conditional blocks can always be executed in parallel. The conditional assignment is associated to a conditional block (the same conditional is used) to enable a variable to receive a result from either the "if" branch or the "else" branch.

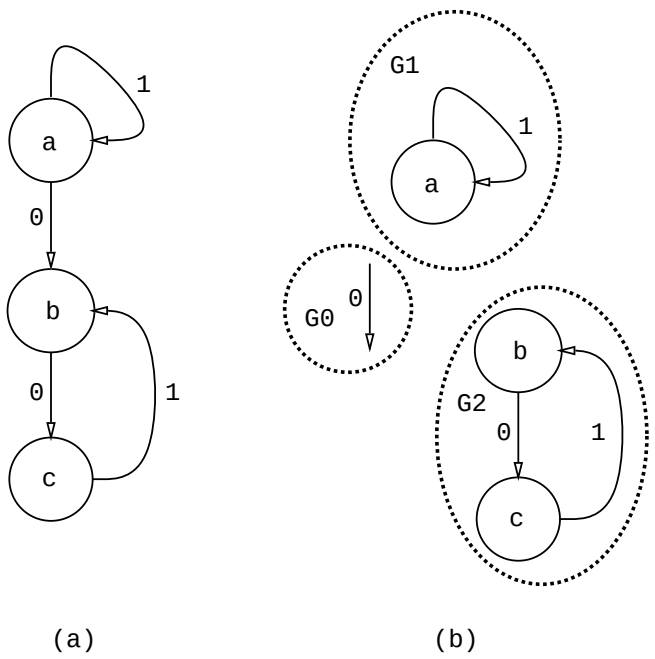

Figure 4: A dependence graph (a) and its two connected components (b).

Forall example

The following examples illustrates the fact that the forall is treated uniformally as a special case of the general framework presented in this paper.

Example 6.1 The loop

$$\begin{array}{ll} \textbf{for } i = 0 \textbf{ to } N-1 & \textbf{do} \\ & a[i] = b[i-1] + 3; \\ & b[i] = 1/X[i]; \\ & c[i] = X[i] + Y[i]; \\ & \textbf{od} \end{array}$$

is optimally parallelized with the schedule

$$t(i, v) = 0 \cdot i + A[v]$$

with $A[b] = A[c] = 0$ and $A[a] = 1$. In this particular case, $T = 0$ and $r = 1$, so the computation rate is infinite (i.e. $1/0$); all iterations are executed in parallel.

The resulting schedule of the first four iterations in figure 5. Here, all iterations have the same schedule.

In this case, there is neither a prelude nor a postlude but only a pattern that contains all the statements of the unrolled loop, i.e. of size $n \times N$ where n is the number of statements of the original loop and N the number of iterations.

Appendix

Connected components

Let us recall the notion of *strongly connected component* of a dependence graph. Basically, the nodes are in the same strongly connected component if there is a directed path connecting any pair of its nodes. A formal definition is given in [1]. Given the dependence graph G, we will note g the number of its strongly connected components. If $g = 1$, the graph is said to be *connected*. We will note G_i the i^{th} component of G; $1 \leq i \leq g$. And will note G_0 the set of arcs which belong to no connected component. In that way,

$$G = G_0 \cup G_1 \cup ... \cup G_g.$$

We will be even more precise in the numbering of the components. Clearly, the connected components G_i ($i \in [1, g]$) connected via arcs of G_0 form a directed acyclic graph. We suggest a numbering such that, when there is an arc from G_i to G_j, $i < j$. In particular, G_1 will have no incoming arc coming from another component.

In [1], an algorithm is given that computes the connected components in $max(n, m)$ time, where m denotes the number of arcs in the graph. In our case, m is in $\mathcal{O}(n)$. Thus, the complexity is in $\mathcal{O}(n)$. The algorithm builds a particular tree to compute the components. This numbering of the connected components corresponds to a particular traversal of that tree: depth-first search but from right to left. The full proof is left to the readers; it is based on the observation that there is never an arc going from left to right in that tree.

Example 2.1 (cont'd) Figure 4(b) shows the two connected components G_1 and G_2 of the dependence graph of figure 4(a), and G_0.

□

When the dependence graph is not connected, it is faster to solve the MCTRC problem on each individual component and then to take the minimal value. Let $r[i]/T[i]$ be the mininum ratio of component G_i,

$$\frac{r}{T} = min(\frac{r[1]}{T[1]}, \frac{r[2]}{T[2]}, ..., \frac{r[g]}{T[g]}).$$

If G_i contains n_i nodes, the complexity to find $r[i]/T[i]$ is in $\mathcal{O}({n_i}^2 \lg n_i)$ when using Bellman-Ford algorithm to solve the $\mathcal{O}(\lg n)$ shortest path problems. (See section 4.)

The values of $A[v]$'s can be computed for each component G_i individually without taking into account the dependence arcs of G_0. The constants can then be readjusted by taking into account these constraints. First, the ones of G_1 are readjusted by taking into account that the minimal value of $A[v]$ can be zero. Second, the ones of G_1 are readjusted by taking into account the dependence arcs going from G_1 to G_2. Then, the one of G_3 are readjusted, etc.

[6] K. Ebcioğlu. A compilation technique for software pipelining of loops with conditional jumps. In *Proceedings of the 20th Annual Workshop on Microprogramming*, December 1987.

[7] K. Ebcioğlu and A. Nicolau. A global resource-constrained parallelization technique. In *Proceedings of the ACM SIGARCH International Conference on Supercomputing*, June 1989.

[8] H. Gabow and R. Tarjan. Faster scaling algorithms for network problems. *SIAM J. Computing*, October 1989.

[9] G. R. Gao, Y. B. Wong, and Qi Ning. A Petri-Net model for fine-grain loop scheduling. In *Proceedings of the '91 ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 204–218, Toronto, Canada, June 1991.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[11] R. B. Jones and H. A. Allan. Software pipelining: A comparison and improvement. In *Proceedings of the 23th Annual Workshop on Microprogramming and Microarchitecture*, pages 46–56, Orlando, Florida, November 1990.

[12] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 318–328, Atlanta, GA, June 1988.

[13] Eugene Lawler. *Combinatorial Optimization: Networks and Matroids.* Saunders College Publishing, 1976.

[14] A. Nicolau, K. Pingali, and A. Aiken. Fine-grain compilation for pipelined machines. Technical Report TR-88-934, Department of Computer Science, Cornell University, Ithaca, NY, 1988.

[15] Pierre Peladeau. On the length of the cyclic frustrum in a sdsp-pn. Technical Report ACAPS Technical Note 31, McGill University, Montreal, 1991.

[16] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Workshop on Microprogramming*, pages 183–198, 1981.

[17] Raymond Reiter. Scheduling parallel computations. *Journal of the ACM*, 15(4):590–599, 1968.

[18] U. Schwiegelshohn, F. Gasperoni, and K. Ebcioğlu. On optimal loop parallelization. Technical Report RC-14595, IBM T. J. Watson Research Center, 1989.

[19] R. F. Touzeau. A FORTRAN compiler for the FPS-164 scientific computer. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 48–57, June 1984.

6 Conclusions

Conventional software pipelining performs loop scheduling by computing a static parallel schedule to overlap instructions of a loop body from different iterations. An advantage of software pipelining is that it provides a direct way of exploiting parallelism across all iterations of the loop. This is achieved without the explicit use of loop unrolling, and results in highly compact object codes. Software pipelining has been proposed as a powerful loop scheduling method for high-performance architectures.

On the other hand, loop unrolling followed by instruction scheduling on the unrolled loop body has also been proposed as an effective method for loop scheduling. After unrolling, the loop body contains the instructions resulting from several iterations, hence instruction scheduling from multiple iterations becomes possible.

An interesting question is: What is the relation of the software pipelining approach to the unrolling plus instruction scheduling approach, in terms of scheduling time optimality? In this regard, the finding reported in [11] shows that in order to achieve time-optimality, existing software pipelining methods should be applied on the loop first unrolled an appropriate number of times. This paper provides an answer to the above question in a unified mathematical framework.

We share the points of view put forward by [10]: Loop unrolling and software pipelining are among the most important methods proposed for fine-grain loop scheduling, and the suitability of the two techniques and their relation remain to be "the most significant open research area in pipelined processor design"[10]. There are important aspects this paper has not addressed. One question is the handling of resource constraints in the mathematical framework proposed in this paper. At this moment, we can only suggest to apply list-scheduling on the pattern of the optimal schedule. Another question is how to extend the proposed method to multidimensional loops. Finally, an implementation of this method in a real life compiler will also be an interesting task by itself. We are currently exploring these problems.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Co., 1974.

[2] A. Aiken. Compaction-based parallelization. (PhD thesis), Technical Report 88–922, Cornell University, 1988.

[3] A. Aiken and A. Nicolau. Optimal loop parallelization. In *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, June 1988.

[4] A. Aiken and A. Nicolau. A realistic resource-constrained software pipelining algorithm. In *Proceedings of the Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[5] Ron Cytron. Doacross: Beyond vectorization for multiprocessors. *Proceedings of the 1986 International Conference on Parallel Processing*, pages 836–844, August 1986.

each statement, and the pattern instances move from one iteration to the next. When $r > 1$, the first pattern that can be observed contains the "slice" of statements that start executing between $t = A^* - T + 1$ and $t = A^*$ where

$$A^* = \lfloor \frac{T \cdot (r-1) + A}{r} \rfloor.$$

(The value of A^* represents the latest starting time of the first r iterations.) This slice contains exactly r instances of each statement, and the instance of the pattern move by step of r iterations. This is a direct consequence of the fact that the time distance between two successive iterations is either $\lfloor \frac{T}{r} \rfloor$ or $\lceil \frac{T}{r} \rceil$.

5 Polynomial bounds on the scheduling parameters

In this section, we give some boundaries on the values of the parameters of the optimal schedule such as T, r, r/T,m and the pattern size. We use the following notations: D for the largest distance dependence ($D \geq 0$) and E for the largest operator delay ($E \geq 1$).

5.1 Bound on T, r and pattern size

The following can be easily derived.

- The maximal number of dependences in a cycle is n.
- The maximal distance dependence of a cycle is $n \cdot D$. The minimal one is 1. (If $D = 0$, there can be no cycle.) Thus the periodicity r is in $[1, D\ n]$, and the pattern size is in $[n, D\ n^2]$ (As explained before, the pattern size is $r\ n$.)
- The maximal number of nodes in a cycle is n. The maximal operation delay of a cycle is $n \cdot E$. The minimal one is 1. Yet, if ther is no cycle, T can be zero. Thus, $T \in [0, E\ n]$.

5.2 Bound on the computation rate

If there is no cycle in the graph, the best schedule is obtained with $\frac{T}{r} = \frac{0}{1}$. In that case, the N iterations are performed in parallel and the computation rate is infinite.

Consider now that there is at least one cycle. The ratio $\frac{T}{r}$ must obey, for any cycle, the constraint (9). Since $T \in [1, n \cdot E]$ and $r \in [1, n \cdot D]$, clearly $\frac{T}{r} \in [\frac{1}{n \cdot D}, n \cdot E]$. However, the range of T/r is smaller because when $r = n \times D$, the operator delay of the cycle is at least n. So, $\frac{T}{r} \in [\frac{1}{D}, n \cdot E]$. The computation rate r/T is therefore in $[\frac{1}{n \cdot E}, D]$.

4.2 Optimal schedule of a fixed periodicity

Consider now the problem of finding the fastest schedule of the form (4) but whose periodicity r is fixed to a given value r_0. As explained in the introduction, this might be interesting to reduce the size of the pattern. The motivation for keeping the pattern small is twofold: (1) The code size has a strong impact on the performance if I-cache is overflowed, and (2) the number of registers may place a constraint on the pattern size.

This problem can be solved by computing one particular shortest path problem when the optimal computation rate is known. Let T_{opt}/r_{opt} be this optimal value. Clearly the minimal value of T, here noted T_0, that wil lead to a valid schedule of periodicity r_0 is, in order to have $T_0/r_0 \geq T_{opt}/r_{opt}$,

$$T_0 = \lceil \frac{T_{opt} \cdot r_0}{r_{opt}} \rceil .$$

The values of $A[v]$'s leading to a valid schedule of rate r_0/T_0 can then be found by solving the integer programming problem associated to the constraints

$$A[v] - A[u] \leq T_0 \cdot d[u][v] - r_0 \cdot E[u], \tag{12}$$

for all dependence distance $d[u][v]$. As explained before, this is a particular shortest path problem.

4.3 Optimal schedule of a given computation rate

Assume now that we want to find a schedule of a given rate r_0/T_0. Then, the values of $A[v]$'s leading to a valid schedule can be found by solving the integer programming problem associated to the constraints (12), i.e. a particular shortest path problem.

4.4 Pattern size

If $T = 0$, the loop is a *forall*. Otherwise, the *pattern* of the loop constains r instances of each statement. So, it is of size nr. Also, the new loop (the one that will repeatedly execute the pattern) is of the form

$$\textbf{for } i = 0 \textbf{ to } (N-1) \text{ div } r \textbf{ step } r \textbf{ do begin}$$

All these facts can be observed by looking at different examples and analysing where the pattern can always be located. Let A be the maximal value of $A[v]$:

$$A = max(A[0], A[1], ..., A[n-1]).$$

Wen $r = 1$, the first pattern that can be observed contains the "slice" of statements that start executing between time $t = A - T + 1$ and $t = A$. Clearly, this slice contains one instance of

Consider any simple cycle c of the dependence graph. To this cycle corresponds a particular value of the ratio

$$\frac{\sum_c E[u]}{\sum_c d[u][v]},$$

where $\sum_C E[u]$ is the sum of the operator delays along that cycle and $\sum_c d[u][v]$ is the sum of the dependence vectors. Clearly, because of (8) and because it is a cycle, the value of T/r must satisfy the constraint:

$$\frac{T}{r} \geq \frac{\sum_c E[u]}{\sum_c d[u][v]}. \tag{9}$$

This constraint must be satisfied for all simple cycle of the dependence graph. In other words, the problem is to find the simple cycle c of minimum ratio

$$\frac{\sum_c d[u][v]}{\sum_c E[u]}. \tag{10}$$

This is a particular application of the *Minimum Cost-to-Time Ratio Cycle (MCTRC)* problem where the cost is given by the dependence values $d[u][v]$'s and the operator delays by the values of $E[u]$'s.

Let us briefly recall the complexity for solving that problem. A full description of the algorithm can be found in [13]. But basically, the idea is to use a binary search to find the appropriate value of T/r. By doing so, $\mathcal{O}(\lg n)$ particular shortest path problems are to be solved. (It is shown hereafter that the test of a particular value of T/r is achieved by solving a particular shortest path problem.) As explained in [13], the shortest path problem is solved in $\mathcal{O}(mn)$ when using Bellman-Ford algorithm. Because m is here in $\mathcal{O}(n)$, the complexity is in $\mathcal{O}(n^2)$. This complexity is further improved when using the recent Gabow-Tarjan's algorithm [8] which is in $\mathcal{O}(\sqrt{n}\ n \lg n)$ in our case. Thus the overall complexity is clearly polynomial. It is in $\mathcal{O}(n^2 \lg n)$ when using Bellman-Ford algorithm and in $\mathcal{O}(\sqrt{n}\ n \lg^2 n)$ when using Gabow-Tarjan algorithm.

In [17], it is proven that the minimum value of (10) computed on all simple cycles of the dependence graph gives the maximal computation rate that can ever be achieved by any schedule. This result is directly applicable here.

Note that once the value of T/r is known, the values of $A[v]$'s can be easily computed. Indeed, these values must obey the constraints (8) which can be rewritten

$$A[v] - A[u] \leq T \cdot d[u][v] - r \cdot E[u]. \tag{11}$$

As explained in [13], this problem is a particular shortest path problem.As explained here above, it can be solved in $\mathcal{O}(n^2)$ with Bellman-Ford algorithm and in $\mathcal{O}(\sqrt{n}\ n \lg n)$ with Gabow-Tarjan algorithm.

Theorem 1 *The schedule of the form (4) that achieves the maximal computation rate can be computed in polynomial time.*

Proof See the above discussion.

■

where $E[u]$ represents the execution time of node u. With a schedule of the form (4), this constraint (5) is equivalent to

$$\lfloor \frac{T \cdot i + A[v]}{r} \rfloor - \lfloor \frac{T \cdot (i - d[u][v]) + A[u]}{r} \rfloor \geq E[u]. \tag{6}$$

If a schedule satisfies the data dependence constraints, we say that it is *valid*. To find a valid schedule of the form (4), we first consider as in [17] the use of the rational schedule

$$t^*(i, v) = \frac{T \cdot i + A[v]}{r}.$$

We will call $t^*(i, v)$ the approximant of $t(i, v) = \lfloor t^*(i, v) \rfloor$.

Lemma 4.1.1 *If $t^*(i, v)$ is valid then $\lfloor t^*(i, v) \rfloor$ is valid.*

Proof By definition, $t^*(i, v)$ is valid if and only if

$$t^*(i, v) \geq E[u] + t^*(i - d[u][v], u) \tag{7}$$

for all dependence $d[u][v]$. Because $E[u]$ is integer, we have

$$\lfloor t^*(i, v) \rfloor \geq E[u] + \lfloor t^*(i - d[u][v], u) \rfloor$$

for all $d[u][v]$. Therefore $\lfloor t^*(i, v) \rfloor$ is valid.

■

Lemma 4.1.2 *The computation rate of $t^*(i, v)$ is equal to the one of $\lfloor t^*(i, v) \rfloor$.*

Proof The computation rate of $t^*(i, v)$ is r/T. On the other hand, $\lfloor t^*(i, v) \rfloor - \lfloor t^*(i+r, v) \rfloor = T$, for all i and v. So, the computation rate is also r/T.

■

Because of the two above lemmas, the strategy to compute a schedule of the form (4) is to find its approximant first. The approximant is subject to the constraint (7) for all $d[u][v]$. This constraint is equivalent to

$$T \cdot d[u][v] + A[u] - A[v] \geq r \cdot E[u]. \tag{8}$$

In [17], it is shown that the maximal value of r/T that satisfy these constraints for all $d[u][v]$ of the dependence graph is a particular application of the well-known *Minimum Cost-to-Time Ratio Cycle (MCTRC)* problem. Let us recall this problem here.

portion of a loop's execution history in this way reveals some repetitive behavior, which can be used to obtain a good schedule [3, 9].

A drawback with the greedy schedule is that gaps of increasing length can appear between the statements. In [3], a scheduling method is presented that keeps the gaps of constant minimal size without affecting the optimality of the final code. Yet, the pattern cannot be kept of polynomial size.

On the other hand, some people suggested the use of *regular* schedules [12, 5]. In that case, all iterations are scheduled in the same manner and the pattern is equal to the size of the original loop body. However, these schedules can not always achieve the maximal computation rate as demonstrated by the second example of section 2. In [11], a solution has been proposed that a loop be first unrolled x times before the application of Lam's scheduling method [12]. It was demonstrated that this combined method may achieve the maximal computation rate with a properly chosen x.

In this paper, we are interested in a uniform analytical framework to express the time-optimal loop scheduling problem for software pipelining, without requiring an extra unrolling phase. We are interested in x-periodic schedules, since they can always achieve the maximal computation rate and we present an efficient analytical method for computing them. We suggest to use schedules of the form

$$t(i,v) = \lfloor \frac{T \cdot i + A[v]}{r} \rfloor, \tag{4}$$

where $t(i,v)$ represents the starting time of the i$^{\text{th}}$ instance of statement S_v. The parameters of such a schedule are the following.

- T is an integer called the *period* (or *initiation interval*) of the schedule.
- r is a non-negative integer called the *periodicity*; T and r are relatively prime.
- $A[v]$ ($v \in [0, n-1]$) is a positive integer called the *offset* for statement S_v.

This class of schedules has shown to be very useful. In [17], its was proven that these schedules can always achieve the optimal computation rate for Karp-Miller computation graphs. We therefore apply this technique to loop scheduling.

Given a schedule of the form (4), its *computation rate* is r/T. Indeed, every T time units, a new set of r iterations starts executing. Note that in the particular case when $T = 0$, the computation rate is infinite. In loop scheduling, this simply means that the loop can be executed as a *forall*, all instances in parallel. An example of such a case is given in the appendix.

4.1 Statement of the problem

Our problem is to find the schedule of the form (4) that maximizes r/T while satisfying all the data dependence constraints. For all dependence $d[u][v]$ of the dependence graph, the schedule must satisfy

$$t(i,v) - t(i - d[u][v], u) \geq E[u], \tag{5}$$

We now introduce some useful notations. Consider statement S_v. The number of its input arguments will be noted $L[v]$. Consider its k$^{\text{th}}$ argument in particular ($0 \leq k \leq L[v]-1$). The place where this argument is being computed will be noted $U[v][k]$ and its input dependence $D[v][k]$. So, if this argument is $b(i-d)$, $D[v][k] = d$, and $S_{U[v][k]}$ is of the form

$$b[i] = ...$$

Let us now recall the definition of *dependence graph*. The dependence graph contains as many node as there are statements in the loop body. Each of its n nodes represents in fact a particular statement. Each node v is labeled $E[v]$ which represents the time it takes to execute S_v. When all statements are unit-delays, $E[v] = 1$. The arcs between the nodes represent dependences between the associated statements. There is an arc going from node u to node v if S_v has for input argument the output variable of S_u. Now, if there are several dependences from node u to node v, only the one of minimal value is of importance (the others can be safely eliminated); the arc will be noted with that value $d[u][v]$. In other words,

$$d[u][v] = min_k(D[v][k], \text{ with } U[v][k] = u).$$

It is important to observe that the number of arcs of the dependence graph, which will be noted m, is in $\mathcal{O}(n)$. Indeed, for each node, the number of incoming arcs is less than the number of arguments of the corresponding statement. Each statement being *elementary*, the number of arguments is a constant independent on n. Thus, m is in $\mathcal{O}(n)$.

Example 2.1 (cont'd) Here, $n = 3$.

- $L[0] = 1$, $D[0][0] = 1$ and $U[0][0] = 0$.
- $L[1] = 2$, $D[1][0] = 0$, $D[1][1] = 1$, $U[1][0] = 0$ and $U[1][1] = 2$.
- $L[2] = 2$, $D[2][0] = 0$, $D[2][2] = 1$, $U[2][0] = 1$ and $U[2][1] = 1$.

The associated dependence graph is shown in figure 4(a). It consists of three nodes, here noted a, b and c and the arc labels are the following: $d[a][a] = 1$, $d[a][b] = 0$, $d[b][c] = 0$, and $d[c][b] = 1$. For example,

$$d[b][c] = min(D[2][1], D[2][2]),$$

because both $U[2][1]$ and $U[2][2] = 1$ (=b).

□

4 Periodic schedules

One approach for scheduling a given loop is by looking at a partial execution history of the loop, say the first i iterations [3]. A *greedy schedule* is obtained when the statements of those i iterations are scheduled as soon as they are ready for execution. Scheduling a large enough

Our program model consists of a class of *loops* which define a number of arrays (vectors). To simplify the model, we allow loops of the following form:

$$\begin{array}{ll} \textbf{for } i = 0 \textbf{ to } N-1 & \textbf{do} \\ & S_0; \\ & S_1; \\ & \ldots \\ & S_{n-1}; \\ & \textbf{od} \end{array}$$

which has the following features:

- The loop has a unit step size.
- Each of the n statements in the loop body, S_v $(0 \leq v \leq n-1)$, is an assigment statement defining an array (vector). All the statements in the body are of the form:

$$a[i] = f(b[i-d], ...) \tag{3}$$

  where

  - a and b are two vectors of N entries computed by two statements in the loop
  - $a[i]$ is the i$^{\text{th}}$ component of a
  - $b[i-d]$ is some component of b
  - d is a constant integer
  - f is some elementary function whose arguments are either external inputs or are of the form $b[i-v]$

In (3), the constant d is called a *dependence distance*. When $d = 0$, we call it *intra-body dependence* else it is a *loop-carried dependence*.

The basic model provides a framework to capture a reasonably wide class of loops important in scientific computation. First, the majority of time in such programs is spent in innermost loops. The main data structures computed in such loops are *arrays* and a significant fraction of their index expressions are in the form allowed by our model. For example, statements of the form

$$a'[i-u] = f(b[i-d], ...)$$

can be temporally transformed into (3) by simple renaming of $a'[i-u]$ into $a[i]$; the input arguments of the form $a'[i-d']$ must then be transformed accordingly into $a[i+u-d']$. The kernel of the body are statements defining arrays. If an assignment to a scalar variable appear in the loop body, it can be temporally transformed into an array by *scalar expansion* — a technique frequently used in vectorization/parallelization. Also, if the index range of the loop is different from $[0, N-1]$, and/or if the step of the loop is different from one, the loop can be first normalized.

Some statements of the loop can be conditional. This particular case requires a particular attention, as explained in the appendix.

time	i=0	i=1	i=2	i=3	i=4
0	a f				
1	b g e				
2	c h	f			
3	d	g a			
4	i	h b e			
5	j	c	a f		
6		d	b g		
7		i	c e h		
8		j	d	a f	
9			i	b g e	
10			j	c h	f
11				d	g a
12				i	h b e
13				j	c
14					d
15					i
16					j

Figure 3: For the loop (2), the optimal schedule is 3-periodic.

The *pattern* of the loop includes the instructions executed from time $t = 3$ to time $t = 10$ included. Its size is $3n$, where n is the number of instructions in the original loop. Also, the pattern is a loop of the form:

for $i = 0$ **to** $(N - 1)$ div 3 **step** 3 **do begin**

The postlude will also be a bit more complicate than in the 1-periodic case because it will depend on the value of N mod 3.

The scheduling method proposed in this paper enables the user to automatically find this schedule of optimal computation rate. Other methods are restricted to 1-periodic schedules [12]. Using that model in this example would result in a computation rate of 1/3 instead of 3/8, which is 12.5% slower. In [11], it is shown that x-periodic schedules can be found by unrolling the loop x times and by applying a 1-periodic schedule on the unrolled loop. This is therefore an alternative to the one-step analytical procedure presented hereafter.

In this paper, we have also extended our procedure to find the optimal schedule of a given periodicity. This will enable the user to find efficiently for example this 1-periodic schedule of rate 1/3 whose resulting pattern and postlude are simpler.

□

3 Program Model

In this section, we introduce the program model to be handled by our transformation techniques. We take the opportunity to recall the notion of dependence graph and its connected components.

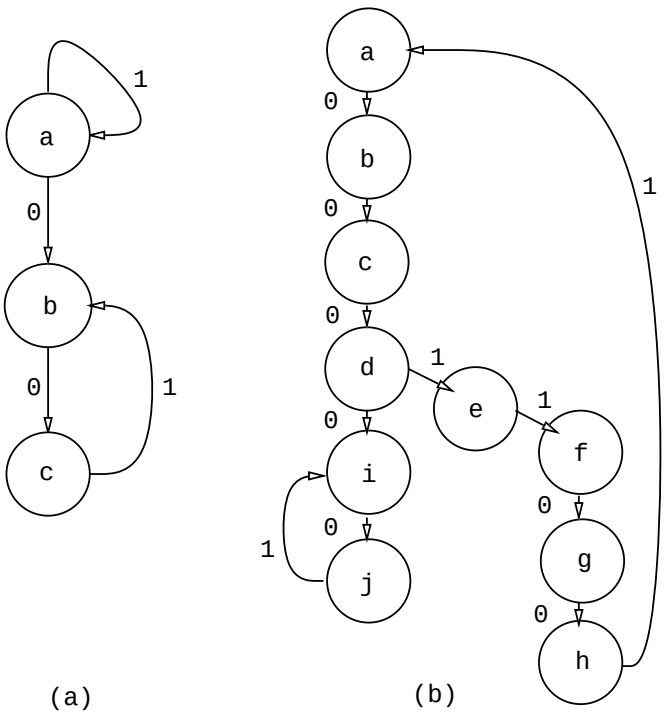

Figure 2: Dependence graphs: (a) for the loop (1) and (b) for the loop (2).

$$
\begin{array}{ll}
\textbf{for } i = 0 \textbf{ to } N-1 & \textbf{do} \\
& a[i] = h[i-1] + R[i]; \\
& b[i] = a[i] + a[i-1]; \\
& c[i] = b[i] + b[i-1]; \\
& d[i] = c[i] + c[i-1]; \\
& e[i] = d[i-1] + d[i-2]; \\
& f[i] = e[i-1] + e[i-2]; \\
& g[i] = f[i] + X[i]; \\
& h[i] = g[i] + Y[i]; \\
& i[i] = d[i] + j[i-1]; \\
& j[i] = i[i] + Z[i]; \\
& \textbf{od}
\end{array}
\tag{2}
$$

Assume that all instructions are unit-delays. The optimal schedule is then given by

$$t(i, v) = \lfloor \frac{8 \cdot i + A[v]}{3} \rfloor,$$

with $A[f] = 0$, $A[a] = 1$, $A[g] = 3$, $A[b] = 4$, $A[e] = 5$, $A[h] = 6$, $A[c] = 7$, $A[d] = 10$, $A[i] = 13$, and $A[j] = 16$. The resulting schedule of the first five iterations is shown in figure 3. Clearly, the first three iterations are scheduled differently, but the fourth iteration is scheduled as the first one, etc. We call such a schedule *3-periodic*.

with $A[a] = 0$, $A[b] = 1$, $A[c] = 2$ (where $A[a]$, $A[b]$ and $A[c]$ denote the values of $A[v]$ for the instructions computing $a[i]$, $b[i]$ and $c[i]$ respectively). The computation rate is 1/2; every two units of time, a new iteration begins. The resulting schedule of the first three iterations is shown in figure 1. All iterations are clearly scheduled the same manner. We call this type of schedule *regular* or *1-periodic*.

time	i=0	i=1	i=2
0	a		
1	b		
2	c	a	
3		b	
4		c	a
5			b
6			c

Figure 1: For the loop (1), the optimal schedule is 1-periodic.

Clearly, the instructions executed from time $t = 2$ to $t = 3$ form a *pattern* for the loop. In other words, the loop can be rewritten in terms of this *pattern*, a *prelude* (instructions before the pattern) and a *postlude* (instructions after the pattern). The loop (1) is rewritten:

```
%prelude
a[0] = a[-1] + R[0];
%pattern
 for i = 0 to N - 2    do
                       b[i] = a[i] + c[i - 1];
                       a[i + 1] = a[i] + R[i + 1]||
                       c[i] = b[i] + b[i - 1];
                        od
%postlude
b[N - 1] = a[N - 1] + c[N - 2];
c[N - 1] = b[N - 1] + b[N - 2];
```

Note that in the *pattern*, the two instructions $a[i + 1]$ and $c[i]$ are executed in parallel; this is here simply noted with the delimiter "||". Note also that in this case, the number of instructions in the pattern is equal to n, the number of instructions in the original loop. □

Example 2.2 Consider the following example whose dependence graph is shown in figure 2(b).

- We demonstrate the feasibility of the proposed method by (1) presenting a polynomial time algorithm to find the *optimal schedule* in this r-periodic form that maximizes the computation rate (in fact, we show that this schedule maximizes the computation rate theoretically possible) (2) establishing polynomial bounds for the optimal schedule, i.e. bounds on the period, the periodicity r, the pattern size, and the computation rate.
- Practical constraints, such as the size of high-speed program memory (e.g. instruction cache) and register files, may limit the size of the periodic pattern. Instead of using the maximal value of r for the r-periodic schedule, some other given value may have to be used. A fast automatic method for computing such a schedule is derived.

This paper is organized as follows. In section 2, we present two examples of loops scheduled with the proposed model. In section 3, we present our program model; i.e., the type of loops to which our scheduling technique applies. We also review the notion of dependence graph as a basis for later sections. Then, in section 4, the main results of this paper are presented. In section 5, boundaries on the parameters of the schedule are given. Section 6 presents our conclusions and ideas for future work.

In the appendix, there are three subsections. In the first one, we recall the notion of connected components of a dependence graph, and we show how these can be used to further accelerate the computation of the schedule. In the second one, an example of a forall loop is given and treated with the general scheduling technique of this paper. In the last one, we illustrate what can be done in the presence of conditional branches. We introduce two models for expressing conditionals, namely the speculative one and the non-speculative. We show how to optimize the execution rate by adding conditional jumps to the code; the resulting schedule is then *dynamic*.

2 Motivating examples

We hereafter consider two examples of loop scheduling. The first one is very simple. Its schedule of optimal computation rate is 1-periodic. In that case, the loop is easily rewritten in terms of its *prelude* (sometimes called *prologue*), *pattern* and *postlude* (sometimes called *epilogue*). The second example illustrates the case when the optimal schedule is 3-periodic. In the appendix, two additional examples are presented: one illustrates when the loop can be rewritten as a **forall** (to illustrate the fact that the forall is treated uniformally as a special case of the general framework), and the other contains *conditional statements*.

Example 2.1 Consider the following simple example.

$$\begin{array}{l} \textbf{for } i = 0 \textbf{ to } N-1 \quad \textbf{do} \\ \qquad a[i] = a[i-1] + R[i]; \\ \qquad b[i] = a[i] + c[i-1]; \\ \qquad c[i] = b[i] + b[i-1]; \\ \textbf{od} \end{array} \tag{1}$$

The dependence graph of this loop is shown in figure 2(a). (For those who are not familiar with the concept of dependence graph, see the next section.) If all instructions take one unit of time, the schedule of maximal computation rate is

$$t(i, v) = 2 \cdot i + A[v],$$

from a number of consecutive iterations. In the technique proposed in [3], the instructions are scheduled as early as possible by a *greedy* machine model with the constraint of having *bounded gaps* between the computations. This technique is called *OPTimal loop parallelization (OPT)* (a particular case of *Perfect Pipelining*). In [9], the loop is modeled by a timed Petri net executed under the "earliest firing rule". There are two problems with this approach: (1) for loops with multiple *critical cycles*, no polynomial time bound is known to occur for such a periodic pattern; (2) the patterns may be exponential in size under the earliest firing rule [15].

The second approach generates a fixed scheduling pattern for one iteration, and repeats it for all iterations with a specific initiation time interval [12]. Unlike the operational approach described above, this method gives a mathematical (linear programming) formulation of the scheduling problem, and tries to minimize the initiation interval between the successive iterations. One problem with this approach is that the formulation of the problem may not be optimal; i.e., the requirement that only regular schedules (such that all iterations are scheduled in the same manner) are considered may exclude some optimal solutions. This was demonstrated in [11], and an improvement was suggested in which the loops may first be subjected to an *unrolling* transformation. The work along this line has mainly focused on scheduling problems with resource constraints — which are NP-hard [12]. As a result, both the original and improved methods are based on heuristic job-scheduling techniques such as the *list-scheduling* method to find an approximate optimal solution.

In this paper, we propose a different alternative, based on an elegent and powerful scheduling technique for Karp-Miller computation graphs [17]. Using this technique, the schedule is of the form

$$t(i,v) = \lfloor \frac{T \cdot i + A[v]}{r} \rfloor,$$

where the parameters T, r and $A[v]$ are integers, and $t(i,v)$ is the starting time of the i^{th} occurrence of the v^{th} statement of the loop body. That is, the schedule of all statement is periodic with periodicity r, period T and offset time $A[v]$. Surprisingly, this mathematical form of schedule and its exact analytical solution didn't receive a great deal of attention from previous work in the area of software pipelining.

There are several reasons which motivated us in adopting this formulation to fine-grain loop scheduling. Intuitively these reasons are: (1) The formulation does not have the restriction that the schedule must follow an earliest firing rule. In fact, the values computed for the offsets in general will be different from those derived from the methods of [3, 4, 9]. (2) The formulation also does not have the restriction that the schedule of each individual iteration must be the same. In fact, the schedule of two consecutive iterations may not be the same, but will repeat every r iterations; hence called *r-periodic*. (3) As a result, the formulation provides a uniform analytical framework to express the time-optimal loop scheduling problem. That is, we have a single mathematical formulation which considers both software pipelining and loop unrolling.

The main contributions of this paper are the following.

- We propose to express the fine-grain loop scheduling problem (in particular, software pipelining) on the basis of the mathematical formulation of r-periodic scheduling. It overcomes some of the problems encountered by existing software pipelining methods as described above.

A Polynomial Time Method for Optimal Software Pipelining

Vincent H. Van Dongen
Centre de recherche informatique de Montréal

Guang R. Gao
McGill University, School of Computer Science

Abstract

Software pipelining is one of the most important loop scheduling methods used by parallelizing compilers. It determines a static parallel schedule – a periodic pattern – to overlap instructions of a loop body from different iterations. The main contributions of this paper are the following: First, we propose to express the fine-grain loop scheduling problem (in particular, software pipelining) on the basis of the mathematical formulation of *r-periodic scheduling*. This formulation overcomes some of the problems encountered by existing software pipelining methods. Second, we demonstrate the feasibility of the proposed method by (1) presenting a polynomial time algorithm to find an *optimal schedule* in this r-periodic form that maximizes the computation rate (in fact, we show that this schedule maximizes the computation rate theoretically possible), and by (2) establishing polynomial bounds for the optimal schedule, i.e. bounds on its period, its periodicity, the pattern size, and the computation rate. Practical constraints, such as the size of high-speed program memory (e.g. instruction cache) and register files, may limit the size of the periodic pattern. Instead of using the maximal value of r for the r-periodic schedule, some other given value may have to be used. A fast automatic method for computing such a schedule is derived.

In an appendix, we illustrate what can be done in the presence of conditional branches.

1 Introduction

The efficient exploitation of fine-grain parallelism in loops has been a major challenge in the design of optimizing compilers for high-performance computer architectures. Fine-grain loop scheduling involves the detection of parallelism from a partially ordered set of operations of the loop body which are to be performed repetitively over a sequence of iterations. With loop-carried dependencies, the dependency relation between operations in the loop is a digraph with cycles, thus presenting a specific challenge to the effectiveness of classical scheduling theory.

Recently, *software pipelining* has been proposed for fine-grain loop scheduling. This method derives a static parallel schedule – a periodic pattern – that overlaps instructions of a loop body from different iterations. Software pipelining has been proposed for high-performance pipelined processor architectures, as well as super-scalar and VLIW architectures [2, 3, 4, 6, 7, 12, 14, 16, 19].

To motivate our work, we briefly discuss two existing approaches for software pipelining. In the first approach [3, 4, 9], a compiler emulates the execution of the loop under a certain operational model, until a periodic pattern is "discovered" which often consists of instructions

List of Figures

1 For the loop (1), the optimal schedule is 1-periodic. 7

2 Dependence graphs: (a) for the loop (1) and (b) for the loop (2). 8

3 For the loop (2), the optimal schedule is 3-periodic. 9

4 A dependence graph (a) and its two connected components (b). 20

5 A 1-periodic schedule of infinite computation rate. 21

6 An optimal 2-periodic schedule of fixed computation rate 1/2. 21

7 Dependence graphs: (a) with no speculation and (b) with speculation. 23

8 Static schedule when speculation is used. 23

9 Dynamic schedule when speculation is used. 24

Contents

1 Introduction 4

2 Motivating examples 6

3 Program Model 9

4 Periodic schedules 11

4.1 Statement of the problem 12

4.2 Optimal schedule of a fixed periodicity 15

4.3 Optimal schedule of a given computation rate 15

4.4 Pattern size 15

5 Polynomial bounds on the scheduling parameters 16

5.1 Bound on T, r and pattern size 16

5.2 Bound on the computation rate 16

6 Conclusions 17

McGill University
School of Computer Science

A Polynomial Time Method for Optimal Software Pipelining

submitted to CONPAR92

ACAPS Technical Memo 37
March, 1992

Vincent H. Van Dongen †
Guang R. Gao

Advanced Computer Architecture and Program Structures Group

†Centre de recherche informatique de Montréal

3480 University St • Montreal • Canada • H3A 2A7