# Speedup of Banded Linear Recurrences in the Presence of Resource Constraints *

Alexandru Nicolau
nicolau@ics.uci.edu

Haigeng Wang
wang@ics.uci.edu

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92714

Technical Report Number 92-12

December 24, 1991

**Abstract**

An $m$-th order linear recurrence system of $N$ equations computes $x_i = c_i + \sum_{i-1}^{j=i-m} a_{ij}x_j$ for $1 \leq i \leq N$. Linear recurrences have a role of central importance in computer design, numerical analysis, program analysis, image processing and vision. However, programs containing banded linear recurrences are difficult to parallelize due to loop-carried dependences. In this paper, we first present a family of schedules, called the *exact schedules*, for parallel evaluation of low order ($m \leq 2$) banded linear recurrences with an execution time $(2m^2+3m)N/(p+(m(m+1)(2m+1))/(2(2+\lfloor \log m \rfloor)))$ for $0 < m \leq 2$ , $N > (p+5)(2p+3)/6$ and number of processors $p > m$. We show that the exact schedules achieve the strict time lower bound under matrix multiplication model. Next, we derive another family of schedules, called the *pipelined schedules*, with better program-space efficiency and with an execution time of *pipeline startup time* $+ (2m^2+3m)N/(p+(2m+1)/2)$ for $m = 1$, and *pipeline startup time* $+ (6m+2)N/(p+(2m+1)$ for $m > 1, m < p \leq 4m+1$, and *pipeline startup time* $+ (2m^2+3m)N/(p+(m-1)(2m+1))$ for $m > 1, p > 4m+1$. This is the first parallel algorithm that achieves this time bound, which improved the fastest prevoiusly published algorithm by a factor $\geq (p+2m^2-m-1)/(p+m+1/2)$ for $m > 1$. We illustrate the technique by parallelizing loops containing linear recurrences and demonstrate the available speedup on a VLIW architecture with experimental results obtained using our pipelined schedules.

## 1 Introduction

An *m-th order linear recurrence system of n equations* computes $x_i = c_i + \sum_{i-1}^{j=i-m} a_{ij}x_j$ for $0 < m < N$, $1 \leq i \leq N$. If the order $m$ is a fixed number independent of the problem size $N$, the linear recurrence is called a *banded linear recurrence.*

*This work was supported in part by NSF grant CCR8704367 and ONR grant N0001486K0215.

```
for(i = 1; i <= N; i++)
  x[i] = c[i] + a[i] * x[i - 1];
```

```
for(i = 1; i <= N; i++)
  {for(j = i - m; j < i; j++)
    x[i] = x[i] + a[i][j] * x[j];
  x[i] = x[i] + c[i]; }
```

Figure 1: (a) first-order linear recurrence. (b) higher-order linear recurrence.

The sequential program for the first and higher order linear recurrence can be written as follows, Clearly, the two loops above cannot be significantly parallelized (because of the loop-carried dependences) without breaking the dependences and introducing redundant operations.

Linear recurrences have a role of central importance in computer design, numerical analysis, program analysis [16, 10]. Furthermore, many programs consisting primarily of loops with loop-carried dependences require real-time response and have a very high frequency of use. Typical examples include image processing, vision[18], and control in embedded systems. Since in all such applications speed is of paramount importance, the parallelization of linear recurrence is critical. Automatic loop parallelization techniques have been extensively studied. These techniques [13, 20, 8, 12, 2] have demonstrated good performance subject to preserving loop-carried dependences, i.e., they do not parallelize the loops beyond loop-carried dependences. Thus, there may still exist considerable amounts of parallelism beyond loop-carried dependences that cannot be extracted by dependence-preserving transformations . Furthermore, many programs consisting primarily of loops with loop-carried dependences require real-time response and have a very high frequency of use. Typical examples include image processing, vision[18], and control in embedded systems. To understand how to parallelize loops with loop-carried true dependence beyond dependence-preserving techniques, it is essential to understand the fundamental forms of loops with loop-carried dependences, i.e., banded linear recurrence. The schedules and the technique used to derive them in this paper can be applied—with some extensions— to parallelizing many sequential algorithms containing loop carried true dependences.

Since in practice a fixed number of resources(i.e., functional units, processors) are available, and since the techniques used to overcome dependence limitation introduce some computation overhead(redundant computation), a trade-off between this overhead and the performance achieved on the given resources exists. Thus, it is critical to devise a scheme which performs the computation in optimal time, given the amount of resources available. In particular, since the number of data elements usually far outnumber the number of processors available, it is important to consider situations in which a fixed number of processors independent of the size of the problem $N$— i.e., the number of results in a linear recurrence system— are available. Since the lower order banded linear recurrences($m = 1, 2$) are most frequently seen in real programs containing linear recurrences, it makes sense to consider parallelizing them first. Needless to say, the method for scheduling parallel computation of lower order banded linear recurrences should be general enough to handle higher order linear recurrences.

Since we are interested in extending the technique to handle more general forms of loops with loop-carried true dependence, we are also concerned with other properties of these optimal schedules, such as the clarity, simplicity of implementation, and extendibility.

Due to the importance of the problem, parallel evaluation of linear recurrences has been studied in various forms by some of the pioneers in the field for the past twenty years. Kogge and Stone[15] developed the recursive doubling technique for computing the first order linear recurrence system and defined some properties that extend applicability of their technique to a broader class of problems. Their technique assumes an unlimited number of processors. Ladner and Fischer[17] later found optimal circuits for a prefix of size $N = 2^k$ for $k > 0$ assuming an unlimited number of resources. Note that the prefix in Ladner and Fischer's work is a form of the simplest first-order linear recurrence—-all coefficients $a_{ij}$ equal 1, i.e., there is no multiplication in the recurrence system. Chen and Kuck[3] first gave time and processor bounds for parallel computing general linear recurrences with no resource constraints. Later, Chen, Kuck and Sameh[5] developed an algorithm for computing banded linear recurrence with resource constraints that achieves a time bound of $(n/p)(2m^2 + 3m) + O(n^2 log(p/m)$. Hyafil and Kung[14] established a time bound of $2/3p + 1/3$ for parallel evaluation of the Horner expression, which is equivalent to evaluating the last equation in a first-order banded linear recurrence, i.e., evaluating $x_N$ only without having to compute $x_1, ..., x_{N-1}$. Hyafil and Kung thus does not require the values of all the equations other than the last, while we require the values of all the equations in the recurrence system. Gajski[9] lowered the time bound of [5] further to $(2m^2 + 3m)N/(p + m + 1/2)$ for $p \geq m + 1$ and $N > p^2$ for computing the banded linear recurrence with resource constraints.

For the analysis of our algorithms and to facilitate a formal comparison with previous techniques—to be discussed shortly— we use the parallel random access(PRAM) computation model: a PRAM consists of $p$ autonomous processors, executing synchronously, all having access to a common memory. Each processor can perform an addition or a multiplication in one step, and different processors may perform different types of operations at any time. We consider a PRAM models: concurrent read, exclusive write(CREW). In a CREW model, several processors may read simultaneously from a memory location but exclusive access is required for writes.

Recently, Nicolau and Wang[19] found the strict time lower bound for parallel evaluation of prefix sums to be $2N/(p + 1)$ for $N \geq p(p + 1)$ and $p > 1$ for a fixed number of processors $p$ independent of the problem size. Although prefix sums is a first-order linear recurrence without multiplication, this result provides a number of insights into the optimal schedules for more general forms of recurrences(we shall mention them in the upcoming sections). Based on these insights, we developed a new method that enables us to systematically approach the minimum possible time(strict time lower bound) and to find the algorithm, and the *exact schedules*, that achieve the execution time of

$$\frac{(2m^2 + 3m)N}{p + \frac{(m(m+1)(2m+1))}{2(2+\lfloor \log m \rfloor)}}$$

for $0 < m \leq 2$ and $N > (p+5)(2p+3)/6$, which is a minimum problem size for which the bound can be achieved. This bound improves Gajski's bound by a factor of $(p + (m(m + 1)(2m + 1))/(2(2 + \lfloor \log m \rfloor)))/(p + m + 1/2)$

for $m = 1, 2$ This time bound applies to $N > (p+5)(2p+3)/6 \doteq p^2/3$, a wider range of problem size than that of Gajski's result, $N > p^2$. We define a *period* in the parallel schedule to be the size of a portion of the given linear recurrence such that the said *execution time ratio* of the schedule(the execution time per result, i.e., the total execution time of the schedule divided by $N$) can be achieved. Intuitively, a period is the number of results produced in an iteration of the loop body of a parallel schedule. Thus, a smaller period of a parallel schedule means a small schedule size since the size of a schedule is proportional to the period of that schedule. For any number of processors $p$, an exact schedule has a period of $p^2/3 + p/2$, a much smaller period than that of Gajski's algorithm, $p^2$. Moreover, a modification of the exact schedules we call the *pipelined schedules* further reduce the period at a very small time cost of pipeline startup, which is provably minimum. The pipelined schedule not only proves to be a natural candidate for compiling linear recurrences because of its program-space efficiency, but can also be used, as a much simpler vehicle than the exact schedules, to approach the time bounds for higher order linear recurrences. Using the pipelined schedules, an $m$-th order banded linear recurrence can be evaluated in time

*pipeline startup time* $+ (2m^2 + 3m)N/(p + (2m+1)/2)$, for $m = 1$, and

*pipeline startup time* $+ (6m+2)N/(p+2m+1)$ for $m > 1, m < p \leq 4m+1$, and

*pipeline startup time* $+ (2m^2+3m)N/(p+(m-1)(2m+1))$ for $m > 1, p > 4m+1$ and . This is the first parallel algorithm that achieves this time bound, which improved the fastest prevoiusly published algorithms[5, 9] by a factor $\geq (p + 2m^2 - m - 1)/(p + m + 1/2)$ for $m > 1$.

This paper is organized as follows. Section 2 gives the exact schedules for parallel evaluation of first and second order banded linear recurrence. In this type of schedules, all arithmetic operations complete precisely at the end of each iteration. In Section 3, we derive more program-space efficient pipelined schedules and their generalization for arbitrary order linear recurrences. We then illustrate the technique by parallelizing two Livermore Kernel loops containing linear recurrences using our pipelined schedules, and we give experiment results from running these benchmarks on a parameterized VLIW architecture simulator implementing the IBM VLIW machine[6] in Section 4.

We now define the terms frequently used in this paper. A computation $A$ can be performed using a sequential schedule $A_{seq}$ on a sequential machine or using a parallel schedule $A_{par}$ on a parallel machine [1]. We denote the time to run schedule $A_{par}$ on a PRAM machine as $T_p(A_{par})$ (or $T_p$ or $T$ when unambiguous), where $p$ refers to the number of processors in the machine. We refer to $T_p$ and $T$ interchangeably as execution time, time steps or steps. The time to compute $A$ sequentially is denoted $T_1(A_{seq})$. The *speedup* of a machine with $p$ processors over a uniprocessor, for computation $A$, is denoted $S_p(A) = T_1(A_{seq})/T_p(A_{par})$, or simply $S_p = T_1/T_p$ when unambiguous. The *efficiency* of this computation is $E_p = S_p/p$, which can be interpreted as actual speedup divided by the machine's peak performance(maximum speedup for the machine) using $p$ processors. Let $O_p$ be the number of operations executed in some computation using $p$ processors. We define *operation redundancy* to

[1] We distinguish between our *algorithm* and *schedule* —our algorithm is used to produce a schedule(for parallel evaluation of the given linear recurrence) which, when run on a machine, will actually evaluate the linear recurrence in parallel.

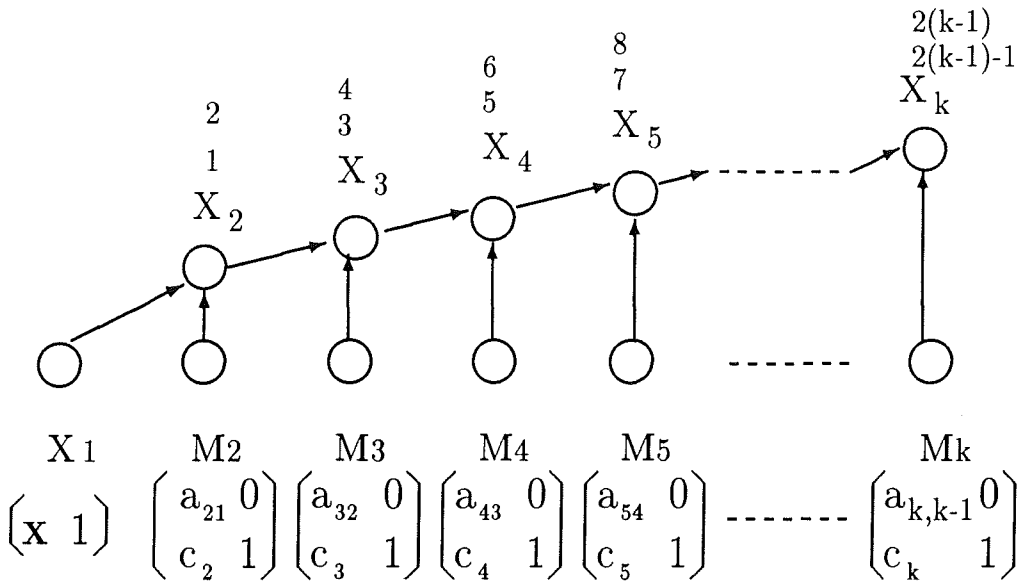


Figure 2: A dependence tree of the sequential computation of a 1st-order linear recurrence

be $R_p = O_p/O_1(\geq 1)$, where $O_1 = T_1$. Finally, we define *utilization* as $U_p = O_p/pT_p \leq 1$, where $pT_p$ is the maximum number of operations that $p$ processors can perform in $T_p$ steps. The *final values* of a linear recurrence are the values computed by the definition of the recurrence, i.e., the values assigned to the left-hand side of the statements in the loop body in Figure 1. The *final operations* of a linear recurrence are operations that compute the final values in the sequential computation. The *redundant operations* are operations that compute auxiliary values introduced by the parallel schedules in an effort to speed up the final values computation. The *redundant values* refer to the auxiliary values computed by the redundant operations. In accordance to the PRAM model and the IBM VLIW model, we assume that each operation takes one step to complete. However, our techniques extend to multi-step operations.

## 2 The Exact Schedules

The sequential evaluation of $N$ first-order linear recurrences can be expressed as a chain of matrix multiplications:

$$[x_k 1] = [x_1 1] \begin{bmatrix} a_{21}0 \\ c_2 1 \end{bmatrix} \begin{bmatrix} a_{32}0 \\ c_3 1 \end{bmatrix} \begin{bmatrix} a_{43}0 \\ c_4 1 \end{bmatrix} \cdots \begin{bmatrix} a_{k,k-1}0 \\ c_k 1 \end{bmatrix}$$

for $1 \leq k \leq N$. This is referred to as the *matrix decomposition* of the evaluation, which was used first by Chen[5], then by Kung[14] and Gajski[9]. The sequential matrix chain multiplication above can be represented as a dependence tree as shown in Figure 2.

The leaves of the computation tree, $X_1, M_2, ..., M_k$, represent the matrices, called the *operand nodes.* The nodes on the top fringe of the tree, $X_1, ..., X_k$, represent the results of each recurrence in the system packaged in matrix form, called the *result nodes* or *final nodes.* A result node represents the result of the multiplication of the two matrices below it, i.e., $X_i = X_{i-1}M_i$ for $2 \leq i \leq k$. In a matrix multiply only a sequence of two

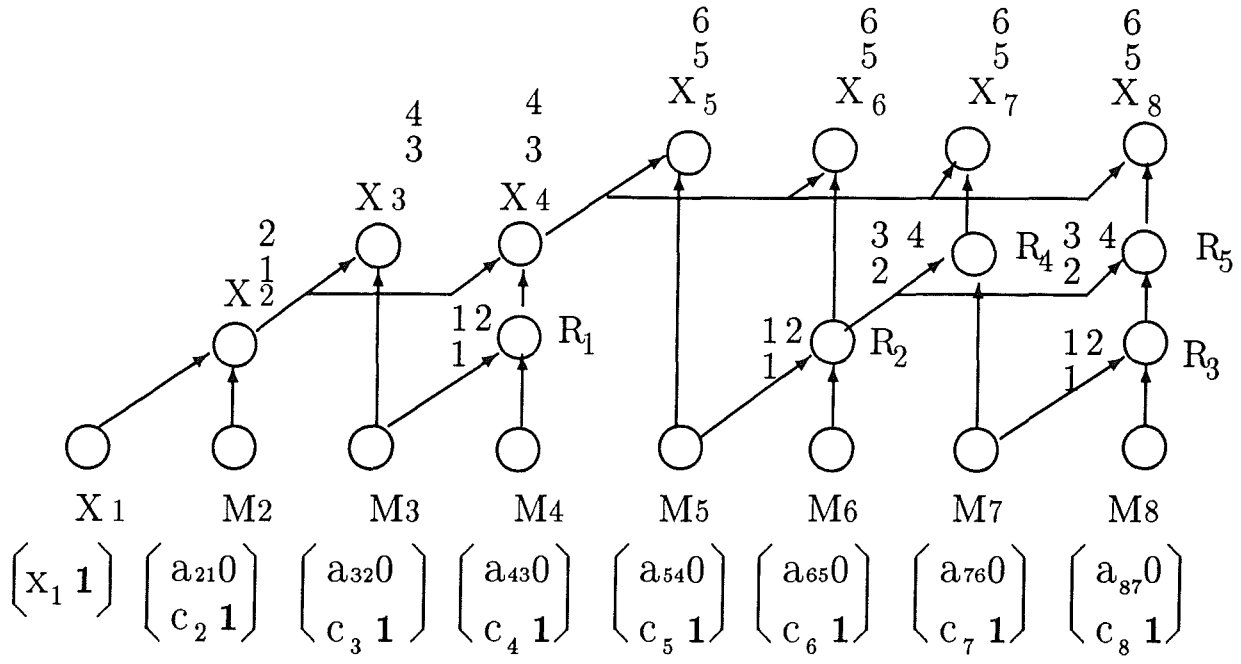


Figure 3: A dependence tree for parallel computation of a 1st-order linear recurrence

arithmetic operations, a multiply followed by an add, are necessarily done. The column of numbers above a result node represents the time steps at which the processor is allocated to execute the operations in that node. The number of operations in a sequential evaluation of $N$ $m$-th order banded linear recurrences is $2mN$. Clearly, the computation above cannot be significantly parallelized without breaking the dependences and introducing redundant operations, because each result node $X_i, 2 \leq i \leq k$, depends on the previously computed result node $X_{i-1}$.

By introducing redundant operations, a parallel evaluation of a system of eight 1st-order linear recurrence can be represented as the tree in Figure 3. In addition to the result nodes and operand nodes, a parallel computation tree uses the *redundant nodes*, $R_1, ..., R_5$ in Figure 3. A redundant node represents the matrix resulted from the multiplication of two nodes with dependence arcs reaching it, for example, $R_1 = M_3M_4$ and $R_4 = R_2M_7$. For first-order linear recurrences, a redundant matrix is composed of three arithmetic operations, which can be done in two parallel chains of arithmetic operations, one consisting of a multiply followed by an add, called the *long chain*, and the other consisting of one add, called the *short chain*. As for the result nodes, the columns of numbers associated with the redundant nodes represent the time steps at which a processor in our parallel machine is allocated to execute the operations in those nodes. The parallel schedule in Figure 3 computes the given recurrence in six steps on seven processors.

The idea of the exact schedule is to do all operations on multiple processors so that minimization of redundant operations and full utilization of all the processors throughout the evaluation are simultaneously achieved. We know that the number of final operations in any schedule equals the number of operations in the sequential schedule and cannot be reduced— by the definition of the required outputs. Thus one way to

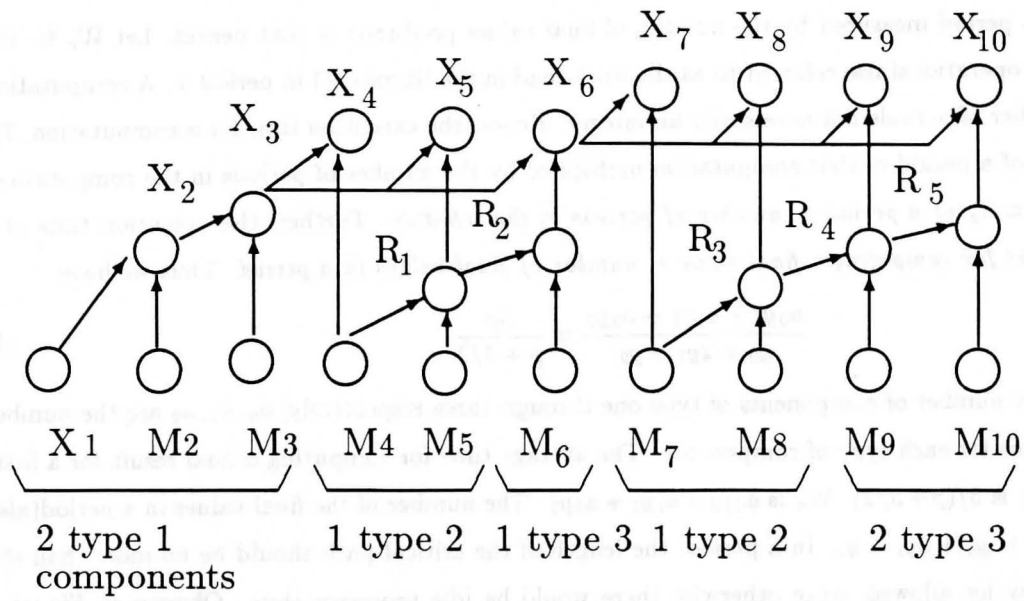


Figure 4: Three types of components in a parallel schedule for first-order banded linear recurrence.

speed up the computation is to use multiple processors to compute redundant values ahead of the final value computation, thus shortening the critical path in the computation of some final values, and then to obtain multiple final values in as a few parallel steps as possible by using previously computed (final, redundant) values in parallel. Thus we trade multiple processors and redundant operations for parallelism that can yield speedup. The speed of a schedule is therefore the average number of final values produced in a step. In order to compute as many final values as possible, given a fixed number of processors, a schedule should do as few redundant operations as possible. However, one cannot reduce the number of intermediate operations arbitrarily, say, to zero, since this will sequentialize the computation of final results, thus making a parallel schedule degenerate to a sequential schedule in the limit. Intuitively, the fastest parallel schedules for a fixed number of processors would use the fewest possible intermediate operations to achieve full utilization of all processors.

A parallel schedule can be divided into a number of *periods* that have the same organization. In each period, the computation should be organized in a way such that full utilization and use of the least possible number of redundant operations are simultaneously achieved. The optimality of the entire parallel schedule may be proved based on the optimality of each period. We refer to this method as *harmonic scheduling* in [19].

We shall construct our schedule with three types of components as shown in Figure 4. The first type of component is composed of an operand node with a dependent result node(equivalently, we can say a result node with a reaching operand node), for example, $X_2$ with $M_2$ and $X_3$ with $M_3$ make two components of the first type respectively. The second type of component is composed of two result nodes that depend on a redundant node and two operand nodes, for example, nodes $X_4, X_5, R_1, M_4$ and $M_5$ make a component of the second type. The third type of component is composed of a result node, a redundant node and an operand node chained by dependences, for example, node $X_6, R_2$ and $M_6$ make such a component. These three types of components are sufficient for constructing optimal parallel schedules for a first-order linear recurrence.

Let $\pi$ stand for a period measured by the number of final values produced in that period. Let $W_\pi$ be the number of arithmetic operations(also referred to as the work-load in the literature) in period $\pi$. A computation is composed of a number of periods not necessarily an integer. Hence, the execution time for a computation, $T_p$, is the execution time of a period of that computation multiplied by the number of periods in the computation, i.e., $T_p$ *of a schedule* $= T_p$ *of a period* $\times$ *number of periods in the schedule*. Further, the execution time of a period is $W_\pi/p =$ *time for computing a final value* $\times$ *number of final values in a period*. Thus we have

$$\frac{a_0y_0 + a_1y_1 + a_2y_2}{y_0 + 2y_1 + y_2} = \frac{5p}{p + 3/2} \tag{1}$$

where $y_0, y_1, y_2$ are the number of components of type one through three respectively, $a_0, a_1, a_2$ are the number of arithmetic operations for each type of components. The average time for computing a final result for a first-order linear recurrence is $5/(p+3/2)$. $W_\pi$ is $a_0y_0 + a_1y_1 + a_2y_2$. The number of the final values in a period(also called the *period size*) is $y_0 + 2y_1 + y_2$. In a period, the length of the critical path should be no more than the number of processor cycles allowed, since otherwise there would be idle processor slots. Observe in Figure 3 that the length of the critical path in a period is $2y_0 + 2y_1$ for a first-order linear recurrence. We then have

$$2y_0 + 2y_1 \leq \frac{a_0y_0 + a_1y_1 + a_2y_2}{p} \tag{2}$$

This is a necessary condition for an optimal parallel schedule. Other schedule constraints are $y_0 \geq 0, y_1 > 0$ and $y_2 \geq 0$, which says that the number of any type of schedule component should not be smaller than zero, and in particular that a parallel schedule should at least have some second type of components(since parallelization is impossible without type two components).

Solving equation (1) for all the integer solutions, we have

$$\begin{aligned} y_0 &= \frac{-(2p-7)t_1}{5} + t_2 \\ y_1 &= (\frac{(2p-7)}{5} + 5)t_1 - t_2 \\ y_2 &= t_2 \end{aligned}$$

where $t_1$ and $t_2$ are integers. Our task is to find all those solutions out of which feasible schedules can be constructed. This is a very difficult task. First, it requires solving a system of linear inequalities for all the integer solutions, which describe all the conditions for feasible schedules, which itself is a hard problem. Secondly, even if we find the system of linear inequalities encompassing all the conditions for feasible schedules, there is no known method for finding all the feasible integer solutions to a system of linear *inequalities*[11]. However, it is possible sometimes to construct a set of feasible solutions from the solutions for equation (1).

We partition the set of possible numbers of processors into six congruence classes, $(p + j) \equiv 0, \pmod 6$ for $0 \leq j \leq 5$ and $p > 1$, and provide a solution for feasible schedules for each congruence class. We can verify that the following are a set of solutions for $p > 1$.

$$y_0 = 1,\ y_1 = 5i - 1, \quad y_2 = 12i^2 - 7i + 1, \quad \textit{for}\ \ p = 6i,\ \ i \geq 1$$

$$y_0 = 1,\ y_1 = 5i - 1, \quad y_2 = 12i^2 - 9i + 1, \quad \textit{for}\ \ p = 6i - 1,\ \ i \geq 1$$

$$y_0 = 2,\ y_1 = 5i-2,\quad y_2 = 12i^2 - 11i + 2,\quad \textit{for}\ \ p = 6i-2,\ \ i \geq 1$$

$$y_0 = 2,\ y_1 = 5i-2,\quad y_2 = 12i^2 - 13i + 2,\quad \textit{for}\ \ p = 6i-3,\ \ i \geq 1$$

$$y_0 = 3,\ y_1 = 5i-3,\quad y_2 = 12i^2 - 15i + 3,\quad \textit{for}\ \ p = 6i-4,\ \ i \geq 1$$

$$y_0 = 3,\ y_1 = 5i-3,\quad y_2 = 12i^2 - 17i + 3,\quad \textit{for}\ \ p = 6i-5\ \ i > 1.$$

The following algorithm can be used to construct a feasible schedule using the solutions above.

**Input:** $p$ processors and components for constructing a schedule for parallel evaluation of a first-order banded linear recurrence. $y_0$ components of the first type, $y_1$ of the second type and $y_2$ of the third type.

**Output:** a schedule for parallel evaluation of the first-order linear recurrence.

**procedure** construct_schedule_for_lr($p$, components)
1. construct a computation tree for a period of the computation
   using the given components;
2. call procedure processor_slot_allocation(tree, slot_sets);
3. transform the computation tree with allocated processor slots into a schedule;

**end** (construct_schedule_for_lr)

In step one, the computation tree should be constructed such that the height of a redundant computation tree increments by one when possible, starting with the first redundant tree of height one. Without loss of generality, let us do an example for $p = 6i$. Consider first constructing the components into a computation tree for a period of the schedule. The number of schedule components of type 1, 2 and 3 are $y_0 = 1$, $y_1 = 5i-1$, $y_2 = 12i^2-7i+1$ respectively. The number of redundant trees should be the number of components of the second type, $5i-1$ in this case. The question is how many components of type 3 should be allocated to each redundant tree. The criterion for allocation is that the longest operation chain in a redundant tree should be no longer than the critical path to any final operation on top of this redundant tree. There are more than one way of allocation. We choose to allocate to the second redundant tree one component of type 3, to the third tree two components, ..., to the $(5i-1-(i(i-1)/2))$th tree $5i-2-i(i-1)/2$ components, and to the rest of trees $5i-2-i(i-1)/2$ components. The heights of redundant trees in a schedule period form a sequence:

$$1, 2, 3, ..., 5i-1-i(i-1)/2, \underbrace{5i-1-i(i-1)/2, ..., 5i-1-i(i-1)/2}_{i(i-1)/2}.$$

For example, given $p = 6(i = 1)$ processors, a schedule can be constructed from the solution $y_0 = 1, y_1 = 4, y_2 = 6$, as shown in Figure 5. Recall the organization of components of type one, two and three in Figure 4.

We can verify that this tree height allocation satisfies the constraint (2).

The second step can be done by the following procedure.

**Input:** a computation tree for a period, and $60i^2$ processor slots evenly divided into $10i$ sets that are marked 1 through $10i$.

**Output:** a computation tree for a period with allocated processor slots.

**procedure** processor_slot_allocation(tree, slot_sets)
  **for** time_step=1 to $10i$ do

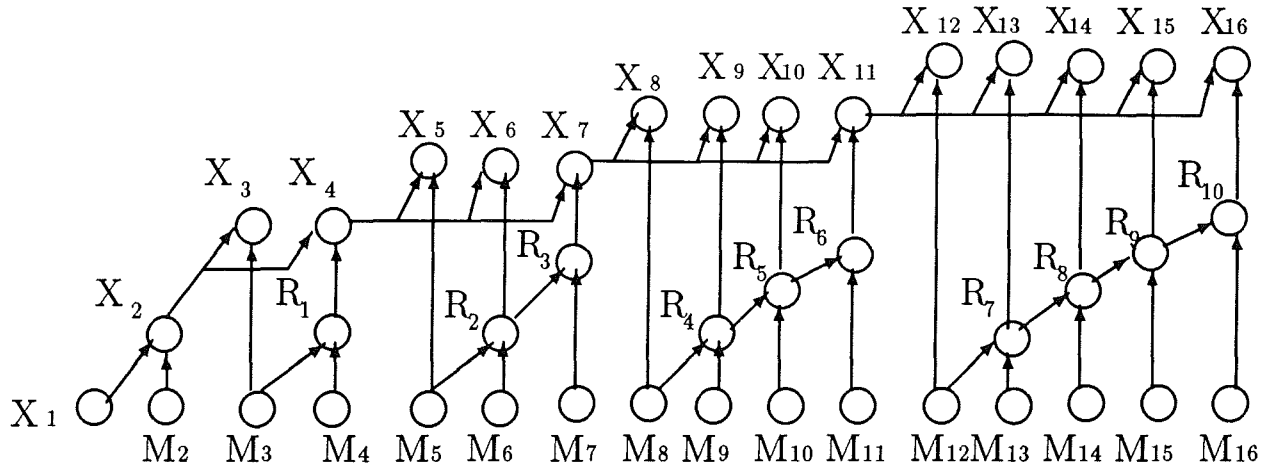


Figure 5: The computation tree constructed out of the solutions for $p = 6$.

```
    while (the slot set marked with time_step is not empty)
      {if (there is a vacancy on the critical path)
        fill the slot in;
      else if (there is a vacancy in any long chain for redundant tree 1 through 5i − 1)
        fill the slot in;
      else if (there is a vacancy in any short chain for redundant tree 1 through 5i − 1)
        fill the slot in;
      else fill the slot with some available final operation that is not on the critical path;
      end if }
      end while
    end for
  end (processor_slot_allocation)
```

Let us run the procedure with the computation tree for $p = 6$ produced in the first step of the algorithm. See Figure 6. The boxes represent the arithmetic operations in the computation tree for a period. The numbers inside the boxes represent the time steps at which processor slots compute the operations. Each column of boxes in the two top rows corresponds to a final node. Each column of boxex in the three bottom rows corresponds to a redundant node. The numbers associated with redundant operation boxes are the indexes for temporary array $t$ that stores the redundant values.

In each step of the procedure, we have a set of $p = 6$ processor slots. In step one, we allocate one processor slot to the first operation on the critical path, and five slots to the redundant trees, four of which are allocated to the four long chains and the last one to the first short chain. We thus can see all the operation boxes marked with "1". Step two is done similarly to step one: the first slot is allocated the operation on the critical path, and the other five allocated to the redundant tree, four of which are allocated to the second operations on all the long chains and the last slot to the second long chain. In step five, we allocate the first slot to the operation on the critical path, and the next four to the operations in the third and fourth redundant trees. Then there are no available operations in the redundant trees for the last slot in step five; thus we fill the slot with an available final operation that is not on the critical path. The allocation proceeds until in step ten all the remaining final operations are allocated—our 60 processor slots are precisely filled with 60 operations in the computation tree

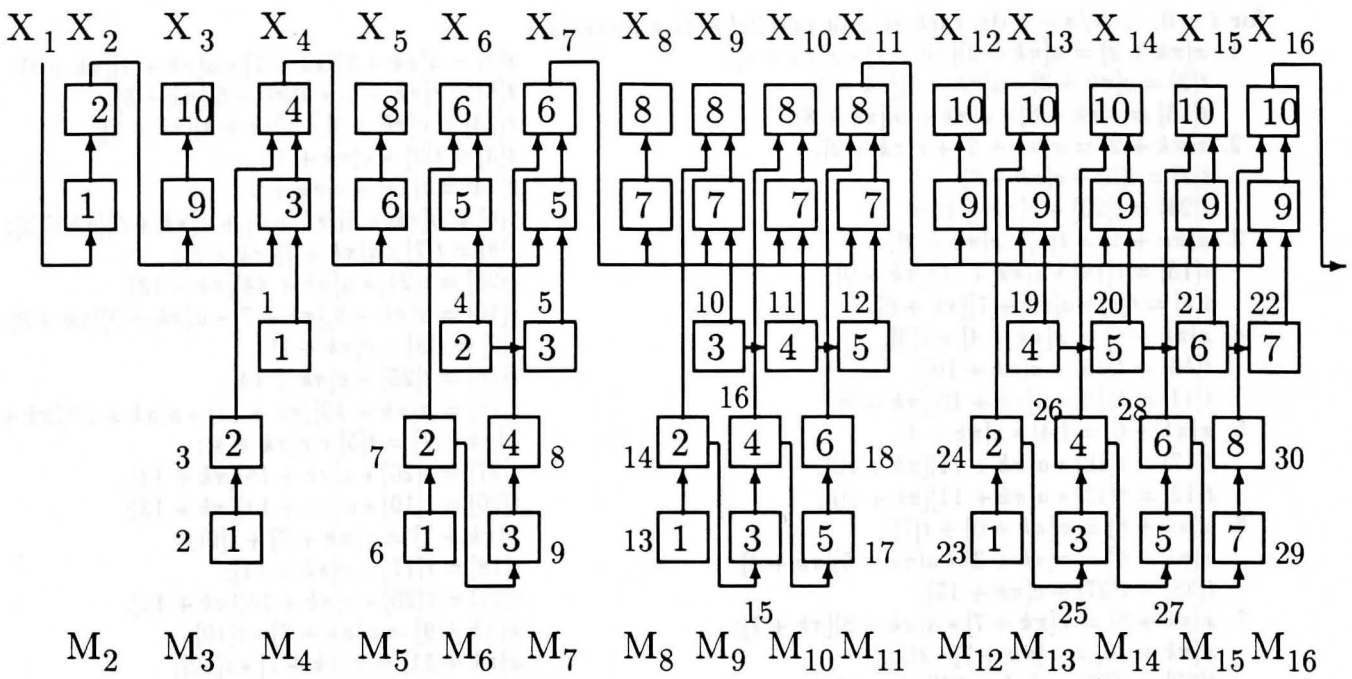


Figure 6: The computation tree with allocated processor slots for $p = 6$.

for a period.

It is straightforward to transform the computation tree with processor slot allocation into an exact schedule as follows. In the schedule, array $a[N][N]$ holds the coefficients, $c[N]$ holds the constant terms, $x[N]$ holds the results and $t[\eta]$ holds the results of the redundant operations of the first-order banded linear recurrence, where $\eta$ is the number of redundant operations in a period, $\eta = 30$ for $p = 6$. Obviously, a careful use of memory can reduce memory size to an order of $N$. We chose to neglect the memory efficiency in the following schedule in favor of readability.

**for** $i = 0, ..., N/\pi - 1$ **do** *each step in parallel with p processors*

1. $x[\pi k+2] = a[\pi k+2][\pi k+1] * x[\pi k+1];$
   $t[2] = c[\pi k+3] * a[\pi k+4][\pi k+3];$
   $t[13] = c[\pi k+8] * a[\pi k+9][\pi k+8];$
2. $x[\pi k+2] = x[\pi k+2] + c[\pi k+2];$
   $t[7] = t[6] + c[\pi k+6];$
   $t[24] = t[23] + c[\pi k+13];$
3. $x[\pi k+4] = t[1] * x[\pi k+2];$
   $t[15] = t[14] * a[\pi k+10][\pi k+9];$
   $t[5] = t[4] * a[\pi k+7][\pi k+6];$
4. $x[\pi k+4] = x[\pi k+4] + t[3];$
   $t[16] = t[15] + c[\pi k+10];$
   $t[11] = t[10] * a[\pi k+10][\pi k+9];$
5. $x[\pi k+6] = t[4] * x[\pi k+4];$
   $t[17] = t[16] * a[\pi k+11][\pi k+10];$
   $t[12] = t[11] * a[\pi k+11][\pi k+10];$
6. $x[\pi k+6] = x[\pi k+6] + t[7];$
   $x[\pi k+4] = x[\pi k+3] * a[\pi k+5][\pi k+4];$
   $t[28] = t[27] + c[\pi k+15];$
7. $x[\pi k+8] = x[\pi k+7] * a[\pi k+8][\pi k+7];$
   $x[\pi k+10] = x[\pi k+7] * t[11];$
   $t[29] = t[28] * a[\pi k+16][\pi k+15];$
8. $x[\pi k+8] = x[\pi k+8] + c[\pi k+8];$
   $x[\pi k+10] = x[\pi k+10] + t[16];$
   $x[\pi k+5] = x[\pi k+5] + c[\pi k+5];$
9. $x[\pi k+12] = x[\pi k+10] * a[\pi k+12][\pi k+11];$
   $x[\pi k+14] = x[\pi k+11] * t[20];$
   $x[\pi k+16] = x[\pi k+11] * t[22];$
10. $x[\pi k+12] = x[\pi k+12] + c[\pi k+12];$
    $x[\pi k+14] = t[26] + x[\pi k+14];$
    $x[\pi k+16] = t[30] * x[\pi k+16];$

**end for**

$t[1] = a[\pi k+3][\pi k+2] * a[\pi k+4][\pi k+3];$
$t[6] = c[\pi k+5] * a[\pi k+6][\pi k+5];$
$t[23] = c[\pi k+3] * a[\pi k+4][\pi k+3];$
$t[3] = t[2] + c[\pi k+4];$
$t[14] = t[13] + c[\pi k+9];$
$t[4] = a[\pi k+5][\pi k+4] + a[\pi k+6][\pi k+5];$
$t[8] = t[7] * a[\pi k+7][\pi k+6];$
$t[25] = t[24] * a[\pi k+14][\pi k+13];$
$t[10] = a[\pi k+8][\pi k+7] * a[\pi k+9][\pi k+8];$
$t[9] = t[8] + c[\pi k+7];$
$t[26] = t[25] + c[\pi k+14];$
$t[19] = a[\pi k+12][\pi k+11] * a[\pi k+13][\pi k+12];$
$x[\pi k+7] = t[5] * x[\pi k+4];$
$t[27] = t[26] * a[\pi k+15][\pi k+14];$
$t[20] = t[19] * a[\pi k+14][\pi k+13];$
$x[\pi k+7] = x[\pi k+7] + t[9];$
$t[18] = t[17] + c[\pi k+11];$
$t[21] = t[20] * a[\pi k+15][\pi k+14];$
$x[\pi k+9] = x[\pi k+7] * t[10];$
$x[\pi k+11] = x[\pi k+7] * t[12];$
$t[22] = t[21] * a[\pi k+16][\pi k+15];$
$x[\pi k+9] = x[\pi k+9] + t[14];$
$x[\pi k+11] = x[\pi k+11] + t[18];$
$t[30] = t[29] + c[\pi k+16];$
$x[\pi k+13] = x[\pi k+11] * t[19];$
$x[\pi k+15] = x[\pi k+11] * t[21];$
$x[\pi k+2] = x[\pi k+1] * a[\pi k+3][\pi k+2];$
$x[\pi k+13] = t[24] + x[\pi k+13];$
$x[\pi k+15] = t[28] + x[\pi k+15];$
$x[\pi k+3] = x[\pi k+3] + c[\pi k+3];$

The most important properties of these schedules we are concerned with are the correctness, the execution time, and the optimality under the matrix chain multiplication model. We now prove them in turn.

**Theorem 2.1** There exists a processor slot allocation for an exact schedule, called a *feasible allocation*, which satisfies the full utilization of all processors and preserves the semantics of the sequential evaluation of a first-order linear recurrence. The procedure "processor_slot_allocation()" gives a feasible allocation.

**Proof:** To prove the correctness of the exact schedules, it suffices to show that the procedure "processor_slot_allocation()" allocates the processor slots correctly since the first step and the third step in procedure "construct_schedule_for_lr()" are straightforward.

We prove the claim for the first congruence class(i.e., $p = 6i$). The proofs for $p$ in other congruence classes can be done similarly. We prove the claim by showing an allocation scheme as shown in the following table. Note that there are many allocation schemes and we choose to show this particular one because it is easier to use in the proof. The column with header "Step" gives the step numbers from 1 through $10i$. The columns with header "number of slots allocated to final ops" and "number of slots allocated to redundant ops" give

respectively, for each step, the number of processor slots allocated to the final operations and the redundant operations in the computation tree. The sum of these two columns for each step in the table should be $6i$. The columns with header "number of slots allocated to redundant ops in the long chain" and "number of slots allocated to redundant ops in the short chain" give, for each step, the number of processor slots allocated to redundant operations in the long chain and short chain of the redundant trees respectively. The sum of these two columns for each step should be the same number as in the column titled "number of slots allocated to redundant ops".

| Step | number of slots allocated to final ops | number of slots allocated to redundant ops | number of slots allocated to redundant ops in the long chain | number of slots allocated to redundant ops in the short chain |
|---|---|---|---|---|
| 1 | 1 | 6i-1 | 5i-1 | i |
| 2 | 1 | 6i-1 | 5i-1 | i |
| 3 | 1 | 6i-1 | 5i-2 | i+1 |
| 4 | 1 | 6i-1 | 5i-2 | i+1 |
| 5 | 1 | 6i-1 | 5i-3 | i+2 |
| 6 | 1 | 6i-1 | 5i-3 | i+2 |
| ... | ... | ... | ... | ... |
| 4i | 1 | 6i-1 | 3i | 3i-1 |
| 4i+1 | i+1 | 5i-1 | 3i-1 | 2i |
| 4i+2 | i+2 | 5i-2 | 3i-2 | 2i-1 |
| 4i+3 | i+3 | 5i-3 | 3i-2 | 2i-1 |
| ... | ... | ... | ... | ... |
| 8i-1 | 5i-1 | i+1 | i | 1 |
| 8i | 5i | i | i | 0 |
| 8i+1 | 6i | 0 | 0 | 0 |
| ... | ... | ... | ... | ... |
| 10i | 6i | 0 | 0 | 0 |

The processor slots allocation can be seen as proceeding in three phases, the first phase going from step 1 through $4i$, the second phase going from step $4i+1$ to $8i$ and the third phase going from step $8i$ through $10i$. In each step of the first phase, only one processor slot is allocated to a final operation on the critical path and the rest of slots are allocated to the redundant operations. As presented in the procedure, we prefer the long chain over the short chain for each redundant computation tree, and the lower numbered redundant tree over the higher numbered redundant tree, because otherwise we cannot keep progressing along the critical path in each step. In each step of the second phase, we allocate, starting with $i+1$, an increasing number of processor slots to the final operations and a dual decreasing number of slots to the redundant operations. By the end of the second phase, all the redundant operations are allocated to processor slots. In each step of the third phase, we allocate all the slots to the remaining final operations. □

**Theorem 2.2** The execution time of the exact schedule for a first-order linear recurrence on $p$ processors is

$$T_p = \frac{10N}{2p+3}$$

for $p > 1$, $N > 1/6(p+5)(2p+3)$ and for $N$ being a multiple of period. When $N$ is not a multiple of the period, the performance formula holds subject to a ceiling effect. We shall not discuss it due to the length of the proof.

**Proof:** We only need to show equation (1) holds when we substitute into the equation the solution set $y_0, y_1$ and $y_2$ for each of the six congruence classes. Here we prove the claim with the solution set for the sixth congruence class (i.e., $p = 6i - 5$ and $p > 1$). The truth of the claim with the solution sets for other congruence classes can be shown similarly. Note $a_0 = 2, a_1 = 7, a_2 = 5$, and $i = (p + 5)/6$. Substituting $y_0, y_1, y_2$ into equation (1), we have

$$\begin{aligned}
\text{the left of (1)} &= \frac{2(3) + 7(5i - 3) + 5(12i^2 - 17i + 3)}{3 + 2(5i - 3) + 12i^2 - 17i + 3} = \\
&= \frac{10(6i - 5)}{12i - 7} \qquad (\text{note } p = 6i - 5) \\
&= \frac{10p}{2p + 3} \\
&= \text{the right of (1).}
\end{aligned}$$

□

**Theorem 2.3** There does not exist a parallel schedule under the matrix multiplication model such that it gives a better execution time for $m = 1, p > 1$, $N > 1/6(p + 5)(2p + 3)$ and for $N$ a multiple of the period.

**Proof:** It suffices to show that a contradiction would result from assuming that there existed a parallel schedule with better execution time. A schedule with better execution time, say $S_0$, would turn equation (1) into a strict inequality:

$$\frac{a_0 y_0 + a_1 y_1 + a_2 y_2}{y_0 + 2y_1 + y_2} < \frac{5p}{p + 3/2}.$$

By substituting $a_0 = 2, a_1 = 7$, and $a_2 = 5$ into the inequality above, we have

$$(6 - 6p)y_0 + (21 - 6p)y_1 + 15y_2 < 0.$$

Assume that this schedule $S_0$ satisfies constraint (2). Thus the constraint can be written into an equivalent inequality:

$$(2p - 2)y_0 + (2p - 7)y_1 - 5y_2 \leq 0.$$

Adding this inequality multiplied by three to the preceding inequality would result in $0 < 0$, a contradiction.

If schedule $S_0$ did not satisfy constraint (2), i.e.,

$$2y_0 + 2y_1 > \frac{a_0 y_0 + a_1 y_1 + a_2 y_2}{p},$$

This means that the critical path of the period would be longer than the number of steps needed to fully utilize the processors. Hindered by the critical path, the computation tree for the period could not be completed in $(a_0 y_0 + a_1 y_1 + a_2 y_2)/p$ steps, resulting in a slower schedule than the exact schedule. Therefore, the exact schedule gives the optimal execution time under the matrix chain multiplication model. □

The following corollary gives the ratio of final operations to redundant operations in a period of an exact schedule for $p > 1$. This ratio will be useful in deriving simpler, more concise and more program-space efficient schedules than the exact schedules.

**Corollary 2.1** The ratio of the number of final operations to redundant operations in a period of an exact schedule is $(2p+3)/(3p-3)$, for a first-order banded linear recurrence of size $N > (p+5)(2p+3)/6$ and $p > 1$.

**Proof:** The proof is done by obtaining the ratio of the number of final operations $(2y_0 + 4y_1 + 2y_2)$ to the number of redundant operations $(3y_1 + 3y_2)$ and substituting into this ratio the solution $y_0, y_1, y_2$ for equation (1) for each congruence class. □

Next, we state the results by our exact schedules for computing second-order linear recurrences. The algorithms and the proofs are in [22].

**Theorem 2.4** There exist exact schedules for a second-order linear recurrence on $p$ processors that achieves the strict time lower bound under the matrix multiplication model,

$$T_p = \frac{(2m^2 + 3m)N}{p + 2m + 1}$$

for $p > 2$, $N > 1/6(p+5)(2p+3)$.

In addition to the properties above, the exact schedules have other desirable properties. The generation of exact schedules depends only on the number of processors $p$, for $N > (p+5)(2p+3)/6$ and $p > 1$. When the number of processor $p$ is known, an exact schedule can be generated at compile time in $5p^2/3$ time steps for $p = 6i$(the time for generating exact schedules for $p$ in other congruence classes is about the same). In comparison with previously proposed algorithms for first-order linear recurrences, the exact schedules not only have the best execution time(strict time lower bound) but also the smallest schedule size.

# 3 The Pipelined Schedules

The exact schedules specify the parallel evaluation with an execution time of $10N/(2p+3)$ for a first-order banded linear recurrence of size $N \geq (p+5)(2p+3)/6$ and $p > 1$. Note that an exact schedule has $10i$ steps in the loop body for $p = 6i$ processors and $i > 1$. Although this does not make the exact schedules less practical than any of the previously published algorithms (in fact it is already more space-efficient), we wondered whether more program-space efficient schedules than the exact schedules exist. By Corollary 2.1, the ratio of final to redundant operations in an iteration(i.e., a period) of an exact schedule for first-order linear recurrence is $(2p+3)/(3p-3)$ for $N \geq (p+5)(2p+3)/6$ and $p > 1$. An exact schedule exhibits this ratio in every $10i$ steps(i.e., an iteration). If a schedule can achieve this ratio in much fewer steps than $10i$, then it will have a much smaller loop body than an exact schedule. Because a program with a small loop body is more likely to fit in processor caches, this gain may well offset any small overhead that may be introduced in a trade for better

space-efficiency. Applying the idea of software pipelining[2, 12], we can derive a simpler, more concise and more program-space efficient schedule than the exact schedule, called a *pipelined schedule.*

The idea of the pipelined schedules is that, in the loop body of a pipelined schedule, we compute the final operations of a period and the redundant operations of the next period. The *size* of the loop body of such a schedule is the number of operations in the period. With such an arrangement, there is no need to match the final computation and the redundant computation strictly inside a single period in order to satisfy the dual criteria for an optimal schedule—full utilization of resources and the minimum number of redundant operations. By allowing the matching of final and redundant computations across multiple periods, we can make a period as small as implied by full utilization of resources and as constrained by the semantic correctness of the schedule. We need not worry about the performance since it is already implied by the ratio. From Corollary 2.1, we establish a system of linear inequalities as follows.

$$\begin{aligned} \frac{2y_0 + 4y_1 + 2y_2}{3y_1 + 3y_2} &= \frac{2p+3}{3p+2} \\ y_0 &\geq 0 \\ y_1 &> 1 \\ y_2 &\geq 0 \end{aligned} \tag{3}$$

The solutions for the first equation of the linear inequalities above cover all the schedules, including the exact schedules and non-exact schedules, for which the ratio holds. All the integer solutions for the first equation of linear inequality system 3 are given by:

$$\begin{aligned} y_0 &= t_0 - t_1 \\ y_1 &= t_1 \\ y_2 &= \frac{6p-6}{15} t_0 - t_1, \end{aligned} \tag{4}$$

where $t_0$ and $t_1$ are integers.

Next, we shall show how a pipelined schedule is constructed for the example in Section 2 and state interesting properties of pipelined schedules. The details of the algorithms and proofs can be found in [22]. We can verify that, given $p = 6$ processors, $y_0 = 0, y_1 = 2$ and $y_2 = 2$ is a solution for linear inequalities (3). Recall that $y_0, y_1$, and $y_2$ are the numbers of components of three types respectively. A pipelined schedule is constructed with parameters $p, y_0, y_1$ and $y_2$, in four stages: (1) generating pipeline startup code, (2) building the computation tree for two consecutive periods of the pipelined schedule, (3) allocating the processor slots to the operations in the computation tree, and (4) transforming the computation tree with processor slots allocated into the pipelined schedule.

The idea of the startup of a pipelined schedule is to use some processor slots to compute the redundant operations in the first pipeline period, and to use the remaining processor slots to compute as many leading final values as possible to reduce startup overhead. In the worst case, the number of overhead steps, i.e., the time steps in which no final results are produced, is no more than that of a pipeline period. A careful design

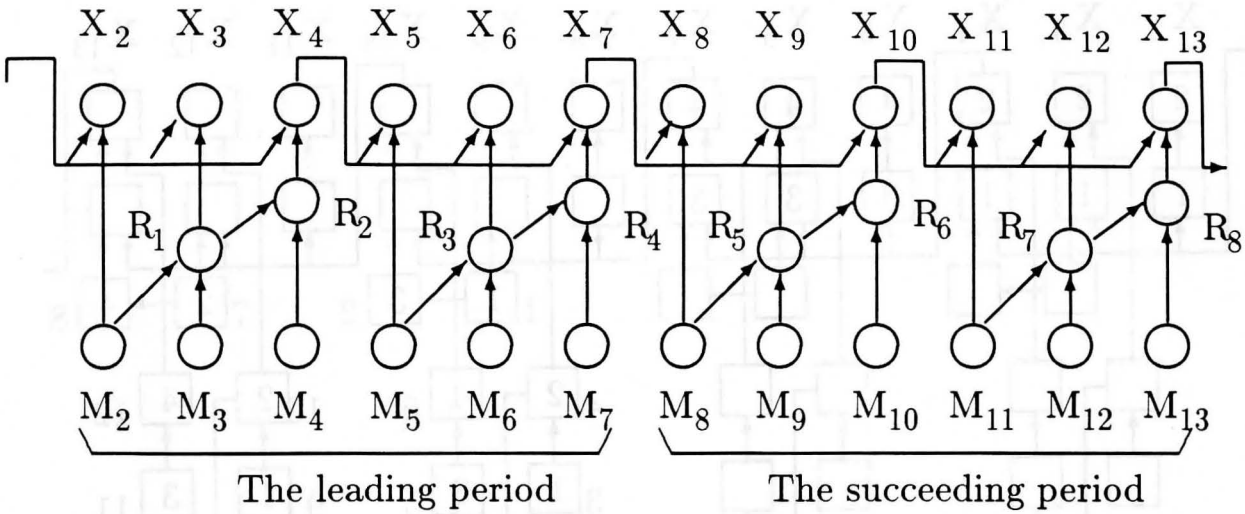


Figure 7: The computation tree for a pipeline period with $p = 6$.

can reduce the startup overhead to very few idle processor slots. In the second stage, we build the computation tree by distributing the components of the third type to all the redundant trees(the number of redundant trees is $y_1$) as shown in Figure 7.

In the third stage, we allocate processor slots to the operations in the computation tree for two consecutive periods, specifically, to the final operations of the *leading period* and the redundant operations of the *succeeding period.*

Figure 8 shows the computation graph of two periods with processor slots allocated for $p = 6$. We have $p = 6$ slots available. For time_step= 1, we allocate the first three slots to the first three final operations in the leading period, and no further allocation can be made onto the leading period. We then allocate the other three slots to redundant operations in the succeeding period, i.e., to the three ready operations in an operation chain with the largest number of vacant operations. When two chains have the same number of vacant operations, we can allocate a processor slot to any of them. For each processing step, we first allocate as many processor slots as possible to the final operations in the leading period, and then allocate the rest of the slots to the redundant operations in the succeeding period following the rule "redundant operation chain with largest number of vacant operations first". Thus in $2 \times y_1 = 4$ steps, the $2y_1p = 24$ processor slots are exactly allocated to 12 final operations in the leading period, and to 12 redundant operations in the successive period.

In the last stage, we compose the two portions of the two periods into the loop body of the pipelined schedule, combined with the startup code, and transform it into a program form as follows. In the schedule, array $a[N][N]$ holds the coefficients, $c[N]$ holds the constant terms, $x[N]$ holds the results and $t[N/\pi][\eta]$ holds the results of the redundant operations of the first-order banded linear recurrence, where $\eta$ is the number of redundant operations in a period, $\eta = 12$ for $p = 6$. Obviously, a careful use of memory can reduce memory size to an order of $N$.

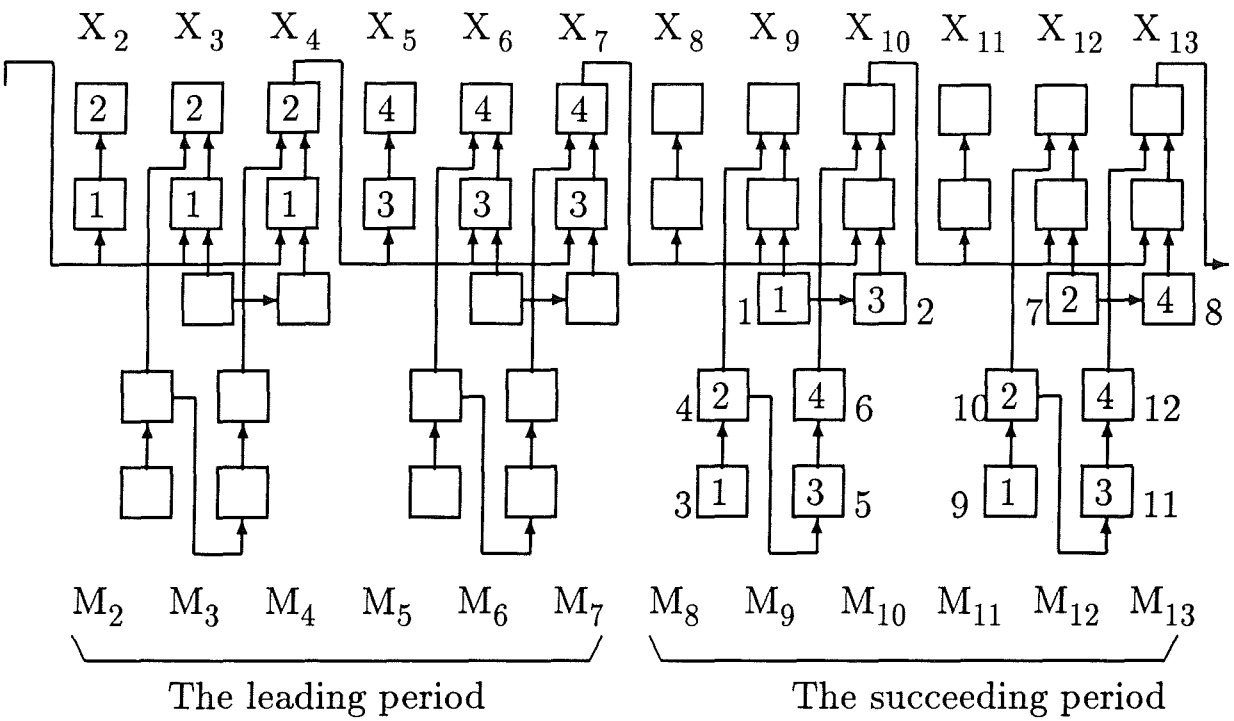


Figure 8: The computation tree with allocated processor slots for $p = 6$ for pipeline schedule.

*pipeline startup code;*
**for** $k = 0$ to $N/\pi - 1$ **do** *each step in parallel with p processors*

1. $x[\pi k+2] = a[\pi k+2][\pi k+1] * x[\pi k+1]$; $x[\pi k+3] = t[k][1] * x[\pi k+1]$;
   $x[\pi k+4] = t[k][2] * x[\pi k+1]$; $t[k+1][3] = c[\pi(k+1)+2] * a[\pi(k+1)+3][\pi(k+1)+2]$;
   $t[k+1][9] = c[\pi(k+1)+5] * a[\pi(k+1)+6][\pi(k+1)+5]$;
   $t[k+1][1] = a[\pi(k+1)+17][\pi(k+1)+16] * a[\pi(k+1)+18][\pi(k+1)+17]$;
2. $x[\pi k+2] = c[\pi k+2] + x[\pi k+2]$; $x[\pi k+3] = t[k][4] + x[\pi k+3]$;
   $x[\pi k+4] = t[k][6] + x[\pi k+4]$;
   $t[k+1][10] = c[\pi(k+1)+6] + t[k+1][9]$; $t[k+1][4] = t[k+1][3] + c[\pi(k+1)+3]$;
   $t[k+1][7] = a[\pi(k+1)+20][\pi(k+1)+19] * a[\pi(k+1)+21][\pi(k+1)+20]$;
   $t[k+1][10] = c[\pi(k+1)+6] + t[k+1][9]$;
3. $x[\pi k+5] = c[\pi k+5] * x[\pi k+4]$; $x[\pi k+6] = t[k][7] * x[\pi k+4]$;
   $x[\pi k+7] = t[k][8] * x[\pi k4]$; $t[k+1][2] = t[k+1][1] * a[\pi(k+1)+4][\pi(k+1)+3]$;
   $t[k+1][5] = t[k+1][4] * a[\pi(k+1)+4][\pi(k+1)+3]$;
   $t[k+1][11] = t[k+1][10] * a[\pi(k+1)+7][\pi(k+1)+6]$;
4. $x[\pi k+5] = c[\pi k+5] + x[\pi k+5]$; $x[\pi k+6] = t[k][10] + x[\pi k+6]$;
   $x[\pi k+7] = t[k][12] + x[\pi k+7]$; $t[k+1][6] = c[\pi(k+1)+4] + t[k+1][5]$;
   $t[k+1][8] = t[k+1][7] * a[\pi(k+1)+7][\pi(k+1)+6]$;
   $t[k+1][12] = t[k+1][11] + c[\pi(k+1)+7]$;

**end for**

The pipelined schedule on $p = 6$ processors has four steps and $4p = 24$ operations in its loop body as opposed to 10 steps and $10p = 60$ operations in the loop body of its equivalent exact schedule, a saving of 60% in the program space. This program-space efficiency is obtained at a small time cost of the pipelined startup

| number of processors | loop body size in number ops exact schedule | loop body size in number ops pipelined schedule | saving in % |
|---|---|---|---|
| 2 | 20 | 20 | 0% |
| 4 | 20 | 20 | 0% |
| 8 | 160 | 80 | 50% |
| 16 | 480 | 320 | 33% |
| 32 | 1920 | 960 | 50% |
| 64 | 7040 | 3200 | 55% |
| 128 | 28181 | 12800 | 55% |

Table 1: A comparison of program sizes of exact and pipelined schedules.

code of no more than the time steps in the loop body of the pipelined schedule, e.g., no more than four steps of the pipelined startup for $p = 6$ processors.

As with the exact schedules, we are interested in the execution time of the pipelined schedules and the reduction in the size of the loop body of the pipelined schedules. In general, full utilization of processors can be realized in the loop body of a pipelined schedule. Thus the execution time for a pipelined schedule with $p$ processors is: *pipeline startup time* $+ (2m^2 + 3m)N/(p + (2m + 1)/2)$ for $m = 1$.

For simplicity of comparison, we show the savings in the size of the loop body by the pipelined schedules from the exact schedules for some frequently-used numbers of processors in Table 1.

In addition to the better program-space efficiency, the pipelined schedules also share other desirable properties with the exact schedules. In contrast with the previous techniques, t he generation of pipelined schedules depends only on the number of processors $p$, regardless of the problem size $N$. It offers flexibility in processor slot allocation—one can allocate processor slots differently within the computation tree of a pipelined schedule without affecting execution time. In all the previously published methods, the allocation schemes are fixed.

The pipelined schedule not only proves to be a natural candidate for compiling linear recurrences because of its program-space efficiency, but can also be used, as a much simpler vehicle than the exact schedules, to approach the time bounds for higher order linear recurrences. Using the pipelined schedules, an $m$-th order banded linear recurrence can be evaluated in time

*pipeline startup time* $+ (2m^2 + 3m)N/(p + (2m + 1)/2)$ for $m = 1$, and

*pipeline startup time* $+ (6m + 2)N/(p + (2m + 1)$ for $m > 1, m < p \leq 4m + 1$, and

*pipeline startup time* $+ (2m^2 + 3m)N/(p + (m - 1)(2m + 1))$ for $m > 1, p > 4m + 1$ and $N >$ (a period of the pipelined schedule) .

This is the first parallel algorithm that achieves this time bound. Our results improve on the fastest prevoiusly published algorithm by a factor $\geq (p + 2m^2 - m - 1)/(p + m + 1/2)$ for $m > 1$. Due to the length of this paper, we omit the details of the proofs. Interested readers can refer to [23] for details of the schedules and the proofs. In Table 2, we illustrate the speedup obtained by our pipelined schedules in comparison with the best two previously published algorithms[5, 9] for a range of number of processors and order of linear recurrence.

In Table 2, the speedups by Chen, Kuck and Sameh's schedule are worse than one at some coordinates of $p$ and $m$. That was because that the speedups were calculated directly from their speedup formula $2p/(2m + 3)$

| $p$ | | order of linear recurrence | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | $1st$ | $2nd$ | $3rd$ | $4th$ | $5th$ | $6th$ | $7th$ | $8th$ |
| | | $S_p$ | $S_p$ | $S_p$ | $S_p$ | $S_p$ | $S_p$ | $S_p$ | $S_p$ |
| 2 | Chen's | 0.8 | | | | | | | |
| | Gajski's | 1.4 | | | | | | | |
| | ours | *1.4* | | | | | | | |
| 3 | Chen's | 1.2 | 0.857 | | | | | | |
| | Gajski's | 1.8 | 1.571 | | | | | | |
| | ours | *1.8* | *2.286* | | | | | | |
| 4 | Chen's | 1.6 | 1.143 | 0.889 | | | | | |
| | Gajski's | 2.2 | 1.857 | 1.667 | | | | | |
| | ours | *2.2* | *2.571* | *3.3* | | | | | |
| 5 | Chen's | 2.0 | 1.429 | 1.111 | 0.909 | | | | |
| | Gajski's | 2.6 | 2.143 | 1.889 | 1.727 | | | | |
| | ours | *2.6* | *2.857* | *3.6* | *4.308* | | | | |
| 6 | Chen's | 2.4 | 1.714 | 1.333 | 1.091 | 0.923 | | | |
| | Gajski's | 3.0 | 2.429 | 2.111 | 1.909 | 1.769 | | | |
| | ours | *3.0* | *3.143* | *3.9* | *4.615* | *5.313* | | | |
| 7 | Chen's | 2.8 | 2.0 | 1.556 | 1.273 | 1.077 | 0.933 | | |
| | Gajski's | 3.4 | 2.714 | 2.333 | 2.091 | 1.923 | 1.8 | | |
| | ours | *3.4* | *3.429* | *4.2* | *4.923* | *5.625* | *6.316* | | |
| 8 | Chen's | 3.2 | 2.286 | 1.778 | 1.455 | 1.231 | 1.067 | 0.941 | |
| | Gajski's | 3.8 | 3.0 | 2.556 | 2.273 | 2.077 | 1.933 | 1.824 | |
| | ours | *3.8* | *3.714* | *4.5* | *5.231* | *5.938* | *6.632* | *7.318* | |
| 16 | Chen's | 6.4 | 4.571 | 3.556 | 2.909 | 2.462 | 2.133 | 1.882 | 1.684 |
| | Gajski's | 7.0 | 5.286 | 4.333 | 3.727 | 3.308 | 3.0 | 2.765 | 2.579 |
| | ours | *7.0* | *6.0* | *6.667* | *7.692* | *8.438* | *9.158* | *9.864* | *10.56* |
| 32 | Chen's | 12.8 | 9.143 | 7.111 | 5.818 | 4.923 | 4.267 | 3.765 | 3.368 |
| | Gajski's | 13.4 | 9.857 | 7.889 | 6.636 | 5.769 | 5.133 | 4.647 | 4.263 |
| | ours | *13.4* | *10.5714* | *10.222* | *10.727* | *11.692* | *12.933* | *14.353* | *15.68* |
| 64 | Chen's | 25.6 | 18.286 | 14.222 | 11.636 | 9.846 | 8.533 | 7.529 | 6.737 |
| | Gajski's | 26.2 | 19.0 | 15.0 | 12.455 | 10.692 | 9.4 | 8.412 | 7.632 |
| | ours | *26.2* | *19.714* | *17.333* | *16.545* | *16.615* | *17.2* | *18.118* | *19.263* |

Table 2: The comparison in speedup of three parallel schedules.

in [3]. And their schedules were designed for $N, p, m$ all being some powers of two. For any one of $N, p, m$ that is not a power of two and $p = m + 1$, a direct application of their schedules would give an execution time worse than sequential schedule.

# 4 Experiments with Pipelined Schedules

Our architectural model is similar to the IBM VLIW machine[7]. It has a single flow of control(single PC), thus is totally synchronous, has multiple functional units and a multi-ported register-file. The simulator for this architecture is parameterized for the number of functional units and for functions performed by each functional units. Because this architecture facilitates mapping of our parallel schedules in source code to parallel instructions, we chose to run some illustrative benchmarks on this model. Obviously, this choice of

| bench-mark | problem size | number of functional units | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 8 | | 16 | | 32 | | 64 | |
| | | $R_p$ | $S_p$ | $R_p$ | $S_p$ | $R_p$ | $S_p$ | $R_p$ | $S_p$ |
| LL5 | 1000 | 1.58 | 3.531 | 1.767 | 6.194 | 1.76 | 10.804 | 2.333 | 12.942 |
| | 2000 | 1.58 | 3.573 | 1.708 | 6.56 | 1.76 | 11.956 | 2.333 | 16.184 |
| | 5000 | 1.574 | 3.604 | 1.697 | 6.758 | 1.76 | 12.757 | 1.871 | 22.075 |
| | 10000 | 1.571 | 3.616 | 1.685 | 6.842 | 1.76 | 13.048 | 1.871 | 23.673 |
| | $\infty$ | 1.569 | 3.627 | 1.683 | 6.898 | 1.751 | 13.403 | 1.785 | 26.427 |
| LL11 | 1000 | 1.369 | 5.757 | 1.468 | 10.602 | 1.493 | 19.381 | 1.493 | 31.957 |
| | 2000 | 1.381 | 5.766 | 1.467 | 10.747 | 1.517 | 20.036 | 1.543 | 36.024 |
| | 5000 | 1.381 | 5.781 | 1.464 | 10.873 | 1.503 | 20.849 | 1.533 | 39.346 |
| | 10000 | 1.383 | 5.781 | 1.464 | 10.903 | 1.508 | 20.997 | 1.533 | 40.51 |
| | $\infty$ | 1.383 | 5.786 | 1.464 | 10.929 | 1.508 | 21.214 | 1.531 | 41.786 |

Table 3: Speedup and redundancy for benchmarks.

experimental environment does not imply that our schedules can only fit onto VLIW architectures.

In order to illustrate the speedup obtained by our parallel schedules, we did experiments on two Livermore Kernels: Livermore Kernel 5 and 11. Actually, six Livermore Kernels out of 24 Livermore Kernels contains loop-carried dependences, and five of those six consist primarily of first-order banded linear recurrences. We have only been able to conduct our experiments on two of them due to time constraints on this paper(**we will have more results in the final version of the paper**).

Table 3 demonstrates the speedup of actual runs using our pipelined schedule on the benchmarks and the redundancy, for some frequently used numbers of processors and a range of problem sizes (i.e., the upper-bound of loop in a sequential benchmark). As defined in Section 1, the speedup $S_p$ is *dynamic*, i.e., the ratio of the number of cycles running a sequential program to the number of cycles running the parallel version using our pipelined schedule for a given problem size. For comparison, the speedup for an arbitrarily large problem size $N$ is calculated by

$$\lim_{N\to\infty} \frac{N \times \mathrm{CSL} + \mathrm{CSNL}}{\frac{N}{period}\mathrm{CPL} + \mathrm{CPNL}},$$

where CSL is the number of cycles in the loop body of the sequential program, CSNL is the number of cycles in the prologue and epilogue of the sequential program, CPL is the number of cycles in the loop body of the parallel program and CPNL is the number of cycles in the prologue and epilogue of the parallel program. The redundancy is the dynamic ratio of the number of operations executed in the parallel version to the number of operations executed in a sequential program. The redundancy for an arbitrarily large problem size is given by

$$\lim_{N\to\infty} \frac{\mathrm{CPL}}{\mathrm{CSL}}.$$

The functional units in these runs are assumed to be homogeneous, i.e., they all can execute the same set of operations.

As indicated in Table 3, the speedup is linear as predicted (at the source level) by the formulas. The speedups are obtained at the cost of redundant operations.

# 5 Summary

Linear recurrences have a role of central importance in computer design, numerical analysis, program analysis, image processing and vision. However, programs containing banded linear recurrences are difficult to parallelize due to loop-carried dependences. In this paper, we first presented a family of schedules, called the *exact schedules*, for parallel evaluation of low order ($m \leq 2$) banded linear recurrences with an execution time $(2m^2 + 3m)N/(p+(m(m+1)(2m+1))/2(2+\lfloor \log m \rfloor))$ for $0 < m \leq 2$ , $N > (p+5)(2p+3)/6$ and number of processors $p > m$. This is the first parallel algorithm that achieves this time bound. We showed that the exact schedules achieve the strict time lower bound under the matrix multiplication decomposition. Next, we derive another family of schedules, called the *pipelined schedules*, with better program-space efficiency and with an execution time of *pipeline startup time* $+ (2m^2 + 3m)N/(p + (2m + 1)/2)$ for $m = 1$, and *pipeline startup time* $+ (6m + 2)N/(p+(2m+1)$ for $m > 1, m < p \leq 4m+1$, and *pipeline startup time* $+ (2m^2+3m)N/(p+(m-1)(2m+1))$ for $m > 1, p > 4m + 1$. This is the first parallel algorithm that achieves this time bound, which improves upon the fastest prevoiusly published algorithms by a factor $\geq (p + 2m^2 - m - 1)/(p + m + 1/2)$ for $m > 1$. In comparison with the previously published algorithms, these schedules not only have the best execution time, but also better program-space efficiency and more flexibility in being adapted to different architectures and higher order recurrences. In addition, our schedules are parameterized only in terms of the number of processors available, i.e., they can be generated efficiently in time proportional to the period size of a schedule at compile time when the number of processors is known.

# References


[1] Aho, A., Sethi, R., Ullman, J., "Compilers-Principles, Techniques, and Tools", Addison Wesley, 1986.

[2] Aiken, A., Nicolau, A., "Perfect Pipelining: A New Loop Parallelization Technique", Proceedings of ESOP, France, Springer-Verlag, 1988.

[3] Chen, S. C., "Speedup of Iterative Programs in Multiprocessing Systems", Ph.D Thesis, January 1975.

[4] Chen, S. C. and Kuck, D., "Time and Processor Bounds for Linear Recurrence Systems", IEEE Transactions on Computer, Vol. C-24, pp. 701-717, July, 1975.

[5] Chen, S. C., Kuck, D. and Sameh, A. H., "Practical Parallel Banded Triangular System Solvers", ACM Transactions on Math. Software, Vol. 4, pp. 270-277, Sept, 1978.

[6] K. Ebcioglu. "Some Design Ideas for a VLIW Architecture for Sequential-natured Software", Proceedings IFIP, 1988.

[7] K.Ebcioglu. "A Compilation Technique for Software Pipelining of Loops with Conditional Jumps". Proceedings of the 20th Annual Workshop on Microprogramming, pp. 69-79, ACM Press, 1987.

[8] J. A. Fisher. "Trace Scheduling: A technique for global microcode compaction". IEEE Transactions on Computers, No. 7,pp. 478-490, 1981.

[9] Gajski, D., "An Algorithm for Solving Linear Recurrence Systems on Parallel and Pipelined Machines", IEEE Transactions on Computers, Vol. c-30, No.3, March 1981.

[10] K. A. Gallivan, R. J. Plemmons, A. H. Sameh, "Parallel Algorithms for Dense Linear Algebra Computations", SIAM Review, Vol. 32, No. 1, pp. 54-135, March 1990.

[11] D. Hirschberg, a correspondence with the authors, Department of Information and Computer Science, University of California Irvine, November, 1991.

[12] T. Gross, M. Lam, "Compilation for High Performance Systolic Array", Proceedings of 1986 SIGPLAN Symposium on Compiler Construction, July, 1986.

[13] J. R. Allen, K. Kennedy, "PFC: A program to Convert Fortran to Parallel Form", Rice University Tech. Rep. MASC TR 82-6, Houston, TX, 1982.

[14] Hyafil, L. and Kung, H. T., "The Complexity of Parallel Evaluation of Linear Recurrence", JACM, Vol. 24, No.3, pp. 513-521, July 1977

[15] Kogge, P. and Stone, H., "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations", IEEE Transactions on Computer, Vol., C-22, No.8, August 1973.

[16] Kuck, D., "The Structure of Computers and Computations", Vol. 1, John Wiley & Sons, Inc., 1978.

[17] Ladner, R., Fischer, M., "Parallel Prefix Computation", JACM, Vol. 27, No.4, October 1980, pp. 831-838.

[18] Little, J., Blelloch, G. and Cass, T., "Algorithmic Techniques for Computer Vision on a Fine-Grained Parallel Machine", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 11, No. 3, March 1989.

[19] Nicolau, A., Wang, H. G., "Optimal Schedules for Parallel Prefix Computation with Bounded Resources", Proceeding of Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming , Williamsburg, Virginia, April 21-24, 1991.

[20] Padua, D., Wolfe, M., "Advanced Compiler Optimizations for Supercomputers", CACM, Vol. 29, No. 12, December 1986.

[21] B. R Rau, C. D. Glaeser, "Efficient Code Generation for Horizontal Architectures: Compiler Techniques and Architectural Support". Proceedings of the 9th Symposium on Computer Architecture, April 1982.

[22] H. Wang, A. Nicolau, "Speedup of Programs Containing Banded Linear Recurrence with Resource Constraints", Technical Report, Department of Information and Computer Science, University of California Irvine, December, 1991.

[23] H. Wang, A. Nicolau, "Optimal schedules and new time bounds for parallel evaluation of banded linear recurrences", Technical Report, Department of Information and Computer Science, University of California at Irvine, Jan. 1992.