JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 14, 15–28 (1992)

Estimating the Potential Parallelism and Pipelining of Algorithms for Data Flow Machines*

BILHA MENDELSON

IBM Israel, Science and Technology, Haifa 32000, Israel

AND

ISRAEL KOREN

Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, Massachusetts 01003

When porting an application to a parallel data driven machine is considered, the maximum achievable parallelism and pipelining need to be estimated. These estimates can be obtained in a hierarchical manner from a data flow graph representation of the given algorithm. A method for estimating these performance measures has been developed and is presented in this paper. Examples illustrating our method and comparing the estimated performance to simulation results are also included. © 1992 Academic Press, Inc.

1. INTRODUCTION

When adapting an algorithm to a parallel data driven machine one has to ask what is the speedup that can be achieved for the given algorithm that may justify the necessary changes in the algorithm. The speedup that an algorithm may achieve on a data driven machine is determined both by the algorithm structure, as well as by the translation of the algorithm to the parallel machine, referred to hereafter as the *Mapping process*. The algorithm structure imposes limitations on its speed of execution, specifically on the maximum potential parallelism that can be achieved and the maximum potential throughput of the outcoming results. This paper presents a method to estimate the bounds on the performance of a given algorithm. When trying to find upper bounds on the performance of a given algorithm the overhead associated with a particular implementation of the given algorithm should not be taken into account. Therefore, operation under "ideal" conditions is assumed. The ideal conditions include:

- Unbounded number of execution units.
- Inputs to the algorithm are accessible to every execution unit.
- Outputs from the algorithm are accessible from every execution unit.
- Full connectivity—any two execution units can communicate with each other with negligible cost.

Estimating the potential parallelism and pipelining is essential to any procedure for mapping algorithms onto parallel data driven architectures. Knowing the estimates for the above performance parameters allows the user to modify his/her algorithm if the required performance is not met. The performance estimations can also serve as bounds for evaluating the quality of the mapping to a data driven machine. An example for using the presented method for designing special-purpose data driven coprocessors can be found in [19].

The performance of an algorithm can be analyzed through either macroanalysis or microanalysis [5]. Macroanalysis of algorithms is a complexity analysis (e.g., [17]) while microanalysis is concerned with evaluating the execution time of a program as a function of the time needed to execute each operation in the program and is the focus of this paper. Only few works have attempted to analyze the potential parallelism and pipelining of given application algorithms using microanalysis. A method for microanalysis of sequential algorithms executing on a sequential machine has been described by Cohen [5]. Some estimations of parallelism in FORTRAN programs were reported in [14]. There, D. J. Kuck surveys theoretical estimations of performance measures for some general programming constructs (e.g., arithmetic expressions and linear recurrences) but not for a particular program. Other works have attempted to estimate the parallelism and speedup, but they were restricted to specific architectures such as [16] which suggests a software tool for measuring parallelism in large scientific/engineering applications using simulations of various machines. Other research efforts were aimed at analyzing the potential parallelism and pipelining under limited resources resulting in the need for scheduling

* This work was done while B. Mendelson was with the University of Massachusetts at Amherst and was supported in part by SRC under Contract 90-DJ-125.

0743-7315/92 $3.00

since not all the operations that are ready at the same time can be executed simultaneously [4, 9, 22].

Very few studies have concentrated on the analysis of the performance of algorithms for data driven machines. Such algorithms are usually represented as a data flow graph (DFG). Data flow graphs are composed of nodes which represent the operations and arcs which represent the data that are transferred from one node to another. Incoming arcs carry the operands and the outgoing arcs carry the results of the operation. The most important property of a DFG is its ability to exhibit all the data dependencies in the computations being performed. The potential parallelism and pipelining of a given algorithm for a data driven machine can therefore be estimated through a detailed analysis of all possible execution paths in the DFG of the given program. Arvind *et al.* [2] presented a method, using the data flow graph of a given algorithm, to assess the benefits of fine-grain parallelism in programs. The performance is estimated there by executing the DFG on an interpreter. In [20], experiments to measure the maximum parallelism of an ideal Very Long Instruction Word (VLIW) architecture were described. These measurements, also obtained by using an interpreter, were claimed to be equivalent to the execution of an idealized data flow machine. The authors conclude that there are substantial amounts of fine-grain parallelism that can be found in many algorithms. In [24] a method is presented for analyzing the performance of a given algorithm by first translating the algorithm to operation nets and then producing specification equations that can be solved to find the execution time. The analysis presented there is restricted to the worst and best case latencies. We present a method that, in addition to the above, also analyzes the pipeline period of a given algorithm and the average case of both latency and pipeline period. Such an analysis is very important for data driven machines as well as other machines but has not been done until now [15].

Our method for estimating the performance parameters for a given algorithm on a data driven machine can be applied to any data driven machine and is not restricted to certain architectures since the estimations are done under ideal conditions. The performance estimation is based on the data flow graph representation of the algorithm and is obtained automatically at compile time when the DFG representation of the algorithm is produced. In our implementation the given algorithm is expressed in SISAL [18]. We have extended the original SISAL compiler to include the estimation of parallelism and pipelining. In addition to the DFG, the modified compiler produces the performance estimation for the given algorithm based on the method presented here. Although most of the results presented in this paper hold for any parallel machine, some apply only to data driven machines.

This paper is organized as follows. In the next section we present our general approach for estimating the potential parallelism and pipelining. In Section 3 we explain in detail how to estimate the above measures for various program structures. In Section 4 we demonstrate the process of analyzing given algorithms through several examples and compare the estimated performance to the results obtained by an event simulator. A summary and conclusions are presented in the last section.

2. GENERAL APPROACH

To find the potential parallelism and pipelining for a given algorithm its data flow graph is generated first. The parallelism of an algorithm can be evaluated based on the critical path, i.e., the longest sequence of operations that have to be done sequentially. By comparing the overall execution time of the algorithm on a sequential machine to the accumulated execution time of the operations on the critical path, we can estimate the maximum speedup achievable when executing the algorithm on a parallel data flow machine (e.g., [8]). On the other hand, the longest operation in the graph is a good measure for the pipeline period. As will be shown later, the pipeline period is sometimes determined by a set of operations rather than a single operation in the case of if–then–else and loop structures.

We have implemented the method presented below for estimating the performance of a given algorithm. Initially the algorithm is written in SISAL which is a functional high level language. The SISAL compiler translates the algorithm into an intermediate code (IF1) [18]. We have added an additional phase that translates this intermediate code into a DFG and produces the estimations for the performance. The compiler identifies the basic structures in the program and for each one of them creates the appropriate subgraph. Then the subgraphs are combined to generate the complete DFG. Estimates for the potential performance of the program are obtained in parallel to the DFG creation. For each basic structure in the program we compute at compile time the performance parameters. Each basic structure thus carries a set of estimated performance parameters which are then employed to calculate the performance measures for the compound structures. The estimated measures are upper bounds on the performance. These bounds do not take into account some implementation overheads of the architecture.

Two performance measures are used: *latency* and *pipeline period.* The latency is the time, in clock cycles, elapsed from entering the input operands until the output is produced. The latency measures the potential parallelism of the given algorithm. We distinguish between two types of latencies:

• *worst case latency*—the input to output latency if the longest possible execution path is taken;

• *average latency*—the average input to output latency, based on branch probabilities and estimated number of loop iterations.

The first one is clearly a bound for the algorithm latency, while the second one provides a better estimate for the typical behavior of the algorithm.

The pipeline period is the mean time between successive results, allowing us to calculate the throughput, which is its reciprocal. We define two types of pipelining measures:

• *worst case pipeline period*—the elapsed time between successive results if the longest operation in the algorithm is always executed;

• *average pipeline period*—the average pipeline period based on branch probabilities and estimated number of loop iterations.

The worst case pipeline period provides a bound for the possible pipelining that can be achieved when the longest operation is executed. For example, in an if–then–else structure, the longest operation may be part of the *Then* path or the *Else* path and, consequently, the throughput of the algorithm will depend on which path is taken. The average measure yields a better estimate for the typical throughput of the algorithm than does the worst case one.

Many studies have been conducted to characterize the typical behavior of an algorithm (e.g., [11, 26]) to be used for compile time optimization. This characteristic behavior includes number of iterations in loop structures and branch probability in if–then–else structures. There are two ways to obtain the characteristic behavior of a given algorithm: to use known statistics of the characteristic behavior of similar algorithms [11, 26] or to run the given algorithm with typical data on any machine and analyze its execution. The latter was justified by J. A. Fisher [6] who found strong correlation between the behavior of a program (in terms of branch probability and number of loop iterations) with one set of data and its behavior with a different set of data.

3. PERFORMANCE MEASURES FOR THE BASIC STRUCTURES

We decompose the DFG into three types of basic structures: arithmetic/logic expressions, if–then–else expressions, and loops, and estimate the potential parallelism and pipelining for each one of them. By combining the performance estimates for the basic structures hierarchically we analyze the performance of the complete algorithm. In the next three subsections we present the data flow graphs of the three basic structures and derive expressions for the above-mentioned performance measures.

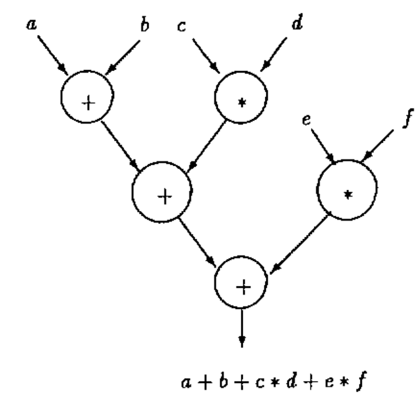


FIG. 1. An arithmetic expression.

3.1. *Arithmetic/Logic Expressions*

A common way to implement an arithmetic/logic expression and achieve the best performance is through a computation tree [1, 23] like the one shown in Fig. 1. The best parallelism can be achieved when the computation tree is balanced [13, 25]. We balance the computation tree not only according to the number of operations but also according to their execution time. Such computation trees require the data to pass through all possible paths of the arithmetic expression. Therefore, the latency of the arithmetic/logic expression is given by the length of the critical path.

All the operations in the computation tree need to be executed to produce the correct result of the expression. Therefore, there is no difference between the worst case and average values of the latency and pipeline period of an arithmetic/logic expression and we use the same notation for both.

The latency of an expression is denoted by *L(expression)*. The execution time of an operation is denoted by *EX(op)*. Given the estimated latencies of two subexpressions, the estimated latency of the compound expression is calculated by the recursive formula

$$L(expression) = Max\{L(sub\_expression\_1), L(sub\_expression\_2)\} + EX(op),$$

where *op* is the operator that generates the final expression out of these two subexpressions. The above formula assumes binary operation but it can be easily extended to *n*-ary operations.

The pipeline period of an expression is given by the execution time of the longest operation in the expression. This is true when a single execution unit serves the stream of operations executed within a single node of the DFG. However, if the pipeline period is of importance

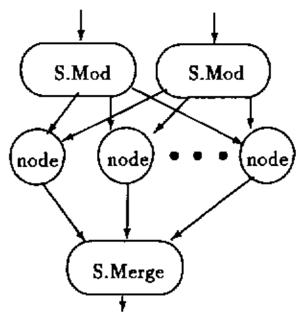


FIG. 2. *Multi-Node* structure—a structure for replicating an execution unit.

then multiple execution units can be used in order to increase the throughput. In such a case, we can replace the single node with the structure shown in Fig. 2, the *Multi-Node structure*, which consists of replicated execution units and additional synchronization nodes. The *Stream Modulo* node (*S.Mod*) routes the incoming data stream to the appropriate outgoing link in a round Robin fashion. The *Stream Merge* (*S.Merge*) guarantees the proper ordering of the results.

The input to the structure in Fig. 2 is a stream of data elements with interarrival time t. For simplicity, we assume that t is a constant. If the interarrival time is not a constant we may use its expected value. Replicating the node can reduce the pipeline period of the original node to

$$P(\textit{Multi-Node structure}) = max\left\{t,\ EX(S.Mod), EX(S.Merge),\ \frac{EX(op)}{m}\right\},$$

where m is the number of replications.

We want to find the optimal number of node replications needed to reduce the pipeline period of the new structure to its minimum value of

$$max\{t,\ EX(S.Mod),\ EX(S.Merge)\}.$$

It is denoted by m_{opt} and is equal to

$$m_{opt} = \left\lceil \frac{EX(op)}{max\{t,\ EX(S.Mod),\ EX(S.Merge)\}} \right\rceil$$

Note that this replication with its additional synchronization nodes is worthwhile only when $m \geq 2$. When we do replicate a node, $EX(op)$ will be replaced by

$$EX(S.Mod) + EX(op) + EX(S.Merge)$$

which increases the latency but the pipeline period is reduced.

3.2. *If–Then–Else Expressions*

When pipelining through an if–then–else expression, the outputs produced should be in the same order as their corresponding inputs. This can be accomplished by introducing three types of synchronization nodes: *True*, *False*, and *Merge*. The *True* and *False* nodes are denoted by T and F, respectively. These nodes receive a data input and a Boolean control input. When the control value is true (false) the T (F) node passes the data to the outgoing arcs or consumes it otherwise. The *Merge* node has two data inputs and a Boolean control input. The control input determines which one of the two data inputs will pass to the output.

A DFG representation of a general if–then–else expression is shown in Fig. 3. This structure is composed of three parts: computing the condition, executing the *Then* or *Else* part, and routing the result of either branch through a *Merge* node. Routing of input data to either the *Then* or the *Else* part is achieved using the *True* (T) and *False* (F) nodes.

The DFG representation of the if–then–else structure, shown in Fig. 3, enables us to achieve the best throughput because it allows overlapping between consecutive passes through the if–then–else structure. We have examined other possible DFG representations that require less synchronization nodes but after a careful analysis we concluded that they do not achieve the same throughput as that of the proposed structure.

Unlike the arithmetic/logic expression, the if–then–else structure is not deterministic, since the computation performed depends on the input data, and the path taken by the computation cannot be determined a priori. Therefore, the average and the worst case performances may differ.

The worst case analysis assumes that data passes always through the critical path or the path with the longest operation in the structure when the latency or the pipe-

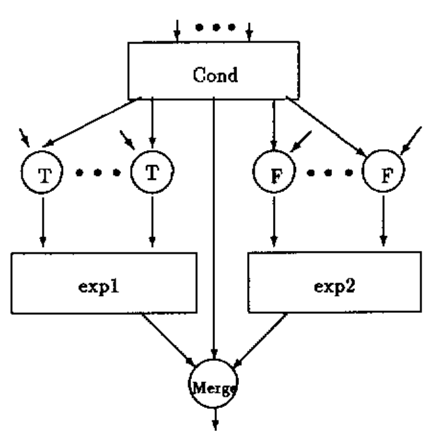


FIG. 3. If–then–else structure.

line period is considered, respectively. In the average case analysis we consider the probability that the *Then* or *Else* part is taken. An estimate for this probability is assumed to be known.

The worst case latency of an expression is denoted by $WL(expression)$. The latency of the *Then* and *Else* parts is equal to $WL(exp1) + EX(T)$ and $WL(exp2) + EX(F)$, respectively. $EX(T)$ can be assumed to be equal to $EX(F)$ because of their similarity. Therefore, we use the notation $EX(T)$. The worst case latency of the if–then–else structure is

$$WL(if\_then\_else) = Max\{WL(exp1), WL(exp2)\} + EX(T) + WL(Cond) + EX(Merge)$$

The probability of passing through the *Then* and the *Else* parts is denoted by p and $(1 - p)$, respectively. The average latency of the *Then* part is equal to $AL(exp1) + EX(T)$ and that of the *Else* part is equal to $AL(exp2) + EX(T)$. The average latency of the complete if–then–else structure is, therefore,

$$AL(if\_then\_else) = p * AL(exp1) + (1 - p) * AL(exp2) + EX(T) + AL(Cond) + EX(Merge).$$

The worst case pipeline period is given by the longest operation in the structure. The calculation of the average pipeline period of an if–then–else structure is based on the average pipeline periods of the *Then* and *Else* paths. We term the path with the largest pipeline period the "long" path while the other is termed the "short" path. The corresponding average pipeline periods are denoted by $AP(long)$ and $AP(short)$, respectively. We also denote the ratio between the average pipeline periods of the two branches by R, i.e., $R = AP(long)/AP(short)$. Let the probability of passing through the short path be denoted by p_s. Clearly, p_s equals either p (the probability of passing through the *Then* part) or $(1 - p)$. Assuming that path selections in consecutive passes are independent, the probability to select the same path successively follows a geometrical distribution. Consequently, the average number of times that the short path will be selected successively is denoted by D and is equal to $D = p_s/(1 - p_s)$.

If D is smaller than R, then R computations in the short path are completely overlapping the single preceding computation in the other branch. Therefore, the average pipeline period will be equal to $AP(long)$ divided by $(D + 1)$. On the other hand, if D is larger than R, then the time to complete $(D + 1)$ consecutive computations is determined by the time needed to complete the D computations in the short path. The computation in the short path can start only $AP(Cond)$ time units after the computation in the long path has started. Hence, we need to add this term to the overall computation time. The average pipeline period of the if–then–else structure is, therefore,

$$AP(if\_then\_else) = \begin{cases} \dfrac{AP(long)}{D + 1} & \text{if } D < R \\ \dfrac{D * AP(short) + AP(Cond)}{D + 1} & \text{otherwise.} \end{cases}$$

The above estimations are based on the assumption that the incoming data are always available when needed; i.e., the input rate is not smaller than the internal throughput or, in other words, the input bandwidth is sufficiently high. Also, we assume that complete overlapping between *Then* and *Else* paths is possible. As was shown in [3], the addition of some delay nodes might be necessary to achieve the maximum throughput. Linear programming can be employed to produce the optimal allocation of the delay nodes in the DFG [7]. Such delay nodes may be needed in loop structures as well.

3.3. Loop Structures

A data flow graph of a typical loop structure is shown in Fig. 4. This structure is composed of two parts: the body and the control. The body is the computation that must be repeated. The control part determines the number of iterations to be executed. There are two types of control parts: one corresponds to *For* loops and the other corresponds to *While* loops. The first is count controlled while the second evaluates a Boolean expression to decide whether to perform an additional computation.

The DFG representation depicted in Fig. 4 allows parallelism between the control and body parts. As mentioned earlier, if the number of iterations is not specified in the program, we estimate the latency and pipeline period based on user supplied values of the average or worst case number of iterations which will, in turn, yield average or worst case estimations, respectively. Therefore, we use, for simplicity, the same notation for the average and worst case measures.

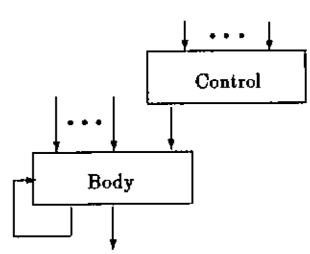


FIG. 4. Iterative structure.

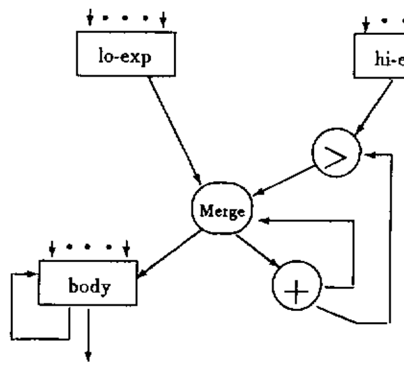


FIG. 5. *For* loop structure.

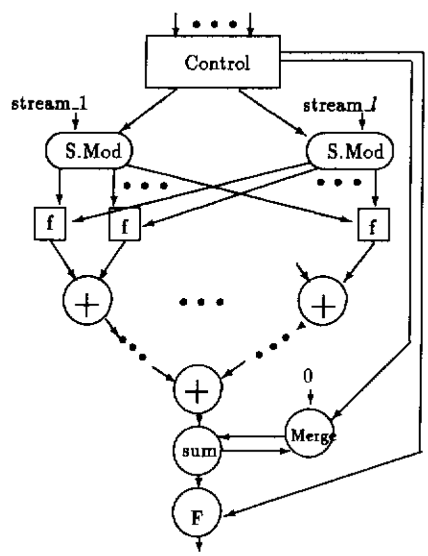


FIG. 6. Single result loop structure.

The *For* loop control part includes two subexpressions *lo_exp* and *hi_exp* to compute the bounds on the iteration count and three additional nodes as shown in Fig. 5. The *plus* (+) node increments the current count which is then checked by the *greater_than* (>) node. The latter gets two values, compares them, and produces a Boolean value accordingly. The current value of the count is passed through the *Merge* synchronization node.

In general, a loop may generate either a single result or a stream of results. We next estimate the performance when the loop structure produces a single result and then we analyze the other case.

3.3.1. Single Result Loop Structure. The DFG of a typical loop structure with a single result is shown in Fig. 6. The control signal (which can be the count value in a *For* loop or a Boolean value in a *While* loop) passes to the synchronization nodes, which control the input streams (stream_1, ..., stream_l). These synchronization nodes are *Stream Modulo* nodes (*S.Mod*). To accelerate the execution of the loop structure, we may replicate the iteration body several times. This way we reduce the number of passes through the loop and as a result reduce the overall latency of the loop structure. We want to point out that replicating the body in a single result loop can be done only if it consists of an associative operation (summation of partial results, finding minimum, etc.). With a nonassociative operation, replications are not always permitted. For example, if the algorithm must find the average number of positive elements between two consecutive negative elements then replicating the body can yield wrong results.

In what follows we analyze a loop structure containing several replicas of the body,[1] denoted by f blocks in Fig. 6. The partial results generated by all copies of the f block have to be accumulated to produce the loop output. This task is accomplished in two steps: first the partial results are combined using a computation tree (consisting for example, of *plus* (+), *Max*, or *Min* nodes) and then the partial results of the various iterations are accumulated using, for example, a *Sum* node. In summary, the loop body consists of m replicas of the f block and a computation tree which may be, for example, a summation tree as shown in Fig. 6.

The input to the loop body in the DFG is a stream of data elements with interarrival time t. For simplicity, we assume, as was done in Section 3.1, that t is a constant. The delay associated with the result of the loop structure depends on the original number of iterations, n, the latency of an f block, denoted by $L(f)$, the interarrival time, t, and the number of f block replications, m. The lower bound on the delay of the single result loop structure is determined by the bandwidth of the inputs. The goal of the f block replications is to reduce the latency of the loop structure and make it as close as possible to its lower bound nt.

To produce the final result, the data must pass through the f blocks and the summation tree. The latency of the tree depends on the number of replications, m. We can divide the summation tree into two subtrees. One is a complete binary tree with $\lfloor \log m \rfloor$ levels and the second is a partial tree. We can further divide the partial tree into two subtrees. One is a complete tree with log $\lfloor m - 2^{\lfloor \log m \rfloor} \rfloor$ levels and the other is a partial tree. We may continue this process until the partial tree is either a complete tree or is empty.

The above subtrees have been introduced to take into account the possible overlap between the subtrees' com-

[1] Note that this analysis includes, as a special case, the situation where no replication is done.

putations. The latency of the summation tree (accounting for the overlap) is given by the recursive formula

$$L_{sum.\,tree}(m) = Max\{LC(m), LP(m)\} + \delta(m)EX(+),$$

where $LC(m)$ is the latency of the complete tree and is given by

$$LC(m) = (2^{\lfloor \log m \rfloor} - 1)t + EX(+)\lfloor \log m \rfloor,$$

$LP(m)$ is the latency of the partial tree and is given by

$$LP(m) = \begin{cases} 0 & m = 0 \text{ or } m - 2^{\lfloor \log m \rfloor} = 0 \\ 2^{\lfloor \log m \rfloor}t + L_{sum.\,tree}(m - 2^{\lfloor \log m \rfloor}) & \text{otherwise,} \end{cases}$$

and

$$\delta(m) = \begin{cases} 0 & m = 2^k \ (k \text{ is an integer}) \\ 1 & \text{otherwise.} \end{cases}$$

The latency of the body, as a function of the number of replications, can be therefore estimated as

$$L(body, m) = \left(\left\lceil \frac{n}{m} \right\rceil - 1\right) P(body) + CL + L_{sum.\,tree}(m),$$

where n is the original number of iterations,

$$CL = L(f) + EX(S.Mod) + EX(Sum) + EX(F),$$

and $P(body)$ is the pipeline period of the body determined by $Max\{IP(body), mt\}$. $IP(body)$ is the internal pipeline period of the body. This can be either the execution time of the longest operation in the body or the execution time of a sequence of operations in the body when there is a dependency between iterations, as is illustrated in Section 4.

Theorem 1 gives the optimal number of f block replications, denoted by m_{opt}, that minimize the latency of the loop structure. Even if the number of replications is determined at run time rather than compile time, the value of m_{opt} still allows us to calculate the minimum latency of the loop structure.

THEOREM 1. *The optimal number of f block replications necessary to achieve the minimum possible execution time of a single result loop structure is*

$$m_{opt} = \begin{cases} \left\lceil \dfrac{IP(body)}{t} \right\rceil & \left\lceil \dfrac{IP(body)}{t} \right\rceil \le \dfrac{1}{2} + \sqrt{\dfrac{1}{4} + \dfrac{n \cdot IP(body)}{t + EX(+) + IP(body)}} \\ \left\lceil \dfrac{n}{\left\lceil \dfrac{n}{\left\lceil \frac{IP(body)}{t} \right\rceil} \right\rceil} \right\rceil & \dfrac{1}{2} + \sqrt{\dfrac{1}{4} + \dfrac{n \cdot IP(body)}{t + EX(+) + IP(body)}} < \left\lceil \dfrac{IP(body)}{t} \right\rceil < n \\ n & \textit{otherwise,} \end{cases}$$

where IP(body) is the internal pipeline period of the body.

The proof can be found in the appendix.

Finally, the latency of the single result loop structure is

$$L(loop) = L(control) + L(body, m),$$

where $L(control)$ denotes the latency of the control part of the loop. The pipeline period of the single result loop is the same as its latency.

3.3.2. Stream of results loop structures. Consider the general loop structure shown in Fig. 4. The latency of the first result is

$$L(first\ result) = L(body) + L(control)$$

and the pipeline period is equal to

$$\text{Max}\{t, P(control), P(body)\},$$

where $P(body)$ and $P(control)$ are the pipeline period of the loop structure body and the control part, respectively. We wish to check whether replicating the body can reduce the pipeline period of this structure.

When the pipeline period is determined by the interarrival time, t, or by $P(control)$, we cannot achieve better performance by replicating the loop body. On the other hand, when the pipeline period of the stream of results loop structure is determined by $P(body)$, we can improve the performance by replicating the body part. This way we allow overlapping of consecutive computations within the loop, producing new results at a rate higher than $1/P(body)$.

A stream of results loop structure with replicated body part (shown as f block) is depicted in Fig. 7. For synchronization purposes we use two nodes: *Stream Modulo* (*S.Mod*) which synchronizes the stream of inputs and

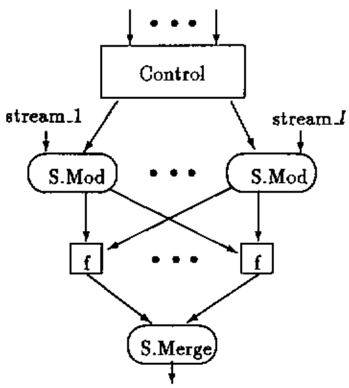


FIG. 7. Stream of results loop structure.

Stream Merge (*S.Merge*) which guarantees the proper ordering of the results. The pipeline period of this structure is

$$\frac{Max\{mt, EX(S.Mod), EX(S.Merge), P(control), P(f)\}}{m}.$$

We must find the optimal number of f block replications, m_{opt}, to achieve the best performance. If there are only a few replications then the same f block will receive new input data with interarrival time smaller than $P(f)$. On the other hand, if m is too large then the same f block will receive new input data at a very low rate resulting in idle time periods. The smallest pipeline period that can be achieved is

$$Max\{t, P(control), EX(S.Merge)\}.$$

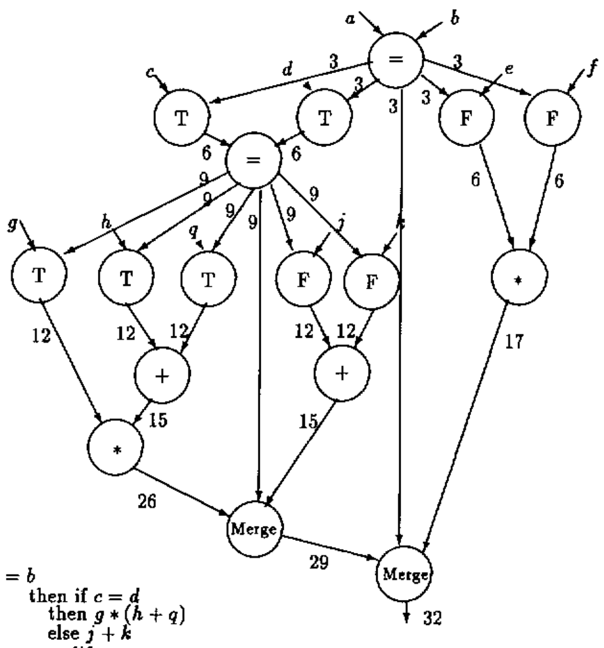
Note that when a complete overlap is achieved, the pipeline period is independent of $EX(S.Mod)$. The best m is therefore

$$m_{opt} = \left\lceil \frac{Max\{P(f), EX(S.Mod)}{Max\{t, P(control), EX(S.Merge)\}} \right\rceil,$$

where $m_{opt} \leq n$.

4. NUMERICAL EXAMPLES

In this section we demonstrate the performance estimation procedure through several examples. We start with a simple nested if–then–else program. Figure 8 shows the program and its corresponding DFG generated by our compiler. For the analysis of this example, we use the execution times from [12]. The number marked on each arc represents the accumulated worst case latency at that point. As can be seen from the figure, the latency of the complete DFG in the worst case is equal to 32




```
if a = b
    then if c = d
        then g * (h + q)
        else j + k
        endif
    else e * f
endif
```

FIG. 8. Nested if–then–else expression and its DFG representation.

clock cycles and the pipeline period is 11 clock cycles. The above results correspond to the length of the critical path and the longest operation (multiply) in the graph, respectively.

In Figure 9 we compare the estimated values of the average pipeline period of the example in Fig. 8 to the simulation results obtained using the PARET [21] event simulator, developed at AT&T Labs. The purpose of the simulation is to evaluate the performance of a data driven machine which operates under the ideal conditions as outlined in Section 1 in order to verify the previously presented analytical expressions.

Figure 9 shows the average pipeline period as a function of the probability of taking the *Then* path of the outer if–then–else. The *Then* path in this example is the short path. In Fig. 9a the probability to pass through the *Then* path of the inner if–then–else is 0.2. We can see that as the probability to pass through the outer *Then* path increases, the average pipeline period approaches the average pipeline period of the *Else* path of the inner if–then–else. In Fig. 9b, the probability to pass through the *Then* path of the inner if–then–else structure is equal to 0.8 instead of 0.2. In this case, as the probability of passing through the outer *Then* path increases, the average pipeline period decreases. This continues as long as there is a complete overlap between the *Else* and *Then* branches of the outer if–then–else. Further increase in the probability of taking the outer *Then* path reduces the overlap and results in an increase in the average pipeline period. As

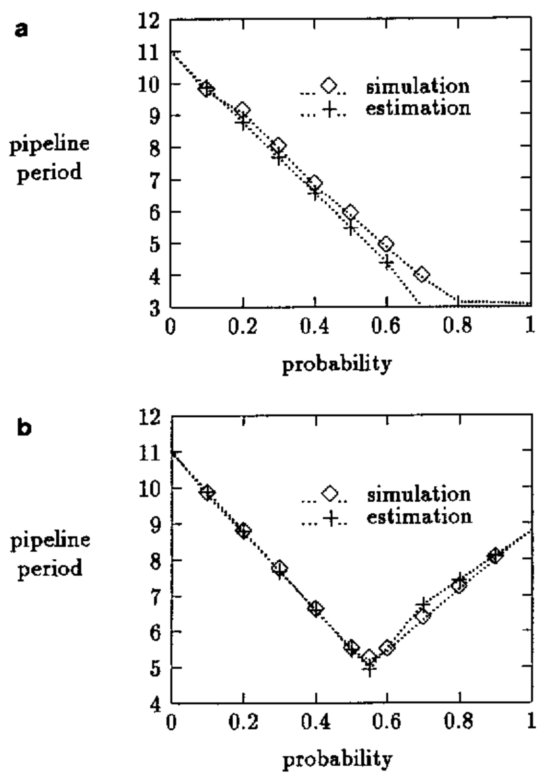


FIG. 9. Comparing the estimated pipeline period to simulation results for the example in Fig. 8. (a) The probability to pass through the *Then* path of the inner if–then–else is 0.2. (b) The probability to pass through the *Then* path of the inner if–then–else is 0.8.

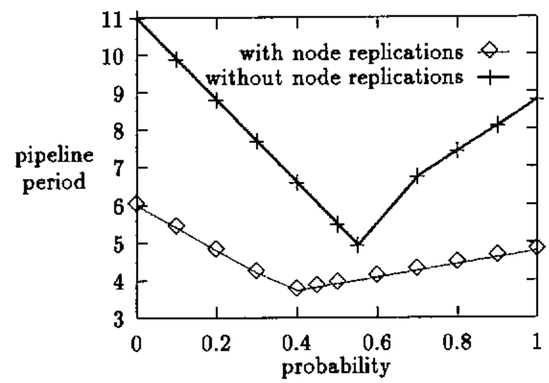


FIG. 10. Comparing the estimated pipeline period for the example in Fig. 8 with and without node replications (the probability to pass through the *Then* path of the inner if–then–else is 0.8).

can be seen from the figure, the estimated values are very close to the simulation results. Therefore, the calculated estimates are sufficiently accurate and lengthy simulations can be avoided. In both cases the worst case pipeline period is 11 clock cycles which is substantially higher than the average case pipeline period.

If we allow node replications, we can reduce the pipeline period of the algorithm as was discussed before. By replacing the *multiply* nodes in Fig. 8 with *Multi-Node* structures consisting of four *multiply* nodes, the pipeline period reduces to 3 clock cycles. The latency of the complete DFG in the worst case will increase from 32 to 38 clock cycles while the pipeline period will decrease from 11 to 3 clock cycles. The average pipeline period in this case will also equal 3 since this is the execution time of the *Cond* node. The average and worst case pipeline periods will differ if a smaller number of node replications are used. Figure 10 compares the average pipeline period depicted in Fig. 9b to the average pipeline period when the *multiply* node is replicated only twice. The worst case pipeline period now decreases from 11 to 6 clock cycles while the minimum average pipeline period is reduced from 5 to 3.7 clock cycles. In summary, by replicating the nodes with the highest execution time in a DFG, both worst case and average case pipeline periods of an algorithm can dramatically improve.

As an example for a single result loop, we have chosen an inner product algorithm, as shown in Fig. 11a. The interarrival time between the inputs is equal to 1 time unit. Figure 11b shows the latency of the loop as a function of the number of *f* body replications. In this figure we compare the estimated latency, for an expected value of $\bar{n} = 1000$, to simulation results where n is uniformly distributed in the range [800, 1200]. We can see that as the number of replications increases, the latency of the inner product loop decreases. This continues until m reaches the optimal number of *f* body replications necessary to achieve the minimum possible latency, which is $m_{opt} = 11$. Further increase in m will not decrease the latency (in some cases it may even slightly increase) and it is therefore not recommended. The minimum latency is $L(body, 11) = 1030$ which is very close to the lower bound $\bar{n}t = 1000$.

Figure 11c shows a graph of the cost-performance of the loop as a function of m. The number of nodes in the DFG has been chosen for evaluating the cost of the hardware associated with the algorithm implementation. By sharing hardware the number of functional units required to execute the algorithm can be smaller than the total number of operations and still achieve the best performance. However, we can use the number of nodes in the DFG representation of the given algorithm as a first approximation for the overhead that is associated with the algorithm implementation. Consequently, we have chosen the product of the number of nodes in the DFG by the latency as a cost-performance measure. The cost-performance decreases sharply at low values of m while as we reach m_{opt} (11 in this case) it decreases modestly; for $m > m_{opt}$, the cost increases with no gain in performance. Therefore, in order to reduce the cost, one may choose a value for m smaller than m_{opt} in the region where the a

$$\textit{for } k = 1 \textit{ to } n$$
$$Q = Q + Z(k) * X(k)$$

b

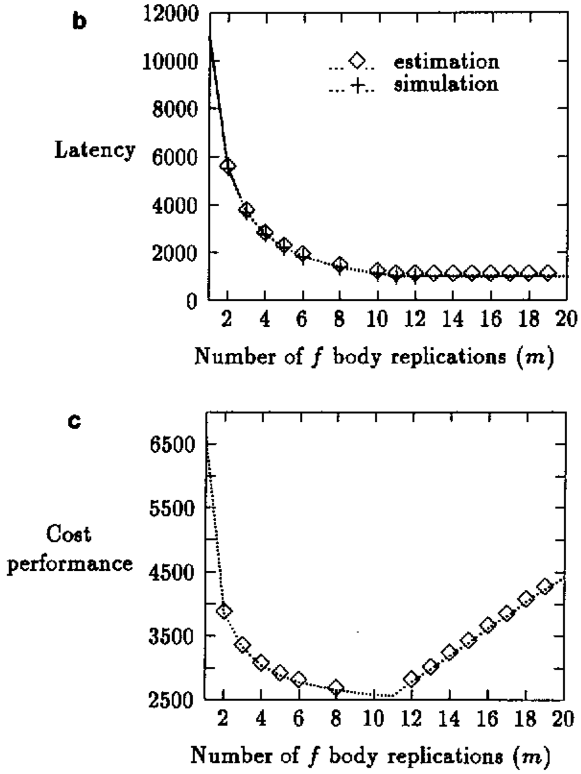
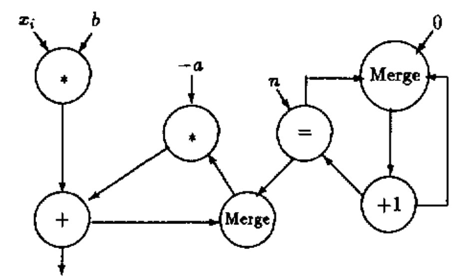


FIG. 11. Analyzing the performance of the inner product algorithm. (a) Inner product algorithm. (b) The latency of the inner product loop. (c) The cost-performance of the inner product loop.

improvement in the performance is not significant while the cost growth is substantial.

Figure 12 shows a first-order impulse response filter [10] as an example for a loop structure where the current iteration depends on the previous iteration. Because of the dependency between successive iterations, the pipeline period of the body is $EX(*) + EX(+) + EX(Merge)$. Replicating the body will not reduce this pipeline period and the latency of the loop structure. In this example the result of the first iteration is produced after 26 clock cycles which is the accumulated execution time of the operations along the critical path. The second result, however, is produced 17 clock cycles later and not 11 which is the execution time of the longest operation in the graph (the multiply operation). Here, the pipeline period of the loop structure is determined by a sequence of operations that cannot be overlapped, which includes the *multiply*, *add*, and *Merge* nodes.

In the last example, we combine the nested if–then–else structure from Fig. 8 with a loop structure as shown in Fig. 13. Figure 14 depicts the optimal number of replications as a function of the probability to pass through the *Then* path of the outer if–then–else (denoted by p) in Fig. 13. This is the number that minimizes the latency of the loop structure, $L(body, m)$. Figure 15 shows the cost-performance product for the above algorithm when the probability p is fixed at 0.4. Note that, unlike Fig. 11c, the cost-performance here achieves its minimum at a value of m lower than m_{opt}, specifically at $m = 4$ instead of $m_{opt} = 7$. This is a result of the high cost associated with replicating the f block.



$$y_i = bx_i - ay_{i-1}$$

FIG. 12. First-order impulse response filter.

Figure 16 shows the sensitivity of the latency to changes in the probability p for a fixed value of m, say m_0. To analyze this sensitivity we use the ratio of $L(body, m_0)$ to $L(body, m_{opt})$. If $m_0 = 5$, i.e., $m_0 = m_{opt}$ for $p = 0.55$ (see Fig. 14), the latency is very sensitive to changes in p. If, however, we assume that the estimated probability to pass through the outer *Then* path is anywhere in the range $0.55 - 0.2 \leq p \leq 0.55 + 0.2$ then a value of $m_0 = 8$ should be selected according to Fig. 14. As might be expected, the latency ratio for $m_0 = 8$ is less sensitive to changes in p. Consequently, the range for p rather than its expected value should be taken into account when selecting m_0.

5. SUMMARY

Estimating the parallelism and pipelining of a given algorithm is essential when porting of the application to a parallel data driven machine is considered. A method for analyzing the potential parallelism and pipelining was

```
sum:=0
for i = 1 to 1000
  if a = b
    then if c = d
           then r= g * (h[i] + q[i])
           else r = j[i] + k[i]
           endif
    else r = e[i] * f[i]
  endif
  sum:=sum+r
endfor
```

FIG. 13. A nested if–then–else in a loop structure.

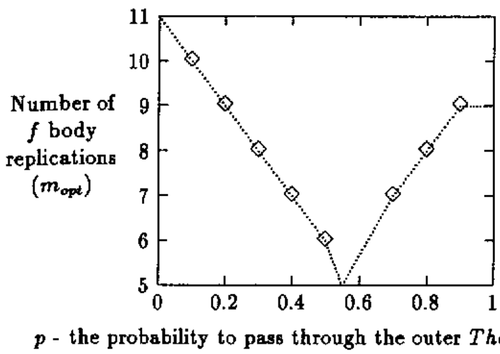


FIG. 14. The dependence of m_{opt} on the probability to pass through the *Then* path of the outer if–then–else in Fig. 13.

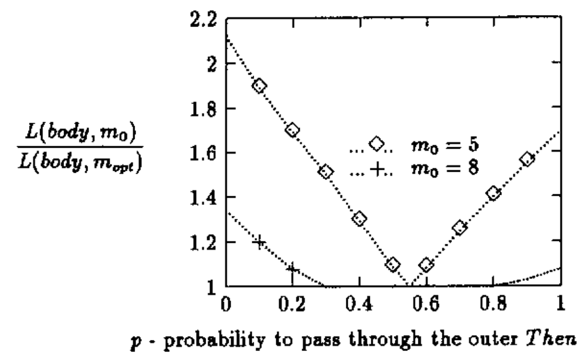


FIG. 16. The ratio of $L(body, m_0)/L(body, m_{opt})$ versus the probability to pass through the *Then* path of the outer if–then–else in Fig. 13.

presented in this paper. The analysis of a given algorithm is performed on its data flow graph representation since it exhibits all the data dependencies in the algorithm that limit the parallelism and/or pipelining. The performance estimation is done automatically by the compiler while producing the data flow graph. It has been demonstrated that the estimated performance measures are very close to the simulation results.

APPENDIX

Proof of Theorem 1. The latency of the body is

$$L(body, m) = \left(\left\lceil \frac{n}{m} \right\rceil - 1\right) P(body) + CL + L_{sum.\ tree}(m).$$

The interarrival time between consecutive items in the input stream is t. We have m f block replications and therefore the interarrival time for any f block is mt. The pipeline period of the body, given by $P(body) = \text{Max}\{IP(body), mt\}$, becomes mt when m increases. Consequently, $L(body, m)$ can be rewritten as

$$L(body, m) = \begin{cases} \left(\left\lceil \frac{n}{m} \right\rceil - 1\right) \cdot IP(body) + CL + L_{sum.\ tree}(m) & IP(body) > mt \\ \left\lceil \frac{n}{m} \right\rceil mt - mt + CL + L_{sum.\ tree}(m) & IP(body) \leq mt. \end{cases}$$

We must prove that $L(body, m)$ achieves its minimum when the pipeline period of the f block equals the interarrival time for any f block; i.e., $mt = IP(body)$. $L(body, m)$ is monotonically decreasing with m as long as the decrease in latency due to the smaller number of iterations ($\lceil n/m \rceil$) is larger than the increase in the latency of the summation tree. Therefore, there is a value for m such that until then $L(body, m)$ is monotonically decreasing with m and afterwards the decrease due to the smaller number of iterations is not always larger than the increase in the latency of the summation tree. As will be shown next, this value is

$$(1/2 + \sqrt{1/4 + n \cdot IP(body)/(t + EX(+) + IP(body))}),$$

denoted by m^*. After this value the latency does not necessarily change for every increase in the number of replications. Therefore, m_{opt} will be chosen as the smallest value of the number of replications that yields the same number of iterations as $\lceil IP(body)/t \rceil$ which is

$$\left\lceil \frac{n}{\left\lceil \frac{n}{\left\lceil \frac{IP(body)}{t} \right\rceil} \right\rceil} \right\rceil$$

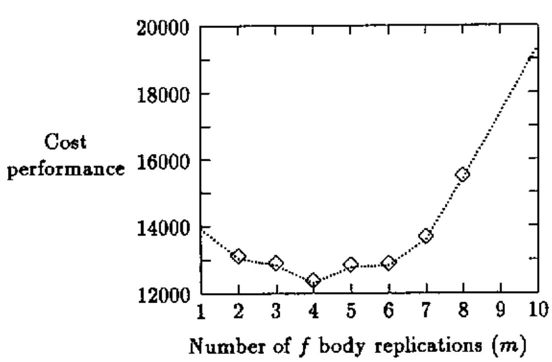


FIG. 15. Cost-performance of the algorithm in Fig. 13 when the probability to pass through the outer *Then* path equals 0.4.

We assume that m_{opt} is a divisor of n. If this is not true we can increase n to $\lceil n/m_{opt} \rceil m_{opt}$, without changing the latency of the loop structure. Consequently, throughout the proof we will assume that m_{opt} is a divisor of n.

We define the difference between the latency of two loop structures with f block replications by $\Delta L(m, i)$ where

$$\Delta L(m, i) = L(body, i) - L(body, m).$$

$\Delta L_{sum.\ tree}(m, i)$ denotes the difference between the latencies of the corresponding summation trees.

We want to show that $\forall m \neq m_{opt}$, $\Delta L(m_{opt}, m) \geq 0$. There are several cases that we need to check:

1. $\lceil IP(body)/t \rceil < m^*$:
   (a) $m > m_{opt}$:

$$\Delta L(m_{opt}, m) = \left(\left\lceil \frac{n}{m} \right\rceil mt - nt\right) - (m - m_{opt})t + \Delta L_{sum.\ tree}(m_{opt}, m)$$

since $nt \leq \lceil n/m \rceil mt \leq nt + mt$, and $\Delta L_{sum.\ tree}(m_{opt}, m) \geq (m - m_{opt})t$.

$$\begin{aligned}\Delta L(m_{opt}, m) &\geq (nt - nt) - (m - m_{opt})t + \Delta L_{sum.\ tree}(m_{opt}, m)\\ &\geq 0.\end{aligned}$$

(b) $m < m_{opt}$: This case will be proved by induction on the difference between $L(body, m)$ and $L(body, i)$, where $1 \leq i < m$ and $m \leq m_{opt}$. We want to prove that $\Delta L(m, i) \geq 0$.

Base of induction. For $i = 1$

$$\Delta L(m, m-1) = \left(\left\lceil \frac{n}{m-1} \right\rceil - \left\lceil \frac{n}{m} \right\rceil\right) \cdot IP(body) + \Delta L_{sum.\ tree}(m, m-1).$$

In the worst case, the above summation tree latency difference is $t + EX(+)$ (adding one replication and increasing by one the height of the summation tree). Therefore,

$$\Delta L(m, m-1) \geq \left(\left\lceil \frac{n}{m-1} \right\rceil - \left\lceil \frac{n}{m} \right\rceil\right) \cdot IP(body) - t - EX(+).$$

The right-hand side of the above inequality is greater than or equal to zero when

$$\left(\left\lceil \frac{n}{m-1} \right\rceil - \left\lceil \frac{n}{m} \right\rceil\right) \cdot IP(body) \geq t + EX(+).$$

This expression holds for

$$m \leq 1/2 + \sqrt{1/4 + n \cdot IP(body)/t + EX(+) + IP(body))} = m^*.$$

Hypothesis of induction. True for $i = j$,

$$\Delta L(m, m-j) = \left(\left\lceil \frac{n}{m-j} \right\rceil - \left\lceil \frac{n}{m} \right\rceil\right) \cdot IP(body) + \Delta L_{sum.\ tree}(m, m-j) \geq 0.$$

Induction step. $i = j + 1$:

$$\Delta L(m, m-j-1) = \left(\left\lceil \frac{n}{m-j-1} \right\rceil - \left\lceil \frac{n}{m} \right\rceil\right) \cdot IP(body) + \Delta L_{sum.\ tree}(m, m-j-1).$$

The difference of the latencies between $m - j - 1$ and m replications can be divided into the difference of the latencies between $m - j$ and m replications and the difference of the latencies between $m - j - 1$ and $m - j$ replications. In this case,

$$\Delta L_{sum.\ tree}(m, m-j-1) \leq \Delta L_{sum.\ tree}(m, m-j) + \Delta L_{sum.\ tree}(m-j, m-j-1)$$

Therefore,

$$\begin{aligned}&\Delta L(m, m-j-1)\\ &\geq \left(\left\lceil \frac{n}{m-j} \right\rceil - \left\lceil \frac{n}{m} \right\rceil\right) \cdot IP(body) + \Delta L_{sum.\ tree}(m, m-j)\\ &\quad + \left(\left(\left\lceil \frac{n}{m-j-1} \right\rceil - \left\lceil \frac{n}{m-j} \right\rceil\right) \cdot IP(body) + \Delta L_{sum.\ tree}(m-j, m-j-1)\right)\\ &\geq \Delta L(m, m-j) + \Delta L(m-j, m-j-1).\end{aligned}$$

$\Delta L(m, m-j) \geq 0$ from the hypothesis. The second term is greater than or equal to zero as proven in the base part.

2. $\lceil IP(body)/t \rceil > m^*$:
   (a) $m > \lceil IP(body)/t \rceil \geq m_{opt}$:

$$\Delta L(m_{opt}, m) = \left\lceil \frac{n}{m} \right\rceil mt - mt - \left(\frac{n}{m_{opt}} - 1\right) \cdot IP(body) + \Delta L_{sum.\ tree}(m_{opt}, m)$$

and since $\lceil n/m \rceil mt \geq nt$ and $IP(body) > m_{opt}t$

$$\begin{aligned}&\geq (n - m)t - (n - m_{opt})t + \Delta L_{sum.\ tree}(m_{opt}, m)\\ &\geq (m_{opt} - m)t + \Delta L_{sum.\ tree}(m_{opt}, m)\\ &\geq 0.\end{aligned}$$

(b) $\lceil IP(body)/t \rceil > m > m_{opt}$:

$$\Delta L(m_{opt}, m) = \left(\left\lceil \frac{n}{m} \right\rceil - \frac{n}{m_{opt}}\right) \cdot IP(body) + \Delta L_{sum.\ tree}(m_{opt}, m) \geq 0.$$

In this case,

$$m_{opt} = \left\lceil \frac{n}{\left\lceil \frac{n}{\left\lceil \frac{IP(body)}{t} \right\rceil} \right\rceil} \right\rceil$$

n

which is the smallest value of the number of replications that yields the same number of iterations as $\lceil IP(body)/t \rceil$. Hence, for $\lceil IP(body)/t \rceil > m > m_{opt}$, $\lceil n/m \rceil = \lceil n/m_{opt} \rceil = n/m_{opt}$ and $\Delta L_{sum.\ tree}(m_{opt}, m) \geq 0$.

(c) $m^* < m < m_{opt}$: Let k denote n/m_{opt} and let $(k + l)$ denote the number of iterations needed when m is the number of replications, and $l \geq 1$.

$$\Delta L(m_{opt}, m) = \left(\left\lceil \frac{n}{m} \right\rceil - \frac{n}{m_{opt}}\right) \cdot IP(body) + \Delta L_{sum.\ tree}(m_{opt}, m) = l \cdot IP(body) + \Delta L_{sum.\ tree}(m_{opt}, m).$$

In the worst case, the above summation tree latency difference is $(m - m_{opt})t + (\log m - \log m_{opt})$ (adding $(m_{opt} - m)$ replications and increasing the height of the summation tree). Therefore,

$$\Delta L(m_{opt}, m) \geq l \cdot IP(body) + \left(\frac{n}{k + l} - \frac{n}{k}\right) t + \left(\log\left(\frac{n}{k + l}\right) - \log\left(\frac{n}{k}\right)\right) EX(+)$$

$$\geq l \cdot IP(body) - \frac{n}{k} \cdot \frac{l}{k + l} t - \log\left(\frac{k + l}{k}\right) EX(+).$$

$IP(body) > m_{opt}t = (n/k)t$ and, therefore,

$$\Delta L(m_{opt}, m) \geq \left(l - \frac{l}{k+1}\right) IP(body) - \log\left(\frac{k + 1}{k}\right) EX(+) \geq 0$$

since $IP(body) \geq EX(+)$ and $(l - l/(k + l) > \log((k + l)/k)$ for $l \geq 1$.

(d) $m \leq m^* < m_{opt}$: In this case $\Delta L(m_{opt}, m) \geq 0$ for the same reason shown in case 1(b). ■

ACKNOWLEDGMENTS

Discussions with D. K. Pradhan, B. Patel, and R. Conrad are gratefully acknowledged.

REFERENCES

1. Aho, A. V., and Johnson, S. C. Optimal code generation for expression trees. *J. Assoc. Comput. Mach.* **23** (Nov. 1976) 488–501.
2. Arvind, Culler, D. E., and Maa, G. K. Assessing the benefits of fine-grain parallelism in dataflow programs. *Proc. Supercomputing '88*. Orlando, FL, Nov. 1988, pp. 60–69.
3. Brock, J. D., and Montz, L. B. Translation and optimization of data flow programs. *Proc. International Conference on Parallel Processing*, Aug. 1979, pp. 46–54.
4. Chen, Z., and Chang, C. Iterative-level parallel execution of DO loops with a reduced set of dependence relations. *J. Parallel Distrib. Comput.* **4,** 5 (1987), 488–504.
5. Cohen, J. Computer-assisted microanalysis of programs. *Comm. ACM* **25,** 10 (Oct. 1982), pp. 724–733.
6. Fisher, J. A. The static jump predictability of programs and data for instruction-level parallelism. Hewlett-Packard Laboratories Tech. Rep., June 1991, HPL-91-59.
7. Gao, G. R. Algorithmic aspects of balancing techniques for pipelined data flow code generation. *J. Parallel Distrib. Comput.* **6,** 1 (1989), 39–61.
8. Granski, M., Koren, I., and Silberman, G. M. The effect of operation scheduling on the performance of a data flow computer. *IEEE Trans. Comput.* **C-36,** 9 (Sept. 1987), 1019–1029.
9. Gui-shong, L., and Yun-gui, C. A model of quantitative analysis for performance evaluation of static data flow computers. *Proc. International Conference on Parallel Processing*, 1986, pp. 611–615.
10. Jackson, L. B. *Digital Filters and Signal Processing*. Kluwer, 1986.
11. Knuth, D. E. An empirical study of Fortran programs. *Software Practice exper.* **1,** 2 (1971), 105–133.
12. Koren, I., Mendelson, B., Peled, I., and Silberman, G. M. Data-driven VLSI array for arbitrary algorithms. *Computer* **21,** 10 (Oct. 1988), 30–43.
13. Kuck, D. J., Muraoka, Y., and Chen, S. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Trans. Comput.* **C-21,** 12 (Dec. 1972), 1293–1310.
14. Kuck, D. J. A survey of parallel machine organization and programming. *Comput. Surveys* **9,** 1 (Mar. 1977), 29–59.
15. Kung, S. Y., Lewis, P. S., and Lo, S. C. Performance analysis and optimization of VLSI dataflow array. *J. Parallel Distrib. Comput.* **4,** 6 (Dec. 1987), 592–618.
16. Kumar, M. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Trans. Comput.* **37,** 6 (Sept. 1988), 1088–1098.
17. Le Métayer, D. ACE: An automatic complexity evaluator. *ACM Trans. Programming Languages Systems* **10,** 2 (Apr. 1988), 248–266.
18. McGraw, J. R., *et al.* SISAL: Streams and Iterations in a Single Assignment Language: Reference Manual Version 1.2. M-146, Rev. 1, Lawrence Livermore National Laboratory, Mar. 1985.
19. Mendelson, B., Patel, B., and Koren, I. Designing special-purpose co-processors using the data flow paradigm. In Bic, L., and Gaudiot, J-L. (Eds.). *Advanced Topics in Data Flow Computing*, Prentice-Hall, Englewood Cliffs, NJ, 1991, Chap. 21, pp. 547–570.
20. Nicolau, A., and Fisher, J. A. Measuring the parallelism available

for very long instruction word architectures. *IEEE Trans. Compu.* **33,** 11 (Nov. 1984), 968–976.

21. Nichols, K. M., and Edmark, J. T. Modeling multicomputer systems with PARET. *Computer* **21,** 5 (May 1988), 39–48.
22. Polychronopoulos, C. D., Kuck, D. J., and Padua, D. Execution of parallel loops on parallel processor systems. *Proc. 1986 International Conference on Parallel Porcessing,* 1986, pp. 519–527.
23. Sethi, R., and Ullman, J. D. The generation of optimal code for arithmetic expression. *J. Assoc. Comput. Mach.* **17** (Oct. 1970), 715–728.
24. Skillicorn, D. B., and Glasgow, J. I. Real-time specification using lucid. *IEEE Trans. Software Engrg.* **15,** 2(Feb. 1989), 221–229.
25. Srini, V. P. An architecture comparison of data flow systems. *Computer* (Mar. 1986), 68–88.
26. Wiecek, C. A case study of VAX-11 instruction set usage for compiler execution. *Comput. Arch. Notes* **10,** 2 (Mar. 1982), 77–84.

BILHA MENDELSON received the B.Sc. and M.Sc. degrees in computer science from the Technion-Israel Institute of Technology, Haifa, in 1979 and 1987, respectively, and a Ph.D. in electrical and computer engineering from the University of Massachusetts, Amherst, in 1991. She is currently a research fellow at IBM, Science and Technology at Haifa, Israel. Dr. Mendelson's current research interests include dataflow, parallel and distributed systems, and compilation optimizations.

ISRAEL KOREN received the B.Sc., M.Sc., and D.Sc. degrees from the Technion-Israel Institute of Technology, Haifa, in 1967, 1970, and 1975, respectively, all in electrical engineering. He is currently a professor of electrical and computer engineering at the University of Massachusetts, Amherst. Previously he held positions with the Technion, the University of California, and the University of Southern California. He has been a consultant to Digital Equipment Corp., National Semiconductor, Tolerant Systems, and ELTA-Electronics Industries. Dr. Koren's current research interests are fault-tolerant VLSI architectures, models for yield and performance, floor-planning of VLSI chips, and computer arithmetic. He has edited and coauthored the book *Defect and Fault-Tolerance in VLSI Systems,* Vol. 1, Plenum, New York, 1989. He was also a co-guest editor for an *IEEE Transactions on Computers* special issue on high yield VLSI systems, April 1989. Dr. Koren is a fellow of the IEEE.

Received December 11, 1989; revised June 13, 1990; accepted April 24, 1991