Correspondence

Performance Based Design of High-Level Language-Directed Computer Architectures

Rajendra S. Katti and Mark L. Manwaring

Abstract— **This paper is concerned with the analytical modeling of computer architectures to aid in the design of high-level language-directed computer architectures. High-level language-directed computers are computers that execute programs in a high-level language directly. The design procedure of these computers are at best described as being ad hoc. In order to systematize the design procedure, we introduce analytical models of computers that predict the performance of parallel computations on concurrent computers. We model computers as queueing networks and parallel computations as precedence graphs. The models that we propose are simple and lead to computationally efficient procedures of predicting the performance of parallel computations on concurrent computers. We demonstrate the use of these models in the design of high-level language-directed computer architectures.**

I. Introduction

Multiprocessor systems are a special class of distributed computing systems that appear to represent the most promising way of obtaining the high-performance computers needed in many application fields. A variety of multiprocessor architectures with different design alternatives have been proposed, implemented, and made commercially available, but their relative merits have not been fully understood. It is thus important to develop methodologies and tools for the prediction of the performance of multiprocessor architectures, so that system designers can verify how well different alternatives suit certain given performance specifications.

Although a variety of multiprocessors have been implemented, programming them is no easy task. We propose that one way to program multiprocessors is by using the concept of high-level language-directed computer architecture (HLLDCA). Such architectures have as their machine language a high-level language (HLL). Due to the high level of abstraction of the instruction set of such an architecture, the pipelining of such instructions can be viewed as parallelism at lower levels of abstraction. Therefore, an HLLDCA can be a multiprocessor. The problem of programming such an architecture reduces to that of studying the interpreter of the machine language of the architecture.

How does one then design an HLLDCA? This work begins to answer this question by providing a means to start the design process. Let us suppose that an HLLDCA exists. If we can evaluate the performance of this HLLDCA rapidly then we can design a new HLLDCA which performs better, by modifying the old one repeatedly and recomputing the performance each time, until the new HLLDCA performs better than the old one. The question of "How to design an HLLDCA?" can therefore be reformulated as "How to evaluate the performance of an HLLDCA rapidly?"

Manuscript received November 22, 1993; revised August 5, 1994 and May 25, 1997.

R. Katti is with the Department of Electrical Engineering, North Dakota State University, Fargo, ND 58105 USA (e-mail:katti@plains.nodak.edu).

M. L. Manwaring is with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99164 USA.

Publisher Item Identifier S 1083-4419(98)02190-6.

The evaluation of the performance indexes of a model of an HLLDCA rapidly, leads us to two questions that must be answered:

1) If the parameters (or workload) of the model of an HLLDCA are known, how does one obtain the performance indexes from the model?
2) How does one obtain the workload or parameters of a model that represents an HLLDCA?

The rest of this paper addresses these two questions by modeling an HLLDCA as a network of queues. In Section II, we introduce the concept of information structures. Section II concludes by defining the research problem we have considered. Section III describes solutions to the research problems described in Section II. It also describes previous work briefly. Section IV verifies the solutions proposed in Section III and also gives some design guidelines to a high-level language-directed computer designer.

II. Preliminaries

Only recently has the design of computer architectures deviated from the Von Neumann style. Most architectures based on the Von Neumann architecture suffer from what is called "the semantic gap." This means that the objects and operations in an HLL are not closely related to the objects and operations in the architectures that execute the HLL [1]. We will address the issues involved in the semantic gap problem by formulating a framework based on "information structures" that would aid in the design of HLL architectures.

A. Von Neumann Architectures and HLL's

Von Neumann architectures usually have a single sequentially addressed memory in which program and data are stored. The only way of differentiating programs from data is by the way they are interpreted. Programs are fetched from memory and then interpreted, thus forming a fetch–execute cycle.

HLL's, on the other hand, deal with data that represent complex structures. HLL's themselves may have a structure that is complex. HLL's distinguish between programs and data. The execution of HLL's does not necessarily imply a fetch–execute cycle.

The above reasons motivate one to think about finding ways to store complex structures in programs and data that would result in their easy access and manipulation. The best that could be achieved would be the ability to access parts of programs or data in one instruction. The type of computer architecture that is designed based on HLL's is a high-level language-directed computer architecture. One could view this as the moving of an architecture toward an HLL until the structures of the HLL and architecture match.

B. Language-Directed View of Digital Computers

Digital computers are usually viewed as being composed of a controller and a controlled subsystem (datapath). The datapath consists of memory, registers and an arithmetic and logic unit (ALU). The controller has as its input the program strings from memory and it outputs control signals that control the datapath. We will return to the above view of digital computers after considering a language-directed view.

A program can be thought of as a realization of a function which when supplied with data will yield the desired results. In order to

1083–4419/98$10.00 © 1998 IEEE

compute the function, the function and data need to be stored in such a way that their access and manipulation is easy.

Programs and data represent information structures. These are called program structure and data structure. In addition to these structures, there are information structures used during execution that are not program or data structures. These are called execution structures. Examples of program structures would be the block structure in a language like Algol and lists in the case of Lisp. Examples of data structures would be arrays and records in Pascal. Examples of execution structures would be structures that are used during execution of a program but are not explicitly specified by the program or data, like stacks. Representation and transformation of these structures and their storage is of concern here. A digital computer viewed in terms of structures has the following functions:

1) stores program, data and execution structures;
2) transforms program, data and execution structures.

A digital computer therefore must consist of a set of devices that store program, data and execution structures and a set of devices for performing transformations on program, data and execution structures.

C. Architecture Design Methodology

We will begin with a view of a digital computer as composed of a controller and datapath. The controller design will be dealt with in terms of recognition of an HLL. The datapath design will be dealt with in terms of information structures.

The controller would therefore function in the following manner:

1) recognize words in an HLL program;
2) after each word is recognized a set of control signals is output by the controller that transform information structures in the datapath.

The datapath consists of:

1) program, data, and execution structures;
2) devices to move information between structures.

The devices in a datapath would contribute a great deal to the efficiency of program execution.

D. Structures and Instructions in Present Day Architectures

The structures in present day architectures are a sequence of memory locations that can hold numbers and are named by addresses that are a sequence of numbers. There are processing units (PU) associated with this memory structure which we call instruction, data and execution PU's.

An example of the contents of the instruction PU would be registers which hold the current instruction being executed and the address of the next instruction to be executed. The data PU may contain registers like the accumulator. The execution PU may contain the ALU. This is shown in Fig. 1.

The PU's associated with the memory structure can be viewed as necessary for the transformation of the structure.

The above architecture can execute mainly two kinds of instructions, transformational instructions (TI) and information moving instructions (MI). The TI's add new information into structures (in this case memory) or delete information from structures or change information in structures. The TI's therefore can cause the structure to shrink or expand. The MI's move information from the memory structure to the PU's or from input/output (I/O) devices to memory or vice versa. The MI's may also move information between structures. An example of such a case is where information is moved from main memory to the cache memory. Here, main memory and cache memory are considered as two structures.

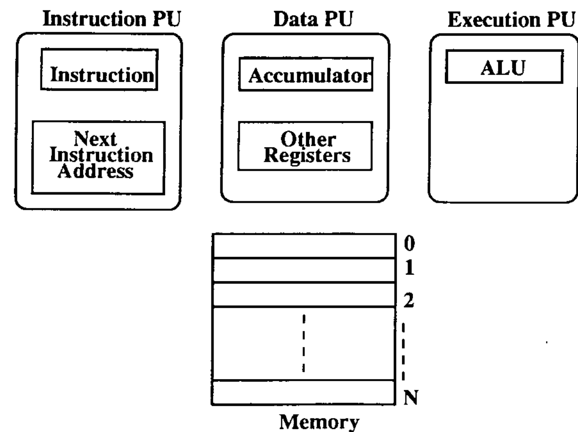

Fig. 1. Structures in present day architectures.

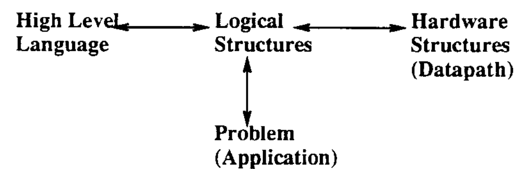

Fig. 2. A conceptual view of language-directed architectures.

E. Proposed Datapath

As seen in Fig. 1, present day architectures store programs and data in a single structure, i.e., memory. HLL's deal with objects and information that have different structures and the memory structure may not be suitable for their storage. Hence, we propose that the datapath in an HLL architecture must have the following components:

1) a set of structures that may be heterogeneous;
2) each structure has its own instruction, data and execution PU;
3) the structures are connected to each other depending on the interaction between structures needed in the HLL.

Fig. 2 describes the basic idea behind this approach. A language or a problem can be represented efficiently by a set of logical structures. These logical structures adhere closely to the geometry of structures in an HLL. An example of this would be the representation of block structure in a language like Algol with a tree. These logical structures can be derived from a language specification or a problem specification. The datapath in an architecture must be able to represent these logical structures. Fig. 3 shows a complete view of a high-level language-directed architecture without the interconnections between structures.

F. Instructions for Each Structure

Instructions for each structure in an HLL directed architecture would be of two kinds, transformational instructions and information moving instructions. The transformational instructions would involve the use of PU's associated with a particular structure. It is possible to achieve parallel transformation of structures. A structure along with its PU's are referred to as a structural unit (SU). A set of SU's can be used to store programs in an HLL, another set can be used to store data and another set can be used during execution of the program. It should be noted that it is possible, if need be, to manipulate all the SU's simultaneously.

G. Problem Definition

A high-level language-directed computer architecture is used to execute programs in the HLL directly. The HLL will be the ma-

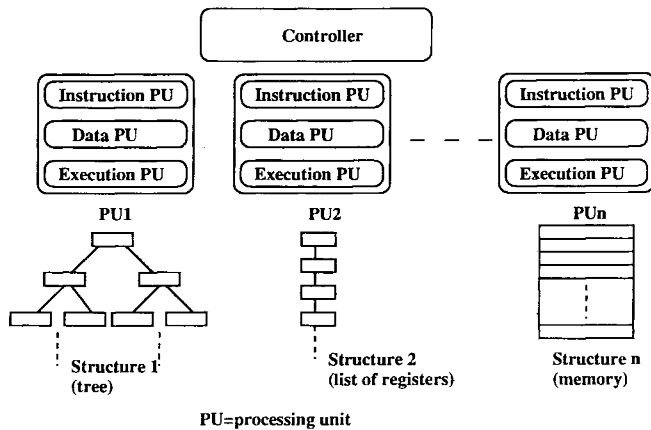

Fig. 3. A complete view of language directed architectures.

chine language of this architecture. The HLL program will first be assembled (in software or hardware) and then executed on the HLL computer architecture. Execution implies manipulation of structures in the architecture. The microprogram in the controller essentially interprets an HLL by associating every operation in HLL to structure transformations.

How does one then design an HLLDCA? The first step is to find the logical structures associated with the language. These structures correspond to the program and data structures in that language and also to the structures involved in the execution of a program in that language. This gives us a set of logical structures needed. Execution of a program in a language can be considered as the interaction of the structural units with each other. The modeling of structural units (SU's) and their interaction will help us evaluate rapidly the performance of an HLLDCA. Such models therefore will be a tool at the disposal of the designer of an HLLDCA. We model each SU as a queue and their interaction as a queueing network. How does one model the interaction of SU's? We start the modeling by assuming that the interaction between SU's is based on one of the following computer organizations:

1) pipelined organization;
2) superpipelined organization;
3) superscalar organization;
4) a combination of the above.

We should note that these organizations involve parallelism and hence there is a need to model issues involved with parallelism, such as synchronization delays, communication delays, queueing delays, and pipeline hazards. Communication delays can be modeled by inserting into the queueing network new queues that represent communication devices such as buses. Queueing networks inherently measure queueing delays. Pipeline hazards can be modeled by using feedback from the last stage in the pipe to previous stages. Synchronization is in general hard to model using queueing networks. We model synchronization with fork–join queues. Customers arriving at the queue get split up into two or more children (fork), and get serviced at different queues. After completing service each child waits for its siblings to complete (join). Such queues do not satisfy the product form solution of queueing networks and therefore new ways are needed to estimate their performance.

HLLDCA's execute instructions that are abstract. This is brought out in the model by allowing any service time distribution. A highly abstract instruction set can be modeled by a service time distribution that has a high mean and an arbitrary standard deviation. Queueing networks that have queues with general distributions (such queues are referred to as G/G/1, where G refers to a general distribution) do not have a product form solution. Therefore new ways of estimating their performance are needed.

If a queueing network represents a computing system, then its parameters represent the software executing on the computing system. Every computing machine is an interpreter of its machine language. Therefore the software we need to model is the interpreter of the machine language of the computing system which is an HLL. An interpreter never stops executing in the sense that after it executes one instruction in a program it immediately tries to fetch another instruction. Thus the queueing network needed to model this must be a closed one (customers do not arrive externally). Since no arrivals occur from the external world in such networks the parameters of such networks are just the service time distributions. An HLLDCA model must have parameters that best represent the execution of all programs in an HLL.

To summarize, if we have a queueing network that models an HLLDCA, then in order to estimate its performance we need to answer the following questions:

1) How does one obtain the performance of a network of G/G/1 queues?
2) How does one obtain the performance of a fork–join queue?
3) How does one obtain the parameters of a queueing network that models a HLLDCA?

We use a procedure that is similar to the one in [3] to estimate the performance of a network of G/G/1 queues. Each G/G/1 server is modeled as a two stage Coxian network. The other two questions will be answered in Section III.

III. Analytical Models

In this section we demonstrate ways to answer each of the questions we enumerated in the Section II. We start off by giving a method to obtain the performance of fork–join queues that have exponential service time distributions. Then the performance of fork–join queues with general service distributions can be obtained using the method of "solving" a network of G/G/1 queues. By solving a network, we mean, obtaining the performance indexes of queues in the network. Finally we give a procedure to estimate the execution time of a parallel program on a concurrent computer.

Section III-A shows that the methods stated in this section are indeed correct and it also demonstrates the use of these methods in HLLDCA design.

A. The Fork–Join Queue

Fork–join queues arise in many applications including parallel processing and flexible manufacturing systems. Programs can be modeled as a collection of tasks with some precedence constraints which can be represented by a graph whose nodes are the tasks and the edges correspond to precedence constraints. Such graphs consist of fork nodes when more than one edge leaves a node and join nodes when more than one edge enters a node. These nodes can be modeled by fork–join queues. Therefore, fork–join queues can model programs in concurrent Pascal [4], communicating sequential processes [5], and Ada [6]. The fork–join queue can also model distributed object-based programming systems [7].

Fork–join queues arise naturally in flexible manufacturing systems. In production lines, objects are built by assembling parts together. The assembly steps can be represented by a graph whose nodes are the assembly operations and whose edges are the precedence constraints between the operations. Such graphs contain fork and join nodes too.

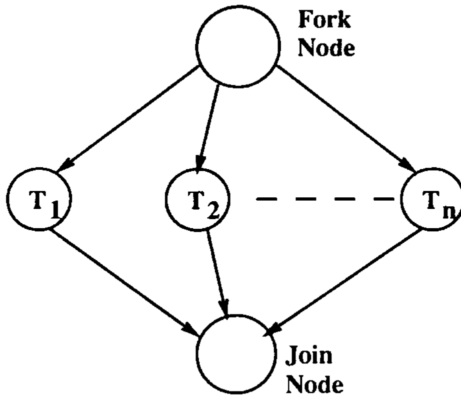

Fig. 4. A simple precedence graph.

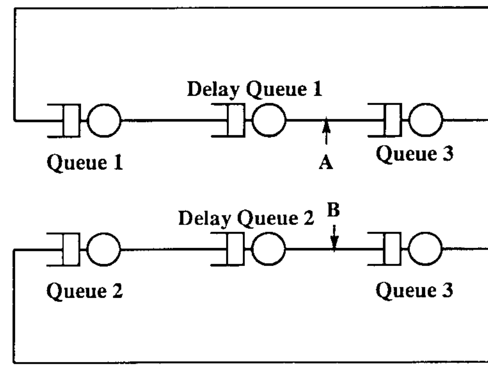

Fig. 5. The two independent networks of a fork–join queueing system.

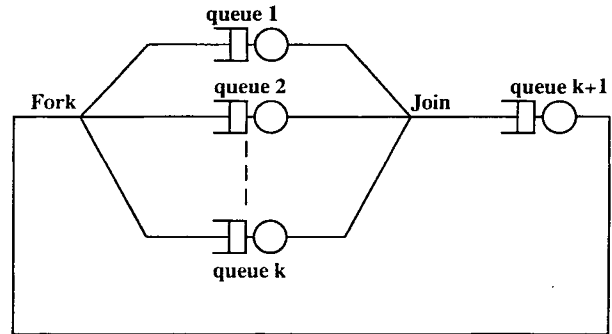

Fig. 6. The fork–join queue with k parallel tasks.

In this paper, fork–join queues are used to model inter-instruction parallelism. Inter-instruction parallelism refers to the fact that different groups of instructions can be executed in parallel. Fig. 4 shows a simple precedence graph with one fork node, N parallel tasks, and one join node.

Exact analysis of fork–join queues has been carried out by Flatto and Hahn [8], but their analysis is restricted to a system with one fork node, two parallel tasks, and one join node. Brun and Fajolle [9] obtain Laplace transforms of the response time distribution for the same model considered by Flatto and Hahn. Nelson and Tantawi [10] present an approximate solution for the mean task response time for any number of parallel tasks. Baccelli, Massey, and Towsley [11] derive the evolution equations that govern the behavior of acyclic fork–join queueing networks. Duda and Czachorski [12] present approximate solutions for series–parallel systems, and Heidelberger and Trivedi [13], [14] present approximate solutions for models with and without synchronization. Balsamo and Donatiello [15] present an algorithmic approach for the performance evaluation of parallel processing systems using fork–join queues.

Most of the above works restrict themselves to exponential service times. Otherwise they are computationally too expensive. Our work introduces a new approximation technique which is very accurate as compared to the other techniques and is applicable to queues with nonexponential service times.

B. The Fork–Join Graph Modeled by Queues

We will first consider a closed fork–join queueing system with two parallel tasks, as shown in Fig. 6 ($k = 2$). Each service center in Fig. 6 consists of a single server, an infinite capacity queue with FCFS (first come first serve) queueing discipline. A job splits up into two tasks T_1 and T_2. T_1 gets serviced at queue 1 and T_2 at queue 2. Queue 3 is not really part of the fork–join queue but is included to make the system more general. All service times are exponentially distributed with mean rates μ_i $(i = 1, 2, 3)$.

The most important aspect of the above queueing network is that at the join node jobs wait for their siblings. Let us split the above network into two independent networks as shown in Fig. 5. Each network in Fig. 5 consists of one queue from the fork–join part, queue 3 which is not in the fork–join part and a delay queue. The delay queues have service time such that the rate at which jobs pass points A and B in the two networks are equal. If we can find the service rates of the delay queues such that the above fact is true then the solution of the two independent networks will also give us the solution of the fork–join queueing system in Fig. 6. Let X_1 and X_2 be two random variables that represent the execution times at queue 1 and 2. Let D_1 and D_2 be two random variables that represent the execution times at the delay queues 1 and 2. Our problem now is to find the distributions of D_1 and D_2. Let us assume that D_1 and D_2 are exponentially distributed. We know that X_1 and X_2 are exponentially distributed, therefore all we need is to estimate the means of D_1 and D_2. Intuitively, a job has to wait at delay queue 1 if the service time at queue 1 is less than that at queue 2. We can thus estimate the E[D_1] and E[D_2] (expected values) by the following equations:

$$\begin{aligned} E[D_1] &= E[\max(X_1, X_2)] - E[X_1] \\ E[D_2] &= E[\max(X_1, X_2)] - E[X_2]. \end{aligned} \tag{1}$$

Reference [19] shows how $E[\max(X_1, X_2)]$ is calculated. We can then solve the two independent networks of Fig. 5 using the algorithm LBLANC [16]. From this solution we can get new estimates of $E[X_1]$ and $E[X_2]$ which now correspond to the rates at which customers arrive at the delay queues. Then, using (1), we can get new estimates of $E[D_1]$ and $E[D_2]$. This iterative process can be continued until the rates at which customers pass points A and B in the two networks are equal. A summary of the above algorithm is as follows:

1) split the fork–join network into two independent networks with delay servers added;
2) calculate the mean service times at the delay servers using (1);
3) solve the two networks using the algorithm LBLANC [16];
4) obtain the new rates at which customers arrive at the delay queues;
5) check if the expectation of the execution time of the delay queues in both the networks are identical;
6) if the departure processes are identical then stop else go to step 2.

Theorem 1: The iterative algorithm for obtaining the performance of fork–join queues converges.

Proof: Suri [17] presents the following definitions to define monotonicity in single class networks:

1) a network is N-monotonic if $X(n) \geq X(n-1)$ for $n \leq N$, where $X(n)$ is the throughput with n customers in the network;
2) server i is N-monotonic if $P(n_i \geq k|n) \geq P(n_i \geq k|n-1)$ for all k and for all $n \leq N$. $P(n_i \geq k|n)$ is the probability of the number of customers at server i (n_i) being greater than k given that there are n customers in the network;
3) a network (server) is ∞-monotonic if it is N-monotonic for all N.

Suri shows that a network with fixed-rate servers only is ∞-monotonic. From [18]

$$\frac{\partial X(n)}{\partial D_i} < 0$$

where D_i is the service demand at server i in the network. This implies that the throughput X_i at server i is monotonically decreasing. We will demonstrate that the algorithm terminates by assuming that X_i are exponentially distributed. Let $Y = \max(X_1, X_2)$. Therefore,

$$E[Y] = \frac{1}{\lambda_1} + \frac{1}{\lambda_2} - \frac{1}{\lambda_1 + \lambda_2}$$

where $E[X_i] = (1/\lambda_i)$. From (1), we obtain the following:

$$E[D_1] = \frac{1}{\lambda_2} - \frac{1}{\lambda_1 + \lambda_2}$$
$$E[D_2] = \frac{1}{\lambda_1} - \frac{1}{\lambda_1 + \lambda_2}.$$

Let us assume that the throughputs of the two networks are T_1 and T_2. In each iteration we would update $E[D_i]$ as follows:

$$E[D_1] = \frac{1}{E[T_2]} - \frac{1}{E[T_1] + E[T_2]}$$
$$E[D_2] = \frac{1}{E[T_1]} - \frac{1}{E[T_1] + E[T_2]}.$$

Without loss of generality we can assume that $E[T_1] > E[T_2]$. From the above equation, it follows that $E[D_1] > E[D_2]$. The solution of the networks will result in new values of $E[T_1]$ and $E[T_2]$ such that the reduction in $E[T_1]$ is more than the reduction in $E[T_2]$. This implies that each iteration will adjust $E[D_1]$ and $E[D_2]$ in a manner that ultimately leads to the following condition being true:

$$E[T_1] = E[T_2].$$

The above equation specifies that the throughput of the two networks are equal. This is the termination condition of our algorithm.

C. The General Fork–Join Queue

The algorithm for solving a fork–join queue in the previous section assumed that the service times of all the queues were exponential. That algorithm can easily be extended to general service time distributions by using the algorithm that solves networks of G/G/1 queues [3].

If we assume that in the fork–join queue the fork results in k parallel tasks then the new fork–join queueing system is shown in Fig. 6. Let $X_1, X_2, \cdots, X_k$ be random variables that represent the execution times at the k queues. Following is a summary of the algorithm to obtain the performance indexes of the network of Fig. 6.

1) Split the fork–join network into k independent networks with delay servers added. Let $D_1, D_2, \cdots, D_k$ be random variables that represent the service times at the delay queues.
2) Calculate the mean service times at the delay servers using the following equations:
$$E[D_1] = E[\max(X_1, X_2, \cdots, X_k)] - E[X_1]$$
$$E[D_2] = E[\max(X_1, X_2, \cdots, X_k)] - E[X_2]$$
$$\vdots$$
$$E[D_k] = E[\max(X_1, X_2, \cdots, X_k)] - E[X_k].$$
Reference [19] gives derivations for the calculations of $E[max(X_1, X_2, \cdots, X_k)]$ where X_i are either all exponential or Coxian. For example if X_i are exponentially distributed and $Y = \max(X_1, X_2)$, then,
$$E[Y] = \frac{1}{\lambda_1} + \frac{1}{\lambda_2} - \frac{1}{\lambda_1 + \lambda_2}$$
where $E[X_i] = 1/\lambda_i$.
3) Solve the k networks using the algorithm to solve G/G/1 networks in [3].
4) Obtain the new rates at which customers arrive at the delay queues.
5) Check if the departure process of the delay queues in the k networks are identical. Let the population at the delay queues be n, then the departure processes at these queues are equal if the throughputs at these queues are equal for all $n \leq N$. N is the population of the network. The condition for termination of the algorithm is that the throughputs at these queues must not differ by more than 1%.
6) If the k departure processes are identical then stop, else go to step 2.

We will adapt the above procedure in the design of HLLDCA's by assuming that each structural unit executes in parallel and then use graph reduction to estimate the response time.

D. Estimation of Execution Time

In this section, we develop a method that allows one to predict the performance of parallel computations running on concurrent systems. In particular we are interested in predicting the performance of an HLL interpreter running on a concurrent system. The interpreter is modeled as a task system with precedence relationships expressed as a graph. Resources in a concurrent system are modeled as queues in a queueing network. Using these two models as input, our method obtains estimates of the execution time of the parallel computation.

Most earlier works deal with restricted computation structures or are computationally expensive. Thomasian and Bay [20] considered a general task structure with precedence constraints expressed as a directed acyclic graph. Their method is based on the concept of hierarchal decomposition [21]. A two-level method is used to solve for the performance measures of the task system. At the higher level, the system behavior is specified by a Markov chain whose states correspond to the combination of tasks in execution. At the lower level, the transition rates among the states are computed using a queueing network solver. Their method is computationally expensive and can be used only for small systems.

Mak and Lundstrom [22] give a computationally efficient algorithm to predict the performance of parallel computations. They model computations by a series-parallel directed acyclic graph and resources by a queueing network. They modify the mean value analysis algorithm [23] to model overlap between two tasks and then develop an iterative algorithm to predict overlap using the mean value analysis algorithm and the task graph.

Ferscha [24] uses Petri nets to model parallel computations and resources. He then models the assignment of tasks to the resources by

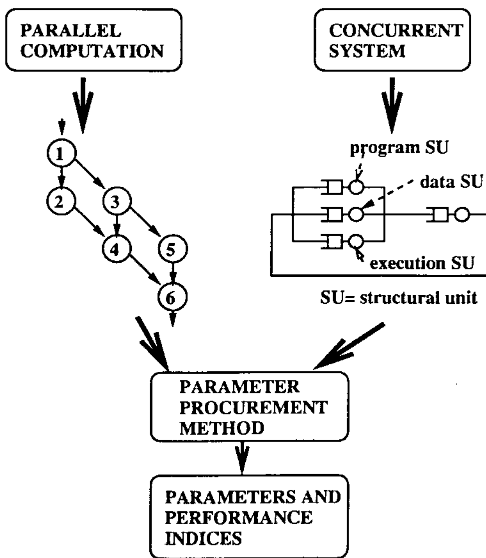

Fig. 7. The modeling of a parallel interpreter and a concurrent system.

arcs that originate in the Petri net that represents parallel computations and terminate in the Petri net that represents resources. Performance indexes of the system are derived by simulation of both the Petri nets and the arcs. This model although powerful is computationally very expensive.

Baccelli and Liu [25] have proposed an exact analytical approach to find the performance of execution of parallel programs on multiprocessor systems. Although their method is exact, it is computationally very expensive.

The method we propose is based on finding different types of delays involved in parallel computations from different models thus making it computationally very inexpensive.

E. System Model

Fig. 7 shows the two models used to represent parallel computations and concurrent systems. The parallel computation represents an HLL interpreter and the concurrent system represents a high-level language-directed computer. The high-level language-directed computer in Fig. 7 consists of four queues. Three queues referred to as program structural unit, execution structural unit and data structural unit form a fork–join queue and the fourth queue represents a communication device such as a bus. The completion of one loop in the queueing network implies the completion of execution of one instruction in the HLL. Therefore the intra-instruction synchronization is expressed in the task graph and the inter-instruction synchronization is expressed in the fork–join queue. We further assume that tasks in the task graph are statically assigned to either the program, execution or data structural unit. We also assume that each task in the task graph has associated with it a random variable that corresponds to the execution time of the task. We assume that the execution time distribution of all the tasks are known. The execution time is a random variable as it has to represent the usage of a task by all programs of an HLL. Besides, the execution time also includes the waiting time due to synchronization and queueing which are often nondeterministic.

F. Execution Time Calculation

The precedence graph models precisely the precedence delays in a parallel computation and the queueing network models the queueing delays precisely. The execution time of the precedence graph is influenced by both precedence and queueing delays. We can obtain the queueing delays from the queueing network by using the known algorithms to obtain the performance of queueing networks. We can obtain the precedence delays by graph reduction (explained later) or by methods given in [26]. In this paper we shall use graph reduction. If we can incorporate the queueing delays into the graph model before graph reduction then the execution time obtained by graph reduction will estimate the actual execution time.

We are given a graph that represents the workload of a network of k queues. Associated with each node in the graph is a random variable Y_i that represents the time taken to execute the task that node represents. We need to find the average execution time of the graph on a concurrent computer represented by a queueing network. Let D_k be the service demand at queue k. D_k is a random variable but when we refer to D_k we mean its expected value. The basic idea behind our approach is to first estimate D_k from the graph. This can be done easily as each node in the graph is statically assigned to execute on some queue k in the queueing network. With this estimate of D_k we can solve the queueing network to obtain queueing delays at each queue k. We can then incorporate these queueing delays into the graph by properly incorporating these delays into Y_i, for each node in the graph. From this new graph we can estimate the execution time, of the computation by graph reduction (explained later). The parameter procurement procedure proceeds as follows.

1) Obtain an estimate of D_k, the service demand at resource k from the task graph. To each node in the graph we assign a resource based on its function. Then an estimate of D_k is just the sum of all the random variables associated with tasks that are assigned to resource k.
2) Solve the queueing network that represents the concurrent system to obtain Q_k, the average time taken by a customer to complete service (this includes queueing delays) at each resource k. The previous algorithms for solving fork–join networks and for solving a network of G/G/1 queues can be used.
3) Use Q_k to find the new task times of the tasks in the graph. Q_k must be distributed among all the tasks that execute on resource k. This distribution is done such that the ratio of the new task time to Q_k is the same as the ratio of the initial task time to D_k.
4) Obtain the execution time from the task graph by using simple merging rules for nodes in the graph. This process is called graph reduction (explained below).

We will demonstrate with an example the workings of the above procedure in the next section. However we give a detailed explanation of graph reduction below.

G. Task Graph Reduction

Let every node in the task graph be associated with a random variable Y_i which denotes the execution time of that task. Task graph reduction is then the process of reducing a graph to a new graph with just one node. The mean of the random variable associated with this node is then an estimate of the mean system completion time.

There are two simple merging rules. We assume that the interaction between two tasks is only of two types. This assumption is not restrictive as we can assume any kind of interaction between tasks and use other ways of graph reduction [26]. Either they execute in parallel [shown in Fig. 8(a)] or in sequence [shown in Fig. 8(b)]. The first merging rule is shown in Fig. 9(a). The nodes 2 and 3 are merged to form a new node and the random variable associated with the new node is $Y_n = \max(Y_1, Y_2)$. Note that Y_n is a random variable which is a function of two random variables Y_1 and Y_2.

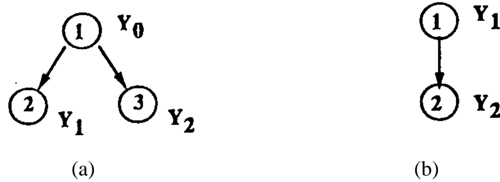

Fig. 8. (a) Tasks 2 and 3 are parallel tasks and (b) Tasks 1 and 2 are sequential tasks.

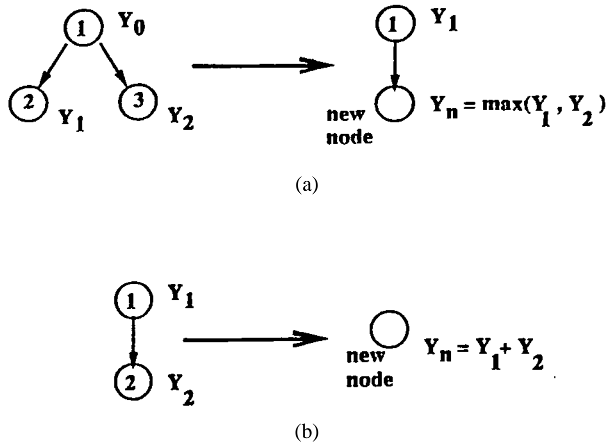

Fig. 9. (a) The first merging rule and (b) the second merging rule.

The second merging rule is shown in Fig. 9(b). The nodes 1 and 2 are merged to form a new node and the random variable associated with the new node is $Y_n = Y_1 + Y_2$.

IV. Model Verification and Application

In this section, the algorithm for predicting the performance of fork–join queues, and the execution time estimation algorithm will be verified. The procedure to obtain the performance of a high-level language-directed computer that executes a simple Pascal-like language is then demonstrated. Finally we conclude this section by showing how queueing networks can be used to analyze design problems in HLLDCA's.

A. Solving the Fork–Join Queue

The fork–join queue network used to verify our algorithm is shown in Fig. 10. We assume that service time distributions of each queue is exponential with rate λ_i ($\lambda_1 = 1.0, \lambda_2 = 5.0$ and $\lambda_3 = 3.0$). For a total customer population of 2 in the network, Table I compares our algorithm to a simulation of the network. The quantities compared are throughput, mean queueing time and mean queue length for queue 1. The simulation gives the upper, mean and lower estimates of the throughput, mean queueing time and mean queue length. These estimates assume a 90% confidence interval. Table I shows that our algorithm produces throughput, mean queueing times and mean queue lengths that are very close to the simulation values. We have tested our method for population sizes of up to 50 and found the errors in the throughput to be less than 20%. Errors in the queue lengths were less than 1% and errors in the queueing time were less than 2%. For example the queue length for queue 1 for a population of 50 is 49.54 using our algorithm and 49.66 using simulation. The amount of computation required increases with population size due to the dependence of the complexity of the LBLANC algorithm on the population size [16].

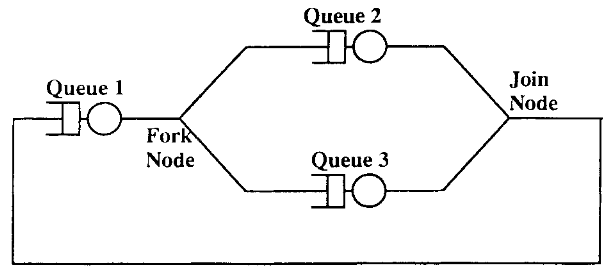

Fig. 10. A fork–join queue network.

TABLE I
A Comparison of Our Algorithm with Simulation

	Our Algorithm	Simulation
Throughput	0.918	1.29 upper 1.07 mean 0.85 lower
Mean queueing time	1.71	1.80 upper 1.44 mean 1.08 lower
Mean queue length	1.57	1.66 upper 1.53 mean 1.42 lower

TABLE II
A Comparison of Our Algorithm with Simulation

Number of Tasks	Completion time from [22] (ms)	Completion time from our algorithm (ms)
1	325.0	325.0
2	415.0	418.4
3	480.0	482.0
4	535.0	532.5
5	580.0	603.8
6	630.0	609.6
7	680.0	671.6

B. The Execution Time Estimation Algorithm

We will verify our algorithm by executing a parallel program on a multiprocessor, the Sequent Balance 8000. We will compare the execution times of a parallel program for different numbers of concurrent tasks to those predicted by our execution time estimation algorithm.

The Sequent Balance 8000 is a single bus, shared memory multiprocessor. The queueing network model of the Sequent Balance 8000 that is similar to the one in [22] will be used. The parallel program has a serial part (master) and N concurrent tasks (slave tasks). The concurrent tasks execute independently of each other except through contention of the system resources like bus and memory. The code for each concurrent task is similar to the code in [22]. Table II compares the execution times obtained by our algorithm with those given in [22]. Despite the computational simplicity of our algorithm these execution times are very accurate with a maximum error of about 4.1%.

C. High-Level Language-Directed Computer Design

The execution time estimation algorithm of the previous section is a method to obtain the performance of running a parallel program on a given computer. In high-level language computer architecture design,

the parallel program is the interpreter of an HLL. The interpreter is modeled as a precedence graph and the computer is modeled as a network of queues. Associated with each node in the interpreter is the mean execution time and the standard deviation of the mean execution time of the task that node represents. Given the precedence graph and the network of queues, the execution time estimation algorithm obtains the execution time of the graph taking into account issues like precedence delays and resource contention. We have used this procedure to estimate the execution time of a Pascal interpreter on a hypothetical parallel computer.

D. Design Guidelines

The fork–join queue can be used to give us insight into the procedure of designing HLLDCA's. We shall first consider the effect of instruction abstraction in pipelined machines. We shall then investigate the relationship between throughput and the coefficient of variation for high-abstraction and low-abstraction instruction sets in pipelined machines. All the results of this section have been obtained using the models and algorithms developed in this paper.

E. Instruction Set Abstraction

Instruction set abstraction or granularity is an important aspect of multiprocessor machines. Higher abstraction usually implies less communication and synchronization overheads and it also implies more opportunity for parallelism at lower levels of abstraction. However a major drawback of a high-abstraction instruction set is that it may lead to load-balance problems. Low-abstraction instruction sets on the other hand lack the load-balance problems, but synchronization and communication overheads become a dominant factor as the abstraction is reduced. Little can be said about the "correct abstraction level" beyond the fact that in every hardware environment, and for every interpreter, a "crossover point" exists beyond which the instruction set abstraction is too low to perform well. Using the models we have described it is possible to know, for a given hardware environment (a queueing network) and an interpreter graph, exactly where the "crossover point" exists.

Consider a hardware environment which consists of two pipeline stages. Let the average execution times in each stage be 1.0 time unit (with $CV^2 = 50.0$). We arbitrarily assume that this models a high-abstraction instruction set. Similarly, if the average execution times in each stage is 0.01 time units (with $CV^2 = 10.0$) then we have modeled a low-abstraction instruction set. Modeling each pipeline stage as a queue we get the throughput T_h for high-abstraction instruction sets to be 0.78 and for low-abstraction instruction sets to be 63.45. Therefore, for the high-abstraction instruction set to be more efficient than the low abstraction one, each high-abstraction instruction must correspond to at least 81 $(= 63.45/0.78)$ low-abstraction instructions. We have just found the "crossover point" for the above-mentioned hardware environment.

F. Throughput and Coefficient of Variation

We must emphasize that a high-abstraction instruction set is modeled by high average service times and a high coefficient of variation. A high-abstraction instruction set always has some instructions with low execution times and hence has to be modeled by a high coefficient of variation. Let us now investigate the effect of the variation of CV^2 on the throughput. Using the same two-stage pipeline considered before as the hardware environment we obtained the following results.

For low-abstraction instruction sets throughput reduces as CV increases. This is due to the fact that as CV increases, instructions with higher execution times are executed more often. For high-abstraction instruction sets the throughput reduces as CV increases up to a certain point and then increases with CV. We predict that the increase in throughput is due to the fact that as the CV gets higher and higher then the high-average execution time of each instruction allows instructions with low execution time to be executed. Note that a higher abstraction instruction set has a higher mean execution time and hence there is room for the actual execution time to be low. However, if the mean is already low then there is no room for the actual execution time to become lower as is the case with low-abstraction instruction sets. This observation is extremely critical as we know now that we should select a high-abstraction instruction set such that the CV that minimizes the throughput is avoided.

V. Conclusions

A major accomplishment of this work is the development of the notion of a structural unit that represents program, data or execution structures, as the basic architectural unit of a HLLDCA. This notion enables us to design HLLDCA's by modeling them as a network of queues. The rest of our work dealt with the modeling of HLLDCA's. The analytical models which we developed are design tools useful to an HLLDCA designer. Analytical models estimate the performance of an HLLDCA rapidly and save the designer a lot of effort when he or she starts out a design. To be more specific, we developed an approximation technique that predicts the performance of the execution of a parallel computation on a multiprocessor system. The multiprocessor system can be represented by a network of fork–join G/G/1 queues and the parallel computation by a precedence graph. We saw that the approximation techniques that we employed were indeed very accurate and computationally inexpensive for small systems.

We can generalize our execution time estimation algorithm by using the algorithms in [26]. to estimate the performance of an arbitrary graph structure.

References

[1] G. J. Myers, *Advances in Computer Architecture*. New York: Wiley, 1982.

[2] R. Sridhar, "An automatic microprogram generator for direct execution machines," Ph.D. dissertation, Washington State Univ., Pullman, 1987.

[3] R. A. Marie, "An approximate analytical method for general queueing networks," *IEEE Trans. Softw. Eng.*, vol. SE-5, pp. 530–538, Sept. 1979.

[4] P. Brinch Hansen, "The programming language concurrent pascal," *IEEE Trans. Softw. Eng.*, vol. SE-1, pp. 199–207, June 1975.

[5] C. A. R. Hoare, *Communicating Sequential Processes*. London, U.K.: Prentice-Hall, 1985.

[6] I. C. Pyle, *The Ada Programming Language*. London, U.K.: Prentice-Hall, 1981.

[7] R. S. Chin and S. T. Chanson, "Distributed object-based programming systems," *ACM Comput. Surv.*, vol. 23, pp. 91–124, Mar. 1991.

[8] L. Flatto and S. Hahn, "Two parallel queues created by arrivals with two demands-I," *SIAM J. Appl. Math.*, vol. 44, pp. 1041–1053, Oct. 1984.

[9] M. A. Brun and G. Fajolle, "The distribution of the transaction processing time in a simple fork–join system," *2nd Int. Workshop Math. Comp. Performance Reliability*, Rome, Italy, May 25–29, 1987.

[10] R. Nelson and A. N. Tantawi, "Approximate analysis of fork–join synchronization in parallel queues," *IEEE Trans. Softw. Eng.*, vol. 14, pp. 532–540, Apr. 1988.

[11] F. Baccelli, W. A. Massey, and D. Towsley, "Acyclic fork–join queuing networks," *J. ACM*, vol. 36, pp. 615–642, July 1989.

[12] A. Duda and T. Cxachorski, "Performance evaluation of fork and join primitives," *Acta Inform.*, vol. 24, pp. 525–553, 1987.

[13] P. Heidelberger and K. Trivedi, "Queueing network models for parallel processing with asynchronous tasks," *IEEE Trans. Comput.*, vol. 31, pp. 1099–1109, Nov. 1982.

[14] ——, "Analytical queueing models for programs with internal concurrency," *IEEE Trans. Comput.*, vol. 32, pp. 73–82, Jan. 1983.

[15] S. Balsamo and L. Donatiello, "Approximate performance analysis of parallel processing systems," *Decentralized Systems*. Amsterdam, The Netherlands: Elsevier, 1990.
[16] K. M. Chandy and C. H. Sauer, "Computational algorithms for product form queueing networks," *Commun. ACM*, vol. 23, pp. 573–583, Oct. 1980.
[17] R. Suri, "Robustness of queueing networks," *J. ACM*, July 1983.
[18] K. D. Gordon and L. W. Dowdy, "The impact of certain parameter estimation errors in queueing network models," *Perform. Eval. Rev. 9*, vol. 2, pp. 3–9, 1980.
[19] K. S. Trivedi, *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1982.
[20] A. Thomasian and P. Bay, "Queueing network models for parallel processing of task systems," in *IEEE Proc. 1983 Int. Conf. Parallel Processing*, pp. 421–428.
[21] P. Courtois, *Decomposibility: Queueing and Computer System Applications*. New York: Academic, 1977.
[22] V. W. Mak and S. F. Lundstrom, "Predicting performance of parallel computations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, pp. 257–270, July 1990.
[23] M. Reiser and S. S. Lavenberg, "Mean value analysis of closed multichain queueing networks," *J. ACM*, vol. 23, pp. 573–583, Oct. 1980.
[24] A. Ferscha, "Modeling mappings of parallel programs onto parallel architectures with the PRM-net model," in *Decentralized Systems*. Amsterdam, The Netherlands: Elsevier, 1990, pp. 349–363.
[25] F. Baccelli and Z. Liu, "On the execution of parallel programs on multiprocessor systems—A queuing theory approach," *J. ACM*, vol. 37, pp. 373–414, Apr. 1990.
[26] S. Madala and J. B. Sinclair, "Performance of synchronous parallel algorithms with regular structures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, pp. 105–116, Jan. 1991.
[27] S. N. Kamin, *Programming Languages, An Interpreter Based Approach*. Reading, MA: Addison-Wesley, 1990.

Minimum Entropy Restoration of Star Field Images

Hai-Shan Wu and Joseph Barba

***Abstract*—In this correspondence, we present an algorithm for restoration of star field images by incorporating both the minimum mean square error and the maximum varimax criteria. It is assumed that the point spread function of the distortion system can be well approximated by a Gaussian function. Simulated annealing (SA) is used to implement the optimization procedure. Simulation results for both Gaussian and square point spread functions with heavy additive independent white Gaussian noise are provided. Visual evaluation of the results indicate that the proposed algorithm performs better than the noncausal Wiener filtering method.**

I. Introduction

Image restoration with known point spread function (PSF) has been discussed extensively [2]–[6]. Unfortunately, the point spread functions are usually not available in most practical situations. If the PSF is unknown, the image restoration problem will be rather difficult since it is a kind of blind restoration in which the estimation is made only based on the observed image. Image restorations with unknown PSF has led to many PSF identification approaches [7]–[11]. A parametric blur estimation algorithm using power cepstrum was reported in [7]. The maximum likelihood (ML) methods that determine the parameters which are most likely to produce the observed images were discussed in [8], [9]. More recently, a method of generalized cross-validation (GCV) for blur identification, which minimizes the GCV measure, was described by Reeves and Mersereau [10]. A spectral matching method which requires *a priori* knowledge of the original image spectrum is studied in [11].

Manuscript received April 24, 1994; revised May 14, 1995 and February 5, 1997. This work was supported in part by the National Science Foundation CISE Directorate under Grant CDA-9114481.

H.-S. Wu is with the Department of Pathology, Mount Sinai School of Medicine, New York, NY 10029 USA.

J. Barba is with the Department of Electrical Engineering, City College of New York, New York, NY 10031 USA (e-mail: barba@ee-mail.engr.ccny.cuny.edu).

Publisher Item Identifier S 1083-4419(98)02191-8.

Assuming that the PSF of the distortion system can be well approximated by a Gaussian function, an algorithm is presented in this correspondence for restoration of star field images by incorporating both the minimum mean square error criterion and the maximum varimax criterion used in the MED approach [12]–[15], which is very successful in deconvolution of seismic signals. The term "minimum entropy deconvolution" (MED) was first introduced by Wiggins [12] for a deconvolution scheme which seeks the smallest number of large spike consistent with the seismic data. The approach assumes the deconvolved version has a simple structure, i.e., it is a sparse spike sequence [12]–[15]. This spike sequence is similar, in nature, to the signals of star field images which have sparse spikes representing the stars. An ideal star field image can be considered as a few bright impulses in a black background. Combined with the mean square error criterion between the observed image and the blurred version of the restored image, the MED method is applicable to the restoration of star field images. Unlike other restoration methods which estimate the PSF prior to restoration, the proposed algorithm produces the restored image directly from the observed image.

II. Image Distortion Model

Suppose the image distortion system can be modeled as

$$y(n_1, n_2) = x(n_1, n_2) * h(n_1, n_2) + \xi(n_1, n_2) \tag{1}$$

where $y(n_1, n_2)$ is the observed image, $x(n_1, n_2)$ is the original image, $h(n_1, n_2)$ is the PSF, $\xi(n_1, n_2)$ is the additive noise, and $*$ denotes two-dimensional convolution of two functions defined as

$$x(n_1, n_2) * h(n_1, n_2) = \sum_{m_1=-\infty}^{\infty} \sum_{m_2=-\infty}^{\infty} x(n_1 - m_1, n_2 - m_2) * h(m_1, m_2). \tag{2}$$

What is desired is an estimate of the original image $x(n_1, n_2)$ from the observed image $y(n_1, n_2)$. In this paper, we consider restoration of star field images distorted by blurring systems whose PSF's can be well approximated by Gaussian functions and corrupted by additive Gaussian white noise. The image distortion can be introduced in the process of acquiring the star field image and may result from atmosphere turbulence, out-of-focus lens, or some other unexpected phenomenon which can be modeled by (1). Atmospheric turbulence blurs are stochastic blurs which, for long term exposures, can be approximated by Gaussian function [1], [11]. In a star field image, the stars are represented by high intensity points in dark background sky represented by black or low-intensity pixels. Thus, a star field image usually has sparse high intensity impulses in a black background.