

A Tool to Help Tune where Computation Is Performed

Hyeonsang Eom, *Student Member, IEEE*, and Jeffrey K. Hollingsworth, *Member, IEEE*

Abstract—We introduce a new performance metric, called Load Balancing Factor (LBF), to assist programmers with evaluating different tuning alternatives. The LBF metric differs from traditional performance metrics since it is intended to measure the performance implications of a specific tuning alternative rather than quantifying where time is spent in the current version of the program. A second unique aspect of the metric is that it provides guidance about moving work within a distributed or parallel program rather than reducing it. A variation of the LBF metric can also be used to predict the performance impact of changing the underlying network. The LBF metric is computed incrementally and online during the execution of the program to be tuned. We also present a case study that shows that our metric can accurately predict the actual performance gains for a test suite of six programs.

Index Terms—Parallel and distributed computing, performance prediction, measurement, tools, online analysis.

1 INTRODUCTION

To successfully tune a distributed or parallel program, the cause of a performance bottleneck must be identified, after which a solution can be proposed and implemented. Finally, the tuned program must be remeasured to verify the problem was corrected. Each step in the process is a difficult and time-consuming task. Performance debugging tools exist to help the programmer with these tasks. However, the majority of the work on performance tools has concentrated on bottleneck identification. While this is an important problem, it is just the first step. In this paper, we concentrate on providing guidance with the next step: choosing between alternative tuning strategies.

Once the source of a problem has been located, a proposed change must be identified. Frequently, there are several different strategies to try, such as changing data decomposition, changing the assignment of processes to processors, or even changing the computation or communication resources. However, each of these options might require significant effort to change the program, debug it, and, then, reexecute it. Performance tools need to help the programmer to evaluate the potential impact of different tuning options before changing a single line of code.

There are several ways for a tool to provide information about the potential benefit of tuning options. First, the tool could use a static prediction of the performance of the changed program based on analysis of the source code. However, such an approach suffers from the problem that the prediction ignores dynamic (execution) data that can provide important information about a program's actual behavior. A second approach is to instrument a program to measure its dynamic behavior and then use this data to

make offline predictions about tuning alternatives. This approach could require a significant amount of data to be collected. Instead, we use a third approach that combines the execution of the current version of the program, online measurements of its execution, and algorithms to predict the impact of different tuning options. The idea is to combine the execution of the original program with a simulation of the proposed changes to the program. This technique has been successfully used to simulate changes in computer architectures [23]. Combining the direct execution of the majority of the system with a simulation of the changed parts permits faster execution than simulating the entire program's execution.

There is a trade-off between efficiency and accuracy when predicting the change in execution time due to tuning. Consider, for example, trying to assess the impact of tuning a single procedure's performance. At one extreme, we could generate very accurate results by performing a detailed execution-driven simulation of the proposed modifications to the original program. Each instruction could be simulated and an appropriate time for that instruction recorded. To simulate the impact of tuning, whenever the tuned procedure is executed, simulation time would advance only by the "tuned" time of the procedure. This would produce a very accurate prediction of the improvement possible by tuning the target procedure. However, the speed of this simulation would likely be too slow to provide timely feedback to the programmer. At the other extreme, we could simply profile the target procedure and predict that any time removed from that procedure would directly reduce the execution time of the program. This produces a simple value to compute, but the accuracy suffers due to the fact that the improvement in execution time of a procedure does not necessarily result in a corresponding improvement in the program's execution time due to communication and work done on other processors. Our goal is to combine reasonable performance and accuracy to provide useful feedback to programmers.

• The authors are with the Computer Science Department, University of Maryland, College Park, MD 20742.
E-mail: {hseom, hollings}@cs.umd.edu.

Manuscript received 24 Sept. 1998; revised 10 Nov. 1999; accepted 20 Apr. 2000.

Recommended for acceptance by D. Eager and A.A. Andrews.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 107454.

Unlike sequential programs in a distributed or parallel program, it is possible to tune where a computation is performed in addition to how it is performed. For example, a process in a producer/consumer pipeline may exhibit *data affinity*. A consumer process has data affinity if it consumes a large amount of data and its performance is improved by colocation with its data source. Data can be either static (a disk file) or dynamic (a producer process). Due to either load balancing or data affinity, it might be more productive to move part of the computation from one processor to another rather than reducing its execution time. In this paper, we concentrate on providing answers to “what-if” questions involving changing where computation is performed rather than changing how the result is computed. We present a new metric called Load Balancing Factor, LBF, that provides programmers with feedback about the performance implications of moving computation between processors. LBF can be efficiently computed during the execution of the current version of the program and does not require postmortem processing. In addition, we present a variant of LBF, called Networking Factor (NF), that predicts the performance gains due to changing the underlying communication network.

In this paper, we introduce the LBF and NF metrics and evaluate them for several distributed or parallel programs. Section 2 introduces the LBF metric, describes an implementation of the metric and its communication model, and quantifies its accuracy at predicting changes. Section 3 illustrates using NF, a network variant of LBF, to predict the change in application execution time due to changing the performance of the networking infrastructure. Section 4 describes the limits of LBF and NF in terms of their accuracy and the overhead of the computation. Section 5 surveys related work. Finally, Section 6 summarizes our work and outlines future directions for this research.

2 LOAD BALANCING FACTOR (LBF)

Load Balancing Factor (LBF) is a metric used to predict the performance of an application when changing its workload distribution. The LBF metric addresses the problem of assessing the impact of process migration by predicting the impact of changing the assignment of processes to processors in a distributed or parallel execution environment. Our goal is to compute the potential improvement in execution time if we change the placement. Our technique can also be used to predict the performance of a distributed or parallel program when it is executed on a larger number of nodes.

To assess the potential improvement, we predict the execution time of a program with a virtual placement during an execution on a current one. Our approach is to instrument application processes to forward data about each message passing event to a central monitoring station that simulates the execution of these events under the target configuration.

Since there could be multiple processes contending for a CPU on a node in a target placement, we must select a realistic policy to schedule processes for an accurate prediction. We assume a fair round-robin scheduling policy, where the OS schedules each nonwaiting process

onto a processor for a fixed quantum of time and then switches to the next nonwaiting process. To speed up the computation of the LBF metric, we do not simulate individual quanta. For each interval of time, every non-blocked process gets an equal share of the processor, effectively making the quantum infinitely small.

Before describing our prediction algorithm, we define a few terms used to describe LBF:

Event. An observable operation performed by a process. A process communicates with other processes via messages. Message passing results in *send*, *startRecv*, and *endRecv* events being generated. Message events can be “matched” between processes. For example, a *send* event in one process matches exactly one *endRecv* event in another process.¹

Process Time. A per-process clock that runs when the process is executing on a processor and is not waiting for a message.

Program Activity Graph (PAG). A graph of the events in a single program execution. Nodes in the graph represent events in the program’s execution. Arcs represent the ordering of events within a process or the communication dependencies between processes. Each arc is labeled with the amount of process time between events or communication time for interprocess arcs. The left half of Fig. 1 shows a simple PAG for a parallel program with three processes.

Happen-Before. The transitive partial ordering of events implied by communication operations and the sequence of local events in a process. For local events, one event happened before another event if it occurred earlier in the program trace for that process. For remote events, a *send* event happens before the corresponding *endRecv* event. Formally, happen-before is the set of precedence relationships between events implied by Lamport’s happen before relationship [15].

Critical Path (CP). The longest process or communication time weighted path through a PAG. For an entire program’s execution, the CP represents the execution time of the program as if there were one process per processor.

Process Group. A set of processes that run on a single processor in a predicted (target) configuration.

Group Time. A per-group clock that runs when any process of the group is executing on a processor. It also runs while a message is being passed between two processes of the group whether there is any process of the group executing during the communication time or not.

Group Activity Graph (GAG). A graph of the events in a single program execution. Nodes in the graph represent events in the program’s execution. Arcs represent the ordering of events within a group or the communication dependencies between groups. Local arcs are labeled with the amount of group time between events. The right side of Fig. 1 shows a simple GAG for a parallel program

1. This definition could easily be extended to include other synchronization or communication events such as locks and barriers.

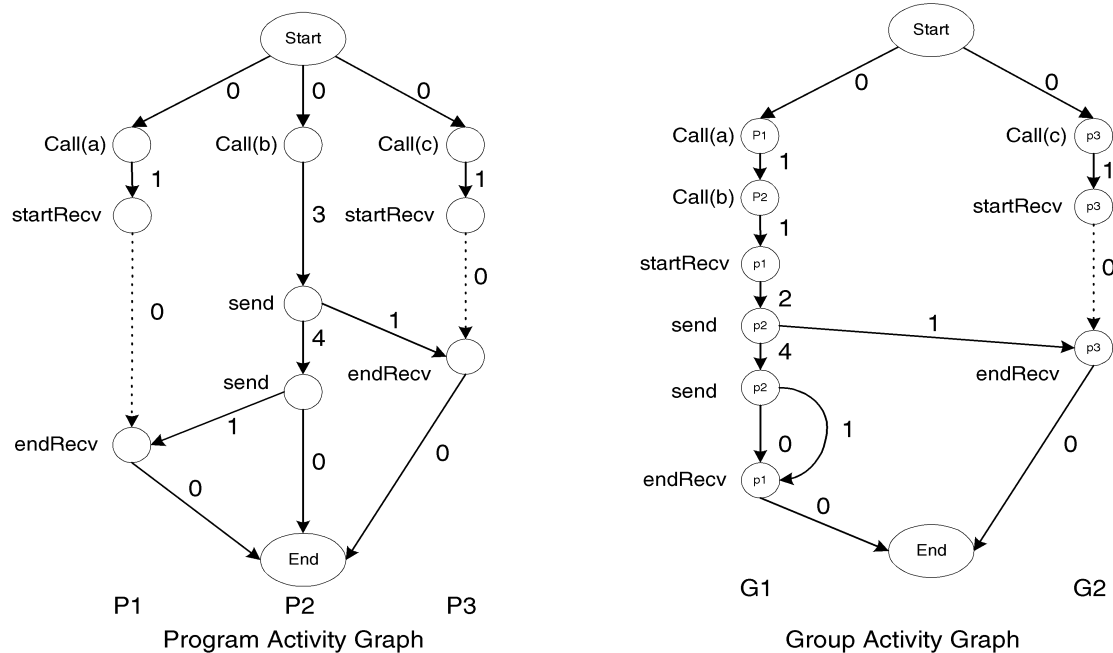


Fig. 1. Transforming a PAG into a GAG.

with three processes and two target groups. A GAG is effectively a PAG with all events from a group collected into a single “virtual” process.

Earliest Possible Time (EPT). The earliest time, measured in group time, an event can occur within a target group. EPT is equivalent to process time when there is only one process in a group.

To compute the execution time of a target configuration, we can construct a Group Activity Graph (GAG) and then compute the length of its longest path. For clarity of presentation, we first introduce the process of converting a PAG to a GAG in a postmortem fashion. We then describe the details of our algorithm that builds the GAG online during application execution.

Given a target process grouping for the execution of a program, the GAG is constructed from the corresponding PAG by combining the PAG components for the processes in a group into a single “process” in the GAG. Each event from the PAG is placed into the GAG in the group time order. The arc between two adjacent events in the same group is labeled with the elapsed group time between them.

Fig. 1 illustrates a PAG and the corresponding GAG. The weights of arcs in the GAG include the effect of the target grouping. The Earliest Process Time (EPT) of the startRecv in Group 1 is 2 because Processes 1 and 2 share a processor up to the startRecv. The EPT of the endRecv in Group 1 is 8 because Process 2 must run on the group processor so that the send event precedes the matching endRecv. The EPTs of the startRecv and the endRecv in Group 2 are all 1s because there is only one process in the group. The endRecvs in the GAG show the two extreme cases of EPT calculation: The EPT of the endRecv in Group 1 is the same as that of the corresponding send event. The EPT of the endRecv in Group 2 is the same as that of the startRecv. The predicted execution time at each message receive is the maximum of

the EPT of the endRecv event and the EPT of the send event plus the message time of flight. Note that there are two arcs between the second send and endRecv in Group 1 because the zero-weighted arc indicates no processing is done between the two events, while the one-weighted arc represents the corresponding message-flight time. LBF is only an approximation of the execution time after migration since we ignore memory contention among processes in a target group. A complete description of the online algorithm is presented in Section 2.1.

An offline algorithm to calculate LBF would build a PAG, convert it to the corresponding GAG, and then compute CP along the longest EPT plus communication time path through the GAG. Since the number of nodes in the PAG is equal to the number of events during the program’s execution, explicit graph construction, conversion, and computation would cause intolerable overhead for long-running programs. Instead, we have developed an online algorithm to compute LBF, building a PAG and converting the GAG incrementally at runtime. Our algorithm permits us to maintain only the part of the GAG that is currently being processed. To incrementally maintain the GAG, we adapt the on-the-fly topological sort algorithm developed by Kimelman and Zernak [14]. Our algorithm simulates the real execution on a target grouping of processes. To compute the predicted execution time of the target configuration during program execution, we use a variation of our online critical path algorithm [12].

Given a target grouping, we must determine the order of events in the grouping to build the GAG incrementally. Like a topological sort, we must choose the next event to process by selecting events such that all events are processed in the order dictated by the happen-before relationship. Events not ordered by the happen-before relationship are ordered based on the round-robin scheduling of a group’s processes onto a processor.

In addition to selecting the next event to add to the GAG, we must also assign the correct weights to its arcs. For intergroup arcs, the communication time supplied in the PAG is used. Computing the weight of the arc between local events is more complicated; the weight is equal to the total amount of processing done by each nonblocked application process between the last event added to the GAG for the group and the current event being processed. The computation is based on the assumption that the clock speeds of the current and target processors are the same; therefore, the weight needs to be scaled accordingly if the target processor clock speed differs from the current processor.

2.1 Algorithm

We now present the details of our algorithm. We describe how to transform a stream of program events arriving from application processes (i.e., a PAG) into a GAG. By calculating the length of the longest path through the resulting GAG, we compute the execution time under the proposed grouping. Events arrive for processing from the application processes and are maintained until they are inserted into the GAG. When events are no longer needed, they are deleted. While an event is being processed, it is in one of four states:

- **Queued.** An event is queued if it has arrived at the monitoring station, but the event immediately before it in the same process has not yet been reported.
- **Current.** A current event is a candidate for processing. There can be at most one current event per process.
- **Pending.** A pending event is an endRecv that is waiting for the corresponding send event to be processed.
- **Reported.** An event is reported when the processing of the event has been completed and is inserted into the Group Activity Graph (GAG). The DAG data structure for a reported event is freed once both its local and remote successors are reported.

Each event arrives from its application process and is processed by the function `EventArrival` (lines 19-44 of Fig. 2). The `EventArrival` procedure inserts the new event into the PAG; the initial state of the event is determined based on the states of its predecessor events. The state of an event is updated in two places: when it arrives and when a predecessor event is reported. An event becomes current when all its predecessors are reported. Since only endRecv events have two predecessors and events from individual processes arrive in FIFO order, only endRecv events can be marked as pending (waiting for nonlocal predecessors to be processed).

The event selected for processing is the earliest current event. To select among multiple current events, we use the function `EarliestEventTime` (lines 14-18 of Fig. 2). The *Earliest Event Time* for an event is the time of an event if it were to be selected as the current event. If the event selected is a nonblocking event, the `groupTime` of the event's group is increased by the total process time used by the runnable events in the group in order to simulate the amount of time it would have executed in the target configuration between

the current and previously reported events in the group (line 32). Otherwise, the `groupTime` for the selected blocking event is advanced by the actual waiting time (line 43). In either case, the `procTime` and `waitTime` of the other current and pending events in the group are also updated to reflect the change of the group time. The `procTime` of the current event for a nonblocked process is decreased by the amount that corresponds to the process' portion of the total process or actual-waiting time consumed. On the other hand, the `waitTime` of the pending events is reduced by the total process or actual-waiting time. The `waitTime` field of the pending events represents the process time consumed by the group since its last event was added into the GAG.

To report an event, we need to know that no other event that causally preceded it remains unreported. If a process is not generating events (i.e., it does not communicate with other processes) for a long period of time, even though it is not waiting for a message, we can't process any current events in other processes. The reason is that the process doesn't have its current event and that the next event to process is globally determined among the current events of all nonwaiting processes. This scenario is possible because events are currently generated only for communication operations in the middle of execution. To prevent this, we use periodic alarms in each application process to create additional keep-alive events. Keep-alive events are treated like normal events and advance the group time of their target group; the difference is that they are discarded rather than being added to the GAG. Therefore, those events do not affect the accuracy of prediction; however, they add overhead, especially for communication-intensive applications, because they cause additional messages to be sent to the central monitoring station.

We illustrate the LBF computation with the execution of a simple application. Fig. 3 shows the attributes of seven events generated from the execution of the application. The order of rows in the table is the same as the event-data arrival at the central monitoring station. The events are generated by three processes in this example. The first digit in the subscript of the event name indicates the id of the process where the corresponding event occurred and the second digit means the arrival (or occurrence) order of the event from the process. The *ProcTime* shown in the last column of the figure represents (interevent) CPU time between the corresponding event and the previous one on the same process. The corresponding space and time diagram is shown in Fig. 4. The diagram shows the advance of time for the occurrence of each event on every process. As event data arrives, LBF is computed for a target configuration where the application is run on two processors with P1 and P2 on one processor and P3 on the other processor.

Fig. 5 illustrates the LBF computation for this example. It shows the changes of attributes of events while they are waiting to be processed at the monitoring station. The attributes are the group time, earliest event time, and wait time for endReceive events. The earliest event time is the time of an event if the event were to be processed next. The time is computed as the current group time

```

1. UpdateState(Event):
2.   IF Event's type is endRecv AND its send event has not been reported
3.     Event.state <- pending
4.   ELSE Event.state <- current
5.   IF Event's type is endRecv AND its send event has been reported AND
6.     Event.remotePred.Cs > Event.localPred.Cr
7.     Event.waitTime += (Event.remotePred.Cs - Event.localPred.Cr)

8. Report(Event):
9.   add Event into GAG
10.  Event.state <- reported
11.  IF (Event.remoteSuc && Event.remoteSuc.state == pending)
12.    UpdateState(Event.remoteSuc)
13.  IF (Event.localSuc) UpdateState(Event.localSuc)

14. EarliestEventTime(Event):
15.  IF Event's type is endRecv
16.    return Event.waitTime + group(Event).time
17.  ELSE
18.    return Event.procTime * |CNER2 events| + group(Event).time

19. EventArrival(Event):
20.  insert Event into PAG
21.  IF (there is no unreported event for Event's Process) UpdateState(Event)
22.  ELSE Event.state <- queued
23.  WHILE (Each Process has a current or pending Event)
24.    neEvent <- CNER Event with the smallest EarliestEventTime(Event)
25.    eEvent <- current endRecv Event with smallest EarliestEventTime(Event)
26.    IF (neEvent AND
27.      (no eEvent OR EarliestEventTime(neEvent) < EarliestEventTime(eEvent)))
28.      FOR EACH (current or pending Event in neEvent's Group)
29.        IF (Event.state == pending)
30.          Event.waitTime -= |CNER Events in Event's Group| * neEvent.procTime
31.        ELSE Event.procTime -= neEvent.procTime
32.        group(neEvent).time += |CNER Events in neEvent's Group| * neEvent.procTime
33.        IF (neEvent is a send event)
34.          neEvent.Cs <- group(neEvent).time
35.        ELSE IF (neEvent is a startRecv event)
36.          neEvent.Cr <- group(neEvent).time
37.        Report(neEvent)
38.      ELSE
39.        FOR EACH (current or pending Event in eEvent's Group)
40.          IF (Event.state == pending)
41.            Event.waitTime -= eEvent.waitTime
42.          ELSE Event.procTime -= eEvent.waitTime / |CNER Events in eEvent's group|
43.          group(eEvent).time += eEvent.waitTime
44.          Report(eEvent)

```

Fig. 2. Pseudocode for LBF. ² CNER (Current Non-End-Receive) events are all current events except endRecv.

plus waiting time for endRecv events and the group time plus a product of remaining ProcTime and the number of (current) nonendRecv events in the group, for events of

Event	Type	ProcTime
E_{11}	startRecv	1
E_{21}	nonendRecv	2
E_{12}	endRecv	0
E_{31}	send	5
E_{22} ³	exit	1
E_{32}	exit	1
E_{13}	exit	2

Fig. 3. Attributes of events. ³ Note that E_{22} arrives after E_{12} and E_{31} in this example. This is possible because the only constraint on the event arrival order is that events arrive in the order they occur on each process.

the other types. Note that this time and the wait time are applicable only for current events (earliest ready-to-process event for each process).

The next event to process is determined by comparing the earliest event time of the current (candidate for processing) events for all processes and selecting an event with the minimum value of the time. However, this selection requires each process to have either a current or pending event. Therefore, no event is processed until event data from all three processes arrive at the central monitoring station. When E_{11} arrives, its ProcTime and earliest event time are initialized. When E_{21} arrives, its time attributes are initialized and the earliest event time of E_{11} is also updated to two (row 3) because there are two non-endRecv events in the group (E_{11} and E_{21}). Similarly, the initial earliest event time of E_{21} is set to four. When E_{12} arrives, its time attributes are initialized. While E_{12} is pending (until its corresponding send event is processed at Step 10), its earliest event time is ∞ .

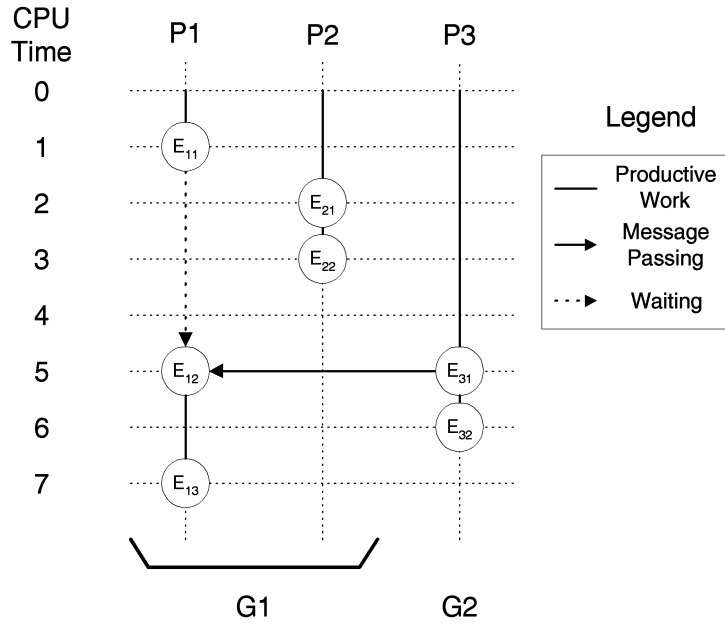


Fig. 4. Space and time diagram for events.

When E_{31} arrives, E_{11} is processed because all processes have an event pending for processing. E_{11} is selected because it has the smallest earliest event time among the current events, E_{11} , E_{21} , and E_{31} . After E_{11} is processed, the group time of G1 becomes two because P1 and P2 run on the same processor for the ProcTime of E_{11} , which is one. Consequently, the remaining ProcTime of E_{21} is reduced to one (row 6). Also, its earliest event time is recomputed as the group time of G1, 2, plus the remaining ProcTime, 1, times the number of nonendRecv events in the group, 1. Next, E_{21} is selected because its earliest event time is smaller than that of E_{12} and E_{31} . As a result of processing E_{21} , the group time of G1 is

advanced to the earliest event time of E_{21} , 3. The wait time of E_{12} is updated to -1 because P2 has run for one unit of CPU time. A negative value of the wait time for a pending (endRecv) event indicates that the group time has been advanced due to the work done by other processes in the same group while waiting for an event (i.e., the send to enable the receive). Therefore, the amount is subtracted from the total waiting time when the corresponding send event is processed. At this point, the LBF computation cannot proceed because there are no unreported events for P2.

When E_{22} arrives, its time attributes are initialized (row 8) and it is selected to be reported (row 9) because

Arriving Event	Initialized or Processed	Group Time		Earliest Time (Remaining ProcTime) {Wait Time}								Reported	Row #
				P1			P2			P3			
		G1	G2	E_{11}	E_{12}	E_{13}	E_{21}	E_{22}	E_{23}	E_{31}	E_{32}		
Initial	-	0	0	-	-	-	-	-	-	-	-	-	1
E_{11}	E_{11}	0	0	1 (1)	-	-	-	-	-	-	-	-	2
E_{21}	E_{21}	0	0	2 (1)	-	-	4 (2)	-	-	-	-	-	3
E_{12}	E_{12}	0	0	2 (1)	∞ (0) {0}	-	4 (2)	-	-	-	-	-	4
E_{31}	E_{31}	0	0	2 (1)	∞ (0) {0}	-	4 (2)	-	-	5 (5)	-	-	5
	E_{11}	2	0	-	∞ (0) {0}	-	3 (1)	-	-	5 (5)	-	E_{11}	6
	E_{21}	3	0	-	∞ (0) {-1}	-	-	-	-	5 (5)	-	E_{21}	7
E_{22}	E_{22}	3	0	-	∞ (0) {-1}	-	-	4 (1)	-	5 (5)	-	-	8
	E_{22}	4	0	-	∞ (0) {-2}	-	-	exited	-	5 (5)	-	E_{22}	9
	E_{31}	4	5	-	5 (0) {1}	-	-	-	-	-	-	E_{31}	10
E_{32}	E_{32}	4	5	-	5 (0) {1}	-	-	-	-	-	6 (1)	-	11
	E_{12}	5	5	-	-	-	-	-	-	-	6 (1)	E_{12}	12
E_{13}	E_{13}	5	5	-	-	7 (2)	-	-	-	-	6 (1)	-	13
	E_{32}	5	6	-	-	7 (2)	-	-	-	exited	-	E_{32}	14
	E_{13}	7	6	-	-	exited	-	-	-	-	-	E_{13}	15

Fig. 5. An illustration of the LBF computation.

PVM Options		Remote One-Way Communication Time (ms) for Message Size (KB) of						
Direct Route	In Place	0	2	8	32	128	512	2,048
√	√	0.62	0.90	1.37	104.45	104.48	106.55	309.88
√		0.61	0.89	1.85	105.30	10,374.75	>> 30,000 [†]	>> 30,000
	√	1.37	1.82	4.94	14.56	53.70	207.02	828.40
		1.38	1.75	4.51	14.37	52.99	250.57	992.64

Fig. 6. A communication time lookup table. [†] We guessed that the communication times for 512 and 2,048 KB would be greater than 30 seconds in this case because the measured time for 256 KB was about 30 seconds. We could not measure the times; however, it did not affect our study because that option combination was not used due to its instability.

its earliest event time is smaller than that of E_{12} and E_{31} . The group time of G1 is advanced to four and the waiting time of E_{12} is updated to -2, indicating that P2 has again run for one unit of CPU time while E_{12} is pending. The P2 column of row 9 is marked as “exited” because the event is the exit event of the process. Therefore, events of P2 are no longer considered in selecting the next event. Next, E_{31} is chosen as the next event to process because it is the only candidate for processing since E_{12} is still pending. The group time of G2 is advanced to five after E_{31} is processed and reported (row 10). Since processing a message send allows the receive to complete, E_{12} becomes ready to be processed. Its wait time is changed to one, which is the difference in group time between G2 and G1 (row 10), and its earliest event time is updated to five (the group time, four, plus the wait time). After E_{32} arrives, E_{12} is selected and, thus, the group time of G1 becomes five (row 12), the earliest event time of E_{12} .

Finally, when E_{13} arrives, E_{32} is selected. The group time of G2 is advanced to six after the event is processed (row 14). Since E_{13} is the current event of the only remaining process, P1, it is selected (row 15). Processing of the event advances the group time of G1 to seven, which is the final LBF metric value. Note that the LBF value is different from either summing up the CPU time of P1 and P2 (6) or the CPU time of P3 (6) because it includes the effect of synchronization activity and the interleaving of events between P1 and P2 on the G1 processor.

When multiple processes are scheduled on a target processor, they are scheduled in a round-robin fashion, as shown in the above example. Each time we select an event among the nonwaiting processes, we “schedule” all nonwaiting processes on the group processor of the selected event for an amount of time equal to the remaining ProcTime of the event. For instance, in the above example, when E_{11} is processed, the remaining ProcTime of E_{21} is also decreased by one. This technique guarantees fairness and accommodates processes even if they have different execution times.

2.2 Communication Model

For accurate prediction, it is necessary to integrate communication cost into the computation of the predicted time. Communication cost is due to protocol processing time at the end-points and the time of flight of the message. Since protocol processing is local to a single process, it is easy to measure directly. However, due to problems with clock synchronization, it is generally impossible to accurately measure the time of flight of a message. As a result of this difficulty, we use a lookup table based on the number of message bytes transferred and whether the message is local

(same processor) or remote. The values for this table are determined offline (prior to application execution) by measuring one half of the round trip times for messages of varying lengths. We use linear interpolation between the two nearest measured values to produce a communication estimate for message sizes that were not measured. Our approach handles finite link bandwidth, but ignores network cross-traffic contention.

Fig. 6 shows a lookup table consisting of measured remote one-way communication times for the LBF computation. To construct the table, we used PVM [6] with and without its direct route (DR) and in-place (IP) options enabled. DR allows setting up task-to-task communication links while IP leaves data items in place until a send routine is called to copy them out of the user’s memory. We obtained the communication time as half of an average of 100 round trips between two 66.7 MHz IBM RS/6000’s connected via a 320 Mbps IBM SP switched network. Note that only part of the table that was used in the LBF computation is shown.

To scale communication message sizes beyond the measured range, data points can be extrapolated via curve fitting. To refine communication performance to include the number of processors involved in communication as well as message size, we can collectively measure one-to-many or many-to-one communication time in a similar way to Abandah and Davidson [1]. However, it is hard to compute accurate point-to-point communication time from collective time due to nonpredictability of packet interleaving. To derive an accurate point-to-point communication model as a function of the number of other processors communicating with a node, individual messages need to be timed. However, this would require a light-weight timer with very fine granularity. Instead, we choose to ignore communication delay caused by other processors’ messages because the delay effect is not significant for most of current systems.

2.3 Experimental Validation of LBF

We implemented LBF as an extension to the Paradyn Parallel Performance Measurement Tools [19]. Using Paradyn provided an easy way to implement the algorithm since it already included support for instrumentation of a running program and periodic sampling callbacks. We tested LBF by running a collection of application programs. The programs consisted of a Synthetic Parallel Application (SPA), a program to solve the Traveling-Salesman Problem (TSP), and a selection of the Numerical Aerodynamic Simulation (NAS) benchmark programs [3]. The NAS applications are an embarrassingly parallel program (EP), a parallel FFT computation (FT), an integer sort program (IS), and a multigrid solver (MG). The data size used for the

Application Target Config.	Meas. Time	From Actual Config.			From Actual Config.			From Actual Config.		
		Pred.	Δ	(%)	Pred.	Δ	(%)	Pred.	Δ	(%)
SPA			4/4				4/2			4/1
4/4	158.7	159.0	-0.3	(-0.2%)	158.2	0.5	(0.3%)	158.5	0.2	(0.1%)
4/1	240.2	235.5	4.7	(2.0%)	235.1	5.1	(2.1%)	236.2	4.0	(1.7%)
TSP			4/4				4/2			4/1
4/4	85.6	85.5	0.1	(0.1%)	85.8	-0.2	(-0.2%)	85.9	-0.3	(-0.4%)
4/1	199.2	197.1	2.1	(1.1%)	197.7	1.5	(0.8%)	198.9	0.3	(0.2%)
EP (class A)			16/16				16/8			16/4
16/16	258.2	255.6	2.6	(1.0%)	260.7	-2.5	(-1.0%)	267.4	-9.2	(-3.6%)
FT (class A)			16/16				16/8			16/4
16/16	140.9	139.2	1.7	(1.2%)	140.0	0.9	(0.6%)	144.0	-3.1	(-2.2%)
IS (class A)			16/16				16/8			16/4
16/16	271.2	253.3	17.9	(6.6%)	254.7	16.5	(6.0%)	255.0	16.2	(6.0%)
MG (class A)⁵			16/16				16/8			16/4
16/16	172.8	166.0	6.8	(4.0%)	168.5	4.3	(2.5%)	186.7	-13.9	(-8.0%)

Fig. 7. Measured and predicted time (LBF value). For each application, we show one or two target configurations and the second column shows the measured time running on this target configuration. The rest of the table shows the execution times predicted by using the LBF methodology when run under two different actual configurations. ⁵ The PVM option direct route was used for this application.

NAS applications was “class A,” which is intended for execution on a network of workstations. All programs were run on an IBM SP-2 and used PVM [6] for communication. Neither the direct-route option nor the in-place one was used except for MG, where both options were enabled. We measured the execution times of the programs and compared them with the predicted times of LBF. We also report the overhead of computing LBF.

All measurements were conducted on dedicated SP-2 nodes and, so, there was no interference with other applications. The metric computation is not influenced by the overhead of other applications running on the same processors as the target application because the prediction is based only on the measured CPU times of the processes in the application and table-driven communication time. However, the load on the system influences the measured timing of the actual configurations.

The summary of the measured and predicted execution times (in seconds) is shown in Fig. 7. We use N/M to describe a target or actual configuration, where N is the number of processes and M is the number of nodes. For example, the number, 158.5, in the 4/4 row for SPA and prediction column for 4/1 is the execution time predicted for the configuration where four processes run on four processors based on profiling in the configuration that all the four processes run on a single processor. For each target configuration, we ran the program in two actual configurations: one identical to the target configuration and the other with no more than half of the nodes of the target configuration. By predicting the performance of a target configuration that was identical to the running configuration, we were able to evaluate how well our communication prediction information worked. The results show that, in all cases, the predicted values are within 8 percent of the actual execution times. The error of prediction for MG from 16/4 to 16/16 is the highest (8 percent) because the execution in 16/4 is prolonged (up to 3,500 secs) due to factors such as paging and context switching. Part of these effects are included in the CPU time and, thereby, in the predicted time. However, an 8 percent error is quite reasonable for predicting the performance of an application

on a machine configuration four times larger than the actual machine used.

We also measured the overhead of computing the LBF metric. To do this, we ran the same six applications with and without computing LBF. The resulting overhead, shown in Fig. 8, represents the extra time required to run the application when computing the LBF metric. For most applications and configurations, the overhead to compute the LBF metric is under 5 percent. However, for the IS application, the overhead is 7.4 percent. We investigated the source of this relatively high overhead and determined that it was caused mainly by the overhead of running the application program with the Paradyn performance tool. In Section 4, we will provide in-depth discussion on the potential sources of overhead and inaccuracy that cause the variations in prediction error and overhead shown in this section.

We mainly targeted one process per node application in our experiments because parallel machines, our primary target for this work, are normally used in this configuration. When multiple processes are assigned to a node, there could be excessive overhead introduced due to a shortage of resources. For those interested in such configurations, we also tried out the 16/8 and HPS configuration for the

Application Target Config.	Msgs	Msg Bytes	Time		Overhead	
			W/o Inst	With Inst	Sec.	%
SPA						
4/4	56	248	158.7	164.2	5.5	3.5%
4/1	56	248	240.2	247.0	6.8	2.8%
TSP						
4/4	6	2.3K	85.6	88.6	3.0	3.5%
4/1	6	2.3K	199.2	203.6	4.4	2.2%
EP (class A)						
16/16	45	1.8K	258.2	268.8	10.6	4.1%
FT (class A)						
16/16	3,480	1.8G	140.9	146.7	5.8	4.1%
IS (class A)						
16/16	7,725	670.5M	271.2	291.2	20.0	7.4%
MG (class A)						
16/16	3,396	400.2M	172.8	178.7	5.9	3.4%

Fig. 8. Overhead of computing LBF.

Application	HPS		Ethernet		Error	
	Meas.	Meas.	Pred.			
EP (class A)	258.2	257.4	262.3	-4.1	-1.6%	
FT (class A)	140.9	4134.1	135.3	5.6	4.0%	
IS (class A)	271.2	2686.7	251.1	20.1	7.4%	
MG (class A)	172.8	495.0	174.0	-1.2	-0.7%	

Fig. 9. Measured and predicted time (NF value).

NAS applications and found that the errors of the prediction from the same actual configuration for EP and MG were less than 2 percent of the actual execution time. However, the errors for FT and IS were 37.8 and 20.3 percent, respectively. The errors occurred mainly as artifacts of multiple processes being run on the system. We confirmed this by checking that the errors of prediction from 8/8 to 8/8 for FT and IS were 13.3 and 8.7 percent, respectively, even though all execution characteristics (such as total communication volume and allocated memory size) for the 8-process case were the same as those for the 16-process case. Also, the difference between the predicted and measured time was partly due to the fact that paging time was included in the actual execution time while it was not contained in the LBF predicted time. We will provide in-depth discussion on the limitation of the metric in Section 4. We didn't perform further experiments for any other such configurations because of their lack of use in our target environment.

3 NETWORKING FACTOR (NF)

Networking Factor addresses the problem of assessing the impact of a network upgrade by predicting the effect of changing a communication network in a distributed or parallel execution environment. Our goal is to compute the potential improvement in execution time if we change the network. The algorithm can also be used to simulate the performance characteristics of long haul networks when the application is run on a local network. Similarly to LBF, we predict the execution time of a program with a virtual network to assess the potential improvement of using the network rather than the currently available network. To validate the NF metric, we compared the execution times of the programs with the predicted times of NF.

To compute NF, we use the same algorithm used for LBF, substituting the communication cost lookup table of a target (predicted) network for the one of the current network. Since we had access to both networks used in our study, we constructed the table by measuring each network. However, if we wished to evaluate a proposed network, we could simply create an appropriate table based on its expected performance. The overhead of computing NF is identical to that of computing LBF.

We implemented NF as a variation of LBF by using the communication cost lookup table for the target network rather than the one for the current network. We tested NF by running the same subset of the NAS benchmarks used to evaluate LBF. Again, we compared the execution times of the programs running on the real network with the predicted times when running on a different network. The summary of the measured and predicted execution times

Application Target Config., Network	Meas. Time	From Actual Config., Network		
		Pred.	Δ	(%)
EP (class A) 16/16, HPS	258.2	16/8, Ethernet		
		259.9	-1.7	-0.7%
FT (class A) 16/16, HPS	140.9	16/8, Ethernet		
		136.5	4.4	3.1%
IS (class A) 16/16, HPS	271.2	16/8, Ethernet		
		254.4	16.8	6.2%
MG (class A) 16/16, HPS	172.8	16/8, Ethernet		
		174.1	-1.3	-0.7%

Fig. 10. Comparison of measured and predicted time (NF value).

(in seconds) is shown in Fig. 9. For each application, the measured performance is shown for two networks: High Performance Switch (HPS) and a traditional Ethernet. The high performance switch is a 320Mbps switched network and the Ethernet is a bus-based 10Mbps network. We also implemented and tested a combination of LBF and NF by using the target configuration and network communication cost lookup table at the same time. The validation is performed in the same manner as that of NF and its summary is shown in Fig. 10.

The results of running four of the NAS applications with the NF metric are shown in Fig. 9. For each application, the second column shows the measured running time of the applications using the HPS, the third column the measured running time using Ethernet, and the fourth column the predicted running time using the HPS when we were running on Ethernet. The last two columns show the error in the prediction relative to the measured HPS execution time. For the MG application, we were able to predict the execution time on the HPS to within 1 percent even though the measured running time on Ethernet was over twice as long. Likewise, for IS, we were able to predict the running time to within 8 percent when our target and actual configurations had running times that differed by almost a factor of 10. Finally, for FT, our prediction was within 4 percent and the running time was 30 times slower than the target configuration.

The results of running four of the NAS applications with a combination of the LBF and NF metrics are shown in Fig. 10. It shows that, in all cases, the predicted values are within 7 percent of the actual execution times.

4 LIMITS OF LBF AND NF

The accuracy of the LBF and NF metrics and the overhead of the metric computation are affected by several factors. First, overhead is incurred mainly by the execution of instrumented code and the transmission of event data. Also, periodic alarms and Paradyn daemon processes that share the processor with the application processes can slow down the application. Overhead is divided into two parts: static and dynamic overheads.

Static overhead is introduced by running applications under the Paradyn tools without enabling any data collection. This Paradyn-specific overhead accounts for most of the overhead seen. For example, running IS using Paradyn without any events being generated accounts for 87.5 percent of its overall overhead or 7.4 percent of its

execution time. Recall that the overhead of IS was the largest of any of our applications. The Paradyn runtime library introduces this overhead because the data collection routine periodically runs even though there are no events being generated. However, this overhead is not regular and predictable because of an interaction with message passing by the application. When a profiling alarm expires (SIGALRM), Paradyn causes an alarm handler in the application to be executed. If large messages are being passed, then it is more likely that the alarm expires in the middle of the execution of a system call. In this case, the system call gets aborted. When the execution of the Paradyn instrumentation code in the application finishes, the system call is restarted. Therefore, the larger the size of messages, the more static overhead is introduced.

On the other hand, dynamic overhead is caused by an instrumentation of events and, therefore, is proportional to the number of events. The online generation and transmission of event traces can result in their interference with the application execution. This overhead illustrates the standard instrumentation problem: Overhead is inversely proportional to the time between events. However, the total amount of dynamic overhead was small, both absolutely and relatively, compared to the corresponding static overhead in our experiments. For instance, in the above-mentioned IS case, the dynamic overhead was 0.32 msec per message and totaled less than 1 percent of the total measured execution time.

The accuracy of prediction is influenced primarily by the fact that the predicted time for a target configuration is computed based on (online) CPU time measurements and that offline measurements used in predicting execution time are made in a configuration other than the target one. There can be a gap between the predicted and measured execution time because CPU time measurements are used in the LBF computation while execution time is measured using a wall clock timer. Due to the use of different timers, the effects of some system activities, such as paging, that are included in the measured time are not contained in the predicted time. For example, we got 2,232.2 and 366.6 seconds (-83.6 percent error) as measured and predicted times, respectively, for running MG with four processes on an RS/6000 having 64 MB memory. The measured time is much larger than the predicted time primarily because execution blocking time due to paging is included in the measured time while it is not in the predicted time. Although ignoring paging seems like a significant limitation, it is not a significant problem for most systems because memory or inputs are sized to accommodate applications with little or no paging.

The use of table-driven communication costs can introduce errors in our prediction because the offline measurements are made in an environment somewhat different than the actual execution environment. Since communication time is measured using a point-to-point ping-pong benchmark, there can be some error introduced if there is contention inside the network due to messages sharing internal links in the network. However, we currently ignore the prediction error introduced by using point-to-point communication time based on the fact that most networks

are switched and on the assumption that the network capacity is large enough to handle communication involving multiple nodes as in point-to-point communication.

However, alternative communication models can be used to capture the effect of contention or congestion. We could easily plug in a more sophisticated model because the central monitoring station has a global view and determines the total ordering of events. For example, to model contention on a shared network, we can add a different queuing structure to the central monitoring station. To approximate bus contention, a pseudoprocess can be created for the bus-based network. All messages are first sent to the bus pseudoprocess and then they are sent out to their destinations. The bus pseudoprocess would process events at a rate equal to Ethernet speed. For increased fidelity, the pseudoprocess could examine the attributes of events as they arrive and determine if a collision would occur and simulate retransmission times.

Another error source is periodic system activities that consume CPU time during application execution. Since we use measured CPU time of the applications themselves, the duration of system activities is not included. On our SP, system activities include ATM service daemons, NFS client daemons, schedulers, and network time protocol daemons. By using the AIX trace utility, we figured out that the extra system activities account for up to 3.7 percent of the total (elapsed) execution time.

Fundamentally, the accuracy of prediction based on execution depends on the similarity of execution behavior between the current and target runs. For accurate prediction, inputs used to do prediction need to be representative of how the application is run in a production environment.

5 RELATED WORK

There are two areas that are closely related to our online "what-if" computation: performance measurement tools and performance prediction tools. Performance measurement tools quantify the behavior of an actual program execution and allocate time to specific operations or program components. Performance prediction uses a model or simulation to predict the execution time of an algorithm or program.

There are three major types of performance measurement tools: profilers, visualizations, and search tools. Profile metrics [2], [8], [18], [25] associate a value with each component of a distributed or parallel application (frequently procedures) and are presented as sorted tables. Visualizations [10], [16], [17], [21], [26] explain application performance using pictures. Search tools [13], [20], [24] help users to manage performance data information overload by treating the problem of finding a performance bottleneck as a search problem. However, all of these tools focus on the measurement and analysis of a specific program for a single execution. One type of tool that permits programmers to evaluate alternatives is application steering [9], [22]. Application steering permits programmers to change selected aspects of their program while it is in execution. This technique can be very effective in tuning program parameters, but is, by necessity, limited in the type of data decomposition and

algorithmic changes that can be accommodated within the currently running executable image. Complex algorithmic changes require rewriting part of the program.

Performance predictions can be based either on extrapolations of executions of the program in a controlled environment or on stochastic models derived from static program analysis. Lost Cycles Analysis [5] predicts performance at different operating points by running a controlled set of experiments that vary an orthogonal set of parameters and record the resulting execution time. However, this technique requires implementations of the different tuning options to be available for execution. Static prediction [4], [7] uses modeling languages or source code analysis to predict the execution time of a program. By necessity, this technique ignores many details about the interactions between the application, system software, and hardware. An alternative strategy of static prediction is to use a stochastic queuing network model such as [11].

6 CONCLUSIONS AND FUTURE DIRECTIONS

We have presented a new performance metric that provides insights into how proposed tuning strategies will improve an application's execution time. We have shown, for a collection of six parallel programs that our metric is able to accurately predict the execution time of a modified configuration or different workload distribution. The prediction accuracy comes partly from the fact our metric considers dependencies between operations, which cannot be captured by a back-of-the-envelope approach. The metric is computed at runtime with little interference with the execution.

Although LBF is useful for programmers in its current form, there are many directions to expand this research. First, LBF doesn't provide any guidance about what tuning options of a program to evaluate. In most cases, there are multiple tuning alternatives to consider. A future direction is to investigate automatic selection of candidate tuning alternatives. Second, the automated selection of candidate configurations combined with LBF provides a basis for dynamic program adaptation where we automatically change programs during execution based on observed behavior to enhance their performance. Third, to permit automatic adaptation, we will need to consider the dynamic migration between configurations and incorporate migration cost into our metric.

ACKNOWLEDGMENTS

This work was supported in part by US National Science Foundation award ASC-9703212, US Department of Energy Grant DE-FG02-93ER25176, and US National Institute of Standards and Technology CRA award 70-NANB-5H0055.

REFERENCES

[1] G. Abandah and E. Davidson, "Modeling the Communication Time Performance of the IBM SP2," *Proc. 10th Int'l Parallel Processing Symp.*, pp. 249-257, Apr. 1996.

[2] T.E. Anderson and E.D. Lazowska, "Quartz: A Tool for Tuning Parallel Program Performance," *Proc. 1990 SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp. 115-125, May 1990.

[3] D. Bailey et al., "The NAS Parallel Benchmarks," RNR Technical Report RNR-94-007, Mar. 1994.

[4] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, "A Static Performance Estimator to Guide Data Partitioning Decisions," *Proc. 1991 ACM SIGPLAN Symp. Principals and Practice of Parallel Programming*, pp. 213-223, Apr. 1991.

[5] M.E. Crovella and T.J. LeBlanc, "Parallel Performance Prediction Using Lost Cycles," *Proc. Supercomputing '94*, pp. 600-609, Nov. 1994.

[6] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine*. Cambridge, Mass.: MIT Press, 1994.

[7] A.J.C.v. Gemund, "Performance Prediction of Parallel Processing Systems: The PAMELA Methodology," *Proc. Int'l Conf. Supercomputing (ICS)*, pp. 318-327, July 1993.

[8] A.J. Goldberg and J.L. Hennessy, "Performance Debugging Shared Memory Multiprocessor Programs with MTOOL," *Proc. Supercomputing '91*, pp. 481-490, Nov. 1991.

[9] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, J. Vetter, and N. Mallavurupu, "Falcon: On-Line Monitoring and Steering of Large-Scale Parallel Programs," *Proc. Frontiers '95*, pp. 422-429, Feb. 1995.

[10] M.T. Heath and J.A. Etheridge, "Visualizing Performance of Parallel Programs," *IEEE Software*, vol. 8, no. 5, pp. 28-39, 1991.

[11] P. Heidelberger and K. Trivedi, "Queueing Network Models for Parallel Processing with Asynchronous Tasks," *IEEE Trans. Computers*, vol. 31, no. 11, pp. 1099-1109, Nov. 1982.

[12] J.K. Hollingsworth, "Critical Path Profiling of Message Passing and Shared-Memory Programs," *IEEE Trans. Parallel and Distributed Systems*, pp. 1029-1040, Oct. 1998.

[13] J.K. Hollingsworth and B.P. Miller, "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems," *Proc. Seventh ACM Int'l Conf. Supercomputing*, pp. 185-194, July 1993.

[14] D. Kimelman and D. Zernik, "On-the-Fly Topological Sort—A Basis for Interactive Debugging and Live Visualization of Parallel Programs," *Proc. ACM/ONR Workshop Parallel and Distributed Debugging*, vol. 1, pp. 12-20, May 1996.

[15] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, vol. 21, no. 7, pp. 558-564, 1978.

[16] F. Lange, R. Kroger, and M. Gergeleit, "JEWEL: Design and Implementation of a Distributed Measurement System," *IEEE Trans. Parallel and Distributed Systems*, vol. 3, no. 6, pp. 657-671, June 1992.

[17] T. Lehr, Z. Segall, D.F. Vrsalovic, E. Caplan, A.L. Chung, and C.E. Fineman, "Visualizing Performance Debugging," *Computer*, vol. 21, no. 10, pp. 38-51, Oct. 1989.

[18] M. Martonosi, A. Gupta, and T. Anderson, "MemSpy: Analyzing Memory System Bottlenecks in Programs," *Proc. 1992 SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp. 1-12, June 1992.

[19] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn Parallel Performance Measurement Tools," *Computer*, vol. 28, no. 11, pp. 37-46, Nov. 1995.

[20] S.E. Perl and W.E. Weihl, "Performance Assertion Checking," *Proc. 14th ACM Symp. Operating Systems Principles*, pp. 134-145, Dec. 1993.

[21] D.A. Reed, R.A. Aydt, R.J. Noe, P.C. Roth, K.A. Shields, B.W. Schwartz, and L.F. Tavera, "Scalable Performance Analysis: The Pablo Performance Analysis Environment," *Proc. Scalable Parallel Libraries Conf.*, A. Skjellum, ed., 1993.

[22] D.A. Reed, K.A. Shields, W.H. Scullin, L.F. Tavera, and C.L. Ellford, "Virtual Reality and Parallel Systems Performance Analysis," *Computer*, vol. 28, no. 11, pp. 57-68, Nov. 1995.

[23] S.K. Reinhardt, J.R. Larus, and D.A. Wood, "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers," *Proc. ACM SIGMETRICS Conf.*, pp. 46-60, May 1993.

[24] W. Williams, T. Hoel, and D. Pase, "The MPP Apprentice Performance Tool: Delivering the Performance of the Cray T3D," *Programming Environments for Massively Parallel Distributed Systems*, 1994.

- [25] C.-Q. Yang and B.P. Miller, "Critical Path Analysis for the Execution of Parallel and Distributed Programs," *Proc. Eighth Int'l Conf. Distributed Computing Systems*, pp. 366-375, June 1988.
- [26] D. Zernik and L. Rudolph, "Animating Work and Time for Debugging Parallel Programs Foundation and Experience," *Proc. 1991 ACM/ONR Workshop Parallel and Distributed Debugging*, pp. 46-56, May 1991.



Hyeonsang Eom received the MS degree in computer science from the University of Maryland, College Park, in 1996. He earned the BS degree in computer science and statistics from Seoul National University, Korea, in 1992. He is currently a PhD student in the Computer Science Department at the University of Maryland, College Park. His research interests include performance tuning and prediction, high performance systems, I/O-intensive applications, information dynamics, and computer networks. He is a student member of the IEEE, the IEEE Computer Society, and the ACM.



Jeffrey K. Hollingsworth earned the BS degree in electrical engineering from the University of California at Berkeley in 1988. He received the PhD and MS degrees in computer science from the University of Wisconsin in 1994 and 1990, respectively. He is an associate professor in the Computer Science Department at the University of Maryland, College Park, and affiliated with the Department of Electrical Engineering and the University of Maryland Institute for Advanced Computer Studies. His research interests include instrumentation and measurement tools, resource aware computing, high performance distributed computing, and computer networks. His current projects include the dyninst runtime binary editing tool and harmony—a system for building adaptable, resource-aware programs. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

▷ IEEE Computer Society publications cited in this article can be found in our Digital Library at <http://computer.org/publications/dlib>.