A Transparent Monitoring Tool for Shared-Memory Multiprocessors

David F. Robinson, Betty H.C. Cheng, and Richard J. Enbody

Department of Computer Science
Michigan State University
East Lansing, Michigan 48824

Abstract
Monitoring and debugging of parallel programs is complicated by race conditions, which can cause software monitoring to alter program behavior. In order to avoid these unwanted modifications of program execution, we present a flexible scheme for transparently monitoring parallel programs in a shared-memory environment. In order to achieve transparency, the monitor observes causal relations between events in different threads of execution, and intervenes when an impending event would change the order of occurrence of causally related events, as compared to unmonitored execution of the same program. In this paper, constructs used to support this monitoring scheme are developed, including mechanisms to deal with unsynchronized and coarse-grained clocks. A monitor prototype for the BBN Butterfly, implemented to explore the utility of the proposed monitoring scheme, is described. Finally, preliminary performance results produced by the prototype are presented and discussed.

1 Introduction
As the use of parallel computing increases, the complexity of writing, debugging, and tuning parallel programs has become a major obstacle to developing efficient software [1]. In addition to the types of software errors that are present in sequential programs, parallel programs may also contain logic errors and performance degradation caused by the interaction of multiple threads of execution [2]. Advanced monitoring techniques are needed to identify the presence of performance problems and to efficiently locate logic errors and causes of poor performance, within the context of multiple, communicating threads.

In order to monitor the execution of a parallel program in the absence of extensive hardware support [3, 4], software monitoring techniques, which work through the introduction of additional instructions into the target program, are required [5, 6]. These additional instructions suspend execution of the target program at various predetermined points, either to monitor the program state or to provide breakpoint facilities. After the monitor activity is complete, normal program execution is resumed (assuming the user did not modify the program or issue a halt instruction at the breakpoint).

Processor events that occur at a particular time with respect to the start of a program may occur at a different time due to the presence of software monitoring. These perturbations of program execution timing are referred to as probe effects [7]. In sequential programs, probe effects do not generally create adverse side-effects since there is only one thread of execution, which can be arbitrarily suspended and resumed without altering the eventual program outcome.

A software monitor whose purpose is to gather debugging data on a target parallel program will add arbitrary delays to the various threads of execution, thus disturbing the relative execution speed among these threads. Two parallel program events from two different threads of execution that occur in a particular order may thus be made to occur in a different order by the introduction of a software monitor [6, 7, 8, 9].

A race condition is a point at which a value read from or written to shared memory is affected by relative processor speeds. For example, when two or more processors write a value to a single shared memory object, the final value of that object is dependent on the order in which the processors performed their respective writes. As another example of a race condition, when one processor writes to a memory object and another processor reads from the same object, the value read depends on the relative order in which the read and write occurred.

Every race condition has a margin of victory, that is, the amount of time by which the "winner" of the race leads the loser. During monitoring, if the probe effect delays the winner more than the loser by an amount greater than the margin of victory, then the outcome of the race will be changed. Since this change will modify the computations performed by at least one thread, it may also lead to changes in the results produced by the program. In fact, a program that would otherwise exhibit incorrect behavior may run correctly while being monitored, making it difficult or even impossible to use the monitor to find the source of the incorrect behavior.

Designers of parallel debugging tools often try to limit the effects of monitoring on program behavior by limiting monitor intrusion during program execution [1, 7, 8, 10]. By striving to minimize the execution time of monitor code inserted in the program execution stream, this approach limits both the amount and complexity of information that can be gathered...
at run time. However, the occurrence of complex program events and states cannot usually be determined quickly, and breakpoints are, of course, incompatible with short monitor delays. Even when monitor intrusions are kept to a minimum, there may still be sufficient disturbance of target program execution to affect the outcome of race conditions, along with the resulting possibility of a change in program behavior [1, 11, 12].

Cai and Turner [9] present a debugging method for distributed memory machines that makes use of logical clocks to achieve monitor transparency. However, they do not give a method of dealing with clock issues such as granularity and synchronization, and their method is incompatible with shared-memory architectures.

We describe a software monitoring scheme for shared-memory multiprocessors (SMM) that does not disturb program behavior. Program errors that are visible during unmonitored execution are not hidden by the effects of the monitor. The proposed monitor can be used as a basis for any activity that requires access to the state of an executing parallel program without altering the manner in which the program executes. Such activities include parallel debuggers, execution monitoring tools, and performance monitoring tools [6].

Although our monitoring scheme requires the use of a software readable clock, there are no requirements placed on the granularity of this clock, nor on the synchronization of clocks between processors. Clocks with coarse granularity may cause monitored programs to run slowly due to the increased monitor overhead delays, but the unmonitored execution behavior is correctly reproduced. Breakpoints are supported such that the system is always halted in a globally consistent state [13], and can always be correctly restarted.

2 Monitor Model

Let \( P_i \) represent a single thread of execution of a parallel program. For purposes of discussion, assume that each \( P_i \) represents one processor. At any particular time \( P_i \) can be executing either the original target program instructions or program monitoring instructions. In the former case, we say that \( P_i \) is in an e-phase (execution phase), while in the latter case we say that \( P_i \) is in an m-phase (monitor phase). In the case of an unmonitored program, each thread consists entirely of one continuous e-phase.

Let thread \( E_i \) correspond to the e-phases of \( P_i \) and thread \( M_i \) to the m-phases of \( P_i \). The \( E_i \) threads represent the threads of the original target program, while the \( M_i \) threads represent the instructions added to accomplish the task of program monitoring. It follows that \( P_i \) is composed entirely of alternating segments of \( E_i \) and \( M_i \), where \( E_i \) and \( M_i \) are chronologically distinct, and for any unmonitored thread \( P_i \), \( E_i \equiv P_i \) and \( M_i \) is null. Figure 1 shows a portion of two execution threads \( P_i \) and \( P_j \) of a parallel program. Notice that the e-phases and m-phases of one thread need not correspond to those of another thread.

Lamport defines the happens before relation, represented by the symbol \( \Rightarrow \), with respect to message passing architectures [14]. We adapt this notation for SMMs by defining \( \Rightarrow \) as follows.

![Figure 1: m-phases and e-phases of two threads of execution.](image)

1. If \( a \) and \( b \) are events in the same execution thread and \( a \) comes before \( b \), then \( a \Rightarrow b \).

2. If \( a \) and \( b \) are events of accessing shared-memory object \( X \) and if \( X \) is accessed by \( a \) before \( X \) is accessed by \( b \), then \( a \Rightarrow b \).

3. The transitivity property states that if \( a \Rightarrow b \) and \( b \Rightarrow c \) then \( a \Rightarrow c \).

4. The well-formed property states that the relationship \( a \Rightarrow b \) if and only if one or more of rules 1 through 3 are applicable.

The relation \( \Rightarrow \) imposes a partial ordering on the events in a parallel program. Notice that \( a \Rightarrow b \) if and only if it is possible for \( a \) to causally affect \( b \) [14]. Not all distinct pairs of events are related by \( \Rightarrow \), hence, not all pairs of events are causally related.

A parallel program consists of multiple threads, each acting individually as a sequential process with input and output. In the case of a SMM, a thread obtains input by reading shared-memory objects and produces output by writing shared-memory objects. Taken by itself, a thread of a parallel program is deterministic. As long as the input of the thread is not altered, the local computations as well as the output of that thread will also be unchanged. It is only when the multiple threads interact that the potential for non-determinism arises.

We assume that execution times are also deterministic. Although this assumption can be unrealistic due to such factors as memory system non-determinism, it provides a starting point from which to develop a model of program monitoring. Future work will further address this issue.

If a monitor provides each thread with the same input that it would have had in an unmonitored execution, then each thread will produce the same output as if the program were unmonitored. In order to avoid ambiguity, we explicitly define the following terms.

\[ \text{It is assumed that the memory system imposes an order, perhaps arbitrary, on conflicting accesses to shared-memory.} \]
We say that a program monitor is intrusive if it, in any way, alters the timing of any thread of execution. Thus, by this definition, all software based monitors and debuggers are intrusive [6]. We say that a monitor is transparent if its presence does not alter the ⇒ relation between events, as compared to unmonitored program execution.

Consider a transparent monitor. By definition, this monitor preserves the ⇒ relation on events, thereby preserving the order of occurrence of any two events that are causally related. Through inductive reasoning, it is clear that if the order of all causally related events is preserved, all program output, including incorrect behavior, will also be preserved.

3 Monitor Design

In this section, we describe in detail the design of a transparent monitor for SMMs. Algorithms for implementing the major components of this monitor are also discussed.

Let T denote real time. For simplicity, assume that the value of T is zero when the program under consideration begins execution. The value of T increases continuously while the program is running, and represents the amount of time that has elapsed since the program began.

Let \( V_i \) be the virtual time of thread \( P_i \), which represents the execution length of \( E_i \). Like \( T \), \( V_i \) begins at zero when a program begins execution. But the value of \( V_i \) advances only when \( P_i \) is in an e-phase, that is, only when \( E_i \) is active. For unmonitored programs, \( V_i = T \).

Note that neither the value of \( T \) nor the value of any \( V_i \) is fixed. All are functions of time, and are thus constantly changing. So when we refer to \( T \) or \( V_i \), we are referring to its respective value at a particular instant.

Given that we would like a monitor to be transparent, let us now develop sufficient conditions for transparency in terms of virtual time. A transparent monitor must preserve the ⇒ relation among all program events. An effective monitor will not exchange the order of events that occur in the same thread of execution, but may change the order of causally related events that occur in different threads. Now suppose \( a \) and \( b \) are events in different threads (\( P_i \) and \( P_j \), respectively) such that \( b \Rightarrow a \). In the absence of monitoring, let \( T_a \) be the time at which \( a \) occurs and let \( T_b \) be the time at which \( b \) occurs. Since \( b \Rightarrow a \), it follows that \( T_b < T_a \). Now assume that the same program is executed with a non-transparent monitor, and that the first manifestation of this lack of transparency occurs when \( a \) is allowed to occur before \( b \), thus violating the relation \( b \Rightarrow a \). Since all ⇒ relations have been preserved until this point, the monitor has not altered the virtual times at which \( a \) and \( b \) occur. That is, \( V_i = T_a \) when \( a \) occurs and \( V_j = T_b \) when \( b \) occurs. It follows that when \( a \) occurs, \( V_i = T_a > T_1 \), or simply \( V_i > T_1 \). Since \( b \) will occur when \( V_j = T_b \) and since \( b \) has not occurred yet, we know that \( V_j < T_b \) (when \( a \) occurs). Thus, \( V_i > T_1 > V_j \), or simply \( V_i > V_j \).

If \( M_i \) can prevent \( V_i > V_j \) when \( a \) occurs, then transparency will be maintained. This assertion leads directly to our sufficient condition for transparency:

Rule 1 Before allowing any shared access \( a \in E_i \) to occur, \( M_i \) must ensure that \( V_i \leq V_j \), for all \( j \neq i \).

Let the delay value, \( D_i = T - V_i \), be the cumulative length of m-phases for \( P_i \). \( D_i \) is thus the amount of delay added to \( P_i \) by \( M_i \) (monitoring), and \( D_i = 0 \) for unmonitored threads. Stated in terms of delay values (\( D \)), Rule 1 becomes:

Rule 2 Before allowing any shared access \( a \in E_i \) to occur, \( M_i \) must ensure that \( D_i \geq D_j \), for all \( j \neq i \).

This method is used by the proposed monitor. Two main constructs are used to achieve this goal, the delay register and the shared object guard. The following sections describe these two constructs in detail.

3.1 Delay Registers

Each thread \( M_i \) maintains a delay register \( R_i \), whose granularity is equal to the granularity of the available clock. \( R_i \) can be modified only by thread \( M_i \), but can be read by any thread. Ideally, \( R_i = D_i \) at all times, but unfortunately, this is not possible. Among other reasons, the value of \( D_i \) is a continuously changing function of time. This property, alone, prevents the monitor from providing delay from \( M_i \). This section introduces the shared object guard, which is designed to ameliorate the effects of intermittently inaccurate delay values.

Let \( D_i \) be the cumulative length of m-phases for \( P_i \). \( D_i \) is thus the amount of delay added to \( P_i \) by \( M_i \) (monitoring), and \( D_i = 0 \) for unmonitored threads. Stated in terms of delay values (\( D \)), Rule 1 becomes:

Rule 3 At all times, each \( M_i \) must maintain the value of \( D_i \) such that:

1. \( D_i - \delta \leq R_i \leq D_i \), whenever \( P_i \) is in an m-phase, where \( \delta \) is a constant whose value is selected to be as small as possible such that the above constraints can be maintained.

2. \( R_i = D_i \) whenever \( P_i \) is in an e-phase.

The second condition is possible because \( D_i \) remains constant during the e-phases of \( P_i \). Thus, if \( M_i \) ensures that \( R_i = D_i \) at the transition from m-phase to e-phase, the relationship \( R_i = D_i \) will hold until the next m-phase begins. Choosing a larger value of \( \delta \) does not cause the monitor to become incorrect, that is, non-transparent, although smaller values of \( \delta \) cause less monitor intrusion, thereby causing less monitor induced execution slowdown.

In order to correctly maintain the value of delay register \( D_i \), as required by Rule 3, \( M_i \) must ensure two conditions:

1. During each m-phase, the condition \( D_i - \delta \leq R_i \leq D_i \) is always maintained.

2. At the end of each m-phase, \( R_i = D_i \).

These conditions are satisfied as follows. At the beginning of an e-phase, \( M_i \) executes idle instructions until the clock ticks. These idle instructions have a known execution time, which is recorded as the prelude time. The clock value is also recorded at this
time. At each clock tick (or at most after $\delta$ time) the clock value is again read and recorded, and $D_i$ is incremented accordingly. At the end of an m-phase, $M_i$ recalls the prelude length, sets up a postlude of idle instructions whose length plus the length of the preceding portion of the m-phase is an integral number of clock ticks, $D_i$ is incremented by the combined length of the prelude and postlude, and the postlude is started.

Each m-phase begins with a prelude and ends with a postlude. The purpose of the prelude is to allow the monitor to synchronize with the hardware clock, while the purpose of the postlude is to cause the duration of the entire m-phase to be an integral number of clock ticks. Additionally, the postlude transfers control to the next e-phase. Figure 2 illustrates an example m-phase, showing how the prelude and postlude are used.

![Figure 2](image)

Figure 2: The components of an m-phase.

Since $R_i$ is set to its final value for the m-phase at the beginning of the postlude, and $R_i = D_i$ only at the point of transition to the e-phase, then the relationship $R_i > D_i$ is true during the postlude. Although $R_i > D_i$ is a violation of Rule 3, no harm is done, since $P_i$ is guaranteed to not access any shared object until after the violation ends.

### 3.2 Shared Object Guards

In order to maintain the condition stated in Rule 2, an m-phase must precede each access to shared memory objects. These m-phases are installed through source code instrumentation, either by the use of macros or directly by the compiler. In either case, in order to provide for optional monitoring, the inclusion of m-phases can be selected by the user.

Before $E_i$ is allowed to access shared memory object $X$, an m-phase is inserted to ensure that $D_i > D_j$ for each thread $P_j$, $j \neq i$. If there is a thread $P_j$ such that $D_i < D_j$, then $M_i$ must delay, thus causing $D_i$ to increase, until $D_i > D_j$. In order to determine if $D_i \geq D_j$, for a particular $P_j$, $M_i$ compares $R_i$ and $R_j$. Given that $D_i - \delta \leq R_i \leq D_i$ and $D_j - \delta \leq R_j \leq D_j$, there are three possible cases:

1. If $R_i \geq R_j + \delta$ then $D_i \geq D_j$.
2. If $R_i < R_j - \delta$ then $D_i < D_j$.
3. Otherwise ($R_i - \delta \leq R_i < R_j + \delta$) we cannot be sure by only comparing $R_i$ and $R_j$ whether or not $D_i \geq D_j$.

In the first two cases, the values of $D_i$ and $D_j$ are sufficiently different for $M_i$ to decide if $D_i \geq D_j$ simply by examining the values of the delay registers $R_i$ and $R_j$. The cooperation of $M_i$ in this decision is not required, except that, as stated in Rule 3, $M_i$ must maintain the value of $D_i$ within certain bounds.

In the third case, $D_i \approx D_j$, and their relative values cannot be distinguished by comparing $R_i$ and $R_j$. Thus, $M_i$ must use other means to determine whether the relationship $D_i \geq D_j$ is true.

If $P_i$ is in an e-phase, then $M_i$ is inactive and $D_i$ and $R_i$ are static. Since $M_i$ is active, $R_i$ is increasing, indicating that $M_i$ need only wait until $D_i \geq D_j$ becomes true (or until $P_i$ enters an m-phase — this situation is discussed next).

If $P_i$ is in an m-phase, then in order to determine whether $D_i \geq D_j$ is true, $M_i$ must make a request to $M_j$ to participate in a “race” whose outcome will determine which of $D_i$ or $D_j$ is larger. This race resolves the inequality at a finer resolution than is available from the clock. Once $M_j$ responds to the request the race proceeds as follows. $M_i$ and $M_j$ agree on a marker $Q$ to reach in their race, and on a delay value, $D$ such that $D > D_i$ and $D > D_j$. The marker $Q$ is simply a shared-memory object that does not belong to any e-phase. Then, for $k \in \{i,j\}$, $M_k$ waits until $D_k = D$ and then writes the value $k$ to the marker $Q$. During the race, the thread $M_k \in \{M_i, M_j\}$ that has the smaller delay value will need to wait longer for $D_k = D$. $M_k$ will be the latter of $\{M_i, M_j\}$ to write to the marker $Q$, hence, after the race has been completed, $Q = i$ is the index of the thread with the smaller $D_k$ value.

In practice, $M_k$ cannot write to $Q$ instantaneously when $D_k = D$. There will be some delay, $\epsilon$, between the time that $D_k = D$ and the time that $k$ is written to $Q$. Since $M_i$ and $M_j$ execute the same predetermined sequence of instructions in $M_i$ and $M_j$ during the race, $\epsilon$ will be the same for $M_i$ and $M_j$. In this manner, $\epsilon$ will not affect the outcome of the race. Once $M_i$ has ensured that $D_i \geq D_j$ for all threads $P_j$, $P_i$ is allowed to proceed with the access of shared object $X$.

![Figure 3](image)

Figure 3: A monitor “race” between $M_i$ and $M_j$.

Figure 3 shows an example monitor “race” between $M_i$ and $M_j$. In this example, $E_i$ writes to marker $Q$. 230
last, hence when the race is finished, \( Q = i \). According to the previous discussion, \( Q = i \) at the end of the race indicates that \( D_1 \leq D_2 \). As the figure illustrates, \( D_1 \leq D_2 \) is the case.

Although m-phases must be inserted at each access to shared memory, monitoring is not limited to these points. Additional m-phases may be inserted at any point at which monitoring is desired. For example, m-phases might be used to track invocations of a certain procedure, or to record values written to a particular shared-memory object.

Additionally, any m-phase can be used as a breakpoint. That is, the user is free to interact with the monitor during an m-phase. While one thread is stopped at a breakpoint, each remaining thread will reach a point where continued execution would violate the \( \Rightarrow \) relation. Because of the monitor's transparency logic, no thread will be allowed to advance beyond this critical point. Thus, the entire system is eventually halted in a globally consistent state [13], that is, in a state in which no \( \Rightarrow \) relation has been violated, and from which the program may be resumed while preserving transparency. Restarting execution after a breakpoint is also automatic. Once the breakpointed thread resumes execution, threads waiting for this halted thread will also continue. From the perspective of the monitor, a breakpoint is no different than a normal m-phase, except that its duration is very long.

The monitor requires a hardware clock, or timing facility, to aid in measuring the duration of each m-phase. In order to achieve flexibility with respect to hardware requirements, we have been careful in our design to avoid the need for special clock features. In particular, we have the following conditions.

1. Clock ticks need not be synchronized with the beginning of an m-phase.
2. Multiple clocks need not be synchronized between processors.
3. Clock granularity need not be as fine as the granularity of instruction execution.
4. The clock may run continuously, with no facility for resetting it.

### 4 Performance Evaluation

Our monitoring scheme requires the instrumentation of every shared-memory access. In order to measure the overhead created by this intrusion, a prototype monitor has been implemented. The prototype is written in C and runs on the BBN Butterfly TC2000. The memory system of the Butterfly is physically distributed among the processing nodes, which communicate through a multi-stage interconnection network to provide a shared address space. At the software level, the Butterfly appears to have a single globally shared-memory system, except that accesses to physically local memory are several times faster than physically remote accesses.

In order to accommodate the different memory access rates for shared and local access, the prototype implementation has maintained a symmetry with respect to shared and local memory. For example, when two threads race for a shared marker \( Q \) (see Section 3.2), \( Q \) is chosen to reside physically at a third processor. The available 62.5\( \mu \)sec clock is used by the prototype to measure the duration of each m-phase.

A target program, also written in C, consists of 2000 iterations of a simple loop. Each loop iteration contains two shared-memory accesses, one read and one write, and two floating point multiplications, although this configuration can be changed to provide for various blends of communication and computation. The prototype provides a procedure, which is invoked at each shared-memory access in the target source code, to implement the shared-memory guard.

In order to determine the overhead created by monitoring, the target program was run with and without the monitor, on configurations of from 2 to 32 processors. The resulting execution times are shown in Table 1. Although the results indicate significant monitor overhead, the performance investigations are preliminary and only intended as a rough guideline for evaluating monitor overhead. Some portion of the indicated high overhead can be attributed to the artificial target program created for this test, which probably has a much higher communication to computation ratio (1: 1, in terms of number of operations) than most real programs. Also, there are several ways in which monitor overhead can be significantly reduced (these are described under future work).

Overall, we feel that the observed slowdown in execution time is acceptable, given the intended purpose of the monitoring tool, the transparency benefits that will result from its use, and the potential for significant reduction in monitor overhead.

<table>
<thead>
<tr>
<th>Processors</th>
<th>Execution Time (unmonitored)</th>
<th>Execution Time (monitored)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>28sec</td>
<td>2.37sec</td>
</tr>
<tr>
<td>4</td>
<td>29sec</td>
<td>7.38sec</td>
</tr>
<tr>
<td>8</td>
<td>51sec</td>
<td>19.98sec</td>
</tr>
<tr>
<td>16</td>
<td>77sec</td>
<td>50.43sec</td>
</tr>
<tr>
<td>32</td>
<td>148sec</td>
<td>141.04sec</td>
</tr>
</tbody>
</table>

Table 1: Test program execution times.

Future work includes the examination of methods to improve prototype performance in terms of execution slowdown. There are several ways in which the overhead of the prototype monitor can be reduced. The system call used to read the clock takes 15\( \mu \)sec. Since the clock value is accessed often, a direct method of reading the value would reduce overhead significantly. In order to reduce timing overhead, it may be possible for the compiler to predator the execution time of each block of code, and to insert instructions into the code that cause these times to be accumulated [15]. This method of measuring virtual time would remove overhead associated with the clock.

Also, it may be possible to specify some subset of shared-memory objects over which access guards are
needed. In other words, the user might specify the set of shared objects where anomalous access is possible, thereby significantly reducing monitor overhead. The user may also be able to define a point within the computation where monitoring is first needed, thus eliminating all monitor overhead prior to that point. Of course, the user would not be required to provide any of these "hints," but would be allowed to provide information to reduce overhead when the information is known and when monitor overhead is annoying.

5 Conclusions

We have shown how the probe effect resulting from the monitoring of parallel programs can cause changes in the programs being monitored. We adapted Lamport's "happened before" relation to shared-memory environments, and related it to sufficient conditions for transparent monitoring. We presented a system of virtual time in a monitored parallel program, along with the concepts of delay register and shared-object guard, and showed in detail how these notions can be used to provide transparent monitoring on a SMM with minimal clock facilities. Finally, we described a prototype implementation on the BBN Butterfly TC2000, used to investigate the proposed monitor scheme. Preliminary data showing monitor induced execution slowdown were reported.

More performance data are needed. It would be helpful to experiment with the mixture of computation and communication on the existing artificial test program, and to monitor real parallel applications with the prototype monitor. These experiments would give a much better indication of the actual execution slowdown caused by the monitor.

Another area of investigation is the creation of a metric of monitor transparency. That is, some method is needed to determine if a monitor has an effect on program execution, and if so, to provide a way to quantify this effect. Such a measurement might involve the introduction of an artificial race condition, with a parameterized margin of victory, into a controlled target program. The smallest margin of victory that the monitor can preserve, within the context of a given program, might be used to evaluate the transparency of the monitor implementation.

References


