Using Perturbation Tracking to Compensate for Intrusion in Message-Passing Systems *

J. A. Gannon †, K. J. Williams †, M. S. Andersland †, J. E. Lummp, Jr. ‡, T. L. Casavant †

† Electrical and Computer Engineering
The U. of Iowa, Iowa City, IA 52242

‡ Electrical Engineering
The U. of Kentucky, Lexington, KY 40506

Abstract

Execution monitoring plays a central role in most software development tools for parallel and distributed computer systems. However, such monitoring may induce delays that corrupt event timing. If this corruption can be quantified it may be possible to determine the intrusion-free behavior. In this paper we describe an algorithm that, given a safe timed Petri net model of the monitored software, can determine the uncorrupted timestamp values, i.e., those that would have been observed had the delays not been present. Monitoring conditions sufficient to ensure correct operation of the algorithm, and examples illustrating the algorithm's applicability to message-passing systems are also presented. This work is part of a larger effort aimed at identifying cost effective software alternatives to custom hardware monitoring.

1. Introduction

This paper describes a trace compensation algorithm developed as part of a larger study aimed at using software instrumentation, minimal generic hardware support, and post-execution analysis to correctly recover a segment of an uncorrupted event trace from an event trace collected by intrusive monitors [1, 9, 11, 19]. The goal of the study is to obtain accurate trace information more flexibly than, and for a fraction of the cost of, architecture-specific monitoring hardware. To this end, we have developed a prototype tool that generates a timed Petri net (TPN) model of intrusively monitored software. The tool uses the TPN model to recover, from the corrupted trace, the trace that would have been observed had monitoring-induced timing changes not been present.

Our work is motivated by the growing need for flexible, cost-effective techniques for monitoring the execution of concurrent software. Verification, debugging, and performance evaluation tools all depend on run-time information that is difficult to obtain in a distributed environment. For verification, run-time monitoring is a primary requirement; the code must be executed on the target hardware [10]. Likewise, debugging may require target hardware experiments [14], and performance monitoring may require observation of resource utilization that is derived from an examination of program state sequences (i.e., traces). In all cases, some form of instrumentation is needed to gather information. The critical problem is that of providing the instrumentation at a minimum cost, while introducing the least possible intrusion in the system, and ultimately providing a completely accurate trace to the user.

Presently, the designers of instrumentation systems must choose between expensive, architecture-specific monitoring hardware and software monitoring techniques. Software monitoring is less accurate because the differential "probe effect" delays it induces in the computation are a form of ***intrusion*** [3, 13], e.g., a debugging print statement that logs a variable induces intrusion through its use of CPU cycles and I/O data paths. Detecting and compensating for intrusion induced changes in a computation's behavior are problems of growing importance. Several perturbation models that can be used to extract approximate aggregate performance data from traces gathered using intrusive monitoring have been proposed [8, 12, 13]. It has also been shown that it is feasible to identify *possible* behavioral changes [6, 16] and event order changes for a limited class of events [18].

Here we focus on developing an algorithm that, given a safe TPN model of monitored message-passing software, can determine the uncorrupted timestamp values, i.e., those that would have been observed had the delays not been present. Details of the modeling and TPN generation process are presented elsewhere [11, 9]. Our algorithmic approach, termed **perturbation tracking (PT)**, allows for intrusive run-time execution monitoring with post-execution compensation for both time delays ***and event order changes*** when possible. The algorithm accurately tracks perturbations in systems for which only a subset of event times (monitor times) are known. The paper describes both the framework for PT that is applicable and the algorithm. A more general PT framework is given in [19].

*This research supported by the Advanced Research Projects Agency under Grant N00174-91-C-0116, and The National Science Foundation under Grant number CCR-9024484.

1063-6927/94 $03.00 © 1994 IEEE

2. Problem Statement

In this section, we first discuss the methodology and system properties that make trace compensation feasible and then state the perturbation tracking problem. The goal of trace compensation is to obtain uncorrupted trace values, i.e., those that would have been observed had the delays not been present, given the trace of a perturbed (intrusively monitored) program. This is a necessary first step if we hope to determine the intrusion-free behavior. To this end, the trace compensation process takes a segment of code and, via static analysis, models the general control flow. Given this information, we can instrument (monitor) the code to collect both information desired by the user and additional information sufficient to ensure that compensation is possible. The code is then executed on the target hardware producing a trace of monitor dumps with timestamps and corresponding processor ID's. The trace is processed a recovery tool that simulates the program on the system to obtain uncorrupted trace values.

Since we are simulating the program on the target hardware, we need a fairly complete model of control flow. We have chosen to use a deterministic TPN model as defined in the next section. This is reasonable because message-passing systems have several properties that reduce both the complexity of the TPN model and the monitoring information required for simulation. In particular:

- P1: The code in each concurrent processor is sequential and is only shifted in time by timing perturbations.
- P2: Monitors can be implemented (possibly with hardware assistance) to assure a known delay is incurred by the computer scheduling of debugging instructions.
- P3: There is never more than one operational program counter (token) in any section of code.
- P4: Sequential code decisions are not affected by perturbation unless message-passing primitives change order.
- P5: The message-passing primitives are fixed and are amenable to TPN modeling.
- P6: A processor identification tag can be affixed to each monitor firing.

Properties P1-P3 ensure that the durations of isolated operations are perturbation invariant, that the durations of the perturbations are known, and that program flow can be modeled by a safe TPN. Properties P4 and P5 limit the points at which branching decisions need be modeled and assure that they can be modeled. Property P6 ensures that we can always identify the last synchronization and monitoring events upstream from the current monitoring operation. Together these properties greatly reduce the complexity of the TPN models by allowing us to aggregate large segments of sequential code into single place/transition pairs. For details, see [11].

The "global clock" assumption inherent to TPN models poses additional challenges. These are addressed by:

(1) timestamping the arrival of all messages to ensure that synchronization delays can be distinguished from transmission delays, and
(2) using causality, and the latency and synchronization inherent in message passing mechanisms, to deduce event order [7].

These issues are discussed further in [4, 11].

In this paper we focus on developing an algorithm and monitoring conditions sufficient for compensation. Formally, we wish to:

(1) use a given TPN model of the monitored computation to determine where monitors should be added to ensure that compensation is possible, and
(2) determine, from the corrupted trace produced by the monitored computation, both the uncorrupted trace and the points at which the intrusion changed the computations behavior.

To compensate for the induced timing intrusion we need to track the propagation of the intrusion through non-trivial program constructs, e.g., points where the intrusion may be partially absorbed due to synchronization delays. To this end, we introduce a deterministic TPN model capable of representing many of the timing intrusion propagation mechanisms present in the system.

3. TPN Notation and Firing Rules

A timed Petri net is a bipartite directed graph of places and transitions used to model concurrent systems. We denote monitored TPNs by the 7-tuple $\mathcal{P} = (T, P, A, F, o, \mathcal{X}_0, M)$, where

- T denotes a finite set of transitions.
- P denotes a finite set of places.
- A $\subseteq (P \times T) \cup (T \times P)$ denotes a set of directed arcs.
- F denotes the set of mappings, $\{f_p(\cdot) : p \in P\}$, which, as a function of the current set of tokens in the net (set K defined below), route tokens at p to transitions in the output set of p.
- o $: T \to \{1, 2, \ldots, |T|\}$ is a bijective mapping that orders all transitions.
- $\mathcal{X}_0$ denotes the initial system state.
- M $\subseteq T$ denotes the set of "monitor transitions" or "monitors".

$I(u)$ and $O(u)$ denote the input and output sets, respectively, of transition or place u, e.g., for $u \in T$, $I(u) := \{p \in P : \exists (p, u) \in A\}$. The notation extends in the usual way to sets of places or transitions. We assume that the TPN is ordinary and safe. TPN events consist of transition firings that remove tokens from the input places of a transition and produce tokens in the output places according to the firing rules given below. The event e corresponding to the

jth firing of transition t is denoted by the pair $\langle t, j\rangle$, $t \in T$, $j \in J_t := \{1, 2, \ldots, J_{t,max}\}$ where $J_{t,max}$ is a finite natural number associated with transition t, i.e., $e_1 = t$ and $e_2 = j$. The set of all events that occur is $\mathcal{E} \subset T \times I\!N$.

The state of the TPN is given by the 3-tuple $\mathcal{X} := (K, N, R)$ where

- K $\subseteq \mathcal{E} \times P \times T \times T \times T$ is the set of tokens in the net, each denoted by, respectively, the event that created it, its holding place, the downstream transition that it is routed to, and the last monitor and the last synchronization events encountered by its ancestors.
- N denotes the set $\{N_u \in I\!N : u \in (T \cup P)\}$, where N_t, $t \in T$, is the current firing number of transition t and N_p, $p \in P$, is the current arrival number for a token entering place p.
- R denotes the set $\{r_t \in \overline{I\!R^+} : t \in T\}$, where r_t is the (extended non-negative real) residual time left on the delay clock associated with transition t.

For a token $k \in K$, $k = (e, p, t, m, s)$, we write $from(k) = e$, $loc(k) = p$, $to(k) = t$, $last_m(k) = m$, and $last_s(k) = s$ when k was created by e, is the token at p, and is routed to t, the last monitor event encountered by an ancestor token of k was m, and the last synchronization event encountered by an ancestor token of k was s.

The TPN input is given by the *firing delay schedule* $V := \{v_e \in I\!R^+ : e \in \mathcal{E}\}$, where v_e is the firing delay incurred by event e. For fixed V, we denote the enabled transitions as

$$\Gamma(K) := \{t \in T : (\forall p \in I(t)) [(\exists k \in K)[(loc(k) = p) \wedge (to(k) = t)]]\};$$

the set of newly enabled transitions, (i.e., transitions that are enabled but not scheduled) as

$$\Gamma_n(K) := \{t \in \Gamma(K) : r_t = \infty\};$$

and the set of tokens consumed by an event $e \in \mathcal{E}$ as

$$C(K, e) := \{k \in K : loc(k) = p, to(k) = e_1, p \in I(e_1)\},$$

where $c(K, e) \in C(K, e)$ denotes an arbitrary element of C(K,e).

The TPNs we study are governed by the following firing rules, where prime denotes "next".

TPN Firing Rules:

Initialize:

$z := 0;$
$\mathcal{X} := \mathcal{X}_0;$
$\forall t \in \Gamma_n(K);\ r_t := v_{\langle t, N_t\rangle};$
$\forall k \in K;\ last_m(k) = \emptyset;\ last_s(k) = \emptyset$

Recursion:

if $(\Gamma(K) = \emptyset)$, stop. (no enabled transitions)
$d := \min_{t \in \Gamma(K)} r_t;$ (time to next event)
$t := \text{argmin}_{i \in \Gamma(K), r_i = d}\, o(i);$ (firing transition)
$z' := z + d;$ (advance current time)

$$m := \begin{cases} \langle t, N_t\rangle & \text{if } t \in M \\ last_m(c(K, \langle t, N_t\rangle)) & \text{else} \end{cases}$$

$$s := \begin{cases} \langle t, N_t\rangle & \text{if } \mid I(t) \mid \geq 2 \vee t \in M \\ last_s(c(K, \langle t, N_t\rangle)) & \text{else} \end{cases}$$

$K' := (K \setminus C(K, \langle t, N_t\rangle)) \cup$ (next token set)
$\{(\langle t, N_t\rangle, p, f_p(K), m, s) : p \in O(t)\};$
$N_t' := N_t + 1;$ (update event score)

$$r_i' := \begin{cases} r_i - z' & \text{if } i \in \Gamma(K) \setminus \Gamma_n(K) \\ v_{\langle i, N_i\rangle} & \text{if } i \in \Gamma_n(K) \\ \infty & \text{else} \end{cases}$$ (update clocks)

if $t \in M$, output$(\langle t, N_t\rangle, z', C(K, \langle t, N_t\rangle))$;

The output of the firing rules is a sequence of 3-tuples (e, τ_e, C_e), where e is a monitored event, τ_e is the *firing time* of e, i.e., the time that transition e_1 fires for the e_2th time, and C_e is the set of tokens consumed by e. When it is necessary to distinguish between perturbed and unperturbed quantities, the latter are denoted by overbars, e.g., $\overline{V}$ is the unperturbed firing delay schedule.

Our TPN description is essentially that of a standard TPN that has transition firing delays and reserved tokens [15], with three additions. First, our TPN uses the routing functions $f_p(\cdot)$ to avoid the ambiguity that arises when transitions contend for a token. A similar approach is taken in [2], where the firing rules use exogenous sequences to route tokens. Second, our TPN uses the order number function $o(\cdot)$ to impose a firing priority on all transitions to avoid the ambiguity that arises when transitions are simultaneously enabled. Third, in our TPN only the firing times of monitored transitions are observable.

For fixed V, these additions ensure that the TPN's behavior can be characterized in terms of deterministic paths that sequences of tokens take through the net. This characterization is analogous to that provided by the paths of the evolution tree defined in [2]. Because we are only interested in the firing times of monitor transitions, we need only consider paths between monitor firings or paths between an event enabled at the start of net evolution and a monitor timing. Let

$$\mathcal{CE}_e := \{d \in \mathcal{E} : (\exists k \in C_e)(from(k) = d)\},$$

denote the set of all events that produce a token consumed by event e. Then these critical event paths can be defined as follows:

Definition 1 [19]: A *rooted* token track $\{\langle t^i, j^i\rangle\}_{i=1}^n$ is a sequence of events (transition firings) such that $\{\langle t^{i-1}, j^{i-1}\rangle\} \in \mathcal{CE}_{\{\langle t^i, j^i\rangle\}}$, for $i = 2, 3, \ldots, n$, i.e., successive events produce and consume the same token, and when $t^n \in M$, $t^i \notin M$ for all $i = 2, 3, \ldots, n-1$; and $\langle t^1, j^1\rangle$ satisfies at least one of the following conditions: (a) $t^1 \in M$, (b) $\mid I(t^1) \mid = 0$, or (c) $\forall p \in I(t^1), \mid \{k \in K_0 : (loc(k) = p) \wedge (to(k) = t^1)\} \mid \geq j^1$.

The firings of transitions satisfying (b) or (c) are termed *initial firings*. It can be shown (See [5]) that every monitor firing terminates a rooted token track so rooted token tracks indeed characterize the behavior of interest.

4. Perturbation Tracking Algorithm

In this section we describe an algorithm that tracks timing perturbations in TPN models of computations executing on intrusively monitored message-passing systems. The algorithm can recover an uncorrupted trace up to a behavioral change (e.g., an order change of message arrivals) between the real and simulated systems. The approach is similar in spirit to that of [19], but is based on somewhat different assumptions, e.g., unlike [19] it does not assume a priori knowledge of event order.

To begin we note that due to message-passing system properties P1-P3, we may assume:

A1. that for all $e \in \mathcal{E}$ the transition firing delays of the unperturbed and perturbed net, $\overline{v}_e$ and v_e, are independent of when they are initiated.

A2. that only monitors introduce perturbations and that the duration of these perturbations $\Delta_e := \overline{v}_e - v_e$, are known.

A3. that the TPN is safe.

A1 ensures that the delays in the TPN do not induce "second-order" perturbations in the firing delays, i.e., a perturbation in the enabling time of event $e \in \mathcal{E}$ does not change the perturbation Δ_e. The same assumption is used in [13] to recover an approximate trace. In general, A1 may not always hold, but may hold often enough to provide information that is otherwise difficult to obtain. These issues are discussed elsewhere [4, 11]. A2 ensures that perturbations are induced only at monitors and that these perturbations are of known duration. A3 ensures that tokens only idle at *synchronizing* transitions, i.e., at $t \in T$ such that $| I(t) |> 1$.

Next, we assume that, for all $p \in P$, the routing function $f_p(K)$ is of the following form:

$$f_{p(K)} := \begin{cases} t^1 \in O(p) & \text{if } \prod_{q \in D_p} \mathcal{I}\{\text{Property}_q^1\} = 1 \\ \vdots & \vdots \\ t^{|O(p)|} \in O(p) & \text{if } \prod_{q \in D_p} \mathcal{I}\{\text{Property}_q^{|O(p)|}\} = 1 \end{cases}$$

where $\mathcal{I}$ is a characteristic function

$$\mathcal{I}\{\text{Property}_q^i\} := \begin{cases} 1 & \text{if Property}_q^i \text{ holds at } q \\ 0 & \text{else} \end{cases}$$

$$\text{Property}_q^i := \begin{array}{c} (\exists k \in K)[(loc(k) = q) \wedge (to(k) \in Z_q^i) \wedge \\ (from(k) \in Y_q^i) \wedge (last_m(k) \in M_q^i)] \end{array}$$

D := $\{D_1, D_2, \ldots, D_{|P|}\}$ where $D_i \subset P$ is the set of places controlling the routing of place i.

Z_q $\subseteq O(q)$ denotes a set of output transitions of place q.

Y_q $\subseteq I(q) \times I\!N$ denotes a set of events that may create a token at place q.

M_q $\subseteq M$ denotes a set of last monitor events for a token at place q.

and

$$\sum_{i=1}^{|O(p)|} \prod_{q \in D_p} \mathcal{I}\{\text{Property}_q^i\} = 1.$$

Note that this routing function is quite general, e.g., routing can be based on token (process) location, destination, origin, or number.

Finally we note that P4 and P5 are sufficient to ensure that message passing systems can be instrumented such that the systems' TPN models satisfy the following conditions:

C1. $(\forall\, t \in T)[|\, I(t) \,| \geq 2 \Rightarrow I(I(t)) \subset M]$.

C2. $(\forall\, p \in P)[|\, O(p) \,| \geq 2 \Rightarrow I(p) \subset M]$.

C3. $(\forall\, p \in P)[(|\, O(p) \,| \geq 2) \wedge (\forall\, q \in D_p) \Rightarrow ((I(q) \cup O(q)) \subset M)]$.

C1 ensures that all transitions immediately upstream from synchronizing transitions are monitors, implying that the idle times of tokens upstream from synchronizations are observable. C2 ensures that all transitions immediately upstream from multi-output places are monitors implying that all routing decision times, and hence all possible event order change times are observable. Likewise, C3 ensures that all transitions immediately upstream and downstream from places controlling routing decisions are observable.

When the message passing system can be modified such that some of its places satisfy the following property, then it can be shown (See [5]) that the downstream monitoring requirement in C3 can be dropped at those places, i.e., C3 can be replaced by

C3′ $(\forall\, p \in P)[(|\, O(p) \,| \geq 2) \wedge (\forall\, q \in D_p)$ $[q \text{ has property } S] \Rightarrow (I(q) \subset M)]$.

Property S: A place, p, satisfies property S when $(\forall e \in \mathcal{E})[e_1 \in O(p) \Rightarrow (v_t = \overline{v}_t = 0)]$, i.e., when all the transitions immediately downstream are untimed.

Property S holds for message passing systems whenever messages are statically routed.

Our main result can be summarized as follows.
Theorem: Given a TPN satisfying A1-A3 conditions, C1-C3 are sufficient to ensure that the algorithm listed below:

I. Can be used to track perturbations as long as

(i) a behavioral change has not occurred, i.e., the results of decisions have not changed due to intrusion, and

(ii) firing time data is available, i.e., there are still monitor times to be compensated.

II. Will detect violations of conditions I.(i) and I.(ii).

Proof: See Appendix. ■

The algorithm first recovers the initial event times, i.e., those that that satisfy conditions (b) and (c) of Definition 1. When unrecovered events remain, the algorithm recovers the "next" timestamp and treats it as the last event in a rooted token track satisfying condition (a) of Definition 1. First, the subroutine *first_events*(e) identifies the first event in a rooted token track terminated by event e and all other events required to evaluate the idle time between the first and second events in the token track.

first_events(e):

1. $q := last_s(e)$; (last synchronization event)
2. $c := last_m(q)$; (last monitor)
3. $G := \emptyset$; (set of synchronized events)
4. $Q := \begin{cases} I(q_1) \setminus O(c_1) & \text{if } c \neq q \\ I(e_1) \setminus O(c_1) & \text{if } (c = q) \wedge (|I(e_1)| \geq 2) \\ \emptyset & \text{else} \end{cases}$
5. $\forall p \in Q$,

   (a) $e^p := \begin{cases} s \text{ if } (s \in \mathcal{TR} \cup \mathcal{PR})[(\exists k \in K)(loc(k) = p) \\ \quad (last_m(k) = s)(to(k) = q_1)] \wedge \\ \quad (\forall d \in (\mathcal{TR} \cup \mathcal{PR} \setminus \{q\}))[(s,d) \notin Used] \\ \emptyset \text{ else} \end{cases}$ (source of token in place p)

   (b) if $e^p = \emptyset$, then $Q = Q \setminus p$; (initial token in p)

   (c) $Used := Used \cup \{(e^p, q)\}$; (token used to fire q)

6. $\forall p \in Q$, $G := G \cup \{e^p\}$; (update synchronizing events)
7. return (c, q, G)

The algorithm then computes the idle and uses it to compensate the event time of the last event in the token track. This event is removed from the set of unrecovered events and placed in either the totally or possibly recovered event sets. Events are termed *totally recovered* unless they are preceded by an unresolved routing. When the last event precedes an unresolved routing, the events controlling the routing are stored, (until they are recovered), in the set $Y^{p,d}$ indexed by the routing place p and the event d triggering the routing. The *controlling* routine determines the event labels of the tokens in places controlling the routing from p at time τ_d.

controlling$(\{\overline{\tau}_e : e \in (\mathcal{TR} \cup \mathcal{PR})\}, p, d)$:

1. $Y^{p,d} = \{d\}$; (initialize set of events)
2. $\forall q \in D_p$ (places required for routing)

   (a) $i := \text{argmax}_{u \in (\mathcal{TR} \cup \mathcal{PR}), u_1 \in I(q)} \overline{\tau}_u$;

   (b) $j := \text{argmax}_{u \in (\mathcal{TR} \cup \mathcal{PR}), u_1 \in O(q)} \overline{\tau}_u$;

   (c) if $(\overline{\tau}_j < \overline{\tau}_i)$, then $Y^{p,d} = Y^{p,d} \cup \{i, \langle j_1, N_j + 1 \rangle\}$
   else $Y^{p,d} = Y^{p,d} \cup \{\langle i_1, N_i + 1 \rangle, j\}$

3. return$(Y^{p,d})$;

The set *Unresolved* keeps track of those perturbed place/routing event pairs that can not be resolved until further event times are compensated. When the event times required to resolve a routing are recovered, the routing function is evaluated for both the perturbed and unperturbed system. If they evaluate to different routes, condition (i) is violated. Otherwise, the process continues until no monitor event times remain.

Algorithm:

1. Recover the "initial" timestamps

   (a) $\mathcal{TR} := \emptyset$; $\mathcal{PR} := \emptyset$; (recovered events)

   (b) $\mathcal{UR} := \{e \in \mathcal{E} : e_1 \in M\}$; (unrecovered event set)

   (c) $Unresolved := \emptyset$; (unresolved event pairs)

   (d) $Used := \emptyset$; (set of used event tuples)

   (e) $\forall\, e \in \{c \in \mathcal{UR} : (\exists\, k \in C_c \mid (last_m(k) = \emptyset))\}$:

      i. $\tau_e := \overline{\tau}_e - \Delta_e$; (recover initial monitor times)

      ii. $\mathcal{UR} := \mathcal{UR} \setminus \{e\}$; (update unrecovered event set)

      iii. $\mathcal{TR} := \mathcal{TR} \cup \{e\}$; (totally recovered event set)

2. Recover the "next" timestamp

   (a) if $(\mathcal{UR} = \emptyset)$, goto 4;

   (b) $d := \text{argmin}_{e \in \mathcal{UR}}\{\overline{\tau}_e\}$; (next unrecovered event)

   (c) $(c, q, G) \leftarrow$ *first_events*(d);

   (d) $\overline{\omega}_{c_1,q_1} := \begin{cases} \max_{e^p \in G}\{0, \overline{\tau}_{e^p} - \overline{\tau}_c\} & \text{if } Q \neq \emptyset \\ 0 & \text{else} \end{cases}$;

   $\omega_{c_1,q_1} := \begin{cases} \max_{e^p \in G}\{0, \tau_{e^p} - \tau_c\} & \text{if } Q \neq \emptyset \\ 0 & \text{else} \end{cases}$;

   (e) $\tau_d := \overline{\tau}_d - [\overline{\tau}_{e^c} - \tau_{e^c}] - [\overline{\omega}_{c_1,q_1} - \omega_{c_1,q_1}] - \Delta_d$;

   (f) $\mathcal{UR} := \mathcal{UR} \setminus \{d\}$; (update unrecovered event set)

   (g) if $(\exists e \in \{G \cup c\})[(|\ O(e_1)\ | \geq 2) \vee (e \in \mathcal{PR})]$ then $\mathcal{PR} = \mathcal{PR} \cup \{d\}$;
   else $\mathcal{TR} = \mathcal{TR} \cup \{d\}$; (update $\mathcal{PR}$ and $\mathcal{TR}$)

3. Test for event order changes;

   (a) $(\forall\, p \in O(d_1))[(|\ O(d_1)\ | \geq 2)]$

      i. $Y^{p,d} \leftarrow \{$*controlling*$(\{\overline{\tau}_e : e \in (\mathcal{TR} \cup \mathcal{PR})\}, p, d)\}$; (determine controlling events)

      ii. $Unresolved := Unresolved \cup \{(p, d)\}$;

   (b) $\forall\, (p, e) \in Unresolved$ (unresolved event pairs)

      i. $Y^{p,e} := Y^{p,e} \setminus (\mathcal{TR} \cup \mathcal{PR})$;

      ii. $\forall\, Y^{p,e} = \emptyset$, (pairs with no unresolved events)

         A. $Unresolved := Unresolved \setminus \{(p, e)\}$;

         B. if $f_p(\overline{K}) = f_p(K)$, then

            1. $(\forall e \in \mathcal{PR})[\tau_e < \min_{(\cdot,j) \in \text{Unresolved}}\{\tau_j\}]$

               (a) $\mathcal{TR} := \mathcal{TR} \cup \{e\}$;

               (b) $\mathcal{PR} := \mathcal{PR} \setminus \{e\}$;

            2. goto 3b;

            else goto 4; (an order change)

   (c) goto 2a;

4. end;

An important algorithm extension would be to compensate for perturbation induced changes in the transition firing order. Currently, compensation terminates when order changes are detected. If order changes are also "tracked", then it may be possible to reorder segments of a correctly compensated trace by "cutting" incorrectly ordered delays out of the observed (monitored) event trace and "pasting" them in the correct unperturbed order. The number of order changes that can be accounted for in this manner will depend on the number of "paste" operations that can be pending simultaneously, and the length of the recorded traces (i.e., the trace must be long enough for the correct token path to occur.) For message-passing systems, the "cut" and "paste" operations would be flagged so that the programmer could determine if operations have violated A1-A3.

5. Example

The following examples illustrate how, given a TPN model of an intrusively monitored message-passing computation, the algorithm can be used to recover, from a corrupted trace, the trace that would have been observed had monitoring induced timing and event order changes not been present.

Example 1: A message-passing producer/consumer computation.

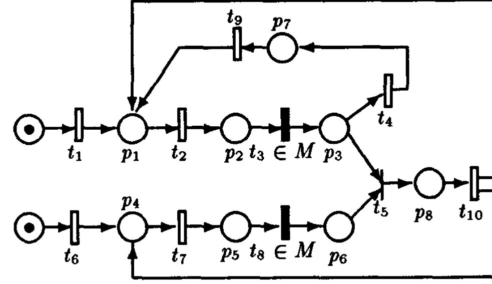
Consider a computation in which there are two concurrent processes, a *producer* generating results and a *consumer* that uses these results. The results are transferred via a synchronous send (**sync_send**) message-passing primitive in the producer process (the producer must wait until the send is acknowledged) and an asynchronous receive (**async_rec**) in the consumer process (the consumer can do other useful work in the absence of results from the producer).

Possible pseudo-code for this example is:

```
producer()
{
   while (forever) {                          (p^4)
      result = Compute_Result();              (t^7)
      Print_Result_and_Time(result);          (t^8)
      sync_send(consumer_pid,result);         (t^5)
   }
}

consumer()
{
   while (forever) {                          (p^1)
      Do_Some_Computation();                  (t^2)
      Print_Result_and_Time(result);          (t^3)
      if ( async_recv(result) )               (p^3)
         Use_Result(result);                  (t^5)
      else
         Do_Other_Work();                     (t^4)
   }
}
```

A TPN model of this code is shown in Figure 1 with its underlying clock schedule. Figure 2 depicts the times at which each monitor completes firing in the resulting trace (the observed trace) along with the output of the algorithm (the uncorrupted trace). Note that the algorithm only requires the timestamps for t^3 and t^8. The result can be verified by using the clock schedule to simulate the unperturbed system, i.e., set $\overline{v}_e = 0$ when $e_1 = 3, 8$.



$$f_{p_3}(k) = \begin{cases} t^5 & \text{if } \exists k \in K : loc(k) = p^6 \\ t^4 & \text{else} \end{cases}$$

$$\begin{aligned} V := \quad & \{v_{\langle t^1,1\rangle} = 1, v_{\langle t^2,1\rangle} = 1, v_{\langle t^3,1\rangle} = 1, v_{\langle t^4,1\rangle} = 1, \\ & v_{\langle t^{10},1\rangle} = 1, v_{\langle t^6,1\rangle} = 2, v_{\langle t^7,1\rangle} = 1, v_{\langle t^8,1\rangle} = 2, \\ & v_{\langle t^9,1\rangle} = 2, v_{\langle t^2,2\rangle} = 1, v_{\langle t^3,2\rangle} = 1, v_{\langle t^4,2\rangle} = 1, \\ & v_{\langle t^{10},2\rangle} = 2, v_{\langle t^7,2\rangle} = 1, v_{\langle t^8,2\rangle} = 3, v_{\langle t^9,2\rangle} = 1, \\ & v_{\langle t^2,3\rangle} = 2, v_{\langle t^3,3\rangle} = 1, v_{\langle t^2,4\rangle} = 1, v_{\langle t^3,4\rangle} = 1, \ldots\} \end{aligned}$$

Figure 1: TPN of a producer/consumer program

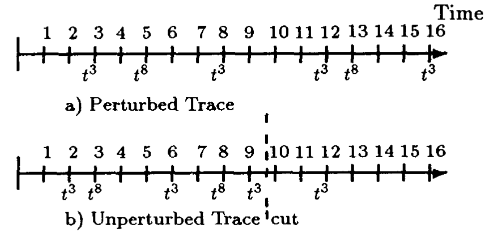


Figure 2: Perturbed and Unperturbed Traces

6. Conclusion

Clearly, the development of software for present-day multiprocessor systems requires high-level support tools. These tools, in turn, rely on accurate run-time information concerning the program state and its interaction with the target architecture. Existing approaches are typically hardware intensive or ad hoc. The perturbation tracking approach presented here is a software intensive approach that recovers from timing changes and detects perturbation induced order changes. Although the algorithm relies on fairly strong assumptions, the requirements are more modest than custom hardware intensive approaches of the past. Many message-passing systems already support software timestamps and any additional hardware that is required will be fairly generic.

Prototype tools that use our algorithm to compensate for monitoring intrusion in nCUBE-2 systems are currently under development. These tools automatically model instrumented explicitly parallel computations as TPNs prior to the algorithm recovery phase. Extensions being investigated include the development of "cut and paste" capability and a mechanism for compensating computations "on-the-fly".

Appendix

Lemma 1: *Given a TPN satisfying A1-A2 and C1-C3, the first and last events of rooted token tracks are known in both the perturbed and unperturbed TPN up to a "token track change," i.e., a behavioral difference in the perturbed and unperturbed TPN. Moreover, token track changes can be detected.*

Proof: See [5]. ■

Proof of Theorem 1: In [19] it is shown that, provided that condition (i) of the theorem holds and provided that the first and last event of every rooted token track is known, every monitor firing (other than initial firings) ended a rooted token track. Hence it suffices to show: (1) that the first and last event of every rooted token track is known, and (2) that the algorithm can compensate the perturbed monitor times for rooted token tracks starting at initial firings, i.e., those satisfying conditions (b) or (c) of Definition 1, and that the firing time of monitors ending rooted token tracks satisfying Definition condition (a) can be compensated as long as conditions (i) - (ii) hold.

Lemma 1 establishes (1). To establish (2) we first characterize the propagation of perturbations along a rooted token track, $\{e^i\}_{i=1^n}, n > 1$ in the unperturbed net. Observe that

$$\begin{aligned}\overline{\tau}_{e^n} &= \overline{\tau}_{e^1} + \sum_{i=2}^{n} (\overline{\tau}_{e^i} - \overline{\tau}_{e^{i-1}} - \overline{v}_{e^i}) + \sum_{i=2}^{n} \overline{v}_{e^i} \\ &= \overline{\tau}_{e^1} + \sum_{i=2}^{n} \overline{\omega}_{e^{i-1},e^i} + \sum_{i=2}^{n} \overline{v}_{e^i} \qquad (1)\end{aligned}$$

where $\omega_{d,e} := \tau_e - \tau_d - v_e,\ d \in \mathcal{C}_e$. Provided that (i) holds, (1) in the perturbed net can be written as

$$\tau_{e^n} = \tau_{e^1} + \sum_{i=2}^{n} \omega_{e^{i-1},e^i} + \sum_{i=2}^{n} v_{e^i}. \qquad (2)$$

Subtracting (1) from (2) we find that the cumulative firing time perturbation $\tau_{e^n} - \overline{\tau}_{e^n}$ at e^n is

$$\begin{aligned}\tau_{e^n} - \overline{\tau}_{e^n} &= \tau_{e^1} - \overline{\tau}_{e^1} + \sum_{i=2}^{n} \omega_{e^{i-1},e^i} - \qquad (3) \\ &\quad \sum_{i=2}^{n} \overline{\omega}_{e^{i-1},e^i} + \sum_{i=2}^{n} v_{e^i} - \sum_{i=2}^{n} \overline{v}_{e^i}.\end{aligned}$$

By A1, A2, Definition 1, and (i),

$$\sum_{i=2}^{n} v_{e^i} - \sum_{i=2}^{n} \overline{v}_{e^i} = \Delta_{e^n}.$$

A3 ensures that *idle times* $\overline{\omega}_{e^{i-1},e^i}$ only occur when $|\ I(e_1^i)\ | \geq 2$, and C1 ensures that $\overline{\omega}_{e^{i-1},e^i} = 0$ for $i \neq 2$ (i indexes an event in a rooted token track). Hence, (3) can be rewritten as

$$\tau_{e^n} - \overline{\tau}_{e^n} = \tau_{e^1} - \overline{\tau}_{e^1} + \omega_{e^1,e^2} - \overline{\omega}_{e^1,e^2} + \Delta_{e^n}. \qquad (4)$$

To verify the algorithm we step through it one step at a time.

Step 1 After the initialization of sets, step 1 recovers the monitored firing times ending each rooted token track satisfying conditions (b) or (c) of Definition 1. All of these token tracks have a $last_m(k)$ field for the consumed token k defined as $\emptyset$.

By the definition of a rooted token track satisfying conditions (b) or (c), C1, and C2, both the perturbed and unperturbed token tracks are unperturbed up to the monitor. Hence, $\tau_{e^1} = \overline{\tau}_{e^1}$, C1 ensures that $\omega_{e^1,e^2} = \overline{\omega}_{e^1,e^2} = 0$, and (4) becomes

$$\tau_{e^n} = \overline{\tau}_{e^n} - \Delta_{e^n} \quad . \qquad (5)$$

At the conclusion of *Step 1*, all monitoring events terminating token tracks that begin with initial firings are placed in the set of totally recovered events, $\mathcal{TR}$, and removed from the set of unrecovered events, $\mathcal{UR}$. These events are the first events of subsequent rooted token tracks satisfying condition (a). *Step 2* compensates for rooted token tracks satisfying condition (a). Suppose that the conditions of Theorem 1 are satisfied, and suppose that we wish to recover the next monitored firing time. It suffices to show: (A) that the next monitored firing time can be compensated; and (B) that the algorithm stops when either (i) or (ii) is violated.

A: *Step 2b* finds the next unrecovered event, d. *Step 2c* uses the subroutine $first_events(d)$ to find the first event in the token track, c, the last synchronization event in the token track, q, (if it exists), and the set of events, G, that generate the other tokens at the input of q. In $first_events(d)$, the first event in the token track and the last synchronization event are trivially found. The events that generate the other tokens for synchronization are the recovered events generating tokens routed to q that have not been *Used* to enable some other event. At this point, the only unknowns in (4) are $\omega_{e^c,q}$ and $\overline{\omega}_{e^c,q}$. From prior discussion, we know that these values are non-zero at synchronizing (multi-input) transitions, i.e., at q. *Step 2d* determines how long a token *idles* at $O(e_1^c)$. *Step 2e* recovers the event time for d and *step 2f* removes d from the unrecovered event set. *Step 2g* places d in the set of possibly recovered events either if a token track change is possible along its token track, or if an upstream event is in the possibly recovered set. Otherwise, d is placed in the totally recovered event set.

B: There are only two conditions under which the algorithm terminates. These conditions are: (a) when firing time data is no longer available, i.e., when there are no additional monitor times to be compensated (*Step 2a*), and (b) when the result of a decision is changed due to perturbation *Step 3b (ii.B)*. It suffices to show that these conditions are equivalent to the

theorem conditions. In fact, (a) is trivially equivalent and (b) can be shown equivalent via the proof of Lemma 1. ■

References

[1] M.S. Andersland and T.L. Casavant, "Recovering uncorrupted event traces from corrupted event traces in parallel/distributed computing systems," *Proc. of the 1991 International Conference on Parallel Processing*, August 1991, pp. II.108-II.116.

[2] F. Baccelli, G. Cohen, and B. Gaujal, "Recursive equations and basic properties of timed Petri nets," *Discrete Event Dynamic Systems: Theory and Applications*, Vol. 1, No. 4, June 1992, pp. 415 - 439.

[3] J. Gait, "A probe effect in concurrent programs," *Software-Practice and Experience*, Vol. 16, No. 3, March 1986, pp. 225-233.

[4] J.A. Gannon, K.J. Williams, M.S. Andersland, T.L. Casavant, and J.E. Lumpp Jr., "Trace Recovery in Multi-Processing Systems: Architectural Considerations", to appear: *23rd International Conference on Parallel Processing*, St. Charles, IL, Aug. 1994.

[5] J.A. Gannon, K.J. Williams, M.S. Andersland, J.E. Lumpp Jr., and T.L. Casavant, "Using Perturbation Tracking to Compensate from Intrusion Propagation in Message Passing Systems", *U. of Iowa Electrical and Computer Engineering Technical Report* TR-ECE-940329, March 1994.

[6] D.P. Helmbold, C.E. McDowell, and J. Wang, "Determining Possible Event Orders by Analyzing Sequential Traces", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 4, No. 7, July 1993, pp 827-840.

[7] L. Lamport, "Time, clocks and ordering of events in a distributed system," *CACM*, Vol. 21, No. 7, July 1978, pp 558-565.

[8] T. Lehr, *Compensating for Perturbation by Software Performance Monitors in Asynchronous Computations*, Ph.D. Theses, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, 1990.

[9] J.E. Lumpp, Jr., T.L. Casavant, J.A. Gannon, K.J. Williams, and M.S. Andersland, "Analysis of Communication Patterns for Modeling Message Passing Programs," *Proceedings of the International Workshop on Principles of Parallel Computing (OPOPAC)*, Lacanau, France, Nov. 1993, pp. 249-258.

[10] J.E. Lumpp, Jr., R.K. Shultz, and T.L. Casavant, " Design of a System for Software Testing and Debugging for Multiprocessor Avionics Systems," *Proceedings of the 15th Annual International Computer Software and Applications Conference (COMPSAC91)*, September 1991, pp. 261-268.

[11] J.E. Lumpp, Jr., *Models for Recovery from Software Instrumentation Intrusion in Parallel and Distributed Systems*, Thesis, Ph.D. in Electrical and Computer Engineering, University of Iowa, Department of Electrical and Computer Engineering, 1993.

[12] A.D. Maloney, *Performance Observability*, Ph.D. Thesis, Dept. of Electrical and Computer Engineering, University of Illinois, 1990.

[13] A.D. Malony, D.A. Reed, and H.A.G. Wijshoff, "Performance Measurement Intrusion and Perturbation Analysis," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 3, No. 4, July, 1992, pp. 433-450.

[14] C.E. McDowell and D.P. Helmbold, "Debugging Concurrent Programs", *ACM Computing Surveys*, Vol. 21, No. 4, December 1989, pp. 593-622.

[15] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, Vol. 77, No. 4, April 1989, pp. 541-580.

[16] R.H.B. Netzer and B.P. Miller, "What is a race? Some issues and formilizations", *ACM Letters on Programming Languages and Systems*, Vol. 1, No. 1, March 1992.

[17] M. Simmons, R. Kosdela, and I. Bucher, *Instrumentation For Future Parallel Computing Systems*, ACM Press, 1989.

[18] M. Spezialetti and J.P. Kearns, "A General Methodology for the System State Characterization of Event Recognitions", *Proc. of the 8th Symposium On Reliable Distributed Systems*, 1990, pp 175-184.

[19] K.J. Williams, M.S. Andersland, J.A. Gannon, J.E. Lumpp Jr. T.L. Casavant, "Perturbation Tracking" *Proc. of 33rd Annual Conference on Decision and Control*, San Antonio, Dec. 1993, pp. 674-679.