An Automatic Trace Based Performance Evaluation Model Building for Parallel Distributed Systems

Ahmad Mizan
System and Computer Department
Carleton University
Ottawa, Ontario
amizan@sce.carleton.ca

Greg Franks
System and Computer Department
Carleton University
Ottawa, Ontario
greg@sce.carleton.ca

ABSTRACT

Performance models can be built at early stages of software development cycle to aid software designers to assess design alternatives and identify fundamental design pitfalls before the implementation phase starts. These models are flexible for varying operational conditions and design alternatives; however, their creation is not trivial and requires considerable efforts. This paper addresses this problem by introducing automation in process of Layered Queuing Network (LQN) performance model creation for traces of events generated from instrumented software programs in the nodes of a distributed parallel software application.

The event-traces are created based on a new timestamp format, which is independent of physical time and uses extremely low count elements. A set of post-mortem methodologies have been introduced to identify the interactions between the service nodes of the parallel distributed software application and determine their workload activities, while supporting concurrent executions in the nodes. It can capture Forward, Asynchronous, Synchronous and loops of Asynchronous or Forward interactions. The final result is a framework of methodologies, specifications and tools which is appropriate for model-based performance evaluation parallel distributed software applications.

Categories and Subject Descriptors

C.3 **[Computer Systems Organization]**: PERFORMANCE OF SYSTEMS, Modeling techniques

General Terms

Performance

Keywords

Performance, distributed parallel systems, performance models, Layered Queuing Systems (LQN), execution graphs, event traces

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICPE'11, March 14–16, 2011, Karlsruhe, Germany.
Copyright 2011 ACM 978-1-4503-0519-8/11/03...$10.00.

1. INTRODUCTION

Performance, defined as the response time and throughput of a system, is a pervasive quality of distributed computing systems because everything from the low level hardware all the way to the application software affects it. It is a function of the amount of communication and interaction between the components of the system and is primarily concerned with the architecture of the software system. The lack of performance is often found to be a serious problem in significant fractions of projects because it causes delays, cost overruns, failure on deployment and abandonment of a project [1].

Evaluation of the performance of a system is often performed by measuring the requests arrival rates, distribution of requests, processing times, queue size, latency and the rate at which queues are serviced. This method of performance evaluation yields accurate knowledge of the system which is built under a particular set of assumptions and design parameters. If the results are not satisfactory the system should be rebuilt under another set of assumptions which is laborious and expensive and inflexible to provide enough room for intuition to find favorite design assumptions.

An alternative to this approach would be to evaluate the performance of a model of the software system rather than working with the software itself. The model is an abstraction of the system, which is distilled from the mass of details essential to its performance. Once the model is defined, it can be parameterized to reflect any of the design alternatives under study. Evaluation of the model also can help to determine the system's behavior under various design alternatives. The models are build by stochastic queuing models such as queuing network (QN) [12] and layered queuing networks (LQN) [14, 15, 17], or state-based systems such as a Petri Net [13], and solved by either analytic or simulation techniques. The models can provide performance predictions under various operation conditions or design architectures, which give valuable hints to detect root cause of performance issues. It means that by help of the models it is possible to design a software system and analyze its performance before even it is implemented. The down side to this approach is that a right amount of architectural features from the software documents or its source code must be abstracted in order to parameterize these models. This process is very difficult and time consuming, so the general trend is to postpone the performance evaluation until the software is developed, when it might be too late to fix fundamental architectural problems [7].

This paper describes an approach to simplify this process by automating the model making process and parameterization from recorded information of an instrumented live parallel distributed software system. A prerequisite for this approach is the availability of an executable early in the life cycle of the product which is quite feasible as modern software engineering promotes a release early in the development cycle [8]. The final performance model of this work is an LQN model. The LQN model is an extended queuing network [12] that includes the visits between processes so that their layered requests for service, and hence their contention effects, are represented.

A distributed system is composed of multiple autonomous computers or hardware nodes, which interact with each other in order to achieve a common goal, such as solving a large computational problem. The interactions between the nodes are solely performed by message passing. A problem is divided into many tasks, each of which is taken up by one particular computer. Tasks are software components whose execution are scheduled and may be performed in parallel on different nodes. From another perspective, a task is a resource that could potentially be shared by other entities therefore the set of active tasks are dynamic and constantly changing.

To have a trace of executed events in the computing system, instrumentation is added to the corresponding application software. This instrumentation consists of splicing probes, or analysis code, at specific locations in the source code. This enables the system to automatically create a time-stamped trace of events as it runs which is termed here as event-traces. By identification of the order in which the events are executed in an event-trace one can create an execution graph. An execution graph characterizes the partial order or independence relationship among the events that are executed in a scenario application of a parallel computing system. It consists of linear (consecutive events) sub-graphs which are termed as threads. A more comprehensive characterization of an execution graph is given in section 3.

The model making process involves capture of traces of events, creation of execution graph, identification of tasks and their interactions, estimating the workload imposed on tasks, recording of some environmental information associated with the distributed system, and transformation of the collected information into an LQN model.

The advantage of constructing models form an event-trace over using the software documentation and source code is that the former takes care of the dynamic features of a software system that are hard to impossible to extract for the latter. This feature of trace-based approach allows for identification of interactions in complicated software contexts such as dynamic bounding, inheritance, polymorphism and data dependent branching.

It is highly desirable to have a notion of global time in a distributed system. It can be used as the timestamps attached to events to enable one to identify the causality and independence relationship among the events. Taking local time of one particular node as the reference for the global time, as have been assumed by some works [5, 9], is not a practical solution. Some factors such as poor clock granularity and synchronization, clock drift and communication delays will not allow this reference to be perceived uniquely by all the nodes of a distributed system. It is not difficult to imagine a different mechanism, termed as virtual time, can reflect the same feature as time does to enable one to capture the causal and independence relationship among the events. Devising a format for virtual time is an essential part of constructing a distributed system performance model-making system, which has great impact on its performance, accuracy and complexity.

Fidge [4] and Mattern[21] introduced "vector time" and Fidge used it along with some rules to extract the execution graph of an application. Each component of the vector time corresponds to the current logical time of one live thread of execution. Each process, during its life span, maintains a variable holding its corresponding current vector time. In each thread only one element of the vector, which corresponds to the same thread, is immediately updated. The other elements are updated when the threads can exchange information which happens when they join. The rules dictate how the vector time is initialized, its components are inherited, incremented, and finally terminated. The causality of events and construction of the execution graph are performed by comparing the vector times of the different events.

An alternative way to represent the virtual time in a distributed application is "proper time" which was introduced in [3,6]. In this method, the events in executing software are characterized based on their situation in two different event graphs, namely their task and thread of operation event graphs. In the event operation graph, the event is executed in a thread of execution in which the event has a specific event number. In a task event graph, the same event has a name and a different event number which corresponds to the order in which that event is executed in that particular task. Although these two graphs are independently created, they are overlaid and give new characterization to the event.

The number of elements in "proper time" method is fixed, while in "vector time" it increases with the number of execution threads. This tends to make the "vector time" method incur an excessive communication overhead due to increased vector elements count as the distributed computing system gets larger, with potentially more threads of operation simultaneously being executed. Conversely, the proper time method suffers from some technical challenges in which the most significant one concerns with correct recording of the task event indices when the task performs concurrent operations. Separately indexing the events in concurrent threads of one task is not a trivial job. The same author has enhanced the method to solve the problem by recording one more parameter, the service period index, to be able to distinguish the events of different threads when concurrently are executed by the same task [19]. The task service period corresponds to the period in which a task fulfills a request, including all the nested interactions made with the servers in the lower layers. To do so the service period index must be traced through all the lower layer servers until it is recovered in the request acknowledgment. This operation significantly increases the model making process complexity because it must deal with a large amount of redundant information.

This works introduces a new format to represent a virtual time for a distributed system. Despite the "vector time", the number of elements in this format is constant and does not change as the number of threads in the distributed system increases. These elements are only associated with the operation context of a scenario application which makes the format significantly different and simpler than that of "proper time". This simplicity has significantly reduced the number of ways, or patterns, by

which the interactions between tasks in a distributed application can be represented.

A performance evaluation model builder based on this new logical time format is constructed which is able to identify all the various types of interactions supported by an LQN model even when the tasks are executing concurrent threads including Synchronous, Asynchronous, Forward and loops of Forward and Synchronous interactions. This system is also able to identify interactions among different entries of the same tasks, determine workload activities of the entries, and provide the final LQN model through a limited number of processing steps and transformations.

The paper is structured to illustrate the performance evaluation model building process based on the new approach. Section 2 gives a brief introduction to LQN model. Execution graph and its parameters are explained in Section 3. The methodologies used and steps needed to identify the interactions of the tasks are brought in Section 4. A general explanation of the actions involved in adding instrumentation to a system for each type of events is given in Section 5. In Section 6, the application of the method in a Building Security System is studied and Section 7 concludes the paper.

2. LQN MODEL OF A DISTRIBUTED SYSTEM

In this work a distributed system is modeled by a layered queuing network (LQN) performance evaluation model [14, 15, 17]. In the LQN model, software resources are represented by tasks. Each task is characterized by a queue, a level of concurrency and one or more classes of services, which are called entries. Entries have directed arcs to other entries to represent requests for services. An entry may either block, until it receives a reply, or continue operation. The former case is referred to as Synchronous interaction or remote procedure call (RPC) and the latter as Asynchronous interaction. The receiving entry may forward the request to an entry in another task rather than issuing a reply. Forwarding can continue until the reply is sent to the task made the original request. This model of task interactions is called Forward. An entry may continue to be busy after it sends a reply to an initiating interaction. This is referred to as a “second phase” of operation and is a common way of performance optimization, for example, for transaction cleanup logging and delayed writes. An entry is characterized by its number of phases, the type of interactions it makes with the entries of other tasks and its host and service demands. Each phase has its individual demands. Service demand specifies the mean number of service requests the entry makes to the entries of the tasks in a lower layer. The host demand specifies the mean total required CPU time for the entry in units of time [17]. Figure 1 shows an LQN model where task A makes Synchronous interactions with the task B which, in turn, makes a different Synchronous interaction with the task C. In this figure the parallelograms represent tasks and rectangles are entries, and arcs between entries represent messages or requests to a responding task. The task B is multi-threaded and this is indicated in the Figure 1 by the stack of parallelograms. The maximum number of the concurrent threads in task B is shown inside braces. Task A is a pure customer task, which is a surrogate for the workload generator of the system. The ovals represent the hosts where the tasks are executed. The numbers in brackets are the host demands and numbers in the parenthesis are request rates between entries.

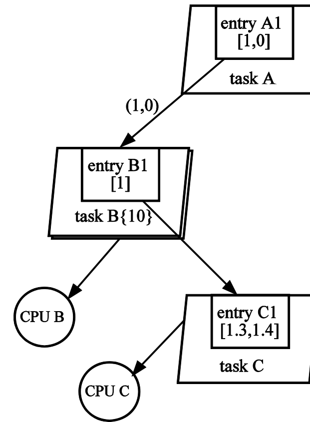

Figure 1: An LQN model

In distributed software systems, the delays and congestions are heavily influenced by Synchronous interaction types. An LQN model captures these delays by incorporating the lower layer queuing and service time into upper layer servers. Therefore the holding time for one class of service, or entry service time, is not a constant parameter but determined by its lower servers.

To construct the LQN model of a distributed system in this work, the elements of the model, including tasks and entries, and the interactions among the elements are identified from the parallel distributed system’s execution graph. The execution graph is created from event-traces. An event-trace is constructed by recording of events while the distributed system is running. The events are simply points of interest in the course of program execution. The most common event types are:

- Call and return of functions or subroutines
- Send and receive in a point-to-point message passing operations.

There are some other specifications that are to be provided for entries and tasks, which are referred to as “resource information” here. The resource information is a database which contains:

- The scheduling policy of the resources,
- The concurrency level of the tasks,
- Resource assignment (e.g. task to CPU assignment)
- The workload intensity or the rate at which requests are made by the reference task.

Host demands can either be provided through the resource database or automatically evaluated from the event-traces by addition of adequate extra instrumentation provided that the system clock’s resolution is high enough. To do so the events should be annotated by high resolution CPU time usage while being recorded.

Part of the validation process of the model is performed by iterative assessment of the performance results and adjustment of parameters in the resource database until optimum performance indices are achieved; however, achievement of satisfactory performance results sometimes requires assessing different architectural design alternatives.

The steps involved to construct the LQN model from an executable are summarized below.

1. Adding instrumentation probes into the software
2. Executing the program to create its execution graph
3. Identifications of tasks and entries involved
4. Identification of interactions between the entries from the execution graph
5. Estimation of host and resource demands
6. Using the resource database to complete the LQN model

3. EXECUTION GRAPH

An execution graph is characterized by a network of nodes which are connected together by a set of directed arcs. A node represents an event which is a uniquely identifiable runtime instance of an atomic action performed in an non-interleave manner by a single task. The relationship between two events is represented by a directed arc which is the indication that they are sequentially executed. An execution thread, or thread, is characterized by a sequence of events connected by directed arcs. If two threads are shared in one or two point, the points are either join or fork events. A fork event is where a new thread is spawned and a join event is where two threads merge to a single thread.

A node in the execution graph is a 3-port entity to represent various types of events including activity, fork and join events. One of the ports is of input type which receives a transition from another event. The two other ports are of output type; one to send a transitions to an event in the same thread and the other to a different thread, when it represents a fork event. Figure 2 shows the execution graph of a simple Synchronous interaction (RPC), which has been performed in one thread of execution.

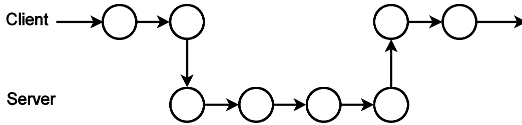

Figure 2: An execution graph

3.1 Event-Trace Creation

The computing system is loaded when a request for service is made by a reference task. This initiates an operation that can be performed through one or more threads of operations. Each thread consists of several activities performed sequentially in one or several tasks. The requests for services and replies are in the form of messages passed between the entries. An event index is attached to each recorded event. This index is incremented and passed to the following event as the program runs. The passed event index includes other information in order to specify in which thread and application scenario or operation the event is executed, i.e. the passed information includes the thread index and operation identifier to which the event belongs. A fork event is responsible for generating a new thread index. This thread index is used in turn to identify the event indices of the new thread. It is feasible for each fork event to know the indices belonging to the threads that are spawned from it. Similarly, the events in the spawned thread are able to know the thread index of the thread preceding the fork event. This helps to identify the connections between the execution threads, by which the identification of the reply corresponding to a request would be feasible (when the request-reply pair is performed in different threads).

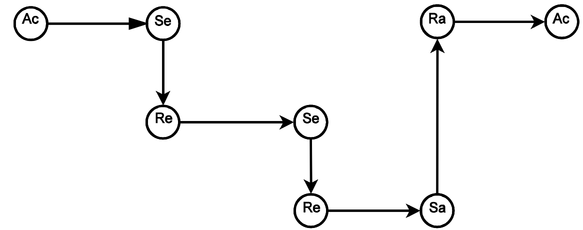

Figure 3: Event types in a Forward interaction

By attaching some predefined strings to each event, named "event types", as shown in Figure 3, the process of finding the start, the end and the places where forks occur would be possible. This also helps to find the initiating request in a Forward and loops of RPC or Forward interactions. In summary the following information are included in the timestamps attached to an event by the instrumentation system:

1. Thread index:
   a. Current
   b. Next
2. Thread event index
3. Event types (External, Begin, Activity, Fork , Send, Receive, SendAck, ReceiveAck and End)
4. Task name
5. Entry name

The thread index consists of two elements, the current and next thread index. This information is used to determine the connectivity of the threads. The execution graph is obtained by ordering of the threads as well as the events in each thread, based on their recorded timestamps. Event ordering is performed by finding out the causal and independence relationship between the events by using the timestamps attached to the recorded events in the event-trace.

Figure 4 shows how timestamps are used to create the execution graph of an Asynchronous interaction. The circles in the figure represent the events and the numerator and denominator of the fractional value inside the circles represent the "thread event index" and the "thread index" respectively.

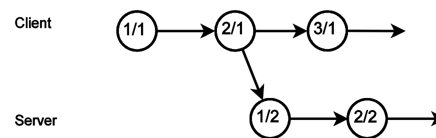

Figure 4: An Asynchronous interaction

Event 2 in Thread 1 is a fork event which uses all its three ports. The external output port of this event goes to the Event 1 of Thread 2 and the internal output port goes to the Event 3 of the same thread.

3.2 Event Types

In every interaction the type of event that an initiating task produces is either "fork" ("Fo"), or "send" ("Se"). When a task receives a request for service the corresponding event type is

"receive" ("Re"). When the task produces a response to a service request, the event type is "send acknowledge" ("Sa"). When a task receives a response to its service request the event type is "receive acknowledge" ("Ra"). Figure 5 and Figure 3 show the event types in RPC and Forward interaction.

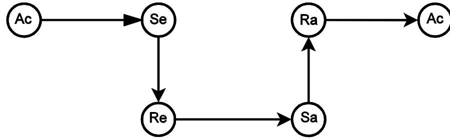

Figure 5: Event types in a RPC interaction

It is notable that a distinction is made between "send" and "send acknowledge" as well as "receive" and "receive acknowledge" event type. This is for convenient identification of interactions of types Forward and loops of either RPC or Forward. For example in the Forward interaction of Figure 3, using the "Sa" and "Ra" types make the service acknowledgement distinguishable which otherwise would have been confusing among the other two "send" and "receive" requests. The same situation applies to the loops of RPC interactions in which there are several requests for service and acknowledgements between two tasks as shown in Figure 6.

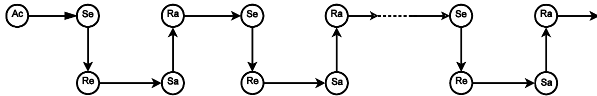

Figure 6: Loops of RPC interactions

If the "send acknowledge" and receive acknowledge" were not used the responding tasks would incorrectly perceived as the initiating task.

3.3 Event-Trace to Execution Graph Transformation

An event in the execution graph is characterized by the following parameters:

- The event's task name
- The event's entry name
- The event type
- The succeeding event in the current thread
- The succeeding event in the forked thread

These parameters can be directly extracted from the timestamps of events in the event-trace. This form of event representation is easier to handle throughout the rest of processing steps.

Transformation from event-trace to execution graph is easy and involves ordering of events with the same thread indices to form the individual threads; ordering of threads and finally finding the fork and join events by looking at the ending events of ordered threads.

4. INTERACTION IDENTIFICATION

This section illustrates how the interactions between the entries of tasks can be identified from their corresponding execution graph. Interactions are higher abstractions level of events, to which events are mapped by "patterns". There might be varieties of patterns that correspond to a particular interaction type. This section starts by describing the possible patterns corresponding to each interaction type and continues to explain the mechanism used to identify them.

4.1 Interaction Patterns

Interactions between tasks are modeled by patterns of events and their interrelationships. A pattern can be constructed by one or several threads. For modeling purpose, it is assumed that a pattern starts with one thread but for any of the following situations it spawns a new thread:

1. A task doesn't block when it request a service from another task
2. A responding task performs a second phase after it replies to the requesting task

Figure 2 and Figure 3 depict the first situation. Figure 7 demonstrates the second situation where the server generates a new thread to respond to the client when it needs a second phase operation.

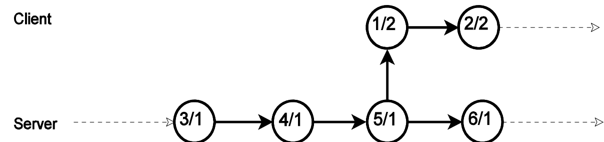

Figure 7: Generation of a thread to respond to the client when the server performs a second phase

The various interaction patterns identified in a distributed application, represented by their corresponding operation graphs, are listed in Table 1.

Table 1. Interaction patterns

Pattern	Graph
1. Simple RPC	Client 1/1 2/1 6/1 7/1; Server 3/1 4/1 5/1
2. RPC with second phase	Client 1/1 2/1 1/2 2/2; Server 3/1 4/1 5/1 6/1
3. Async RPC	Client 1/1 2/1 3/1 1/3 2/3; Server 1/2 2/2 3/2 4/2
4. Multi tier RPC	Client 1/1 2/1 1/4 2/4; Server 3/1 4/1 ?2/3 3/3 4/3 ?2/6
5. Forward	Client 1/1 2/1 1/3 2/3; Server1 1/2 2/2; Server2 3/2 4/2
6. Async	Client 1/1 2/1 3/1; Server 1/2 2/2

Case 1 in this table, the simple RPC, shows a pattern in which the first task blocks and the server doesn't do a second phase, so all the events are executed in the same thread.

In the Case 2, the RPC interaction contains two threads as the server performs a second phase.

In the Case 3 both of the conditions hold, that is the client doesn't block and the server performs a second phase; therefore this pattern of interaction is formed by three threads.

Case 4 shows that the phases of one and two have nested interactions with the server tasks in the lower layers. The types of these interactions, which are not displayed, are identified based on which pattern of the Table 1 they have followed. Since the client and the server have blocked, their requests are both modeled by one thread. The server uses the same thread it was acknowledged by the third-tier server to make a second phase nested service request. This thread spawns a new one to send acknowledge to the client.

Case 5 shows the simplest form of a Forward interaction. With the same principals it uses a single thread. A Forward interaction can have all the same variations of patterns of an RPC interaction that were illustrated through cases 1 to 4.

Case 6 shows a simple Asynchronous interaction, which obviously is performed by a spawned thread.

4.2 Events Abstraction

The method that is used to identify an interaction pattern is to transform the sequence of events in the execution graph to a higher level of abstraction to make the process of interaction identification easier. The sequences of events in the execution graph are decomposed to different types of chains of events that are named here as "transition", "connector" and "segment", as shown in Figure 8. A transition represents a directed arc between events that belong to two different tasks in the execution graph. A connector represents a directed chain of consecutive events that have been executed within a single entry of a task (A connector bridges the gap between two consecutive transitions in an execution graph). The overlapping event between a connector and a transition is shared by each of them. A segment is a directed arc that consists of a pair of consecutive transition and connector, starting with the segment. Segments are used to identify the task interactions and connectors and are used to identify the phase of an interaction throughout the interaction pattern matching.

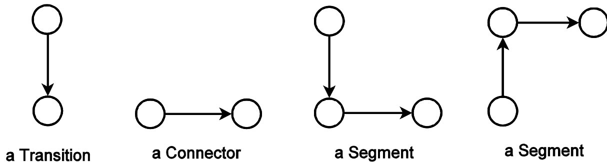

Figure 8: Abstracting event sequences in an operation graph

4.2.1 Anti-parallel segments

Two segments are anti-parallel when both of their interacting tasks (and entries) are identical but the directions of their interactions are different. Figure 9 shows two anti-parallel segments

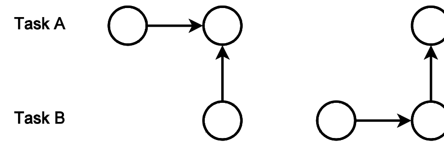

Figure 9: Anti-parallel segments

4.2.2 Matched segments

Two anti-parallel segments are matched when either of the following happens:

1. They have a shared event. This event is located at the end of one segment and beginning the other one. This case happens when the requested task is a pure server, i.e. it doesn't have a nested interaction with a lower layer task.
2. The ending event of one segment and the beginning event of the other one are not identical but are attached by a connector.

These are named as "type-one" and "type-two" matched anti-parallel segments respectively throughout the rest of this paper. For example Figure 10 shows a pair of type-two matched segments which corresponds to the interaction of case 4 of Table 1.

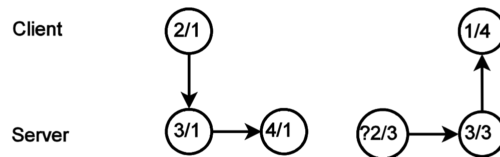

Figure 10: Two type-two matched segments

4.2.3 Double-segment

A double-segment is an extended segment which consists of two connected segments, as shown in Figure 11.The segments share an event which is the head event of one and end event of the other or vice versa, but the two segments are not matched. A double-segment can be considered like a new type of segment whose head event is that of the first segment and whose end event is that of the second segment. By this in mine a double-segment can match up against a single segment just like the way two segments do.

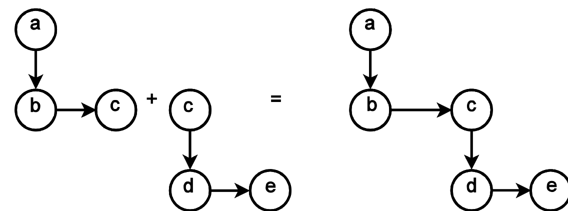

Figure 11: Formation of a double-segment

4.2.4 Multi-segments

A multi-segment is a further and the same way extended of a double-segment, or another multi-segment, with a single segment, as shown in Figure 12.

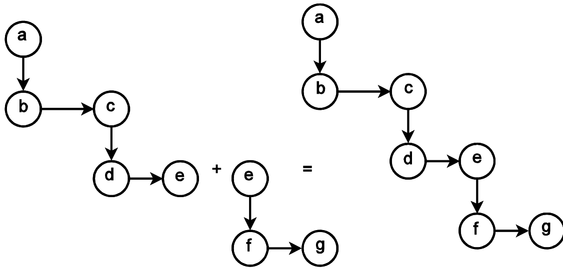

Figure 12: Formation of a multi-segment

4.3 Identification of Simple Forms of Interactions

By using the segments and multi-segments identified, various interaction types can easily be identified. This section discusses simple forms of these interactions in the sense that they don't have a nested interaction.

4.3.1 Synchronous interaction

Two matched anti-parallel segments can identify a Synchronous interaction between their corresponding two tasks, as shown in Figure 13. In a Synchronous interaction, an event from one task requests a service from that of another. The initiating task blocks until it receives a response. This is usually found in remote procedure calls.

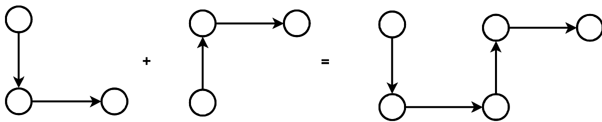

Figure 13: An RPC interaction with two matched anti-parallel segments

4.3.2 Forward interaction

Figure 14 shows that when a multi-segment is matched against a single segment, a forward interaction is identified.

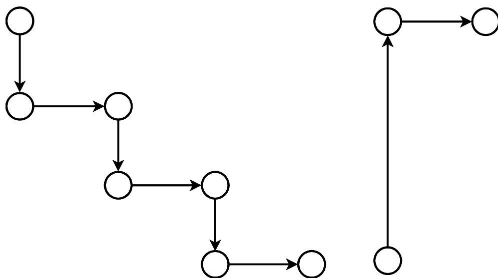

Figure 14: A matched multi-segment and segment to form a forward interaction

4.3.3 Asynchronous interaction

When all the synchronous and Forward interactions are identified, the remaining segments that have not participated in any of the Synchronous or Forward interactions specify Asynchronous interactions.

4.4 Identification Complex Forms of Interactions

Simple forms of interactions consist of interactions of type-one which occur at lowest two layers, i.e. the pure server layer and the one immediately above it. Interactions of type-two, in which the serving task performs nested interaction, are first converted to interaction of type-one and then identified by the methods of the previous section. This may prevent any uncertainty in terms of matching two wrong segments in complex unanticipated behavior of a system.

4.4.1 Type-two to type-one interactions conversion

A nested interaction places a gap between the two segments of its corresponding type-two interaction which should be removed to convert the type-two interaction to that of type-one. This is done by extending the connector of the first segment with a pseudo connector, which is a replacement for its identified (type-one) nested interaction at the lower layer, as shown in Figure 15.

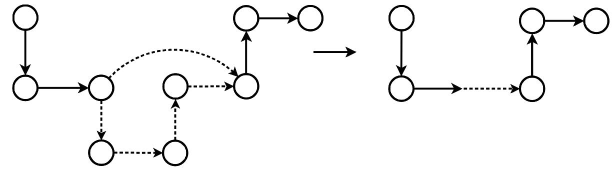

Figure 15: Bridging over an iteration and connector extension

Obviously the identification system must first identify the nested interactions and it starts from the lowest layer. The process of replacing a nested interaction with a pseudo connector and using it to extend its succeeding connector will be referred to as a "gap-filling" process throughout the rest of this paper. Now it is time to illustrate the whole interaction identification process in the following subsection.

4.4.2 Interactions identification algorithm

The main steps involved in identification of interactions are summarized as follows:

1. Transformation of execution graph to transitions, connectors, segments and multi-segments
2. Identification of matched anti-parallel segment pairs
3. Identification of matched pairs of multi-segments and single segments
4. Forming new connectors which are constructed by bridging over identified interaction
5. Extending the available connectors by the bridging connectors
6. Looping to step 3 until no new interactions identified
7. Remaining segments form asynchronous interactions
8. Determining the phase in which an interaction took place

The first iteration of these steps identifies the interactions between the entries of the tasks in the lowest two layers, i.e. the pure server layer and the one immediately above it. The gap-filling process will remove all the interactions at lowest layer and qualify the layer above with having only type-one interactions. Other iterations to step 3 of the above algorithm will further remove the layers, one per iteration, until it reaches to the top two layers wherein all interactions are identified. Therefore, the number of iterations required to identify all the interactions will be "n-1", where "n" is the number of layers.

4.5 Phase Two Nested Interaction Determination

Figure 16 shows the Interaction A which has a second phase nested Interaction B. The specifications of the relationship between these two interactions are listed below:

- The Connector C connects Interaction A to that of B
- First event of Connector C is shared with first event of the second segment of Interaction A
- Second event of connector C is shared with the first event of first segment of Interaction B

Therefore the phase two nested interaction determination process consists of examining the connectivity of one interaction to all the other identified interactions through one connector, in the way described above.

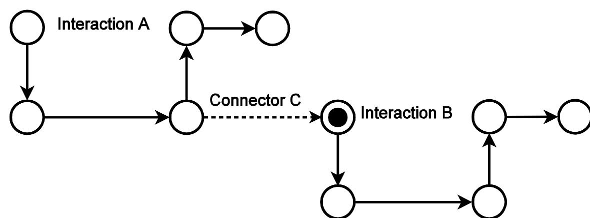

Figure 16: Second phase interaction identification

4.6 Merging of the Interactions

Many of the interaction identified from the same execution graph might be analogous in the following sense:

1. They have both the same source and the same destination tasks
2. Their entries both at the source and the destination ends are the same
3. Their interactions are of the same type
4. They have happened at the same phase of other analogous interactions

The analogous interactions are classified into what is termed as an "interaction-class". An entry request of an LQN model is determined by finding its corresponding interaction-class, resource demand and host demand.

4.7 Workload of a Task

An interaction-class specifies a request for service from a specific entry of one specific task, termed as entry-task pair, to that of another.

4.7.1 Partial resource demand

When all the interactions mapped to a particular interaction-class are quantified, it specifies a *partial resource demand*.

4.7.2 Total resource demand

Summation of all partial resource demands of all interaction-classes with the same target task-entry pair specifies the total resource demand of that target.

4.7.3 Resource demand

The resource demand of the entry-task pair "A" from the entry-task "B" is the ratio of the partial resource demand of entry-task "A" to entry-task "B" over the total resource demand of entry-task "B".

4.7.4 Host demand

The host demand of an entry is determined by measuring the total CPU time consumed by an entry operation. Measurement of this time is not the early target of this work and it is provided through the resource database by estimation.

5. INSTRUMENTATION

Instrumentation simply means splicing probes or analysis code consisting of e.g. logging instruction or print statements at particular locations in the source code. There are various mechanisms for adding instrumentation to a program such as using: compiler, libraries, direct source code and binary instrumentation. The basic requirement for an instrumentation infrastructure is to have zero probe effect when disabled and must be absolutely safe when enabled. There must be no way to accidentally induce system failure through system misuse.

To create an executable, instrumentation probes of this work are added to a simulation program as has been explained in the following section. Adding the probes to a real system and dealing with the corresponding practical details is an ongoing project and is not the main emphasis of this paper. In the following the pseudo code of the probes required to get the executable of a software system to generate the suitable event traces for identification of the interaction based on the methodologies illustrated is provided.

Regardless of the interaction types, the following basic operations will be performed to record and update the state of an event for instrumentation:

- The event index is incremented
- The event's timestamp is recorded
- Indices of event, thread and operation are passed to the following event

For specific events types, the above operation will be complemented with other operation specific to the event type. These operations per event types are illustrated below.

5.1 Fork Events

In addition to the above steps, the fork event performs the following:

- The fork event is responsible for generating the new thread index for the thread that was spawned.
- The event type label of this event is "Fo"
- The new thread index will update the next thread element of timestamp in the fork event

5.2 Begin Event

A "Begin" event starts when the sending event type is "Fork". The following extra steps are performed for a begin event:

- The current thread index is changed to the next thread index coming from a fork event.
- The event type label of this event is "Be"
- The event index is set to 1

5.3 External Event

An external event initiates an operation; therefore this is where the operation index is incremented. The following steps are performed for an external event:

- The operation index is set to “1”
- The thread event index is set to “1”
- The event index is set to “1”

6. CASE STUDY

This case study demonstrates automatic LQN model building by using an application called Building Security System (BSS). This application has been previously used in [18] and [19] to create its LQN performance model based on different methodologies. This work has adopted the same application to demonstrate the method of automatic LQN model building from event-traces generated from an executable. The executable of the BSS application is created by simulating it with PARASOL [20] which is a simulation environment by C and C++ libraries for distributed parallel applications, explained in the following paragraph. The resulting simulation is instrumented based on the rules of the previous section and its executable is used to generate the event-traces for the illustrated post-mortem analysis. It will be shown that this work is able to construct the same LQN models developed in [18] and [19] by using the new explained methodologies. Figure 17 shows the activities performed in this case study.

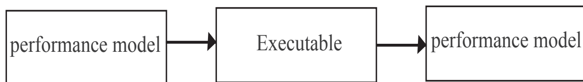

Figure 17: Case study

The PARASOL software interface looks like a primitive distributed operating system, containing a number of functions to support task management including dynamic task creation/destruction and inter-task communication. It allows true concurrency to be simulated in a multi-processor system, with scheduled concurrent tasks. The simulated execution environment on which PARASOL tasks execute is constructed from nodes and one-way communication links. A PARASOL node may have one or several processors. Each node has a single ready-to-run queue and is managed by either a built-in scheduler or by a user-defined scheduler. Network connections are made through point-to-point one-way links.

The BSS is intended to control access and to monitor activities in a building like a hotel or a university laboratory. To assess the performance and improve the design, two main scenarios of this application are modeled. The first scenario, Access Control, is used for the control of door locks by access cards. In this scenario, a card is inserted into a door-side reader where its contents are read and transmitted to a server. The server checks the access rights associated with the card in a database of access rights, and then either triggers the lock to open the door, or denies access. The second scenario, Acquire/Store Video, is used for video surveillance. In this scenario, video frames are captured periodically from a number of cameras located around the building, and stored in the database. The referenced papers perform the assessment of the performance, improvement of design and planning of capacity for the number of cameras for future scaling by looking at various alternatives such as system configuration. The sequence diagrams of the two scenarios described earlier are shown in Figure 18 and Figure 19.

The video surveillance scenario has a 5-tier architecture. The tasks of Buffer, StoreProce, AquProce, DB and Disk are multi-threaded with each having 15, 3, 12, 10 and 2 threads respectively. Tasks DB and Disk are shared by both scenarios. All of the interactions except one are of types RPC or Forward.

The access control scenario has a 4-tier architecture. With the exception of the user task, all of the tasks are multi-threaded. Tasks SCR, AccCtrl, DLA, DB and Disk each have 50, 60, 50, 10 and 2 threads respectively.

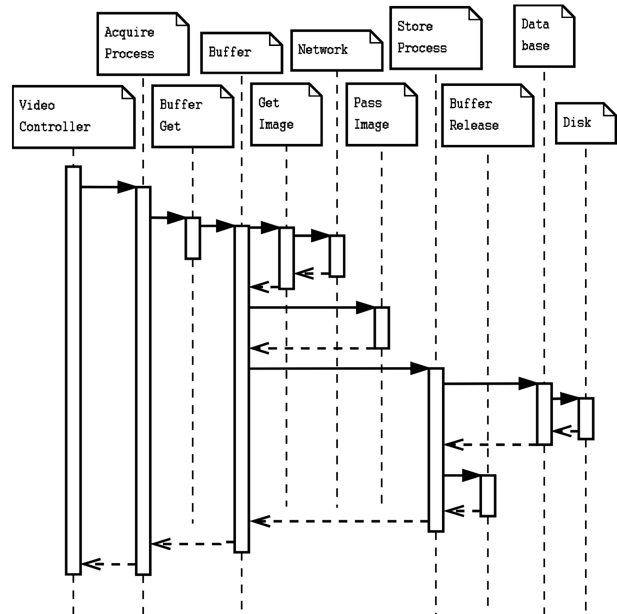

Figure 18: Sequence diagram of video surveillance scenario

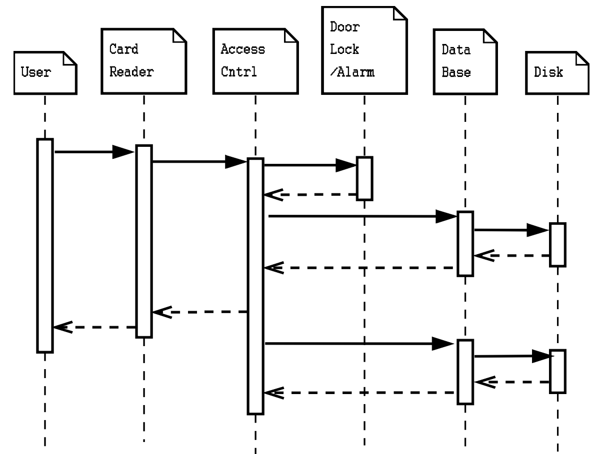

Figure 19: Sequence diagram of access control card scenario

6.1 Making an Executable

A PARASOL [20] simulation is used to create the event-traces required for construction of the execution graph. PARASOL is an execution-based tool, which provides a flexible software

prototyping environment for distributed and/or parallel computer systems.

PARASOL is built up of multi-processor nodes interconnected with communication devices of various types and capacities. The concurrent software is created explicitly with user-defined tasks written in C or C++. Operationally, a PARASOL-based simulator runs as a single task in a POSIX compliant host environment.

The workload of this case study is created by the “User” and “vidCtrol” tasks, which initiate interactions by sending their service requests to the lower layer tasks of SCR and AcqProc, respectively. The arrival rate is defined by an exponential random delay time between each request in the simulator.

After prototyping the application, instrumentation is added to enable the resulting executable to generate the desired event-traces. The details regarding creation of the simulation of the application will be provided in an extension paper.

6.2 LQN Model

The state of knowledge before tracing is the resources service demands associated with the entries, the task multi-threading level, CPU multiplicity and task to CPUs allocation map. The user provides this information through the resource database. The system identifies the tasks, entries, the interactions between them and the number of times they have been repeated in a specified period. It also identifies if an interaction is nested in another one and whether it is in “first phase” or “second phase. After merging each interaction into various interaction-classes identified, the system determines the visit ratios and resource demand associated with each request. After taking care of a few other details such as counting the total number of tasks, entries and CPUs, the process of building the LQN model is finished. The visual representation of the LQN performance evaluation model of this case study is shown in Figure 20, which is very similar to the model presented in referenced paper [20], from which the executable of this case study was created.

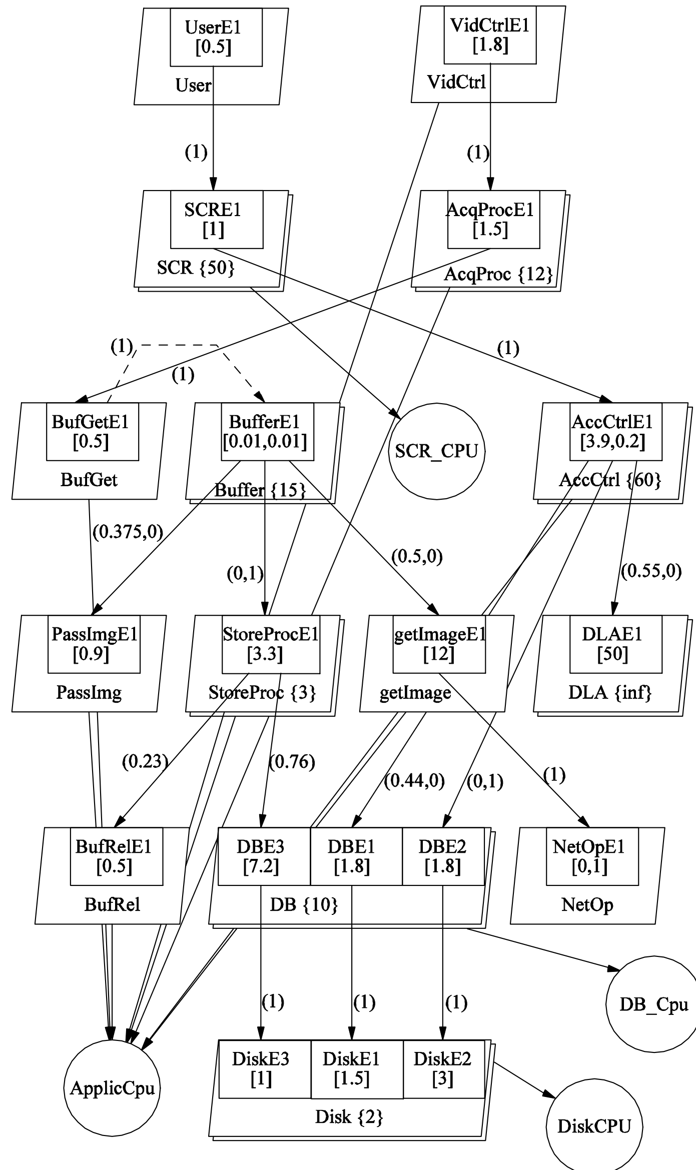

Figure 20: The resulting LQN model of the BSS case study

7. CONCLUSION AND RESULTS

This research introduces a framework consisting of specifications, methods and tools to capture the workload and interaction architecture between the nodes of a parallel distributed software application to generate its LQN performance evaluation model in an automatic fashion. To capture interaction architecture a trace-based methods is utilized. Timestamps, with a new structural format, are attached to the events by an instrumented software program which currently is the simulation of a parallel distribution application. A formal post-mortem analysis approach is used which takes the event-traces as input and produces the LQN performance evaluation model. This approach of performance model generation is appropriate for systems without a time reference, such as distributed software systems, since the new timestamp format is independent of physical time. It is also appropriate for parallel or concurrent processing systems since the interaction identification and workload detection methods used in this work allow for internal concurrency in individual nodes. The contributions of this paper are:

1. Introduction of a new lightweight format for logical time
2. Providing pseudo code of the instrumentation probes to be added to the source code of the parallel distributed application based on the new timestamp format
3. Providing a low overhead communication for interactions in a distributed application
4. Introducing a framework of specifications, algorithms, methodologies and tools to identify the architectural structure and workload activities of a distributed computing system from the event-traces
5. Automatic, end to end, creating of LQN performance evaluation model from the executable of a computing system
6. Demonstrating the automatic performance evaluation model creation method using PARASOL as a distributed parallel software simulator

The definition of the probes in the instrumentation system and the interaction identification method are designed in a way to capture the sequence of consecutive interactions in a loop interaction between two tasks, which is a common behavior in distributed systems. The system has demonstrated a strong ability to correctly identify the various patterns which map to the Synchronous and Forward interactions, which are the crucial part in any performance model making process for distributed systems. Application of the method on different event-traces extracted from same prototype has always led to the exact same identified distributed software architecture, which is an indication of robustness of the method. This is compared to the case of proper time method [3,6] in which the accuracy and reliability of the end result is dependent to the correct determination of the service period of a task.

The hosts service demands, tasks to CPUs mapping and the anticipated multiplicity of the CPUs and threads in a task are the only manually provided information to complete the LQN performance evaluation model. Automatic determination of these parameters is an ongoing research and will be addressed in a separate work, which analyzes the application specifications and information of the involved components for their determination.

Application of this method allows for automatic construction of the LQN evaluation performance model for a software system and would prevent human interventions error, which is likely to occur in large systems. This method is suited to be applied in a software performance engineering (SPE) fashion by which the target systems performance is automatically monitored as the system evolves.

8. ACKNOWLEDGMENTS

Financial support for this research was provided by the Natural Sciences and Engineering Research Council of Canada.

9. REFERENCES

[1] A. D'Ambrogio and P. Bocciarelli, "A Model-Driven Approach to Describe and predict the Performance of Composite Services", In Proc. Sixth International workshop on software and performance (WOSP'07), pp. 78-89, Buenos Aires, Argentina, 2007, ACM Press.

[2] Curtis E. Hrischuk , C. Murray Woodside, Jerome A. Rolia, Rod Iversen, "Trace-Based Load Characterization for Generating Performance Software Models", IEEE Transactions on Software Engineering, v.25 n.1, pp. 122-135, January 1999

[3] Curtis E. Hrischuk, "Trace-Based Load Characterization for Automated Development of Software Performance Models", PhD Thesis, System and computer engineering department, Carleton University, 1998

[4] C. Fidge, "Logical Time in Distributed Computing Systems", Computer, Volume 24, Issue 8, pp. 28–33, Aug. 1991.

[5] T. Israr, "A Light Weight Technique for Extracting Software Architecture and Performance Models from Traces", Master Thesis, System and Computer Department, Carleton University, April 200.

[6] C. E. Hrischuk , C. M. Woodside, "Logical Clock Requirements for Reverse Engineering Scenarios from a Distributed System", IEEE Transactions on Software Engineering, v.28 n.4, pp. 321-339, April 2002

[7] C.U. Smith, Performance Engineering of Software Systems. New York: Addison-Wesley, 1990.

[8] I. Jacobson, G. Booch and J. Runbugh, " The United Software Development Cycle", Object Technology Series, Addison-Wesley, Boston, MA, 1999

[9] C. Hrischuk, J. Rolia, and C.M. Woodside, "Automatic Generation of a Software Performance Model Using an Object-Oriented Prototype", Proc. Int'l Workshop Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'95), pp. 399–409, 1995

[10] S. Vaidehi, D. J. Ram, A. Shukla, "Difference Clock: A new Scheme for Logical Time in Distributed Systems", IEE Proc., Comput Digit Tech, Vol. 143, No. 6, November 1996

[11] H. Koetz and W. Ochsenreiter, "Clock synchronization in distributed real-time systems IEEE", Transactions on Computers Volume 36 , Issue 8 (August 1987), Special Issue on Real-Time Systems, pp. 933 - 940

[12] E.D. Lazowska, J. Zahorjan, G. Graham and K. Sevcik, Quantitative System Performance. Englewood Cliffs, N.J., Prentice Hall, 1984.

[13] C.U. Smith, “Robust Models for the Performance Evaluation of Software/Hardware Designs,” Proc. Int’l Workshop Timed Petri Nets, pp. 172–180, Torino, Italy, July 1985.

[14] J. Rolia and K. Sevcik, “The Method of Layers”, IEEE Trans. Software Eng., vol. 21, no. 8, pp. 689–700, Aug. 1995.

[15] C.M. Woodside, J.E. Neilson, D. Petriu, and S. Majumdar, “The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-Like Distributed Software,” IEEE Trans. Computers, vol. 44, no. 1, pp. 20–34, Jan. 1995.

[16] G. Franks, S. Majumdar, J. Neilson, D.C. Petriu, J. Rolia, and C.M. Woodside, "Performance Analysis of Distributed Server Systems," in the Sixth International Conference on Software Quality (6ICSQ), Ottawa, Ontario, pp. 15-26, 1996

[17] “Layered Queuing Network Solver and Simulation User Manual”, Department of System and Computer Engineering, Carleton University, April28, 2010, Revision 9242

[18] Jing Xu, Murray Woodside, Dorina Petriu, “Performance Analysis of a Software Design Using the UML Profile for Schedulability, Performance and Time”, Proc. 13th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation, ,2003

[19] Lianhua Li, Franks G., *“*Performance modeling of systems using fair share scheduling with Layered Queuing Networks”, Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS '09. IEEE International Symposium on

[20] J. E. Neilson, "Parasol: A simulator for distributed and/or parallel systems," Tech. Rep. SCS-TR-192, Carleton University, 1991.

[21] F. Mattern, “Virtual Time and Global States of Distributed Systems,” Proc. Int’l Workshop Parallel and Distributed Algorithms, pp. 215–226, Amsterdam, Bonas, France, North-Holland, 1988