Monitoring of Component-Based Systems

Jun Li
Imaging Systems Lab
HPL-Palo Alto

Abstract

The current state-of-the-art techniques are not sufficient to debug, understand and characterize multithreaded and distributed systems. In this report, we present a software monitoring framework for distributed and multithreaded systems which are built upon component technology, as the attempt to explore software development tools to address this need. Our monitoring framework captures multi-dimensional system behaviors—application semantics, timing latency and shared resource usage. We adopt the approach of compiler-assisted auto-generation of instrumentation probes into the stub and the skeleton. With our system-wide causal tracing techniques, global propagation of resource consumption and timing is captured from locally recorded monitoring data. Furthermore, the monitoring framework allows users to select different types of system behaviors and different subsets of candidate components for monitoring. As the result, not only the amount of monitoring data can be reduced, but also the monitoring associated interference can be eliminated or mitigated. System behavior characterization tools are also developed to support this monitoring framework, including reconstruction and visualization of dynamic system call graphs, and characterization of end-to-end timing latency for function invocation.

1. Introduction

The successful development, maintenance and evolution of a software system demand a full understanding of the targeted system behaviors. Examples of the involved software engineering practice are: to pinpoint the root cause of the observed system failure, to identify performance bottlenecks, to deploy the system under a constrained resource budget, and to estimate how impact change propagation occurs in the system before conducting the actual change, etc. Application-level semantic behavior (such as which function calls which function, why an exception is raised in a function execution), timing behavior (such as an end-to-end timing latency of a function invocation), and shared resource (such as CPU and heap memory) usage behavior, constitute the multi-dimensional system behaviors in a software system. Moreover, different types of semantic behaviors can intertwined with each other. For example, a function invocation takes longer than expected if excessive number of processes are deployed in the system to compete the CPU resource. Out of dynamic memory allocation leads to function invocation exception, which in turn forces the execution to be aborted. Thus, a software system is a complex multi-dimensional network of semantic dependencies.

With such complex semantic behavior and inter-dependencies, in practice, single-threaded and single-processed systems have been realized to be difficult to debug, to understand and to characterize, even though various monitoring tools and profiling tools on different aspects of the systems have been developed. Multi-threaded systems are much more difficult to understand because threading introduces yet another dimension of complexity, e.g., thread concurrency, thread reentrancy. The situation becomes even worse, when such a system is deployed into different processes, located in different processors. Most current existing tools and development environment cannot significantly handle such truly distributed systems, namely, multithreaded, multiprocess-ed and multiprocessor-ed systems. In this report, we refer the processors to the computing hosts that are uniquely identified in the network where the application is running.

We have developed a runtime monitoring framework as the first step towards helping system developers and maintainers understand the behaviors of multithreaded and distributed systems. Our primary target systems are large-scale multithreaded and distributed systems in the embedded application domain. The following are the principle design objectives and requirements that must be met to build such a monitoring framework:

- **System-Wide Multi-Dimensional Behavior Capturing** It should cover not only application-semantic behavior, but also timing behavior and shared resource usage behavior. Moreover, Different behaviors from different components or subsystems should be correlated.
- **Component Level with Application-Centric** We focus our application systems at the component abstraction level, instead of at the level of local procedure call or basic block or machine instruction. All behaviors captured should have a linkage to user-application level and become application-meaningful.
- **Reconfigurable Monitoring** Users should be able to specify their interest focus on different system aspects and different subsystems one at a time.
- **Minimize Monitoring Interference** Perturbation to the system under monitoring is intrinsic to any types of system monitoring, especially software-based monitoring. Minimizing interference is essential to capture and present authentic system behavior.
- **Scalable Monitoring System** Scalability is key to deal with large-scale complex systems. For example, the current HP LaserJet printing system has more than one million lines of source code, and is currently partitioned into tens of threads, with number of processes configurable.
- **Minimize Efforts Imposed to Users to Perform System Monitoring** Users are easy to get frustrated and reluctant to use the tool if the involved efforts are too demanded. The required efforts cover writing monitoring associated specification, modifying existing code implementation to meet monitoring requirement, and performing monitoring reconfiguration, etc.
- **Facilitate Users to Navigate and Inspect Monitoring Information at An Easy and Effective Manner** Large-scale complex systems inevitably produce enormous amount of raw monitoring data and post-processing data. We believe that unless we can provide an intelligent way of presenting monitoring related data, there is little hope of getting engineers to use the monitoring framework in real practice.

The monitoring framework is developed by leveraging various techniques which have been developed in the general distributed computing domain, including distributed object computing, distributed system performance monitoring, distributed system debugging. The approach that we adopt primarily is through source code instrumentation on the stub and the skeleton of each function whose function interface is defined in the IDL as part of the system artifact. The instrumentation process is automatic through a modified version of the IDL compiler. To capture more comprehensive system information, portion of component technology runtime is instrumented, such as thread library and memory management subsystem. The source code of the user-defined application suffers little modification. When the application system is running, the instrumentation probes produce log data locally. Ultimately, the log data scattered in the entire distributed system will be collected and then synthesized in a central repository. Unavoidably, instrumentation interference occurs at runtime. A selective instrumentation framework is developed to perform the instrumentation on only a particular system portion with a particular type of system behavior (one of the following: application semantics, timing, shared resource such as heap memory or CPU) in which users show current interest. The instrumentation can be complementarily performed at both compilation time and runtime.

The analyzer equipped with the monitoring framework performs the post-mortem analysis on the stored log data and present the analysis result to users through a front-end user interface. As a prototyping effort, in this report, we present the dynamic system call graph reconstructed for the system-wide causal tracing, and the end-to-end timing latency for function invocation at the component level.

The component technology infrastructure for our current system prototyping is CORBA [11], in particular, the version the CORBA called ORBlite [10]. With ORBlite, our current implementation has demonstrated the monitoring capability for systems running on a collection of networked machines with different native operating systems: HPUX 11.0, Windows NT and VxWorks. It is believed that the underlying developed monitoring techniques can be adopted quickly to other types of component technology, such as Microsoft COM and Java RMI.

The rest of the report is structured as follows. In Section 2, we present the developed instrumentation techniques, the core of our monitoring framework, for monitoring system-wide multi-dimensional system behaviors. Section 3 outlines the complete monitoring framework to collect and store, to analyze and present, the monitoring data collected at runtime from the deployed instrumentation probes. Section 4 provides in-depth monitoring data analysis and data visual presentation. Section 5 introduces the selective

monitoring techniques to reduce the amount of monitoring data and mitigate monitoring interference. Section 6 describes some related work. And finally in Section 7, some conclusions are drawn and some research directions are identified to improve and extend the developed monitoring framework. Appendix A shows a distributed application example used in this report.

2. Mechanism for Instrumentation

Instrumentation is a process to deploy monitoring probes to the system under monitoring and then from these probes to collect monitoring data when the system is running. CORBA facilitates function invocation across component boundaries at the abstraction level where detailed communication mechanisms including socket communication, shared memory, and local procedure call become transparent to users. In Section 2.1, we identify certain function invocation constraints on such function invocations imposed by our monitoring techniques. The core of our monitoring system is to trace system-wide semantic causality relationships. We explain two major types of relationships in detail in Section 2.2: function caller/callee relationship and thread parent/child relationship. Section 2.3 is devoted to explain in-depth how to place instrumentation probes to collect four different types of run-time monitoring data associated with causality relationship tracing, application semantics, timing latency and shared resource usage. We focus on how causality relationship is captured and traced from function call to function call, from thread to thread. The instrumentation occurs primarily at the stubs and the skeletons, which, however, are not sufficient to collect all the four types of monitoring data. Therefore, In Section 2.4, the instrumentation on the supplementary subsystems, including thread library and memory management subsystem, is introduced.

2.1 Function Invocation across Component Boundaries

CORBA supports different function invocation mechanisms and the associated threading mechanisms to meet different application purposes [11]. Our monitoring and characterization framework cannot support arbitrary CORBA-compliant function calls. We have the restrictions described as follows:

(1) **Function Invocation Synchrony** We assume all function calls are synchronous. That is, when the client invokes a function associated with a distributed object, it will be always blocked until the function returns. Asynchronous calls, e.g., one-way function calls, are precluded in this report. The blocking behavior of synchronous function invocation is analogous to procedure (function) call and return in traditional sequential programs. This assumption allows us to capture the deterministic and ordered causality relationship embedded in the function invocations.

(2) **Dynamic Function Invocation** In this report, only static function invocations are eligible. That is, the stubs and skeletons are generated at the compilation phase in advance. No dynamic invocation interfaces at the client side and dynamic skeleton interfaces at the server side are supported.

(3) **Collocation** Each function defined in the IDL always has a static stub and skeleton associated with. Function invocation always passes through its stub and skeleton. To improve system performance, when the client and the component object are located in the same process, i.e., they are collocated, the stub may be implemented intelligently to acquire the in-process interface function pointer, and then directly invoke the function from the interface function pointer. In this report, we assume such skeleton bypassing is turned off. The Orblite provides the Local Procedure Call (LPC) transportation protocol, and the standard IIOP protocol to transport the request and response messages between the client and the object (server). The LPC is chosen when the client and the object (server) are collocated, whereas the IIOP is applied when the client and the component (server) are separated in different processes. When the LPC is involved, the client and the component object function execution are within the same thread of control. No parameter marshalling occurs in the LPC.

At the user-application level, there is no restriction on how function invocation should be structured. Generally, function invocation can be composed in the following two ways:

- **Cascading** Function F_1 is called from the implementation body of function F. F_1 is immediately followed by the call from the same F to the second function F_2. F_1 and F_2 are said to form a *sibling* function relationship.
- **Nesting** Function F calls function G and function G subsequently calls function H. Function G does not complete and return until after function H completes and returns. F and G, or G and H, are said to form a *parent-child* function relationship.

Note that callback falls into the nesting category. Recursion follows into the nesting category as well.

In this report, the term "function call" or "function invocation" is synonymous with "function call across component boundaries" or "function invocation across component boundaries", unless stated explicitly. Two components can be collocated, or assigned to different processes of the same processor, or separated to different processes on different processors. Under the degenerated circumstance, the entire system can be configured into a single bulk thread, which is likely happened during the early system development phase to simplify debugging. Therefore, our monitoring framework can fallback to perform traditional local procedure call tracing as well.

2.2 Causality Relationships

Two types of causality relationships are associated with our monitoring framework: *function caller/callee relationship* and *thread parent/child relationship*.

The function caller/callee relationship is established when a function caller invokes the function callee located in the other component. The tracing of function caller/callee relationship in the system can cross threads, processes and processors. In a function implementation, cascading and nesting can be arbitrarily composed. Note that such relationship is traced per function invocation without limitation on call depth.

The thread parent/child relationship is formed when a thread *T* spawns a child thread *C* as the additional independent sequence of execution on behalf of *T*. Process-level parent/child relationship can be traced trivially by retrieving the current and parent process identifiers through the standard Unix system calls. According to our best knowledge, none of today's commodity operating systems directly exposes the thread parent/child relationship to user applications. In a multi-threaded application, such parent/thread relationship helps understand how shared resource usage propagate through a chain of such linked threads, since from a resource consumption perspective, what the child thread consumes during its work on behalf of its parent thread should be charged to its parent thread. In component-based systems, we further divide threads into two categories: *runtime infrastructure threads*, the ones which are directly spawned and managed by the component technology runtime (which is called ORB in CORBA), such as worker and listener threads; and *user-application threads*, the ones which are spawned during the execution of user-defined object function implementation. These two types of threads can be actually distinguished at runtime and we only keep track of the thread relationship for the user-application threads, as will be described in Section 2.3. In this report, the threads are referred implicitly to the user-application threads, unless otherwise stated explicitly.

By crossing the above two types of thread and function relationships, we have the following two types of derived relationships: *thread-invoke-function*, which occurs when a spawned thread on behalf of user-application further conducts function calls; and *function-spawn-thread*, which occurs when a thread is spawned out on behalf of the user-application during the execution of a function implementation body. Overall, we have the relationship diagram illustrated in Figure 1.

Causality tracing is to link all the function invocations and all the dynamically created user-application threads together to come up with a comprehensive picture of how a function call is propagated through the entire system. System characterization relies on such causality information to identify system-wide behaviors, e.g., how function end-to-end timing latency is contributed from the intermediate function calls, and how CPU consumption (or other shared resources) is propagated across function call boundaries and thread boundaries.

2.3 Instrumentation in Stub/Skeleton Indirection Layer

As part of the standard CORBA programming support, each function with its interface defined in the interface definition has a stub and the skeleton automatically generated by the IDL compiler. The stub and the skeleton create an indirection layer between the function caller (or the client), and the callee function implementation inside a component (or the server). By inserting probes at this indirection layer, all the different types of runtime behaviors associated with each individual function call can be monitored. Figure 2 shows the probe deployment in this indirection layer. The four probes are located at the start of the stub after the client invokes the function, at the beginning of the skeleton when the function invocation request reaches the skeleton, at the end of the skeleton when the function execution is concluded, and at the end of the stub when the function response is sent back to the stub and is ready to return to the original function caller. The sequence numbers in Figure 2 indicate the chronological order of probe activation when a function invocation is carried out.

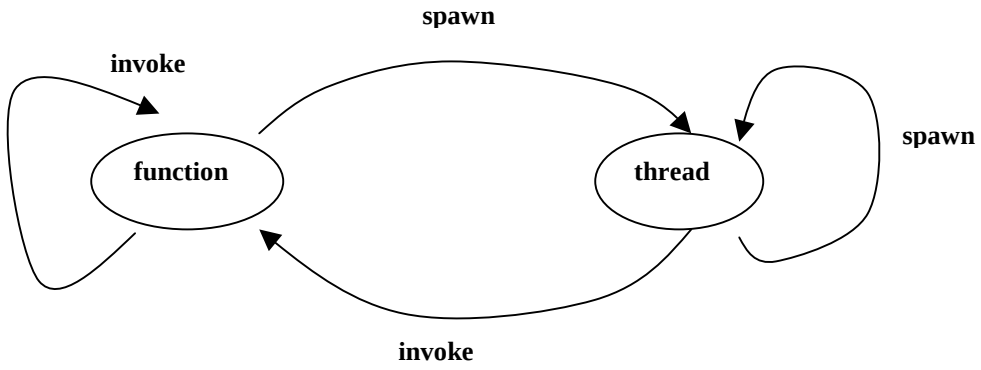

Figure 1: Thread And Function Relationship Captured at The Application Level in the Monitoring Framework

The stub and skeleton is the unique place to deploy instrumentation probes due to the following aspects:

- **Application-Awareness and Loyalty** It interlinks the user-application, and the component technology runtime. The functional behaviors, such as type and value of actual parameters, possible thrown exceptions, are explicit to the stub and the skeleton. The non-functional behaviors, such as timing latency, CPU consumption, are reflected back to the application more closely in the stub and skeleton than any other sub-layers inside component technology runtime infrastructure (e.g., transportation layer), because the stub and skeleton are placed right next to the user-defined code.
- **Automaticity** The stub and skeleton are automatically generated by the IDL compiler, which leads to the automatic instrumentation probe deployment.
- **Flexibleness** Migrating the component technology runtime from one vendor to the other, or from one type of component technology, such as CORBA, to the completely different one, such as COM or RMI, requires only the modification of the IDL compiler to produce the corresponding stub and skeleton. No modification to the well-developed component technology runtime is required.

Four different types of runtime monitoring data can be collected from the probes shown in Figure 2: application semantic information about each function call behavior, timing latency information about end-to-end timing latency associated with each function call, shared resource information concerning different shared run-time resource such as CPU and memory, and causality relationship information which enables the system-wide tracing of timing latency and shared resource usage. Each type of runtime information is collected in individual threads locally without global clock synchronization. The post-mortem system characterization assembles application semantic, timing latency, and shared resource information based on the reconstructed system-wide causal relationship.

Runtime information collection relies on the native operating system and even hardware platform configuration. Not all runtime information can be retrieved from the standard operating system calls, e.g., per-thread CPU consumption information. Not all runtime information can be equally collected with favorable accuracy on each platform and operating system. For example timing latency at microsecond level usually requires an on-chip high-resolution timer.

The primary instrumentation occurs in the stub and skeleton. We will describe other supplementary locations where the instrumentation is carried out in order to have much more comprehensive runtime information collected and then expose such information to the stub and skeleton. The additional instrumentation will be described in Section 2.4.

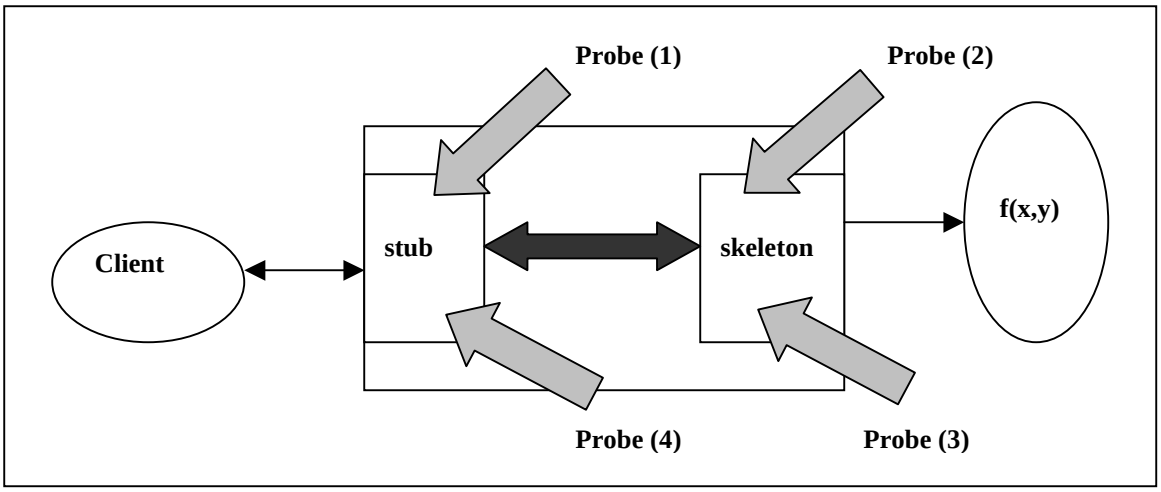

Figure 2: The Probe Deployment in the Automatically Generated Stub and Skeleton

2.3.1 Application Semantic Behavior Monitoring

The application semantic monitoring captures what have happened within a function call, i.e., input/output parameters, function return results, and possible exceptions, apart from the function name and its associated interface name. According to Figure 2, Probe 1 records function name and interface name, and the function input parameters. Probe 4 records the function output parameters and function return, as well as any exception that has occurred. Probe 2 and Probe 3 located at the skeleton side do not record any application semantic monitoring data.

2.3.2 Timing latency monitoring

Each probe obtains a time stamp from the local processor when the probe is initiated and when the probe is finished. All four probes shown in Figure 2 jointly record the timing latency log data. No global time synchronization is required. In our experimental system, on-chip high-resolution timer at the order of microsecond is available in the standard HPUX or Windows NT platform. In VxWorks, similar high-resolution timer is also supported from a third-party add-on component.

2.3.3 Shared resource usage monitoring

For shared resource usage monitoring, such as CPU consumption or heap memory usage, the probes are configured reminiscent of timing latency monitoring. All four probes shown in Figure 2 collect shared resource usage information when each probe is started and when each probe is finished. Different operating systems offer different degrees of accessibility to the CPU and memory usage information retrieval. Some operating systems might require additional libraries in order to collect and transform the available information from the operating system.

One type of shared resource is CPU. Not every operating system provides system call to expose accurate and per-thread CPU usage. In both HPUX 11.0 and Windows NT, both per-thread CPU information can be retrieved but with different resolution (HPUX at microsecond level and NT at millisecond level). In VxWorks, such information is not directly supported. However, VxWorks allows user-defined hook function to intercept thread entry/exit and context switch, which facilitates an instrumentation library to collect the per-thread CPU usage.

One other type of shared resource is heap memory for dynamic memory allocation/de-allocation requested by both a user application and the component technology runtime. It occurs when a library call of “malloc/free” (in C) or “operator new/delete” (in C++) is invoked. Such resource consumption is not usually exported by the current operating systems. A memory management subsystem may be located between the component technology runtime and the native operating system. The memory management subsystem intercepts each memory request and forwards the request down to the operating system. Upon receipt of the memory request, the memory management subsystem may gather per-thread statistical

memory usage information. This information can then be exported to the probes in the stub and skeleton.

2.3.4 Causality Tracing

The causality tracing covers the monitoring of both the function caller/callee relationship and user-application thread's parent/child relationship. With the instrumented stub and skeleton, in conjunction with the instrumented thread library, we are able to create a system-wide causal tracing tunnel completely invisible to the user application.

2.3.4.1 Function Caller/Callee Relationship Tracing

The basic idea of function caller/callee relationship tracing is to somehow annotate the same global identifier to two different threads: one represents the caller (the client), and the other represents the callee (the server), no matter whether these two threads are sharing the same processes or processor. Furthermore, such global identifiers will be propagated system-wide when further function calls are chained, from thread to thread, from process to process, and from processor to processor. By causally linking a collection of runtime thread entities through such global identifiers, we are then able to trace function caller/callee relationship system-wide. In short, we use global tagging to conduct global thread monitoring.

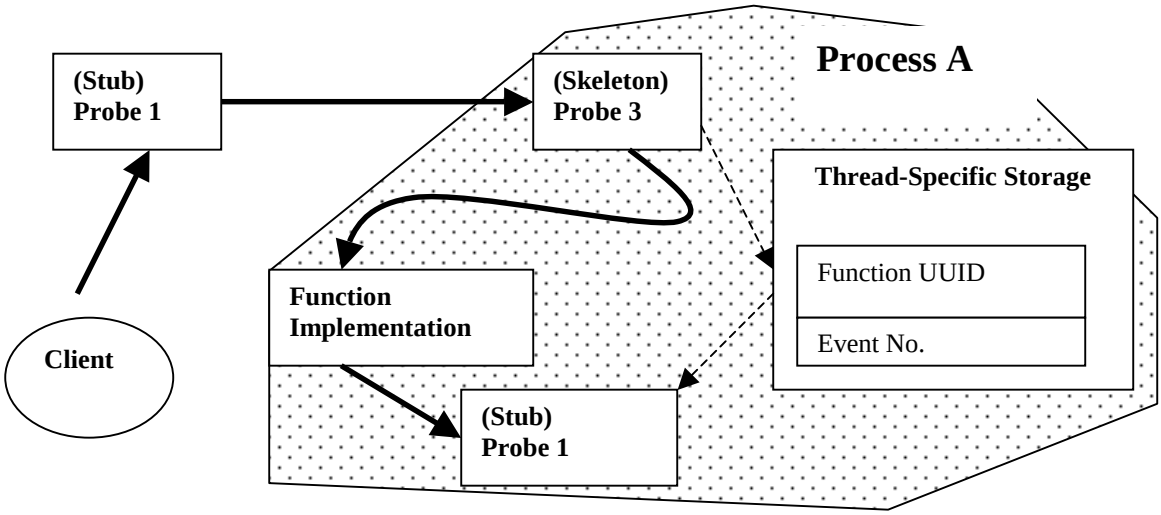

Figure 3: A Tunnel to Trace Function Caller/Callee Relationship System-Wide

Instead of taking over the control of the underlying operating system kernel (adopted by [13]), or the component technology runtime, we take the approach that requires only the instrumented stub and skeleton to fulfill system-wide identifier transportation. Figure 3 shows the virtual tunnel to realize such global tagging.

In order to have a global thread identifier, which is called the *Function Universally Unique Identifier*, or Function UUID, to become transportable system-wide and invisible to user-applications, tunneling is laid out completely by the IDL compiler. According to Figure 3, the tunnel consists of a private communication channel between the instrumented stub and the instrumented skeleton (shown in solid lines), and the transportation from the function implementation body down to the further child function invocation through a thread-specific storage (shown in dashed lines). To identify more detailed hierarchical function call structure reflected as sibling relationship and parent/child relationship, two additional log items besides the Function UUID are introduced: tracing event and the associated event numbers. We have four types of tracing events: *stub start*, *stub end*, *skeleton start*, *skeleton end*, each of which is produced when the corresponding probe is activated. The event number starts from zero when the initial function is invoked and gets incremented along the function chain when a tracing event is encountered. Figure 4 uses two

simple examples to show how the event chaining pattern can reflect the underlying function invocation pattern. The corresponding algorithm will be shown in Section 4.1.

Cascading
void main ( ) {
F(…);
G(…);
}
	F.stub_start→F.skeleton_start→F.skeleton_end→F.stub_end→
	G.stub_start→G.skeleton_start→G.skeleton_end→ G.skeleton_end

Nesting
void F(…) {
G (….);
}
void G (…) {
H(…):
}
	F.stub_start→F.skeleton_start→
	G.stub_start→G.skeleton_start→
	H.stub_start→H.skeleton_start→H.skeleton_end→H.stub_end→
	G.skeleton_end→G.stub_end→
	F.skeleton_end→F.stub_end

Figure 4: Event Chaining Pattern vs. Function Invocation Pattern

Causality tracing for the function caller/callee relationship works as follows. We define a *function-transportable* log data structure to incorporate two log items: the Function UUID and the event number. In order to transfer the function-transportable log data from the instrumented stub to the instrumented skeleton, the IDL compiler is modified such that the instrumented stub and the skeleton are generated in a way as if an additional parameter is introduced into the function interface with the type corresponding to the function-transportable log data and the direction of in-out. Following Figure 3, the function-transportable log data is packaged in the instrumented stub and then transported to the instrumented skeleton, where the event number is further updated and then recorded in a local log item. Subsequently, the updated function-transportable log data is stored into the thread specific storage. Such storage space is declared during the initialization phase of the monitoring process and is invisible to the user-application. If a child function is invoked during the function implementation, the function-transportable log data is retrieved from the thread specific storage at the beginning of the instrumented stub, gets updated and carried further down to the chain. Thread specific storages are only accessible to the function executions sharing the same thread of control. Simultaneous invocations to the same callee do not lose their own Function UUID's and event numbers.

In short, by combing the private communication channels enabled by the component technology runtime, and the thread specific storage to bridge such private channels, the function-transportable log data can be carried along the full function call chain system-wide. And such causal tracing channel is achieved without the awareness from the user-application.

The transportation of function-transportable log data is a two-way process. Figure 3 only shows the forward function call chain process. When each function in the chains get returned, such log data is updated by Probe 4 and then Probe 2 when the function is returned back to the caller. Function-transportable log data is also transported from one function to its immediate follower under the sibling relationship. This is guaranteed when the previous function's termination and the immediate follower function's invocation are always in the same thread, such that the log data is transported via the designated thread specific storage described before.

2.3.4.2 Thread Parent/Child Relationship Tracing

To complete tracing the causal relationships described in Section 2.2, we create a separate tunnel to capture the thread parent/child relationship for the user-application threads. Such tunnels are constructed

jointly by the instrumented stub/skeleton, and the instrumented thread library.

We define a *thread-transportable* log data structure to incorporate two different log items: a marker and a thread identifier. Such thread-transportable log data is stored in a thread specific storage space. The marker of *white* indicates the current thread is a user-application thread, while the marker of *black* indicates the current thread is a runtime infrastructure thread. The marker is always set to be black when the execution is within the stub body. The marker is turned into white in the skeleton right before the execution control is switched to the callee, and then be resumed to be black again when the callee is returned. Thread identifiers are managed by the thread factory located in the instrumented thread library.

Log Data	**Data Type**	**Instrumentation Location**	**Explanation**
Function UUID	unsigned char [20]	Stub/Skeleton	The universally unique identifier to identify the function call chain
Event	enumeration	Stub/Skeleton	An item belongs to set = {stub start, stub end, skeleton start, skeleton end}
Event Number	unsigned long	Stub/Skeleton	To record the ordering of the event encountering along the function call chain
Marker	boolean	Stub/Skeleton	The indicator to distinguish the thread execution of component technology runtime from user-application
Thread Identifier	unsigned long	Stub/Skeleton	The process-wide identifier for threads created by the overall monitored system
Local Function Identifier	unsigned long	Stub/Skeleton	The process-wide identifier for the function stub or the function skeleton
Function Container Identifier	unsigned long	Thread Library	Identical to local function identifier in the stub/skeleton
Self Thread Identifier	unsigned long	Thread Library	Identical to thread Identifier in stub/skeleton
Parent Thread Identifier	unsigned long	Thread Library	The identifier of the parent thread

Table 1: Log Data to Trace System-Wide Causal Relationships

In the instrumented thread library, right before a child thread is spawned during the callee execution, the thread-transportable log data is retrieved and packaged with the normal thread creation required parameters. Right after the child thread is created, the thread-transportable log data is retrieved, the marker of the child thread is set corresponding to the retrieved data. The thread identifier of the retrieved data now corresponds to the parent thread identifier. When the marker is white, the current self thread identifier and its parent thread identifier are recorded in a instrumentation thread table. Such a thread identifier pair uniquely determines a pair of parent and child in a process.

2.3.4.3 Causal Relationship Associated Probes

As the summary for monitoring causality propagation, the log data required to trace the system-wide causality relationships are listed in Table 1. The local function identifier records the unique number associated with each stub or skeleton local to its running process. It belongs to a part of thread-transportable log data in the actual implementation. When a local function identifier is retrieved in the instrumented library, it becomes a function container identifier. Both local function identifier and function

container identifier are introduced to capture the function-spawn-thread relationship and the thread-invoke-function relationship, as will be described in more detail in Section 4.1. All such log data are locally logged.

To reduce tracing overhead for frequent Function UUID generation, a Function UUID is partitioned into two parts. The inter-process portion is generated once for each process during the process initiation stage, which has 16 bytes. The intra-process portion is an in-process counter, which has 4 bytes. This counter gets incremented when a new Function UUID is requested from some instrumented stubs.

In order to attribute each thread with the corresponding host machine and process, some additional process wide information needs to be separately recorded, which includes the machine name, the process identifier assigned by the operating system, and a *Process Universally Unique Identifier*, or Process UUID, to unique identify a process in the distributed application under monitoring.

2.4 Overall Instrumentation Probe Deployment

The instrumentation probes are primarily deployed in the instrumented stubs and skeletons, from which application-semantics and function caller/callee relationship directly captured. In order to have more comprehensive runtime system information exposed to the stubs and skeletons, we need other instrumentation probes statically deployed independent of the application under monitoring. From the preceding subsections, we know that in order to have the thread parent/child relationship tracing, we need the thread library instrumentation. Memory access related statistical information requires the instrumentation on memory management subsystem. Per-thread CPU information requires some runtime library installation.

All probes are designed to be thread-safe for log data consistency in a multithreading environment.

3. The Monitoring Framework

In Section 2, we explained the instrumentation mechanisms to deploy instrumentation probes, capture and correlate runtime monitoring information. To achieve a comprehensive monitoring framework, besides the compiler-assisted instrumentation probe deployment, other facilities are necessary to collect the scattered raw monitoring data, to analyze the collected data, and to present the analysis result. In this section, we describe the data collector, the back-end analyzer and the front-end user interface to achieve such objectives. The complete monitoring framework is shown schematically in Figure 5.

3.1 Data Collector

The Data collector deals with monitoring data accumulation and their storage in a permanent location.

The data accumulation in performed in two stages. One happens at runtime, when the system is running. The other one happens offline when the system has finished running. To reduce logging overhead, the runtime data accumulation is further divided into a two-layer hierarchy. A per-process in-memory buffer is created during the process initialization to hold monitoring data records. Buffer sizes can be customized by users before the system is started. The buffer gets flushed into the local file system asynchronously each time the buffer is full, or when the system finishes running. Each process has a designated log file in its lifetime. The offline data accumulator gets notified each process's log file location during the system initialization phase. A remote file access mechanism is applied to collect all the log files scattered around the entire system into a central location when the system finishes running.

In-memory log data is stored in a machine-dependent data representation. Log data needs to be presented in a machine-neutral format as the system can potentially span through multiple processors. Log data captured in in-memory buffers has to be reconstructed later in the analysis phase, even though log data may be transferred and stored in different types of media and processed by different machines. Such machine independent translation can be fulfilled through the marshaller and demarshaller provided from the corresponding component technology. Marshalling occurs to in-memory log data and then the local file system stores the resulted binary stream. Demarshalling is invoked later when log data needs to be inspected and analyzed.

The preferred ultimate storage location for raw monitoring data is a relational database because relational databases offer the following capabilities:

- **Synthesize Log Data** All log data captured in different corners of the system are fused into such a permanent storage where data values are stored and their semantic relationship are preserved;

- **Easy Back-End Analyses** The database-equipped SQL query language provides a powerful and optimum log data sorting and searching capability to reduce back-end analysis implementation efforts;
- **Simplify Front-End system Behavior Inspection** The database-equipped SQL query language also simplifies system information retrieval issued from the front-end user interface.

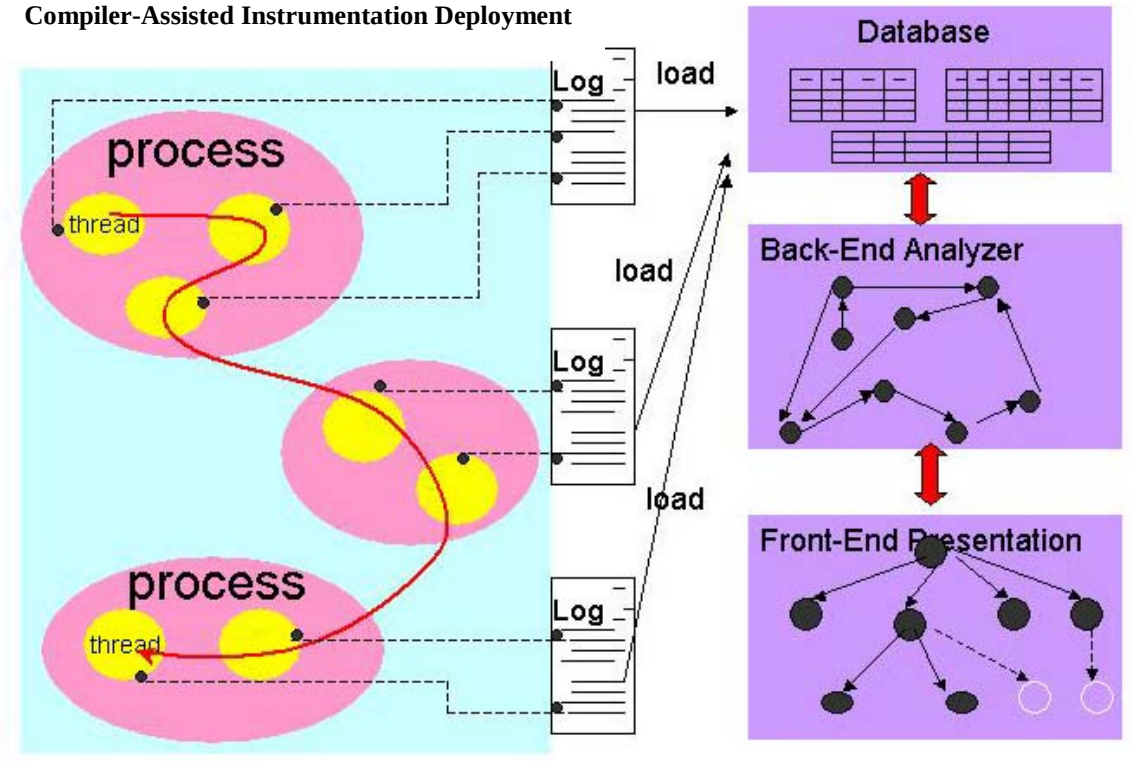

Figure 5: The Complete Component-Based System Monitoring Framework

3.2 Back-End Analyzer and Front-End User-Interface

The backend analyzer provides a set of system characterization tools to help users understand how the system behaves, identify performance bottlenecks of the system, provide impact estimation on a change of a component to the entire system.

We will report dynamic call graph reconstruction and timing latency characterization in more detail in Section 4 and leave CPU consumption characterization to be reported in a subsequent technical report.

The front-end user-interface allows users to browse the system characterization report produced from the back-end analyzer. From the analysis result report, users may issue requests to the backend analyzer for further system behavior information not immediately accessible in the characterization report.

The front-end user-interface is actually an interactive information exploratory system. For large-scale distributed systems in which millions of function calls might pass by even at the component-level, how to present the system characterization results to users in an effective way is challenging. As the first step to tackle this problem, in Section 4 we show how a commercial hyperbolic tree viewer can be applied to produce promising visual effect.

4. Characterization: Dynamic System Call Graph and Timing Latency

This section presents two developed tools to help understand system behaviors. Section 4.1 covers the first tool, which is to construct the system-wide dynamic call graph. The dynamic call graph is handled by a

visual display system, which can potentially handle large applications with millions of function calls invoked at runtime. Section 4.2 describes the second tool, a timing latency analyzer for function calls at the component level. Section 4.2 also presents some timing latency measurement data from the example system described in Appendix A. The measurement data exposes monitoring interference, which leads to the necessity of selective system monitoring in Section 5.

4.1 Dynamic System Call Graph

In Section 2.3, we presented the instrumentation mechanism to perform system-wide causal tracing for the function caller/callee relationship and the thread parent/child relationship. In this section, we show how to reconstruct the dynamic system call graph based on the collected causal tracing associated monitoring data.

As indicated in Section 2.2, causality relationships are divided into two categories: the function caller/callee relationship, and the thread parent/child relationship. There exist other two types of derived relationships: function-spawn-thread relationship, and thread-invoke-function. These four types of semantic relationships form the dynamic call graph of the system. The function caller/callee relationship reconstruction is the following discussion focus.

As described in Section 2.3, a function call chain is with a unique Function UUID. Following the function call chain, for each involved function invocation, events are logged and event numbers are incremented for each event appearance. Moreover, the event repeating patterns uniquely manifest the underlying function invocation patterns. The algorithm to reconstruct function caller/callee relationship is shown in Figure 6.

```
procedure CONSTRUCT_FUNCTION_CALLER_CALLEE_RELATIONSHIP():
  begin
        UUID ← QUERY_FOR_UNIQUE_UUID();
        for each ID g in UUID
          begin
             E ← SORT_EVENT(g);
             T ← SCAN_EVENT( E );
          end
  end
```

Figure 6: Reconstruct Function Caller/Callee Relationship from Monitoring Data

QUERY_FOR_UNIQUE_UUID performs a query on the overall monitoring data and identifies the set of unique Function UUIDs created for the system. *SORT_EVENT* conducts the second query to sort by ascending order the events associated with the function invocations which have the designated Function UUID. These two procedures can be easily realized by some standard SQL queries. *SCAN_EVENT* performs the scanning of the sorted event sets and reconstructs the calling hierarchy based on the state-machine shown in Figure 7.

According to Figure 7, each state is augmented with the name of the invoked function. A transition from one state to another may be used to generate a parsing decision. A decision of "in progress" between two states indicates that the two adjacent log records come from the normal execution of a function call. A decision of "child function starts" indicates that subsequent log records come from a child function of the current log record function. A decision of "child function returns" indicates that subsequent log records come from a parent function of the current log record. A decision of "continued with sibling function" indicates that the next log record comes from a function immediately following the return of the function associated with the current log record. Figure 7 only shows how to determine the function caller/callee relationship when there is no runtime execution failure in the system, such as a crash. There is an additional "abnormal" transition state (not shown) that is called if the adjacent log function records do not follow any of the transition patterns shown in Figure 7. If that happens, the analysis will indicate the failure and the analysis is restarted from the next log record.

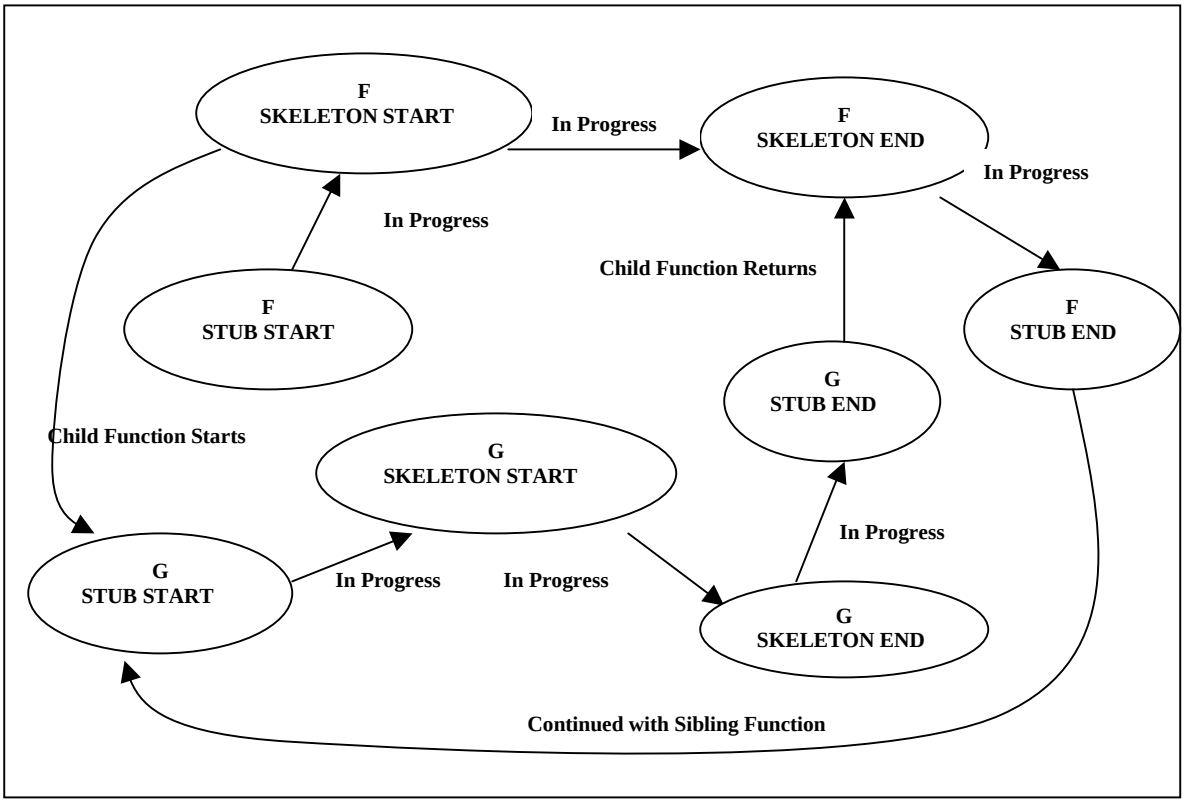

Figure 7: State Machine to Reconstruct System-Wide Function Caller/Callee Relationship

In terms of computational complexity, *QUERY_FOR_UNIQUE_UUID* requires $O(N \log N)$ expected time where N is the total number of function-associated log records. Each function invocation generates 4 such log records. Thus N=4*M where M is the total number of function invocations carried out in the system. *SORT_EVENT* requires $O(L \log L)$ expected time where L is the length of the event chained associated with a particular Function UUID. *SCAN_EVENT* is just a linear parsing which only requires $O(L)$ expected time. Thus, the total time cost C(function) is:

$$\begin{aligned} C(\text{function}) &= O(N \bullet Log(N)) + \sum_i O(L_i Log(L_i) + \sum_i O(L_i) \\ &\le O(4M \bullet Log(4M)) + \log(L_{\max}) O(4M) + O(4M) \\ &= O(MLogM) \end{aligned}$$

Equation 1

In contrast to the function caller/callee relationship determination, the thread parent/child relationship reconstruction is relatively straightforward. In a process, with the available log data, a thread T_1 is a parent to thread T2 if thread T1's self thread identifier is identical to the parent thread identifier of thread T2. The identifier comparison only occurs in individual processes. Under the worst case that the entire system is a bulk process, the total time cost is $O(N_T^2)$ where N_T is the total number of threads spawned in the entire system. Since dynamically creating threads at runtime is rather expensive, especially in the embedded application domain, typically N_T will not be large. Both self thread identifiers and parent thread identifiers have been introduced in Section 2.3.4 and shown in Table 1.

The two derived relationship types can be determined in a straightforward manner. With the available log data, if Function F_1 spawned thread T_6, thread T_6's function container identifier will be identical to Function F_1's local function identifier. It can be determined that thread T_1 invoked Function F_9 if T_1's self thread identifier is identical to Function F_9's thread identifier. The thread-invoke-function relationship is

local to each individual process, in worst case where the entire system is a single bulk process, it requires $O(N_T M)$, which is applied to the cost of determining function-sawn-thread relationship reciprocally. Both local function identifier and function container identifier have been introduced in Section 2.3.4 and shown in Table 1.

Therefore, the total estimated cost to construct system wide causal relationship is:

$$O(MLogM + MN_T + N_T^2)$$

Equation 2

The function and thread based relationship reconstruction results in a collection of trees, because different function call chains can be independently established in the system. Besides function invocation nodes and thread instance nodes, auxiliary nodes are introduced to combine the independent trees into one single tree. The following are the auxiliary node types:

- **Virtual Main Node** is a unique top node, which resembles the top function "main" for a program written in high-level programming language like C/C++, Java, etc., both for sequential and concurrent applications.
- **Global Thread Node** is a node holding a call hierarchy reconstructed from a particular function call chain, as described in Section 4.1. A global thread node is the child of either the virtual main node, or one of the thread nodes.
- **Segment Node** is a node holding a call hierarchy reconstructed from a particular function chain. Unlike global thread nodes, a segment node does not represent a complete call chain. A segment node is created when parsing of the event chain described in Section 4.1 is terminated immaturely and restarted. A segment node is the child of a global thread node.

Therefore, a *dynamic system call graph* is a tree that holds function nodes and thread nodes following the four different types of causal relationships with some necessary auxiliary nodes to combine isolated sub-trees.

For a large-scale distributed application, a dynamic call graph can potentially consist of millions of nodes. Conventional visualization tools based on planar graph display do not have the capability to present such enormous amount of graph nodes and help user effectively navigate and inspect the graph. Fortunately, a commercial graph package [5] based on hyperbolic tree viewing demonstrates promising capability. A portion of dynamic call graph from this graph package is shown in Figure 8, which comes from the example system described in Appendix A. The largest dynamic call graph that we have produced in our experiments is about 16,000 tree nodes from one particular subsystem of a large-scale HP embedded system. In the experiment, by navigating such a dynamic system call graph with tens of thousands of nodes, software developers identified some code implementation inefficiency at the component level within minutes.

The dynamic call graph captures every instance of function invocation across component boundaries in the entire system and preserves the complete call chains. It is unlike Gprof [3] and Quantify [12], which can only keep track of caller/callee relationship with call-depth of 1. Defined at the component level, the call chain that we achieve herein is exactly the call path described in [4]. As a result, the scenarios and techniques introduced in [4] for call path profiling under the single-processed environment can be naturally extended to the multithreaded distributed system with our dynamic call graphs. Section 4.2 is to demonstrate such an analysis extension.

4.2 End-To-End Timing Latency and Its Monitoring Intrusiveness

End-to-end timing latency, or timing latency, is considered as one of the most important non-functional behaviors. We define the end-to-end timing latency in a component-based system as the total elapsed time to fulfill a function invocation request started by the client and then eventually returned to the client. The calculation of timing latency of an invocation of function *f* is rather straightforward:

$$L(f) = T(\mathrm{Probe}_{f,1,end}) - T(\mathrm{Probe}_{f,4,start})$$

Equation 3

The involved probes have been shown in Figure 2. The associated second index in Equation 3 denotes the

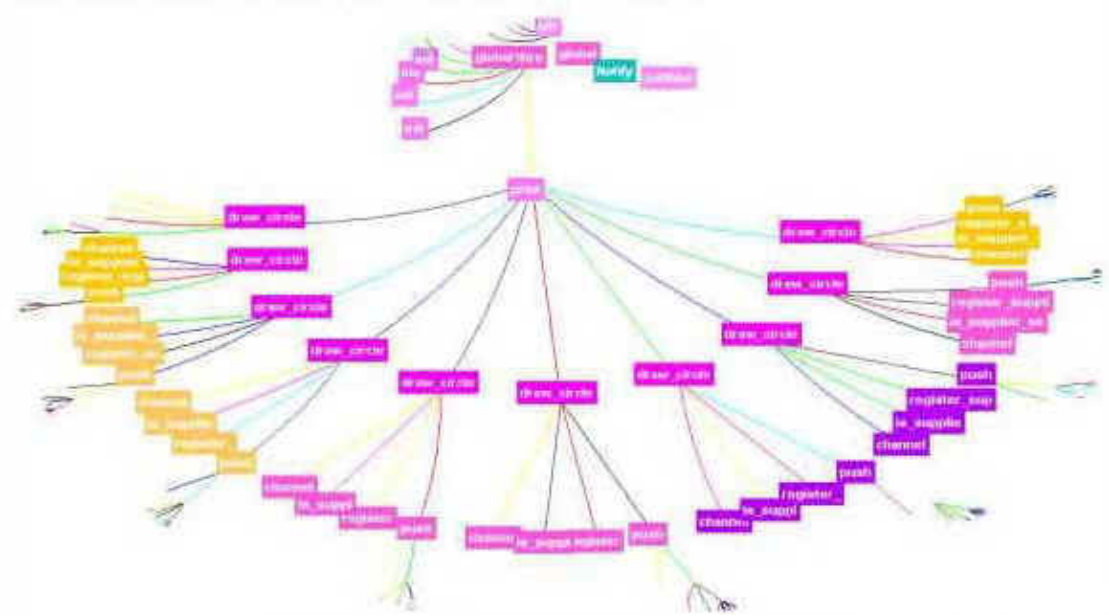

Figure 8: Example of System Dynamic Call Graph Shown in Hyperbolic Tree Viewer (from Inxight's Star Tree SDK)

chorological order number indicated in Figure 2. The third index in Equation 3 refers to whether the time stamp T is captured at the start or the end of the probing activity.

Note that in Equation 3, no timing latency monitoring is required in function skeletons. Therefore, from an interference reduction viewpoint, the timing latency monitoring associated probes in skeletons can be disabled.

Also note that because we are dealing only with synchronous function invocations herein, if function F has child function invocations involved, the latencies of its child function invocations contribute certain portion of F's latency.

Timing latency information can be directly annotated to each function node in the dynamic system call graph as node attributes. By navigating and inspecting the graph, users can easily obtain the insight about the propagation of timing latency in the system. Alternatively, a more concise statistics report on a group of function invocations can be presented in a tabular format. The non-shaded columns of Table 2 show the end-to-end timing latency measurement results from the example system described in Appendix A.

Unavoidably, instrumentation interferes the original system and potentially alters its system behavior. Such intrusiveness becomes more significant in multithreaded systems due to intrinsic system non-determinism. Unlike shared resources such as CPU or memory, whose interference can be localized to individual processes or processors, timing latency is boundary-less across threads, processes, and processors. As a result, the respective monitoring interference is superimposable across threads, processes, and processors.

The root of instrumentation interference is because the instrumentation probes consume CPU, which otherwise should be allocated to other computing activity belonging to the non-instrumented system. Following the control flow of individual function *i*'s invocation, the interference factors can be easily identified as the following:

$$P_i = \sum_{k=0}^{3} I_{k,i} + \left(L_{p,i} + L_{s,i}\right)$$

Equation 4

In Equation 4, $I_{0,i}$, $I_{1,i}$, $I_{2,i}$, $I_{3,i}$ represents the monitoring activity happens at one of the stub start, skeleton start, skeleton end, and stub end probes respectively, with the index number following the probe numbering shown in Figure 2. $L_{p,i}$ ($L_{s,i}$) denotes the computation devoted to construct, to marshall/demarshall, and to transport the transportable-log data both at the function forward and function

return phases at the stub (skeleton) side.

Equation 4 covers only the interference called *intra-function-call-chain interference*. Such interference affects only the corresponding function invocation, or the function invocations belonging to the same function call chain. Within a system, multiple function call chains can coexist and therefore incur *inter-function-call-chain interference* as well. Suppose Function F is issued in Processor 1, and the associated function call chain holds the invocation of Function F' carried out by Processor 2. Another Function G is issued in Processor 3, and its associated function call chain holds the invocation of Function G' carried out by Processor 2 also. F' and G' can be interleaved with each other in Processor 2. Therefore, the monitoring interference for F' brings about intrusiveness to G, so does G' to F, even though from the application semantic perspective, F-associated function call chain is independent of G-associated function call chain. Such inter-function-call-chain can be described as:

$$Q_i = \sum_{\substack{j \in R \\ R \cap S \neq 0}} \tilde{H}_j$$

Equation 5

where R represents the function invocations including function *i* itself and the function call chain started from *i*. S represents all the function invocations which have ever preempted the execution of function *i*'s call chain. $\tilde{H}_j$ denotes the portions of *I* or *L* interference in Equation 4 from function *j* to function *i*.

Table 2 also shows monitoring intrusiveness. To measure the latency of each function *f* under the non-instrumentation situation, we manually insert two time-stamp retrieval functions at the places identical to where the instrumentation associated IDL compiler will deploy $\text{Probe}_{f,1,start}$ and $\text{Probe}_{f,1,end}$. For each function *f* under such manual measurement, no other probes exist in the system except these two time-stamp retrieval functions. The monitoring data are collected in one single run for the instrumented system. However, to reduce the intrusiveness to minimum in the non-instrumented system, monitoring data are collected only for one single function in each system run. In order to have convincing comparison between multiple system runs, each application component is implemented with a deterministic functionality.

The actual interference shown in Table 2 is defined as the relative difference between the instrumented measurement and non-instrumented counterpart. The example system under monitoring is described in Appendix A. The functions selected in Table 2 are representative of the interference range coverage. We have observed the interference of less than 60% without any post measurement compensation. The interference from this particular example is similar to the number reported by IPS-2 [9]. The highest interference in this example comes from the collocated function calls, which includes *UserApplication::notified*, *Render::deposit_to_queue*, *Render::retrieve_from_queue*. The reason is that $I_{0,i}$, $I_{1,i}$, $I_{2,i}$, $I_{3,i}$ in Equation 4 become much more significant in collocated function calls compared to non-collocated function calls, since inter-process communication and marshalling occur only in non-collocated function invocations.

The interference factor is varied from application to application. For example, as long as function *UserApplication::notified* increases its execution time, the interference shown in Figure 2 would decrease. It is believed that at the component level, each function more likely encapsulates a collection of local procedure calls. Therefore, the computation devoted to the non-instrumented portion overwhelms the interference more likely, in contrast to other monitoring and characterization systems like IPS-2 [9], Paradyn [8] and Gprof [3], in which the instrumentation targets are local procedure calls. At the scale of interference shown in Table 2, even though each function invocation's latency cannot be measured precisely, it still can present valuable information about the value range, and its relative scale compared to other function invocations, similar to the merits provided by the Gprof-like tools.

Compensation techniques presented in [7, 15] are helpful to overcome partially the monitoring interference by the back-end analyses. How effectiveness such techniques can be for the component-level latency measurement is under the current investigation.

Function	Average (msec)	Standard Deviation (msec)	Average (msec)	Standard Deviation (msec)	Interference
EngineController::print	1.535	0.158	1.484	1.288	3.4%
DeviceChannel::is_supplier_set	1.865	0.054	1.236	0.025	50.9%
IO::retrieve_from_queue	10.155	0.094	9.636	0.094	5.4%
GDI::draw_circle	95.066	10.206	85.866	11.342	10.7%
RIP::notify_downstream	13.831	2.377	11.557	0.381	19.7%
RIP::Insert_Obj_To_DL	2.502	0.141	1.879	0.127	33.2%
IO::push_to_queue	13.626	0.298	13.580	2.887	0.3%
UserApplication::notified	0.418	0.04	0.282	0.099	48.3%
Render::deposit_to_queue	0.529	0.097	0.358	0.010	47.8%
Render::render_object	7.138	2.104	6.284	0.074	13.6%
Render::retrieve_from_queue	0.501	0.040	0.318	0.010	57.6%

Table 2: Timing Latency From Automatic Monitoring Framework and Its Manual Counterpart

Generally, monitoring intrusion is intrinsic and cannot be completely eliminated. However, it is possible to mitigate such interference and make the measurement result more authentic to the original system behavior by introducing some new instrumentation features into the existing monitoring paradigm. Such objective is explored in Section 5.

5. Selective Monitoring

System monitoring based on a fixed configuration of probes generated at the system compilation phase may gather a significant amount of log data. This may impact the performance and behavior of the system under monitoring. Furthermore, concurrent monitoring probes can potentially interfere with each other, and therefore affect the accuracy of the monitoring result. Such monitoring interference happens when a singe behavior monitoring is performed in the system, as has been demonstrated in Section 4 already for timing latency. The interference becomes more significant when more than one system behaviors are under monitoring. For instance, application semantic monitoring probes unavoidably introduce monitoring interference to both CPU and timing latency monitoring. Therefore, it may be desirable to reduce the data produced by the probes to only the portion significant to an underlying monitoring. Additionally, the monitoring may be further streamlined by allowing probes to be dynamically enabled and disabled at runtime, based on the monitoring being performed.

The selectiveness can be tackled in the following two aspects: monitoring type selection and component selection. There are two schemes to fulfill the monitoring selectiveness: compilation time and runtime. We will briefly describe the compilation-time selectiveness in Section 5.1 and then focus on runtime selectiveness in Section 5.2. In Section 5.3, we present some preliminary measurement result from the example system for selective latency runtime monitoring.

5.1 Compilation-Time Monitoring

Users can specify what the subset of components, and what is the particular type of behavior that they are interested in either through a monitoring specification file or simply just through a command line. The compiler can then deploy the necessary probes into the corresponding stubs and skeletons.

Since the instrumentation is pair-wise at the stub and the skeleton, the instrumented and non-instrumented components can coexist within the system without having the problem of function parameter type mismatching due to the introduction of the additional transportable log parameter in the instrumented components (see Section 2.3.4). Although the function invocation chain will not be severed, the result will only manifest the function invocations which are instrumented.

For the components which are not instrumented, their object function implementation, in addition to the corresponding stubs and skeletons, are just treated as a collection of normal local procedure calls and transparent to the runtime monitoring system. Therefore, if these non-instrumented components are invoked directly or indirectly by a function belonging to an instrumented component, their runtime behavior will be ceiled by the instrumented components and belongs to the ceiling instrumented components.

5.2 Run-Time Selective Monitoring

Compilation-time monitoring only supports one fixed monitoring configuration. The second monitoring configuration requires a new system compilation, linking, and deployment. For large-scale systems, this undoubtedly is a time-consuming process. In this section, we introduce the run-time selective monitoring, to allow users to change the monitoring specification on-the-fly at runtime. Run-time selective monitoring requires all components and all system behaviors that are within the potential candidate list to be instrumented at compilation time. Therefore, compared to compilation-time monitoring, runtime selective monitoring have a compromise between flexibility and low memory footprint.

5.2.1 Run-Time Monitoring Type Selection

The monitoring type selection can be chosen from one or more of application semantic behavior, timing latency, shared resource usage. In this section and the ongoing Section 5.2.2, without explicit indication, the causality tracing is always turned on, such that the monitoring result can be causally linked system-wide. In our discussion, we further assume that the shared resource usage only covers CPU and heap memory, the two most important resources in practice.

5.2.1.1 Framework and Protocol

Each monitoring type can be turned on/off. In total, we have the following event set: *application semantics on*, *application semantics off*, *latency on*, *latency off*, *CPU on*, *CPU off*, *memory on*, *memory off*. An event channel is to allow any of such events to be published and subscribed. The event channel and the related event push/pull action to the monitored application are described in Figure 9.

In Figure 9, the selective monitoring framework includes a monitoring coordinator that is started before any client and server processes are created, and is always running. The monitoring coordinator creates an event channel. The monitoring coordinator then creates an event supplier thread, which can receive user commands from input interfaces (such as the keyboard), and can push the event to the event channel. After the creation of the monitoring coordinator, client and server processes are started. A server or client process then spawns an event consumer thread. The event consumer thread pulls the event from the event channel and performs subsequent actions. The sequence of involved actions is indicated in Figure 9 as well.

Beginning from the start state, where the monitoring coordinator starts, the state transition diagram shown in Figure 10 describes a state protocol for selective monitoring types. When a *Latency_ON* event is published by the monitoring coordinator and is subscribed to by each consumer thread, the monitoring system selects the latency measurement. This also applies for *CPU_ON* and *Memory_ON*. In the selective monitoring framework, a single monitoring type can be selected fro each monitoring session. A monitoring session is defined as a time interval for the transition from *start* to *normal*, or for the transition from normal back to normal. After the monitoring coordinator issues the monitoring off event of *Latency_OFF*, *CPU_OFF*, or *Memory_OFF*, the monitoring type is terminated and the system goes to the normal state within a finite amount of time. From the normal state, the system can transition to any other type of monitoring, but not without returning to the normal state. Each event receiver globally sets the monitoring flag to be on or off which is then visible to the entire process with respect to the designated corresponding monitoring type.

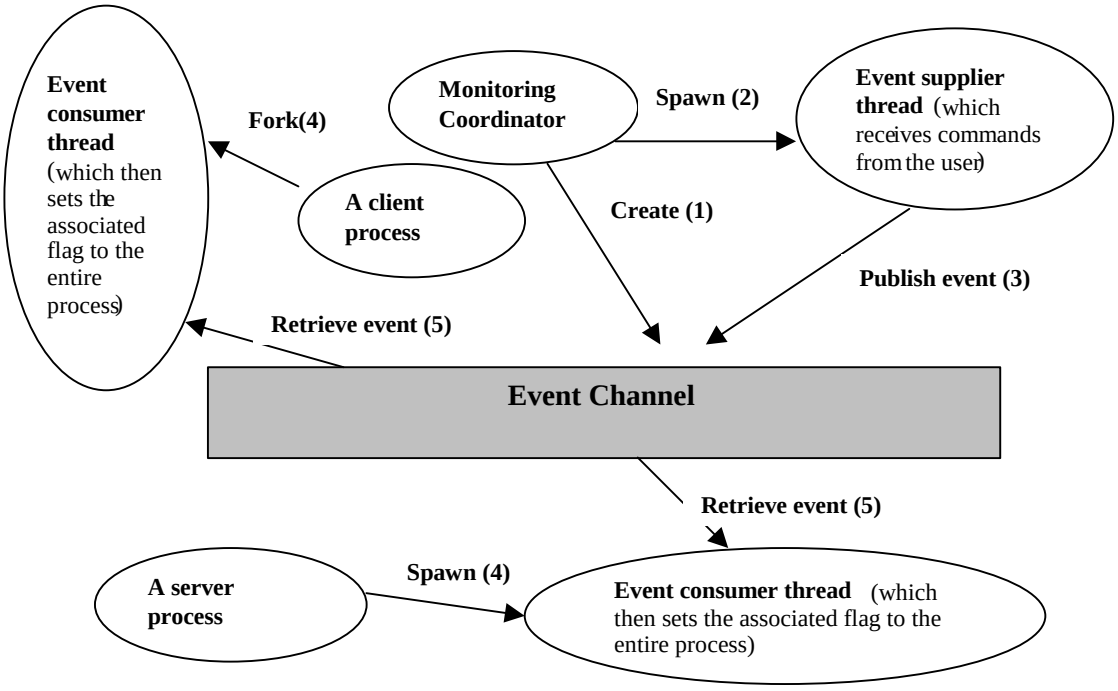

Figure 9: Monitoring Type Selection In Selective Monitoring Framework

As the result, a timing latency monitoring, a shared resource usage monitoring, and an application semantics monitoring are capable of being selectively enabled and disabled.

5.2.1.2 Monitoring Data Consistency

In normal operation, it is possible that a function invocation may occur right before a client or server process receives an off event. This may trigger a set of subsequent function invocations and may therefore cause the issuance of a set of dynamically created threads throughout the system under monitoring. As a result, it will be difficult, if not impossible, to determine the reach or duration of a normal state. It is imperative that these functions and threads execute and are monitored, even though the off event has been allowed to execute and the off event has been set. Therefore, in some instances the normal state should not be immediately reached after the off event is set.

To cope with the fact that the reaching of the normal state cannot be absolutely determined, the monitoring and the analyzer are designed so that either the monitoring result can be guaranteed to be consistent or that the failure to reach the normal state is detected and reported to the user. If the failure is reported, the inconsistency can be avoided in the next run of system monitoring by waiting until the monitoring reaches the normal state.

The consistency is defined by two criteria. Criterion 1 is that no required monitoring is discontinued by a state transition, and Criterion 2 is that no required monitoring is interfered with by a state transition.

Criterion 1 is always guaranteed by the technique called *causal overriding*. When a function is invoked by anther function, or a thread is created by another thread, the causality relationship is continuously tracked, including the function caller/callee relationship and the thread parent/child relationship. After the off event has been received, if a function is still called from another active function requiring the monitoring action, or if the function is contained in the thread whose parent thread requires the monitoring activation, then this function locally turns on the respective monitoring. The criterion may be set via an application event or via a debugger hook in the actual implementation.

An interference happens when the instrumentation probes of the monitoring framework process compete for a shared resource with the application under monitoring. Criterion 2 is automatically satisfied for application semantics monitoring, as no shared resource usage is concerned. Criterion 2 can be

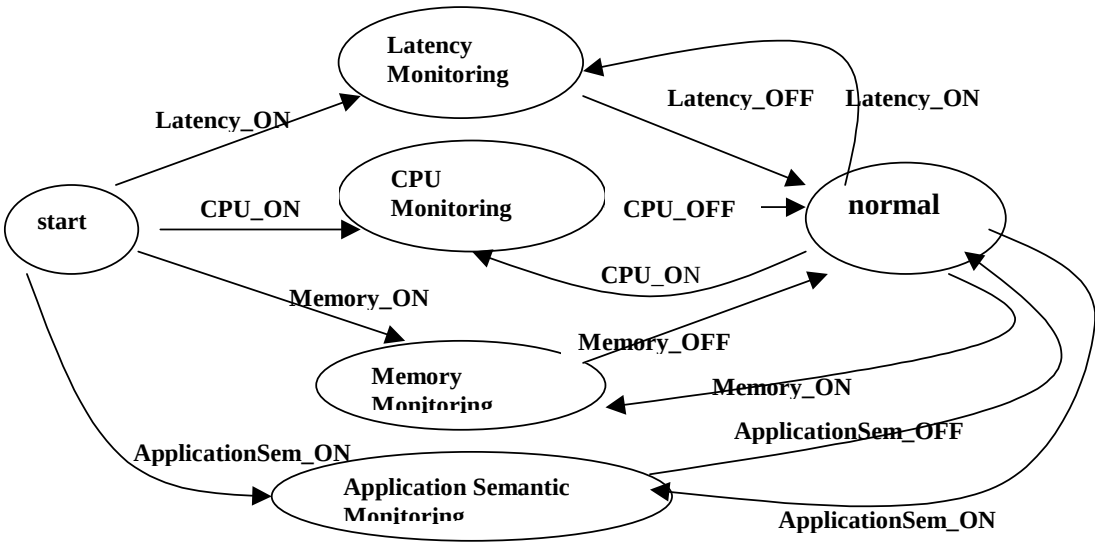

Figure 10: State Transition Diagram to Select Monitoring Type

compensated for by monitoring of the shared resource usage such as CPU and memory usage, as these types of monitoring are thread-based. For shared resource usage monitoring, if the monitoring occurs simultaneously in two different threads then no interference between threads can occur. The only interference left is intra-thread. Since the execution inside a thread is always sequential, the resource spent on the probes (i.e., the interference), can be recorded in log records and be subtracted out later by the back-end analyzer.

For the monitoring of timing latency, the interface cannot be easily measured and compensated when there exist multiple functions with different Function UUID's simultaneously executed on the same processor. The difficulty exists because the timing latency interference system-wide, as indicated in Section 4.3. In this case, if monitoring data captures the start and end of the function invocation in both the stub and the skeleton, it may be determined whether a particular function is interleaved with another function invocation. If the interleaving does occur, it will become part of the report presented to the user, and the user can take action to reduce or eliminate the interference by employing on-the-fly on/off probing as described in Section 5.2.1.1.

Therefore, different monitoring types may be controlled by a user or application in order to minimize impact on the system and confine the monitoring framework only to the degree that is truly desired by the user. For example, the user may wish to turn one or more of the monitoring features off at time when a large amount of processing time is required. Likewise, at times where the user desires more information on the runtime execution of the application, the user may turn on more of the monitoring types in order to collect additional data. For example, the user may choose to enable more monitoring to understand the root cause if the system is experiencing a large number of exceptions.

The instrumentation probing in thread library and in the memory management subsystem may be configured to be permanent. However, they may be implemented so that they can be turned on or off to some degree in order to cope with the probes inside the stubs and the skeletons and therefore to minimize the monitoring interference.

5.2.2 Run-Time Selective Component Monitoring

The theme of selective monitoring type can be employed to achieve a selective component monitoring, wherein a user can specify that a monitoring operation be turned on or off for a component. For selective component monitoring, an event may be encoded as {host, interface, function, on} or as {host, interface, function, off}. The interface and function fields specify the monitoring function and the associated interface. The host filed indicates that only the function invoked from that particular processor will be monitored. Any one of the first three fields can be absent and replaced by a wildcard operator (such as *). The wildcard operator may be interpreted to be any host, any interface, or any function. When such an

event is broadcast system-wide through the framework, the monitoring system can enable and disable specific functions.

Figure 10 is applicable to selective component monitoring with the involved events changed to what have been just described above. The causal overriding is still applied to meet Criterion 1. Generally Criterion 2 can be met by interference avoidance through the feedback of the back-end analyzer, similar to the recommended treatment for timing latency monitoring described previously. Moreover, the selective component monitoring scheme and the monitoring type selection scheme can be superimposed to form a comprehensive selective system monitoring. Consequently, a user may control both monitoring aspects in order to tailor the monitoring operation to the needs of the user.

5.3 Experimental Result

Function	**Average** (msec)	**Standard Deviation** (msec)	**Average** (msec)	**Standard Deviation** (msec)	**Interference**
EngineController::print	1.491	0.145	1.484	1.288	0.5%
DeviceChannel::is_supplier_set	1.878	0.066	1.236	0.025	51.9%
IO::retrieve_from_queue	10.212	0.067	9.636	0.094	6.0%
GDI::draw_circle	93.581	6.674	85.866	11.342	9.0%
RIP::notify_downstream	12.955	0.333	11.557	0.381	12.1%
RIP::Insert_Obj_To_DL	2.698	0.458	1.879	0.127	43.6%
IO::push_to_queue	13.567	0.307	13.580	2.887	-0.1%
UserApplication::notified	0.441	0.051	0.282	0.099	56.3%
Render::deposit_to_queue	0.541	0.100	0.358	0.010	51.1%
Render::render_object	6.475	0.078	6.284	0.074	3.0%
Render::retrieve_from_queue	0.502	0.039	0.318	0.010	57.9%

Table 3: Measurement Data from Selective Monitoring

A preliminary selective monitoring framework is built to show how significantly the monitoring interference can be reduced. In this framework, only timing latency is supported. The causal overriding only happens to the function caller/callee relationship propagation. In our experiment, we choose a particular function interface for monitoring in one system run. The causal overriding turns on the associated function call chain for each function under monitoring. For the example system, as shown in Table 3, since most of the computation is carried out sequentially in different pipeline stages, there is no significant inter-function-call-chain interference observed. And since the causal overriding is enabled, the intra-function-call-chain interference does not get reduced. Therefore, the advantage of selective monitoring is not demonstrated in this particular example in which only timing latency is supported. However, in other testing examples where dynamic user-application threads are created frequently, we do observe the reduction of inter-function-call-chain interference even when timing latency is only supported.

6. Related Work

It is beyond the scope of this report to provide a comprehensive survey on the field of tool support for

monitoring, characterizing and debugging distributed systems, especially for multithread distributed systems. In this section, we present only the tools relevant to distributed debuggers and distributed system profilers that appear to have the closest relationship to what we are exploring. Such tools can be classified as: application-level-logging tools, binary-rewriting tools, debugger-per-thread tools, network/OS-message-logging tools and instrumented virtual-machine tools.

Application-level-logging tools are essentially the use of macros embedded in application code that produce printf()-like logs. The principal disadvantage of these tools is that the source code had to be written with logging in mind (i.e., the developer has to consciously add a log at an important event). The AIMS [15] is such a source re-writing system to address performance tuning for parallel program execution based on message passing. Log entry points are inserted directly into the user-defined source code as well as some associated runtime support library source code. The instrumentation points can be decided by the user at the system compilation phase, and the associated probes can be turned on/off at system initialization but not dynamically at runtime. No threading concepts exist in AIMS. AIMS demands global clock synchronization to partially compensate monitoring overhead for performance index evaluation. Causal linkage between processes (i.e., why an event occurred) happens only at message transportation layer to pair *send* and *receive* events between two processes.

Another variant on the application-level-logging techniques are binary-re-writing techniques. Quantify [12] is a version of a binary-rewriting tool to provide function-based CPU consumption and propagation along individual processes. It re-writes the application code by inserting counting instructions at the basic blocks of the binary program. Quantify does not work on multi-process-ed applications and cannot find causal linkage across processes/threads. The Paradyn [8] is a binary rewriting system for parallel program performance measurement. The binary level instrumentation introduce great flexibility in terms of runtime information capturing but it also raises platform dependency issues. The threading extended version of Paradyn [16] provides certain performance evaluation at thread-level, but no causal linking between different threads has been addressed. In terms of logging, both AIMS and Paradyn cannot automatically filter log message at runtime following on-the-fly user-specification.

Debugger-per-thread tools, such as Microsoft Visual Studio and GDB [14], provide a debugging window per process in a distributed system. Such debugging tools are playing a dominant role in developing distributed systems currently. It is undoubted that for traditional sequential programs, which are single-threaded and single-processed, they are extremely powerful. However, for multithread applications, their capability is limited. There are two key disadvantages to these tools. The first one is the screen real-estate taken up in any large scale system. The second is the inability to correlate between threads and between processes (i.e., it is not possible to tell what caused one process or thread to enter a particular state or who sent the message).

Network/OS-message-logging tools measure system performance and network traffic by intercepting operating system events and network packets. Examples of such tools are ThreadMon from Sun [1] and Desktop Management Interface from Hewlett-Packard [2]. These tools are particularly useful for identifying amount of CPU consumed by a process or bandwidth issues. However, these tools have great difficulty turning operating system call or the network packet into application meaningful events. That is, usually one just gets a packet of bytes and no easy way to interpret why the packet of bytes was set or what the packet is trying to cause to happen. Or, in terms of CPU consumption, there is no clue about what actually is computed in a thread or process and why it happens.

Finally, in the instrumented virtual machine approach, there are systems like JaViz [6] for monitoring applications that span multiple Java virtual machines. It has been report to successfully produce dynamic call graph like ours. However, they differ in the following two aspects. First, Their call graph covers not only the function calls based on RMI, but also local procedure calls. Second, it only deals with function caller/callee relationship but no thread parent/child relationship. The principal disadvantage of this approach is that it is confined to the Java virtual machine and does not allow intermixing multiple languages or platforms.

7. Conclusions

A framework to monitor distributed and multi-threaded systems based on component technology is developed. We have shown the following major features of our developed monitoring infrastructure compared to the existing tools and development environments targeting distributed and multi-threaded application development:

- Application-centric and thread-based multi-dimensional behavior monitoring and characterization in an integrated framework, which includes application semantics, timing, and shared resources, along with the global semantic causality tracing.
- System-wide capturing of shared resource consumption and timing propagation due to causality tracing of threaded applications;
- By leveraging component technology, the monitoring is independent of the hardware platform, operating system, the communication protocol. Moreover, our approach does not require a globally synchronized clock provided in the network. Therefore, our runtime monitoring is applicable to multi-threaded, multi-processed and multi-processor-ed applications.
- Automatic and selective probe insertion through an IDL compiler at the compilation phase and flexible probe activation/deactivation at runtime. By having the flexible probe configuration, the user can eliminate or reduce the perturbation of the system behavior due to the monitoring activity, as well as the interference between different types of monitoring simultaneously performed in the system, besides to reduce the amount of monitoring data produced at runtime

Current system characterization support includes a dynamic system call graph constructor and timing latency analyzer. An equipped web-based interface from a commercial hyperbolic tree viewer allows users to inspect component interaction with the annotated end-to-end latency information.

The monitoring framework serves as the start of the development of a tool family for assisting in understanding system behavior based on component technology such as CORBA, Microsoft COM and RMI. We are currently at the stage of porting the developed techniques into the HP LaserJet printing system whose component technology runtime is based on the Microsoft COM standard. To enhance the functionality and quality of the monitoring framework, the future effort will be spent on improving monitoring system scalability, providing richer characterization support to understand system behavior, and enhancing front-end user interface's capability for system behavior information's effective inspection and navigation.

8. Acknowledgements

Thanks to Keith Moore from Imaging Systems Lab for his suggestions, Emiliano Bartolome from Imaging Systems Lab for helping prepare the example application, and Ming Hao from Software Technology Lab for providing the hyperbolic tree viewer.

9. References

[1] B.M. Cantrill and T.W. Doeppner, "ThreadMon: a tool for monitoring multithreaded program performance", *Proceedings of the Thirtieth Hawaii International Conference on System Sciences*, pp. 253-65, 1997.

[2] DMI web site: http://wojo.rose.hp.com/dmi.

[3] S. Graham, P. Kessler and M. McKusick, "Gprof: a call graph execution profiler," *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices*, Vol. 17, No. 6, pp. 120-126, June 1982.

[4] R. J. Hall, and A. J. Goldberg, "Call Path Profiling of Monotonic Program Resources in Unix," *Proceedings of the Summer 1993 USENIX Conference*, pp. 1-13.

[5] Hyperbolic Tree Toolkit, http://www.inxight.com.

[6] I. Kazi, D. Jose, B. Ben-Hamida, C. Hescott, C. Kwok, J. Konstan, D. Lilja, and P.Yew, "JaViz: A Client/Server Java Profiling Tool," *IBM Systems Journal*, Volume 39, Number 1, 2000, pp. 96-117.

[7] A.D. Malony, D.A. Reed, H.A.G. Wijshoff, "Performance measurement intrusion and perturbation analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol.3, no.4, 1992, pp. 433-50.

[8] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam and T. Newhall, "The paradyn parallel performance measurement tool," *IEEE Computer* 28(11), pp. 37-46, Nov. 1995.

[9] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski, “IPS-2: The second generation of a parallel program measurement system,” *IEEE Trans. Parallel Distrib. Syst*., vol. 1, no. 2, pp. 206-217, Apr. 1990.

[10] K. Moore, and E. Kirshenbaum, “Building evolvable systems: the ORBlite project,” *Hewlett-Packard Journal*, pp. 62-72, Vol. 48, No.1, Feb. 1997.

[11] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.4*, Oct. 2000.

[12] Quantify, http://www.rational.com/products.

[13] F. D. Reynolds, J. D. Northcutt, E. D. Jensen, R. K. Clark, S. E. Shipman, B. Dasarathy, and D. P. Maynard, “Threads: A Programming Construct for Reliable Real-Time Distributed Computing,” *Internal Journal of Mini and Microcomputers*, Vol. 12, No. 3, 1990, pp. 119-27.

[14] R. M. Stallman, and R. H. Pesch, *Debugging with GDB, The GNU Source-Level Debugger*, Fifth ed., for GDB version 4.17, Boston, Mass.: Free Software Foundation, Apr. 1998. Also available at http://www.gnu.org/manual/gdb-4.17/gdb.html.

[15] J. C. Yan, S. R. Sarukkai, and P. Mehra. "Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit," *Software Practice & Experience*. April 1995. Vol. 25, No. 4, pages 429-461.

[16] Z. Xu, B. P. Miller and O. Naim, "Dynamic Instrumentation of Threaded Applications," *7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, Georgia, May 1999.

Appendix A–Example System

The example system is to simulate the behavior of a printing pipeline, starting from a user's printing command, and ended with a hardcopy printout from a printer or an error message reported to the user. The total system consists of the following 9 component interfaces:

- **User Application** to simulate user's printing command;
- **GDI** to simulate Graphical Device Interface in Windows, which consists of a set of graphical object creation and manipulation operations, such as *draw_circle*;
- **Channel** to represent the abstraction for different alternative channel connections between the computer and the printer. It can be a parallel channel, or a serial channel, or through local area network;
- **Channel Administrator** to simulate network management for deciding which channel is allocated to establish the connection between the computer and the printer;
- **IO** to simulate the buffering of data coming from the channel;
- **RIP** to simulate the Raster Image Processing to translating GDI objects to an internal display list;
- **Render** to simulate the conversion from the display list into a bit map format;
- **Engine Controller** to simulate how the physical printer engine controls mechanics and electronics to print physical dots onto the paper.
- **Diagnostic Subsystem** to simulate the printer behavior reporting and the failure cause identification.

We also have four different exceptions to represent abnormal conditions occurred in the printing process. They are *DeviceChannelNotFound* thrown by the channel administrator to the GDI; *QueueOverflowed* and *QueueEmptied* thrown by the IO to the channel; *OutOfMemory* from the render to the RIP; and *PaperJam* from the engine controller to the diagnostic subsystem.

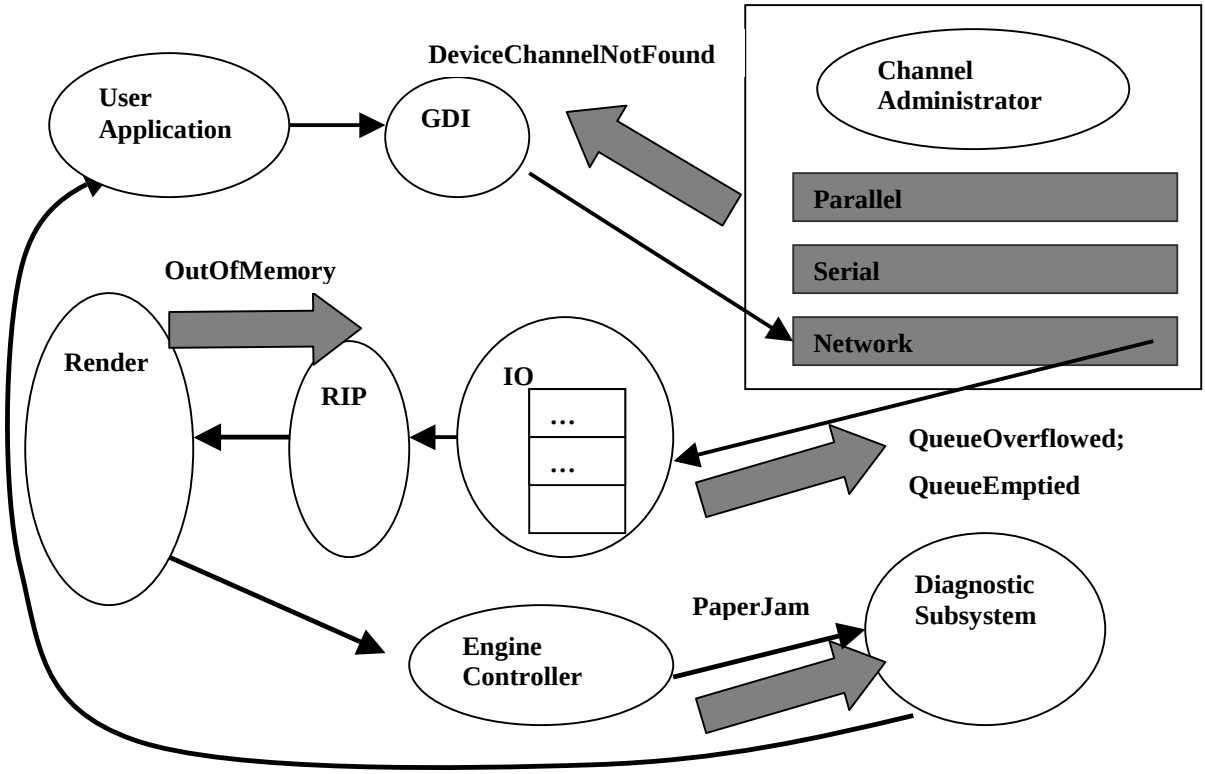

Figure 11: Example System

There are totally 11 component object instances in the system, as shown in Figure 11. Most of the

component function implementations are just to realize the object interaction structure. There is no in-depth behavior simulation. In terms of component object deployment, to collect the measurement data used in Section 4 and 5, totally there are four processes assigned into two machines, one with Windows NT and one with HPUX 11.0. Process ***A*** is a client process in which a print command is issued. Process ***B*** consists of the Device Channel Administrator along with the channel objects under management, the UserApplication, and the Diagnostic Subsystem. Process ***C*** consists of the GDI and the IO. Process ***D*** consists of the RIP, the Render and the EngineController. Process ***A*** and ***D*** are in Windows NT. Process ***B*** and ***C*** are in HP UX 11.0. When the engine controller reports a PaperJam to the diagnostic subsystem, it spawns a user-application thread to work on the reporting.