Monitoring and Characterization of Component-Based Systems with Global Causality Capture

Jun Li
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
junli@hpl.hp.com

Abstract

Current software development techniques and tools lack the capability to characterize function call chains in multithreaded and distributed applications built upon component technologies like CORBA, COM and J2EE. The root cause is that causal linkage information necessary to trace end-to-end call chains is private to each vendor's runtime and often unavailable for logging or analysis. We propose and demonstrate a mechanism for maintaining and correlating global causality information of component-based applications, and using this information to expose and characterize function call chains and their associated behaviors in such multithreaded and distributed applications. Our approach relies on a global virtual tunnel facilitated by the instrumented stubs and skeletons. This tunnel maintains and correlates causal information throughout the end-to-end call chains spanning threads, processes and processors. As a result, monitoring data captured locally can be correlated and system-wide propagation of timing latency and CPU utilization becomes perceivable.

1. Introduction

The successful development, maintenance and evolution of a software system require a full understanding of different system behaviors, including application-level semantic behavior (such as which function calls which function and why an exception is raised), timing behavior (such as the end-to-end timing latency of a function call), and shared resource usage behavior (such as CPU utilization under a constrained budget).

To expose complex semantic behavior, monitoring and characterization tools and debuggers have been developed. Many of these tools perform well on single-threaded and single-processed applications; however, when the system is multi-threaded, or worse, when the system is deployed across different processes, located in different processors, current tools are not able to monitor and analyze these sophisticated applications. Execution profiler GPROF [3] (popular for CPU bottlenecks pinpointing) merely reports the callee-caller propagation of CPU utilization within the same thread context. Well-accepted debuggers like GDB and Microsoft Visual Studio are unaware of how threads interact and call frames propagate beyond thread and process boundaries. The root cause is that the tools that extract causal linkage information are tightly coupled to the specific runtime infrastructure used for deploying the application. Often, gathering such causality information is either impossible, or the retrieved information is insufficient to determine the full function call chains. Moreover, such causality information cannot span across different vendors' runtime infrastructures. As a result, linking semantic behavior information about individual threads, processes and processors becomes a manual process that is error-prone, inefficient, and unmanageable for large-scale distributed applications.

To better understand system-wide behavior for multithreaded and distributed applications, we have developed a runtime monitoring framework for component-based systems to capture both execution behavior and propagation of semantic causality spanning remote component-level function invocations, and the associated characterization tool to analyze and present the captured system behavior.

Our approach exploits source code instrumentation on the stubs and skeletons. For components defined with the IDL interfaces, we use an IDL compiler to automate the instrumentation in the stubs and skeletons, without modifying the user-defined application code. A virtual tunnel is constituted in the instrumented system via the stubs and skeletons. This tunnel facilitates the semantic causality information concerning function invocation to be propagated across threads, processes and processors. It therefore establishes the correlation among the scattered monitoring data captured by the stub and skeleton probes.

The monitored behaviors include timing latency and shared resource utilization like CPU. The behavior characterization is performed on the collected log data. Rather than just offering basic query processing to present raw monitoring data (reminiscent of *printf*), the off-line tool constructs the dynamic system call graph based on the captured system-wide causality information. This graph exhibits dynamic system execution in terms of component object interaction. Timing latency and CPU consumption at the component level are computed. Moreover, with such unveiled causal relationship, latency and CPU utilization propagation across threads, processes and processors becomes perceivable.

Compared with the existing distributed performance monitoring and distributed debugging techniques, our work makes the following contributions:

- **Component Level with Dissimilar Remote Invocation Infrastructures** Our focus is at the component abstraction level and the developed technique is feasible to different runtime infrastructures following different remote invocation standards (e.g., COM and CORBA). Component object programming and invocation model, and multithreading strategies provided by underlying runtime infrastructures, impose a set of new challenges not presented in prior monitoring work both at component level and at more granular level (e.g., local procedure call and basic block);
- **Global Causality Capture and Propagation** Causality tracing is enabled by a virtual tunnel crossing threads, processes and processors. This tunnel can be seamlessly propagated in the distributed application even when different remote invocation infrastructure standards are involved and bridging occurs between these infrastructures (e.g., the CORBA/COM hybrid systems);
- **Application-Level Behavior Capture and Correlation** Our work brings together a collection of available technologies that effectively capture application-level semantic behavior, and application-level timing and resource utilization behavior. The captured global causality correlates all these behaviors that belongs to different subsystems and components to constitute a holistic view of component interactions.

Our current tool implementation explores two different component technology infrastructures: a version of CORBA [14] called ORBlite [13], and an embedded infrastructure similar to COM [11]. The paper proceeds as follows. In Section 2, we present our monitoring technique to both CORBA and COM. Section 3 provides monitoring data analysis. Section 4 shows experimental results from example systems, including a commercial

large-scale embedded system. Section 5 describes some related work and summaries are drawn in Section 6.

2. Monitoring Mechanism

In this paper, the semantic causality relationship refers to the *function caller/callee relationship*. A richer set of causality relationships such as thread parent/child relationship [8, 9], will not be explored here.

The caller/callee relationship is established when a function caller invokes its callee located in the other component following either a cascading (e.g., F calls G_1 and then calls G_2) or nesting (e.g., F calls G and then G calls H) pattern. Such relationship can cross threads, processes and processors, and always manifests itself throughout the call chain without limitation on the call depth. Note that callback and recursion both produce nesting calls.

The CORBA oriented terminologies are used herein. However, the presented monitoring technique is general to both CORBA and COM distributed applications. Also, the term "function call (or invocation)" is synonymous with "function call (or invocation) across component boundaries", unless stated explicitly.

2.1. Stub/Skeleton Based Instrumentation

The IDL compiler automatically generates a stub-skeleton pair for each function defined in the IDL interface. The stub and the skeleton create an indirection layer between the function caller or client, and the callee function implementation of a component. Probing at this indirection layer captures runtime system behaviors. The probe deployment is schematically shown in Figure 1. The four probes are located respectively at the start of the stub after the client invokes the function, at the beginning of the skeleton when the invocation request reaches the skeleton, at the end of the skeleton when the function execution concludes, and at the end of the stub when the response is sent back to the stub and ready to return to the client. The sequence numbers labeled in Figure 1 indicate the chronological order of probe activation along the invocation path. The IDL compiler automates the necessary stub and skeleton instrumentation.

In this section, a simple component object invocation model is considered with the following restrictions:

- Function calls are synchronous;
- No support for Dynamic Invocation Interface (DII) or Dynamic Skeleton Interface (DSI);
- No collocation optimization for in-process component object invocation. Such optimization often allows the stub to intelligently locate the object interface pointer directly and therefore bypass the skeleton;

- Multithreading servers follow only the thread-per-request policy [18].

Section 2.2 will release all but the DSI/DII constraints to support flexible component object programming and multi-threading strategies.

Monitoring of Different Behavior Aspects

The probes in Figure 1 can collect four different behaviors: application semantics about each function call behavior (input/output/return parameter, thrown exceptions), end-to-end timing latency associated with each function call, shared resource usage, and semantic causality information to correlate system-wide application semantics, timing latency and resource utilization. All runtime behavior information is recorded individually by probes without coordination and global clock synchronization.

In this paper, we focus only on timing latency and CPU utilization, along with semantic causality capture. Application semantics capture is primarily useful for application debugging and testing. Note that timing latency and CPU utilization rely on the underlying native operating system and even hardware configuration. Not all runtime information is available with standard operating system calls. For example, per-thread CPU consumption is available in HPUX version 11 but not earlier versions. And not all runtime information can be collected with equally favorable accuracy on different platforms. Timing with microsecond accuracy usually requires an on-chip high-resolution timer.

To perform timing latency or CPU utilization monitoring, each of the four probes retrieves local time stamps or per-thread CPU usage, once when the probe is initiated and once when finished. No global time synchronization is required. To reduce interference, latency and CPU utilization probes are not activated
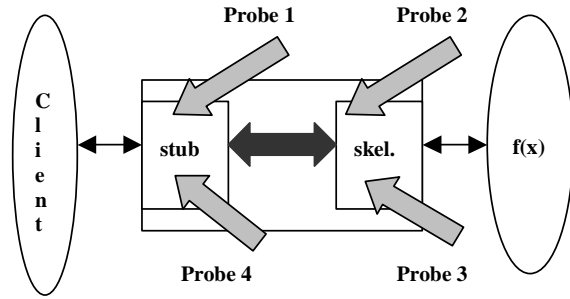

Figure 1. Probe deployment in the stub and skeleton

simultaneously. However, they always perform causality capture.

Causality Capture

The basic idea of causality capture is to annotate a global identifier to two different threads, each of which represents the caller and callee respectively, independent of their physical location (they may degenerate into a single thread). Such global identifiers have to be propagated system-wide when further function calls are chained, from thread to thread, from process to process, and from processor to processor, i.e., causally linking a collection of runtime thread entities.

Rather than manipulating the underlying operating system or runtime invocation infrastructures, we adopt the approach that requires only the instrumented stubs and skeletons to transport system-wide global identifier (called *Function Universally Unique Identifier*, or *Function UUID*), completely transparent to user applications. Figure 2 schematically shows the constituents of the underlying virtual transportation tunnel.
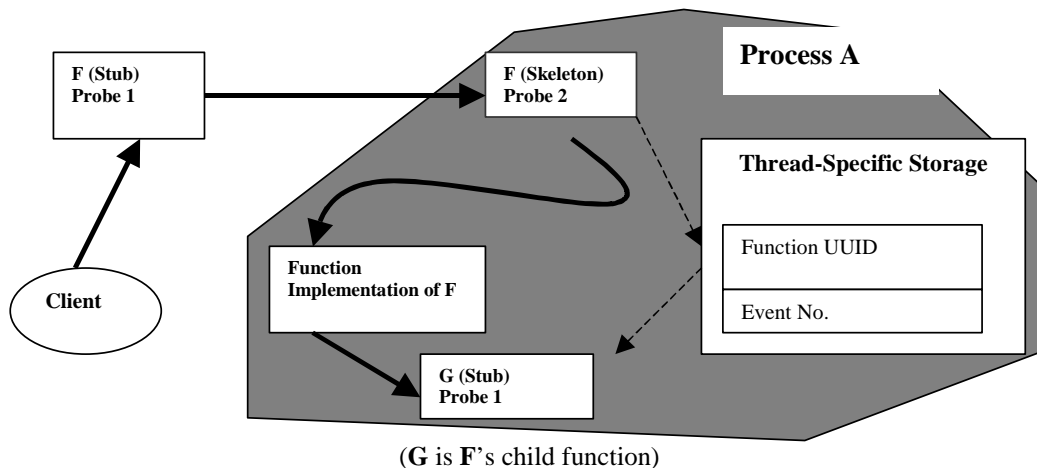

(**G** is **F**'s child function)

Figure 2. The virtual tunnel to trace causality relationship system-wide

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 determine the hierarchical call structure identified as *sibling* relationship and *parent/child* relationship (shown in Table 1), two additional items besides the Function UUID are introduced: tracing event and the associated event number. Tracing events include: *stub start*, *stub end*, *skeleton start*, *skeleton end*, each of which is recorded when the corresponding probe is activated. Event numbers are incremented along the function chain at each time a tracing event is encountered. Table 1 also shows how the event chaining patterns determine the call structures.

Figure 3 shows the *Function-Transportable Log* (FTL) data incorporating the Function UUID and the event sequence number. It is the FTL that propagates along the tunnel. To transfer the FTL from the instrumented stub to the instrumented skeleton, the IDL compiler generates the instrumented stub and skeleton in a way as if an additional in-out parameter is introduced into the function interface with the type corresponding to the FTL. The FTL is packaged at the stub and then

Table 1. Event chaining patterns and function invocation patterns

Sibling
 	
void main(){
    F(…);
    G(…);
}
	F.stub_start→F.skel_start→F.skel_end
	→F.stub_end
	→G.stub_start→G.skel_start
	→G.skel_end→G.stub_end

Parent/Child
 	
void F(…) {
    G(…);
}
void G(…) {
    H(…);
}
	F.stub_start→F.skel_start→G.stub_start
	→G.skel_start→H.stub_start
	→H.skel_start→H.skel_end
	→H.stub_end→G.skel_end→G.stub_end
	→F.skel_end→F.stub_end

Therefore, the private transport channel underneath stubs and skeletons, and the TSS to bridge such private channels, collectively update the FTL along the system-wide call chain transparent to user-applications. The tunnel in Figure 2 only shows the forward chain advancing. At the subsequent call return phase, the FTL is updated by Probes 3 and 4 in sequence. The FTL is also
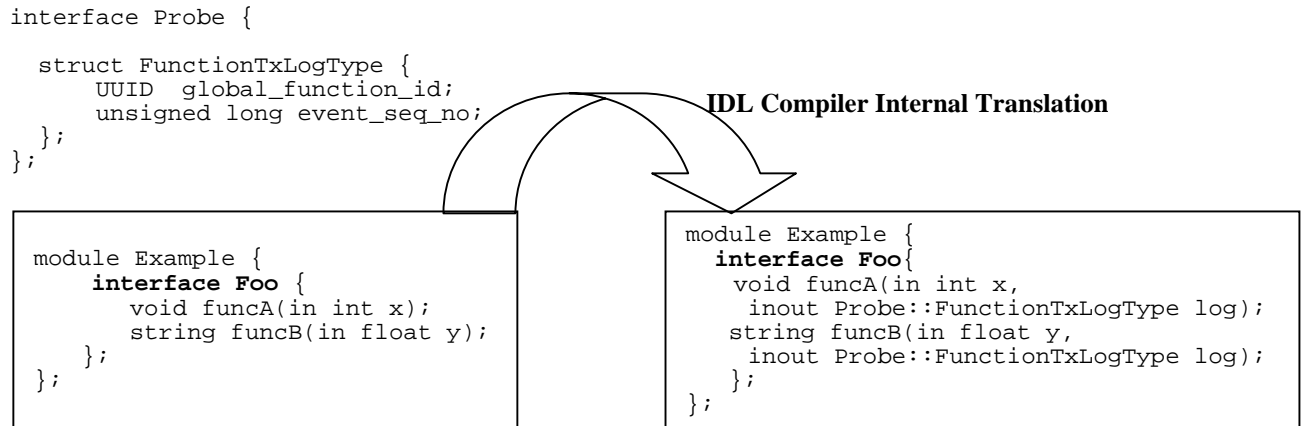
```
interface Probe {

   struct FunctionTxLogType {
       UUID  global_function_id;
       unsigned long event_seq_no;
   };
};
```

IDL Compiler Internal Translation

```
module Example {
    interface Foo {
       void funcA(in int x);
       string funcB(in float y);
    };
};
```

```
module Example {
  interface Foo{
    void funcA(in int x,
     inout Probe::FunctionTxLogType log);
    string funcB(in float y,
     inout Probe::FunctionTxLogType log);
    };
};
```

Figure 3. Code example of the FTL insertion by the IDL compiler

transported to the skeleton, where the event number is further updated. No modification to the runtime infrastructure is necessary for the FTL's transportation. Subsequently, the updated FTL is stored in the thread specific storage (TSS). Such a TSS is created at the monitoring initialization phase by loading the instrumentation-associated library, and is independent of user applications. If a child function *G* is invoked within *F*'s implementation, the FTL is retrieved from the TSS at *G*'s stub, gets updated and carried further down the chain.

transported from one function to its immediate follower (sibling call). This is guaranteed because the previous function's termination and the immediate follower's invocation incurs always within the same thread, and the FTL can be transported by storing/retrieving the TSS.

The FTL is recorded locally by probes associated with either timing latency or CPU utilization. It is lightweight-ed since no log concatenation occurs as the call progresses through the tunnel. Probes only update the FTL. Neither the user application nor the runtime infrastructure needs to be altered for tunneling.

2.2. Flexible Object Invocation and Threading to Monitored Applications

This section releases all but the DII/DSI restrictions introduced in Section 2.1 to support flexible distributed applications. Only causality propagation needs reconsideration, as timing and CPU utilization are collected locally and independent of causality capture.

Object Invocation

Asynchronous (one-way) function calls are always cross-thread. From the perspective of the causality propagation, call dispatching spurs a fresh causality chain out of the callee thread that executes the actual function implementation, in parallel to the chain the caller thread still carrying forward. The original chain is the *parent* chain and correspondingly the newly created chain is its *child*. Such a parent/child chain relationship is recorded in the *stub start* probes of the one-way function calls.

Component object invocation is not necessarily cross-process or cross-processor. They can be collocated calls. To allow collocation optimization, the stub needs to be changed such that when the client and the component object share the process domain, both *stub start* and *skeleton start* probes are triggered before the execution falls into the user-defined function implementation. Similarly, both *skeleton end* and *stub end* probes are turned on at the function return phase. Therefore, the *stub start* (end) and *skeleton start* (end) probes degenerate into a single *start* (end) probe.

Custom marshalling (or marshall-by-value) allows remote object invocation to be actually carried out in the client's thread context, which basically turns remote calls into collocated calls. The probe adjustment suggested above applies.

Server Multithreading

To improve performance, advanced threading architectures are recommended for ORB, such as thread-per-connection and variants of thread pooling [18]. The causality tracing techniques proposed before are still applicable in such complex threading architectures because of the following observations:

(O1) A physical thread T is always dedicated to the incoming call C_1 until C_1 finishes execution. T never gets suspended and switched to serve another incoming call C_2 before C_1 finishes. Only when C_1 finishes, T will be either reclaimed by the ORB (e.g., thread-per-connection and thread pooling) or by the underlying operating system (e.g., thread-per-request).

(O2) When serving the incoming call C_1, thread T is annotated with C_1's latest FTL. When T is reclaimed by the ORB, although T physically survives and holds the stale FTL, each time new call C_2 comes and T is activated, T is always refreshed with the latest FTL of C_2.

Therefore, causality relationship will be clearly distinguished and propagated without being intertwined under different ORB threading policies.

Note that *O1* will not hold true for COM applications. For its Single-Threaded Apartment call dispatching, the server-side up-call is through a message loop [1]. The apartment thread T can switch to serve another incoming call C_2 when the call C_1 that T is serving issues an outbound call C_3 and suffers blocking. C_2 might get blocked later and T switches to continue serving C_1 as C_3 has already returned. Techniques have been devised to avoid causal chain mingling [10]. In the actual implementation, only a very limited amount of instrumentation before and after call sending and dispatching is required to the COM infrastructure.

2.3. Run-time Infrastructure's Instrumentation Impact and Interoperability

To achieve causality tracing, the IDL compiler handles the internal interface transformation and full probe deployment. This requires a back-end compilation flag to instruct the compiler for the instrumented or non-instrumented version of stub and skeleton generation. Currently, for both CORBA and COM applications, our IDL compiler is modified to accommodate such instrumentation demand. The instrumentation-associated library is supplied externally at link time. For CORBA applications, no CORBA runtime modifications are required. For COM applications, as discussed before, due to thread multiplexing between blocking calls, the runtime infrastructure needs instrumentation to prevent causal chain intertwining.

In a heterogeneous environment like a CORBA/COM application where different subsystems are flexibly built upon either CORBA or COM, as long as the bi-directional CORBA-COM bridge is aware of the extra FTL data hidden in the instrumented calls, and delivers it from the caller's domain to the callee's domain, causality will seamlessly propagate across the boundary, and continue to advance in the other domain. Again, timing and CPU utilization deal with individual threads locally, and are therefore not impacted.

3. Application Behavior Characterization

Section 2 detailed the capture and correlation of monitoring information. At runtime, the probes capture the runtime information locally. When the application

ceases to exist or reaches a quiescent state (e.g., finishes processing a collection of transactions), the scattered logs are collected and eventually synthesized into a relational database. The data collector's detailed design is described in [8]. This section presents the monitoring data analysis tools to help understand system behaviors at the component level: the reconstruction of global causal relationship into a dynamic system call graph, and timing and CPU utilization analysis that leverages this call graph. The analyzer currently is implemented as a stand-alone tool, rather than being incorporated into the runtime infrastructure for dynamic system adaptation or management.

3.1. Dynamic System Call Graph

Recall that following the function call chain, in each invocation, events are logged and event numbers are incremented for each event appearance. Moreover, the event repeating patterns uniquely manifest the calling patterns. The reconstruction of system-wide causality relationship first performs a query on the overall monitoring data and identifies the set of unique Function UUIDs ever created. Then for each identified UUID, the second query sorts the events associated with the invocations sharing the UUID by ascending order. Subsequently, the analyzer scans the sorted event sets and reconstructs the call hierarchy following the state-machine shown in Figure 4, similar to the compiler parsing that creates an abstract syntax tree and performs type checking.

In Figure 4, a transition from one state to another produces a parsing decision. The state transitions labeled with solid lines evaluate synchronous function calls. The transitions labeled with dash lined govern asynchronous function calls (both the stub and skeleton sides). A decision about "in progress" indicates that call chain is advanced as expected. An additional "abnormal" transition state (not shown) is taken if the adjacent log function records follow none of the identified transition patterns. If that happens, the analysis will indicate the failure and restart from the next log record.

As the result, each causal chain with a unique UUID will be unfolded into a tree T_i. A **Dynamic System Call Graph (DSCG)** is a tree by grouping $\{T_i\}$. A large-scale application's DSCG potentially consist of millions of nodes. Conventional visualization tools based on planar graph display are incapable of presenting, navigating and inspecting such enormous amount of graph nodes. The hyperbolic space viewer [6] demonstrates its promising capability. Section 4 will show an example of DSCG.

The DSCG captures all component object invocation and preserves the complete call chains the application ever experienced, unlike GPROF [3] or QUANTIFY [16] that maintains the relationship with call-depth of 1. The call chain achieved by the DSCG is exactly the proposed call path [4]. The scenarios and techniques for call path profiling in single-processed procedure call environments can be extended naturally to multithreaded and distributed applications, as shown in Section 3.2.

3.2. End-To-End Timing Latency and System-Wide CPU Consumption

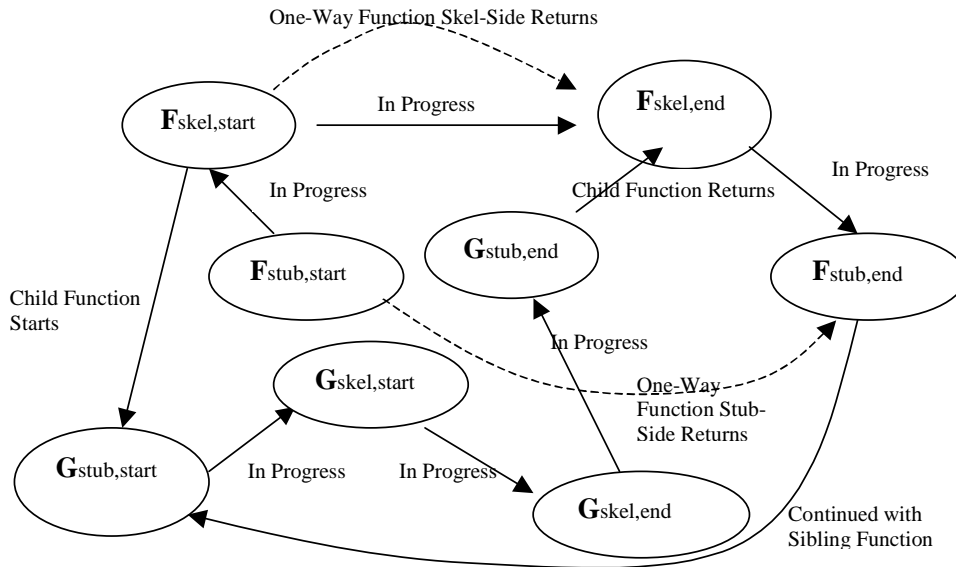The end-to-end timing latency for function F's invocation is obtained by:

Figure 4. State machine to reconstruct causality relationship

$$L(F) = \left(P_{F,4,start} - P_{F,1,end}\right) - O_F$$

if F is a synchronous or stub side's one-way call, and

$$L(F) = \left(P_{F,3,start} - P_{F,2,end}\right) - O_F$$

if F is a collocated or skeleton side's one-way call. O_F denotes the overhead spent for causality capture, which is determined by (N is the total number of child functions):

$$O_F = \sum_{i=1}^{N} \sum_{j \in R(F)} \left(P_{i,j,end} - P_{i,j,start}\right)$$

P denotes the value from latency probing. The second and third scripts respectively denote the sequencing number of Figure 1, and the time stamp at the *start* or *end* of the probing. $R(F)=\{1,2,3,4\}$ if F is a synchronous call and $R(F)=\{1,4\}$ if F is a one-way call.

Latency can be annotated to the DSCG's nodes to help perceive latency dispersed throughout the system-wide call hierarchy, besides reported in a certain statistical format.

Our CPU characterization identifies how function invocation utilizes CPU and how such utilization propagates in a distributed cross-thread, cross-process and cross-processor environment. It involves the following three phases.

First, it computes the CPU consumption of each function invocation, i.e., the *exclusive* or *self* CPU consumption that refers to the only portion consumed during executing the function implementation, with the portion spent inside its child functions excluded. It is computed by (*L* is the number of the immediate child functions):

$$SC_F = (P_{F,3,start} - P_{F,2,end}) - \sum_{i=1}^{L} (P_{i,4,end} - P_{i,1,start})$$

The subscripting follows the above equations.

It then propagates the result following the caller/callee relationship to obtain the *inclusive* or *total* CPU consumption of a function. It accounts for the above *self* portion, and the *descendent* CPU consumption from the child functions calculated by:

$$DC_F = \sum_{f \in \{immediate\text{-}child\text{-}functions\}} (SC_f + DC_f)$$

Note that the result of DC_F is represented generally by $<C_1, C_2,....C_M>$ where M is the number of different processor types in the application.

Finally, it synthesizes the inclusive CPU consumption with the DSCG to form a **CPU Consumption Summarization Graph** (CCSG) that exhibits system-wide CPU utilization propagation. The detailed CCSG construction and visualization can be found in [9]. A CCSG example will be shown in Section 4.

4. Experiments on Example Systems

We use two examples: a commercial large-scale embedded system based on an infrastructure similar to COM, and a simple CORBA-based Printing Pipeline Simulator (PPS), to demonstrate our tool. We have completed call graph construction for both COM and CORBA applications. We use this particular commercial system to demonstrate its scalability and practicality. Because timing latency and CPU utilization

Figure 5. Dynamic System Call Graph (DSCG) shown in hyperbolic tree viewer

characterization have not been completed for COM applications, only the PPS' CORBA-related result is shown here.

The commercial system contains more than 1 million lines of code. The experiment was performed in the HPUX 11.i development environment. The code base is partitioned into 32 threads in a single-processor 4 processes configuration. The largest system run ever conducted so far consisted of about 195,000 calls, with a total of 801 unique methods in 155 unique interfaces from 176 unique components. With the current Java implementation, it took the analyzer 28 minutes to compute the DSCG on a HP x4000 1.7 GHz dual-processor Windows 2000 computer. A portion of the DSCG is shown in Figure 5. By navigating the DSCG, with the superior navigation capability from the viewer [6], within minutes, developers were able to identify certain code implementation inefficiency, demonstrating its potential usage in high-level system review process.

For the PPS, timing latency of each individual function call is presented at the bottom section of the hyperbolic tree viewer when the mouse hovers over the tree node. To understand our end-to-end latency result's accuracy due to overhead on causality information capture, we compared it with manual measurement. The manual counterpart was carried out by having one probe for one target function in one system run. This probe retrieves time stamps at the beginning and end of the target function. With the configuration involved with 4 processes, two on Windows NT and two on HPUX 11.0, we observed that the automatic measurement and manual measurement were matched within 60% [8]. The collocated calls (with optimization turned off) tend to have larger difference compared with the remote calls.

In terms of the PPS's system-wide CPU utilization, Figure 6 shows a snapshot under Internet Explorer (as an XML viewer). It unveils the CPU propagation on a configuration of single-processor 4-process on a HPUX
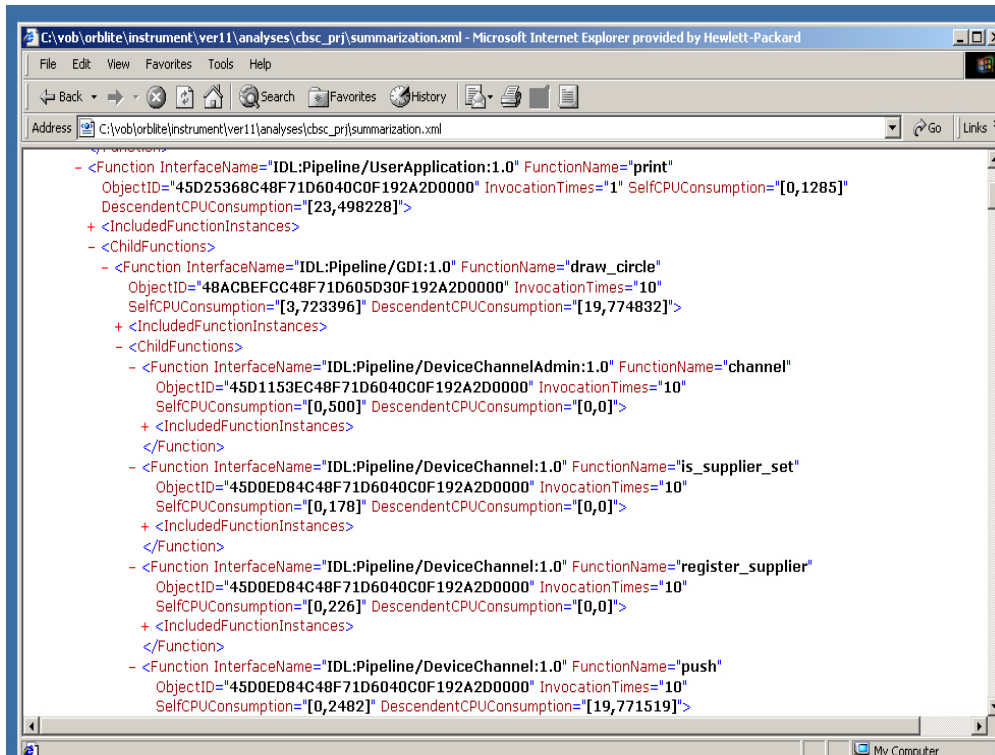
ObjectID: the universal identifier of the object; **InvocationTimes:** number of times the function has been invoked
IncludedFunctionInstances: all the invocation instances occurred to the function
SelfCPUConsumption and **DescendentCPUConsumption** are shown in [second, microsecond] format

Figure 6. A CPU Consumption Summarization Graph (CCSG) from the XML viewer

The PPS system is ORBlite based and consists of 11 components [8, 9]. It has been flexibly configured into multiple processes hosted by different platforms that include HPUX, Windows and VxWorks.

11.0 machine. The self and descendent CPU results are structured following the call hierarchy. Each node is identified by the interface and function names, along with its unique object identifier. To estimate the overhead or interference devoted to the necessary causality

information capture, we first evaluated that the automatic measurement from the monolithic single-thread configuration matches the true manual measurement to within less than 10%. Then we compared the measurement result on the above mentioned single-processor 4-process configuration with this monolithic single-thread configuration under the same HPUX 11.0 machine, and obtained good matching (within 40% difference) between these two configurations.

5. Related Work

Instead of providing a comprehensive survey on monitoring, characterizing and debugging multithreaded and distributed applications, herein we discuss only the existing techniques closest to our approach.

Global causality identifiers have been used for runtime system management. Alpha [17], realizes the control flow migration in cross-thread function invocation, and automatically transfers thread-specific attributes to the new thread, such that logical thread control remains seamless. COM's Object RPC protocol uses logical thread identifier to manage message dispatching and to correlate response/request messages. Our global causality identifier correlates both remote and collocated user-level calls. Also, from Sections 2 and 3, it is clear that without the additional event number in the FTL, the full causality relationship reconstruction into a call graph is impossible.

Interceptors have been standardized or explored in CORBA and COM. CORBA interceptor [14] allows user-defined message manipulation. While it might be employed to capture causality information, timing latency and CPU utilization will be less accurate because of the unknown overhead from the interceptors. Moreover, depending on vendor implementation, the interceptor and the dispatching of the execution of the function implementation might be carried by different thread contexts. This would break both the tracing tunnel and the transparency of the skeleton dispatching since thread-specific storage is key to our monitoring.

OVATION [15] is an example of employing CORBA interceptors for runtime monitoring. It provides a graphical tool to unveil component runtime interaction for CORBA based applications. Similar to our approach, the involved interceptor provides four different timing anchors: client pre-invoke and post-invoke, servant pre-invoke and post-invoke. Object method calls are presented in a sequence chart with respect to time progressing, along with their corresponding runtime execution entities (thread, process, and host). The major difference to our work is that it does not provide global causality capture. As the result, for each method invocation ever happens between two distributed objects,

the tool cannot determine how this particular invocation is related to the rest of method invocations.

Universal Delegator is an application-level COM interceptor [2]. All components under interception are aggregated by an Interceptor Component (IC) handling calls' pre/post-processing. The IC uses a Trace Object (TO) to log call information verbosely. To trace cross-apartment calls, channel hooks are placed to the COM ORPC channel. The IC mandates manual component aggregation. Even though the TO resembles our FTL in terms of global migration, the TO concatenates log info during call progression and unavoidably introduces the barrier for the call chains that exceed tens of thousands calls. Finally, the proposed TO is not sufficient to determine the hierarchical call graph.

The above interceptor mechanism in either COM or CORBA confines distributed application monitoring to their own domain, whereas our approach is applicable across dissimilar standardized infrastructures.

BBN's work on Resource Status Service [21] relies on in-band and out-of-band runtime measurements for QoS parameters evaluation to perform runtime quality of adaptation. In-band measurements are conducted on the remote function invocation paths. It also requires probing on the delegate objects (encapsulating stubs and skeletons) and the ORB, similar to our approach. However, to correlating client and server's measurements, it requires an explicit additional interface for each component. This interface accommodates an additional parameter to every interface method in order to carry trace records back and forth between the client and server. This trace record parameter is essentially the Trace Object in [2]. Consequently, application developers have to be aware of this QoS oriented interface and its implementation. Such tracing records are sufficient to determine individual client-server pairs. However, if the global view of object interaction has to be determined, to large-scale applications, in the runtime adaptation environment, the size limitation for the involved function call chain is certainly the concern, due to the trace record concatenation with respect to the call chain advancing.

Other distributed debugging, profiling and monitoring tools, including tools to produce logging at the source code level like AIMS [20], binary-rewriting tools like QUANTIFY [16], per-thread debuggers like Microsoft Visual Studio, GDB, and SmartGDB [19] and network/OS message logging tools such as HP's OpenView [5], all lack the capability of correlating monitoring information across thread, process and processor boundaries. Finally, JaViz [7] produces call graphs like ours for JVM-hosted applications. Its principal disadvantage is no support for language-intermixing applications other than Java.

6. Summary

A framework to monitor and characterize distributed and multithread applications built upon component technologies is developed. Compared with the existing tools and development environments in the context of distributed and multithreaded applications, our framework provides the unique features of:

- Integrating different well-established monitoring techniques to form an integrated framework that performs application-centric multi-dimensional behavior monitoring, including application semantics, timing, and resource consumption;
- End-to-end capture of dynamic system topology in terms of interface method invocation, which facilitates off-line system-wide characterization of both timing latency and shared resource consumption;
- Full IDL compiler automation for monitoring probes deployment without disturbing user-defined application code;
- Capable of handling both CORBA and COM applications, and feasible of dealing with the applications involved with the CORBA and COM bridging.

We have implemented the monitoring and characterization framework for CORBA applications on Windows NT, HPUX, and VxWorks (the VxWorks CORBA does not currently support CPU), and are finishing the COM version on HPUX. One future effort is to investigate the adoption of our monitoring techniques to the J2EE-based applications. We strive for the monitoring framework capable of monitoring the end-to-end application that consists of different subsystems, each of which is built upon a different remote invocation infrastructure. Other promising avenues for future research are to provide richer end-to-end system behavior characterization support, to apply the global causality capturing technique from the on-line perspective for application-level system management, and to automate or semi-automate test harness generation for multithreaded and distributed systems testing.

7. Acknowledgements

8. References

[1] D. Box, *Essential COM*, Addision-Wesley Pub. Co., 1998.

[2] K. Brown, "Building a Lightweight COM Interception Framework Part I: The Universal Delegator," *Microsoft Systems Journal*, Jan. 1999. Presentation about *Unobtrusive COM Tracing*: http://www.develop.com/kbrown/com/comtrace.ppt.

[3] S. Graham, P. Kessler and M. McKusick, " An Execution Profiler for Modular Programs," *Software - Practice and Experience*, Vol. 13, No. 8, 1983, pp. 671-85.

[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] HP OpenView, http://www.openview.hp.com/

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

[7] 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*, Vol. 39, No. 1, 2000, pp. 96-117.

[8] Jun Li, "Monitoring of Component-Based Systems," HP Labs Technical Report HPL-2002-25(R.1).

[9] Jun Li, "Characterization and Impact Estimation of CPU Consumption in Multi-Threaded Distributed Applications," HP Labs Technical Report HPL-2002-50(R.1).

[10] Jun Li, "Monitoring and Characterization of COM-Based Systems," HP Labs Technical Report in preparation.

[11] Microsoft, "The Component Object Model Specification," Version 0.9, 1995.

[12] 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.

[13] K. Moore, and E. Kirshenbaum, "Building Evolvable Systems: the ORBlite Project," *Hewlett-Packard Journal*, pp. 62-72, Vol. 48, No.1, Feb. 1997.

[14] OMG, *The Common Object Request Broker: Architecture and Specification, Revision 2.4,* Oct. 2000.

[15] Ovation, http://www.ociweb.com/product/ovation.html

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

[17] 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," *International Journal of Mini and Microcomputers*, Vol. 12, No. 3, 1990, pp. 119-27.

[18] D. Schmidt, "Evaluating Architectures for Multithreaded Object Request Brokers," *Commun. ACM*, Vol. 41, pp. 54-60, Oct. 1998.

[19] SmartGDB, http://hegel.ittc.ku.edu/projects/smartgdb/.

[20] 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*, Vol. 25, No. 4, pp. 429-61.

[21] J. Zinky, J. Loyall, R. Shapiro, "Runtime Performance Modeling and Measurement of Adaptive Distributed Object Applications," *Proceeding of International Symposium on Distributed Object and Applications*, 2002.