Characterization and Impact Estimation of CPU Consumption in Multi-Threaded Distributed Applications

Jun Li
Imaging Systems Laboratory
HP Laboratories Palo Alto
HPL-2002-50 (R.1)
May 28th, 2003*

component-based systems, distributed and multithreaded systems, Gprof, CPU characterization, impact estimation, causality propagation, CORBA

The existing CPU resource characterization techniques perform well for the applications running as individual processes, or distributed applications involving simple interaction between two-party systems, such as traditional client/server or database systems. They do not work when function invocation spans processes and processors in the multithreaded and distributed applications built upon component technologies such as CORBA, COM, and J2EE. We introduce a tool to characterize and estimate change impact of CPU resource consumption in such multithreaded and distributed applications. The tool relies on a component-based system monitoring framework to acquire the information about the CPU resource spent on function invocations, and the system-wide causality propagation regarding such function invocations. As the result, the tool is able to perceive the propagation of CPU consumption across threads, processes and processors. A multi-hyperbolic-tree visualization scheme is also devised to facilitate seamless navigation and inspection of the analysis report regarding CPU consumption characterization and its impact change, along with other aspects of system behaviors.

* Internal Accession Date Only Approved for External Publication
© Copyright Hewlett-Packard Company 2003

Characterization and Impact Estimation of CPU Consumption in Multi-Threaded Distributed Applications

Jun Li
Imaging Systems Lab
HPL-Palo Alto

Abstract

The existing CPU resource characterization techniques perform well for the applications running as individual processes, or distributed applications involving simple interaction between two-party systems, such as traditional client/server or database systems. They do not work when function invocation spans processes and processors in the multithreaded and distributed applications built upon component technologies such as CORBA, COM, and J2EE. We introduce a tool to characterize and estimate change impact of CPU resource consumption in such multithreaded and distributed applications. The tool relies on a component-based system monitoring framework to acquire the information about the CPU resource spent on function invocations, and the system-wide causality propagation regarding such function invocations. As the result, the tool is able to perceive the propagation of CPU consumption across threads, processes and processors. A multi-hyperbolic-tree visualization scheme is also devised to facilitate seamless navigation and inspection of the analysis report regarding CPU consumption characterization and its impact change, along with other aspects of system behaviors.

1. Introduction

A significant problem in distributed systems is to understand CPU resource consumption. During the system development lifecycle, the following questions often arise: how much the CPU is spent on each component or subsystem, where is the performance bottleneck, how to schedule different components and subsystems onto the limited number of processors, and how to understand the impact of change to the entire system when modifying a component's implementation or changing one of the processors in the system–will the system run faster, slower, or no effect, etc. The existing techniques to characterize CPU resource consumption are well accepted to deal with applications running as individual processes, or distributed applications involving simple interaction between two-party systems, such as traditional client/server or database systems. However, such techniques do not work when function invocation spans processes and processors in most component-based systems built upon CORBA, COM, RMI and J2EE.

We have developed a technique to not only measure the CPU consumption in each involved runtime process, but also capture semantic causality propagation across the entire system. As the result, how the CPU resource is consumed and propagated in a system-wide fashion can be revealed in the post processing phase. It is based on the runtime monitoring framework that we previously developed [9]. In contrast to the existing techniques that rely on program execution time measurement on either local procedure calls or basic blocks or machine instructions, our CPU consumption measurement is performed at the component level where components interact with each other through function invocation across component boundaries. Component technology allows the applications being measured to be multi-threaded, partitioned into different processes, deployed on different networked processors, and managed by different operating systems. Accordingly, the unveiled CPU consumption by our tool propagates across threads, processes and processors.

The distinction to the profilers such as Gprof [3] can be illustrated by the following simple example. Suppose the application consists of two processes P_1 and P_2. In P_1, function A is invoked 3 times and each time it consumes 1.0 ms CPU. Function A is invoked by the main function in P_1 once, and subsequently by function B from process P_2 twice. If employing a Gprof-like profiler, P_1 and P_2 have to be profiled

independently. These tools will only report that function A totally consumes 3.0 ms. However, our tool will not only reveal function A's consumption 3.0 ms, but also indicate that 2.0 ms out of the total 3.0 ms should be propagated to function B in P_2. Even though when the number of the involved parties (scheduled in different threads, processes and processors) are small and their interaction is simple, users can manage the inter-party relationship based on their application knowledge and manually perform the necessary propagation, for general multithreaded and distributed systems, especially large-scale systems, such automatic propagation is necessary in order to understand the system-wide CPU consumption behavior.

The further disadvantage of the existing profilers is that they mingle the CPU consumed by the functions both for user-defined component implementations and for component runtime infrastructure (e.g., ORB in CROBA) in a single report without distinction. In such a unified report, for a CORBA-based application, the functions to implement the runtime infrastructure, including the stubs and the skeletons and the ORB, can potentially be ranked over many user-defined functions in terms of CPU consumption due to their frequent invocations, whereas such infrastructure-related functions are usually beyond the control of application developers, and therefore out of their interest focus.

Our developed characterization technique can be further extended to perform impact analysis on system-wide CPU resource consumption. Users modify the measured CPU consumption to reflect their intended change to either an interface function's source code implementation, or the processor's processing power. The tool can then automatically construct a new system-wide CPU resource consumption report and identify the difference before and after the change. Such analysis is beneficial to gain the insight about how the new system will behave before the change is actually performed.

The hyperbolic tree visualization technique [6] is capable of managing and displaying large amount of dataset effectively. To deal with large distributed systems that produce large amount of analysis data, both the CPU consumption characterization result and its impact change estimation result are described in a tree-like format and thus viewable by the hyperbolic-tree visualization system. Moreover, to form a comprehensive and coherent view of the entire system behavior, a multi-tree view scheme is devised. The view is comprised of four different hyperbolic trees representing the following system behavior information: the high-level system interface definitions, the system causal relationships, the CPU consumption, and the CPU impact estimation. By navigating within individual trees up and down and between different trees back and forth, users can obtain the insight of the entire system behavior without being overwhelmed by the enormous amount of raw monitoring data and post processing data.

We have finished building the prototyping characterization tool for the applications based on a version of CORBA runtime infrastructure developed in HP Labs [12] and deployed on HPUX 11.0 platforms. In principle, we believe that there is no hurdle to apply our monitoring and analysis techniques for the applications built upon component technology infrastructure other than CORBA, including COM, RMI, J2EE, etc.

The report is organized as follows. Section 2 describes the monitoring framework that we have developed, with focus on how CPU consumption is measured and how the system-wide semantic causality propagation is traced. Section 3 presents the construction of a system-wide CPU characterization. Section 4 extends the characterization tool described in Section 3 for impact change estimation. To help users navigate and inspect the captured behavior information, Section 5 proposes a visualization scheme to present different behavior information through multiple coordinated hyperbolic trees. Section 6 briefly reviews the work on resource consumption characterization and system impact estimation, emphasizing the aspects most closely related to this paper. We end this report in Section 7, where some conclusions are drawn and possible future research directions are offered.

2. System-Wide CPU Consumption Monitoring

Our CPU consumption characterization relies on the dynamic runtime behavior information captured through a monitoring framework, which has bee described thoroughly in [9]. In Section 2.1, we outline the runtime monitoring framework with focus on how to trace system-wide causality propagation and how to measure the CPU consumption at the component interaction level. Section 2.2 addresses some minor extension from [9] to suit the need to characterize the CPU resource consumption. In Section 2.3, we show how to compute the CPU consumed both by the function invocations at the component level, and by the threads on behalf of the user-application, from the captured runtime CPU consumption information. In Section 2.4, the dynamic system call graph is introduced, which is constructed based upon the system-wide causality tracing information.

2.1 The Component-Based System Run-Time Monitoring Framework

This monitoring framework is developed with the objective of helping application developers understand multithreaded and distributed systems' multi-dimensional behavior at the component level. The involved system behaviors are about application semantics, timing latency and shared resources.

The monitoring framework relies on a compiler-assisted automatic deployment of instrumentation probes into the stubs and the skeletons. Each function with its function interface defined in the IDL specification has its stub and skeleton generated in pair. Generally, to monitor a function invocation, four different probes are required. They 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. Application semantics, such as input/output parameters, possible thrown exceptions, are captured directly at the stubs and the skeletons. Timing latency, and the usage of shared resources such as CPU and memory, are captured by these probes as well with necessary programming supports, either directly from the standard operating system, or from some special instrumentation libraries.

For CPU consumption monitoring, particular to the HPUX 11.0 platform, we use the system function *pstat_getlwp* to retrieve per-thread CPU consumption information in a straightforward manner. HPUX 11.0 employs 1x1 thread scheduling model, i.e., a kernel-thread is one-to-one mapped to a user-level thread.

All the application semantic behavior, timing behavior and resource consumption behavior are captured locally in individual runtime processes. In order to correlate such local views of system, the monitoring framework also captures system-wide semantic causality propagation. There are two primary types of causal relationships: the *function caller/callee relationship* and the *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 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*. The threads that the monitoring framework is aware of are the ones called *user-application threads*, which are spawned on behalf of the user-application during the execution of user-defined function implementation. The other threads, which are spawned and managed directly by component technology runtime infrastructure, and are not aware of by the monitoring framework, are categorized as *runtime infrastructure threads*. Two additional types of causal relationships are: *function-spawn-thread*, which occurs when a thread is spawned on behalf of the user-application during the execution of a user-defined function implementation; and *thread-invoke-function*, which occurs when a spawned thread on behalf of the user-application further performs function calls.

The semantic causality tracing is fulfilled by tunneling through the entire system from function callers to their corresponding function callees, and from parent user-application threads to their corresponding child user-application threads. Such tunnels are invisible to user-applications. The monitoring information relevant to causality tracing is locally recorded by each individual running process. The post-mortem analysis reconstructs system-wide semantic causality propagation, with the result shown in a dynamic system call graph, as will be elaborated in Section 2.4.

2.2 Extension of Monitoring Data Type

Our algorithms to perform CPU consumption characterization and impact estimation require object instance information, in addition to the interface information. The reason to distinguish different component objects, even though they might all be instantiated from the same implementation class, is that data members (i.e., internal states) of different objects can be potentially different, and therefore lead to significantly different object runtime behaviors. Unlike objects in traditional sequential programs, objects in component-based systems, especially those commercial off-the-shelves components, are carefully considered and encapsulate a fair amount of functionality. Typically the total number of component objects is moderate and component objects have a long life span with respect to the system run time.

Object Universally Unique Identifier, or Object UUID, is introduced to distinguish component objects system-wide. Object UUID's are only required in skeleton monitoring since function implementation can only be invoked from the skeletons. In ORBlite [12], Object UUID's can be directly retrieved in the skeletons. For other component technology runtime, it might require additional runtime infrastructure instrumentation.

2.3 Function and Thread CPU Consumption

This section covers the computation of the CPU consumption both for function invocations at the component level and for the user-application threads, based on the collected monitoring data described in Section 2.1. We consider only the CPU consumption devoted to user-defined implementation. The CPU spent on component technology runtime infrastructure is precluded.

2.3.1 Function CPU Consumption

Like local procedure calls, function invocation at the component level might be nested. The *exclusive CPU consumption* of a function refers to the CPU resource that is directly consumed by a function during the execution of the function implementation body, without taking into account of the portion that is spent inside its child functions. Correspondingly, the *inclusive CPU consumption* of a function refers to the CPU resource spent on both the function itself and the invoked child functions. The exclusive CPU consumption is also called the *self CPU consumption*. Within the inclusive CPU consumption, the portion contributed from the child functions is called the *descendent CPU consumption*.

With our monitoring framework, assume function F's implementation is comprised of a set of direct child function invocations (with size of N), its exclusive CPU consumption C_F is

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

Equation 1

where P denotes the CPU consumption monitoring data obtained at a particular probe location. The first subscript of P refers to the involved function. The second subscript refers to one of the four following probes with the corresponding index number: 1 for stub-start, 2 for stub-end, 3 for skeleton-start, and 4 for skeleton-end. Each probe has the beginning and termination phases. The third subscript indicates either one of the two phases.

In Equation 1, only the direct child function invocations are involved. All related probes are always located in the same thread, independent of whether the child functions are invoked remotely or collocated to the parent function.

The exclusive CPU consumption takes into account only the user-defined function implementation, because the CPU spent on either the stubs or the skeletons is nullified by subtraction. The interference to the original application from the monitoring activities, which are carried out in the stubs and the skeletons, is therefore eliminated. Defined in [9], a *function call chain* is a sequence of function invocations that are stimulated from an initial function invocation and share the same global causal identifier. Also in [9], two types of monitoring interferences are introduced: *intra-function-call-chain interference*, and *inter-function-chain interference*. The intra-function-call-chain-interference comes from the CPU consumption on each involved instrumentation probe. The inter-function-call-chain interference is due to the interleaving execution of different instrumented functions belonging to different function-call chains on the same processor. Equation 1 removes the intra-function-call-chain interference. The inter-function-call-chain interference does not exist for CPU measurement, because a thread cannot be shared by two function executions simultaneously, i.e., the CPU consumption associated with different function invocations is compartmentalized in different threads. Therefore, the CPU consumption monitoring is compensated in our analysis.

2.3.2 Thread CPU Consumption

As mentioned in Section 2.1, the threads in our monitoring framework are categorized into user-application threads and runtime infrastructure threads. It is the user-application threads' CPU consumption that we are concerned with.

Similar to Equation 1, assume an application thread T has a thread entry function of G, from which a set of direct child functions with size of N is invoked, we have the exclusive CPU consumption of a thread T is computed by:

$$C_T = (Q_{T,start} - Q_{T,end}) - \sum_{i=1}^{N} (P_{i,2,end} - P_{i,1,start})$$

Equation 2

In Equation 2, item Q denotes the CPU consumption monitoring data captured by the instrumentation probes located in the instrumentation library. Such probes are activated at the thread initialization phase and the thread termination phase. Item P denotes the same monitoring data already described in Section 2.2.1. In particular, only the stub associated probes are involved. The computed thread CPU consumption is exclusive since the result only accounts for the portion spent on the thread entry function G, but not the child functions that G invokes. Note that G is a local function call whereas the child functions of G are the ones at the component level.

2.3.3 Remarks

The CPU consumption obtained from Equations 1 and 2 only represent the CPU resource that is devoted to user-defined implementation. We term this portion *User-Application CPU Consumption*. The consumption from the stubs and the skeletons, and necessary runtime management inside the component technology runtime infrastructure, and runtime monitoring activities, are excluded. It is the user-application CPU consumption that will be characterized in Section 3. Even though the user-application CPU consumption does not address the comprehensive CPU consumption in the system, it is significant to meet many practical purposes to understand the system CPU consumption behavior, because in component-based systems, component technology runtime infrastructures are usually fixed, what users can manipulate in order to alter the system behavior are component interface implementations. Furthermore, component technology runtime infrastructures can be separately characterized independent of user-applications.

2.4 Dynamic System Call Graph (DSCG)

As described in [9], the principal graph nodes in a dynamic system call graph are: *function invocation nodes* and *thread instance nodes*. The four types of causal relationships described in Section 2.1 are manifested via the links between these nodes. A dynamic system call graph is actually a tree. Each function invocation node or thread instance node can have different node attributes to represent the associated runtime behaviors, such as the CPU consumption from Equations 1 and 2, timing latency, and memory usage.

It is beyond the traditional planar graph display techniques to display effectively the DSCG of a large distributed system, e.g., the HP LaserJet printing system, in which potentially millions of function calls can easily pass by for just one single system run. The hyperbolic tree visualization technique [6] has demonstrated the promising visual effects from the experiments that we conducted in [9]. The runtime information can be either encoded as node attributes recognizable to the hyperbolic tree viewer, or incrementally queried to the back-end analyzer from the front-end user interface.

The DSCG presents a comprehensive view of how dynamic component interaction actually occurs at runtime. It is what our CPU consumption characterization tool is based upon to summarize and propagate causal relationships. Section 3 will provide the in-depth explanation.

3. System-Wide CPU Consumption Characterization

This section describes the characterization of a system-wide CPU consumption. It can be considered as an extension of Gprof [3] to the distributed and multithreaded application domain. The CPU consumption characterization is based on the CPU consumption consumed on each function invocation and user-application thread described in Section 2.3, and the DSCG described in Section 2.4. The CPU consumption characterization is a graph called *CPU Consumption Summarization Graph* **(CCSG)**.

The CPU consumption on single-processed applications can be simply represented as a scalar value. Such representation is no more feasible for distributed and multithreaded systems, where a function call can potentially span multiple processors and likely processors are distinctive from each other in terms of intrinsic processing capability and the operating system which manages and schedules applications onto these processors. We introduce *Vector CPU Consumption* in accordance with such application domain migration. The CPU consumption in a system that has N processors is denoted as $<C_1, C_2, \ldots\ldots C_N>$. In this report, we refer a processor to a computing host with a uniquely identifier in the network where the application is running. In practice, different processors can be treated equivalently, e.g., a computing cluster in which all processors are identical and the operating systems to manage these processors are also

identical. To simplify the representation of vector CPU consumption, we further introduce *Processor Group*. A processor group is a set of processors treated as identical in the system. With such grouping, the vector CPU consumption can now be represented as $<C_1, C_2, \ldots C_K>$ where K is the number of processor groups in the system. The CPU consumption spent within each processor group is superimposable.

In Section 3.1, we present the detailed algorithm on how to construct the CCSG. In Section 3.2, we identify the differences between our CCSG and the characterization report from the Gprof-like profilers, apart from the core difference that the Gprof-like profilers are only applicable to the single-processed applications, whereas our characterization tool is able to handle the applications spanning across multiple processes and multiple processors.

3.1 Construction of The CPU Consumption Summarization Graph

The construction of CCSG is based on the DSCG, and the CPU consumption information associated with the function invocation nodes and thread instance nodes in the DSCG.

We have two principal node types to characterize function invocation nodes and thread instance nodes respectively:

- A *function node* is denoted by a tuple of <object id, interface name, function name>. A function node N_F is the aggregation of all the function invocation instances $\{INV_F\}$ that share the object id, interface name, and function name in the corresponding DSCG;
- A *thread node* represents a set of thread instances. A thread node does not have an explicit identifier to be associated with. A thread node always has a function node as its parent node, and a function node can only have at most one thread node. Therefore, threads nodes are uniquely addressed in the CCSG since their parent function nodes are uniquely addressed. A thread node N_T, whose parent node is N_F, is the aggregation of all the user-application threads dynamically spawned during any function invocation that belongs to $\{INV_F\}$. If any of the threads spawned during a function invocation ever further spawns user-application threads, the above aggregation is continued to move on to the child user-application threads by following the thread parent/child relationship.

The characterization process involves two phases: the propagation of CPU consumption for function invocations nodes and thread instance nodes in the DSCG, and the aggregation of CPU consumption on these DSCG nodes into the CCSG's function nodes and thread nodes.

The propagation follows both the function caller/callee relationship and the thread parent/child relationship. For a function invocation F, its exclusive CPU consumption C_F is also the self CPU consumption SC_F. The descendent CPU consumption DC_F for a function invocation F is defined recursively as:

$$DC_F = \sum_{f \in child-function(F)} (SC_f + DC_f) + \sum_{t \in child-thread(F)} (SC_t + DC_t)$$

Equation 3

Equation 3 shows that the descent CPU consumption includes both the self and descent CPU consumption of its immediate child functions and child threads. The recursion will be terminated when a function invocation neither has any child function invocations nor spawns child threads.

Similarly, the self CPU consumption SC_T for a thread instance T is same as its exclusive CPU consumption C_T. The descendent CPU consumption DC_T for a thread instance T whose thread entry function is G is defined as:

$$DC_T = \sum_{f \in child-function(G)} (SC_f + DC_f) + \sum_{t \in child-thread(T)} (SC_t + DC_t)$$

Equation 4

Note that for a general distributed and multithreaded applications, the summations in Equation 3 and 4 are all operated in vectors.

The aggregation is rather simple. The self (descendent) CPU consumption of the function node N_F is the summation over the self (descendent) CPU consumption associated with each function invocation node belonging to $\{INV_F\}$ in the dynamic call graph. Similarly, the self CPU consumption of a thread node is the summation over the self CPU consumption of all the thread instances which are aggregated.

Arcs between function and threads nodes in the CCSG are labeled with numbers, which represent the

number of function invocations if the destination nodes are function nodes, and represent the number of threads spawned if the destination node is a thread node. The numbers are updated in each node aggregation.

In a decentralized distributed system, multiple independent call chains might occur, which potentially leads to more than one top function nodes in the CCSG. A *Global Virtual Main* node is introduced to be the root node in the CCSG and the previous top nodes now become its children. Note that the resulted topmost node has its descendent CPU consumption to represent the complete CPU consumption directly devoted to the execution of user-defined implementation in the entire system.

Consider a simple example shown in Figure 1. The main function invokes function *foo* (in processor A). In *foo*'s implementation, it first invokes a function *times* (in processor B) to query how many times the subsequent invocation of a function *say_it* (in processor D) has to be iterated. Then it invokes a second function *what_to_say* (in processor C) to query the string information that the subsequent invocation of the function *say_it* is going to use. It then invokes a function *say_it* for the number of times specified in the function *foo*. The function *what_to_say* spawns two threads in the middle of its execution. Suppose that the three invocations of the function *say_it* respectively consume 2.6, 2.5 and 2.7 ms, the function *what_to_say* consumes 3.0 ms and each of its two threads spends 2.0 ms, and the function *times* consumes 2.7 ms, and the function *foo* consumes 3.2 ms. The vector CPU shows the one spent in processors A, B, C, D in sequence. Figure 1 also shows the resulted DSCG and CCSG.

The detailed algorithm to construct the CCSG is shown in Appendix A. Since the graph manipulation involves only the traversal of the DSCG and the simple node-associated information processing. The complexity to construct the CCSG is $O(N_F + N_T)$ where N_F and N_T are the total number of function invocation nodes and the total number of thread instance nodes in the DSCG respectively.

Note that unlike the DSCG, which is a tree, due to function recursion, the CCSG is a directed graph that can potentially contain cross-links and even cycles. The introduction of thread nodes further increases such possibility.

3.2 Construction Differences to The Gprof-Like Profilers

The CCSG construction process described in Section 3.1 reveals the following major differences between our characterization paradigm and what Gprof adopts. Such differences are also believed to hold for the Gprof-like profilers, such as Quantifier [15].

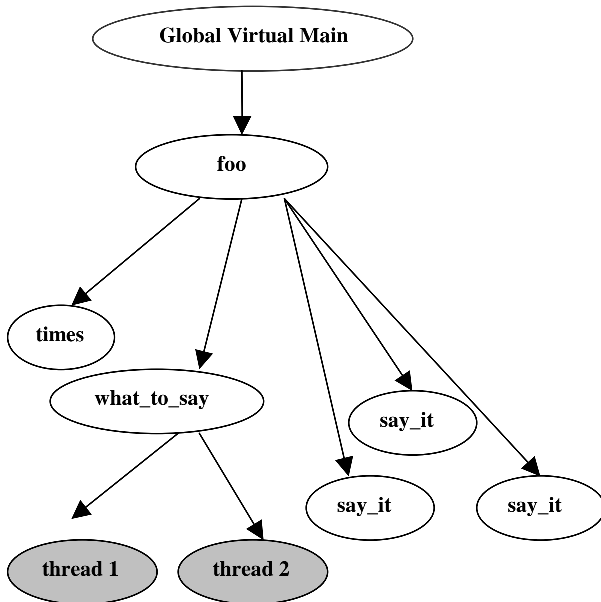
- **Assumption about Even Contribution of Child Function Consumption to The Parents** Because Gprof only captures the function caller/callee relationship with call depth of 1, and the CPU consumption to each function is accumulative via a single scalar value, when a child function tries to propagate the CPU consumption to each of its parent functions, it has to assume that each of its function invocations consumes evenly the CPU resource, such that its total CPU consumption can be evenly propagated out to each of its parents, based on the number of invocations coming from each of its parents. Such assumption often is inaccurate in practice, as pointed out by [14]. However, in the CCSG construction, as shown in Equations 3 and 4, there is no need to have such assumption to deal with propagation from a child function to its parents.
- **Handling Cyclic Sub-Graphs** When recursion occurs, the call graph in Gprof forms a cycle, which makes the CPU consumption propagation impossible. The solution by Gprof is to collapse all the nodes forming the cycle into a composite node, and have the composite node to compound the following attributes from the involved nodes: the self or descendent CPU consumption, and the function caller/callee relationship visible externally to this composite node. However, in our monitoring framework, a recursion is just another function invocation to advance the call depth. Even though the CCSG can be potentially cyclic, since we do not conduct the propagation on the actual CCSG, no graph transformation performed in Gprof is necessary to the CCSG.

```
void ClassA::foo() {
        …..
        //obj1, obj2, obj3 are three remote objects identified already.
        int counter = obj1->times( );
        String content = object2->what_to_say( );
        for (int i=0; i<counter; i++) {
             obj3->say_it(content);
        }
….
}

String ClassA::what_to_say(  ) {
        …
        Thread *thread_1 = new Thread (Thread_Start_Function)f1);
        Thread *thread_2  = new Thread(Thread_Start_Function)f2);
}

void main( ){
      ….
      //assume objA is with type of ClassA and have been identified as a remote object.
      objA->foo();

}
```



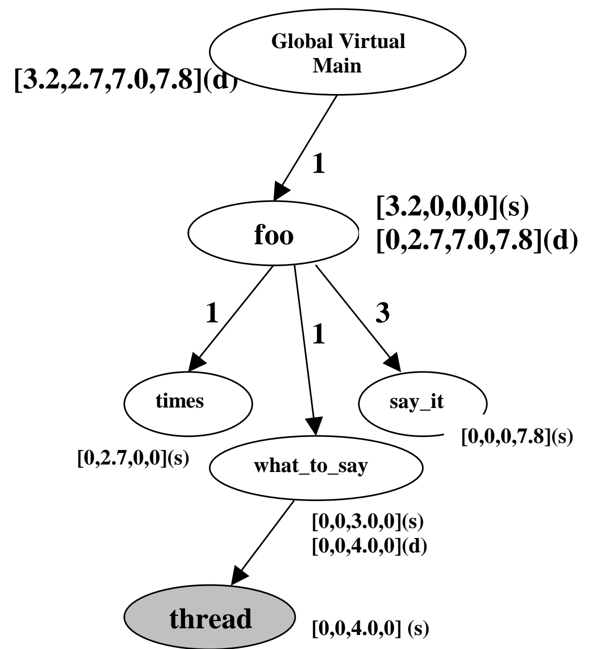


Figure 1: An Example Program with Its DSCG and CCSG

- **Aggregating Function Nodes in Different Level of Granularity.** The aggregation of function node described in Section 3.1 is based on component objects, because objects can have different internal states that may lead to dramatically different object behaviors. However, for large systems with enormous amount of object instances, the following scheme can be adopted to construct the dynamic CCSG. The CCSG becomes dynamic in the sense that the major portion of the CCSG is based only on interface names and function names, with object identifiers excluded; but those objects within a particular subsystem currently under the user's interest focus can be expanded and become object instance based. When the component objects

are out of the user's interest, nodes are automatically collapsed and become insensitive to object instances. Such flexible and dynamic node expansion and coalescence in our tool is feasible because the construction of the DSCG from the CCSG is merely a graph transformation process with different summarization criteria to follow.

3.3 Visualization of the CPU Consumption Summarization Graph (CCSG)

The CCSG can be encoded into a XML format and then displayed in a XML viewer like [19]. A XML representation describes a tree structure. However, the CCSG is typically a directed graph probably containing cross-links and cycles. In order to produce the primary tree structure into the XML representation, we conduct the breadth-first-search traversal starting from the topmost global virtual main node in the CCSG, the result is the breadth-first tree [2] covering all the tree nodes in the CCSG. The links traversed are called *primary* tree links. Other links not visited are then called *secondary* tree links. The secondary links are separately encoded into the XML format via a second graph traversal.

For a large distributed system, such as the HP LaserJet printing system, which consists of up to one hundred interfaces, and several thousands of interface functions, to visualize the CPU consumption summarization graph effectively in a normal planner graph display system has been proved to be difficult.

Hyperbolic tree visualization technique has been explored for non-tree graph structures [13, 4]. Similar to construct the XML file to represent the CCSG, a tree structure with only the primary (direct parent-child, no-cross) links are presented in the hyperbolic tree, and all other cross-links are only presented to users upon request. Both the primary links and secondary links can be constructed via graph traversals just described above.

3.4 Characterization on An Application Example System

The CPU characterization experiment has been conducted on the example application system described in the appendix of [9] in a HP Workstation 9000/750 running with HPUX 11.0.

Function/Thread	**4-Process Configuration (usec)**	**2-Process Configuration (usec)**	**Difference on Configurations**
EngineController::print	779	599	30.1%
DeviceChannel::is_supplier_set	167	123	36.7%
IO::retrieve_from_queue	8394	8044	4.4%
GDI::draw_circle	36320	25257	43.8%
RIP::notify_downstream	70998	70530	0.7%
RIP::Insert_Obj_To_DL	1970	1909	3.2%
IO::push_to_queue	16196	16379	-1.1%
UserApplication::notified	832	767	8.5%
Render::deposit_to_queue	1418	1348	5.2%
Render::render_object	185768	181925	2.1%
Render::retrieve_from_queue	2085	1923	8.4%
Thread spawned by EngineController::print	3349	2689	24.5%

Figure 2: CPU Consumption Characterization for The Example System's Functions and Threads

To validate the accuracy of the system function *pstat_getlwp* from which the CPU consumption information is acquired, the on-chip cycle counter is used. The counter only measures elapsed time, instead of the actual CPU consumption. Therefore, accuracy estimation only can be performed on single-threaded and single-processed applications without being swapped out in the middle of execution. From the initial experiment result, we found that the overhead introduced by *pstat_getlwp* is between 10 µs to 30 µs, which

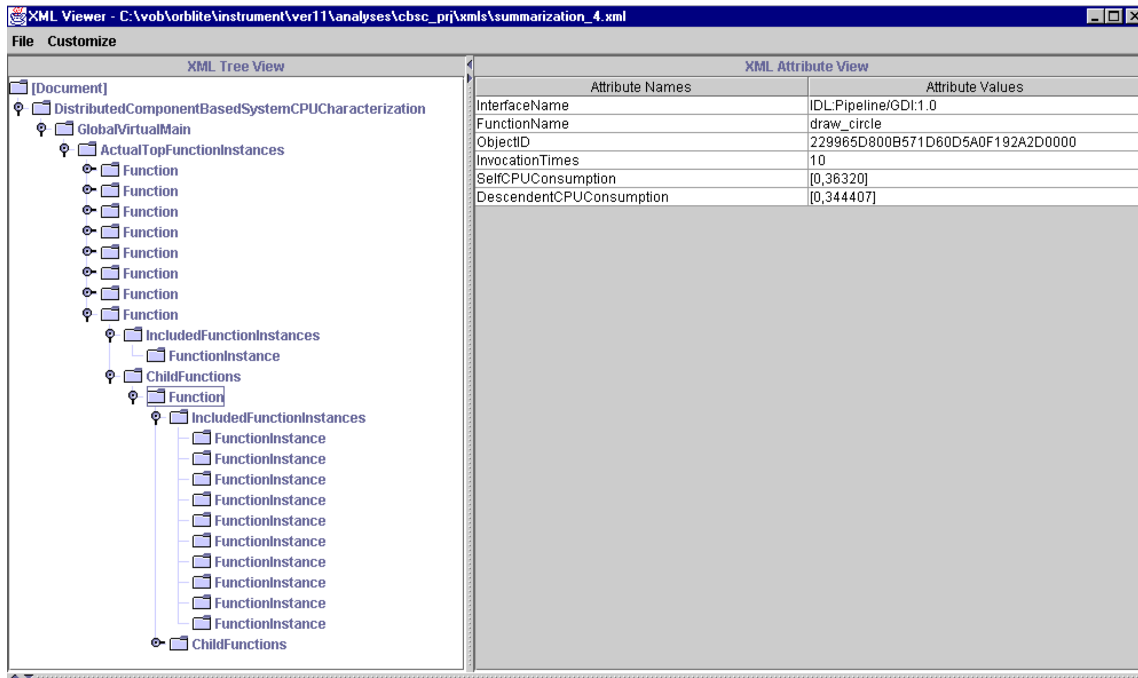


Figure 3: The CCSG of The Example System Presented in The XML Viewer

fluctuates from run to run. For applications without IO (e.g., *printf*) and system calls, the CPU consumption measurements from *pstat_getlwp* and the cycle counter vary within the 5% range. IO and system calls manifest unstable measurement results from run to run.

The functions chosen in [9] for latency measurement are shown in Figure 2 again for their self CPU consumption characterization report. We used two system configurations for comparision. The four-process one is described in [9]. The two-process one is to aggregate all the components in one server process, and to start the printing from the client process. Therefore, for two-process one, all function invocations at the component level are executed in one bulk thread. As stated in Section 2.2, our CPU consumption measurement is at the user-application level, ideally, the CPU characterization result should be independent of system configurations. As we are confident about the measurement for the single-threaded and single-processed applications, the difference shown in Figure 2, defined as the relative difference between the measurement for the four-process configuration and the measurement for the two-process configuration, infers the measurement accuracy for multi-threaded applications. The implementation of the example system is conducted in such a way that its execution is as deterministic as possible, so that the comparison between different runs is meaningful.

The difference shown in Figure 2 indicates the good matching between different system configurations. *GDI::draw_circle* has the difference exceeds 40%, which can be attributed to the intensive system calls invoked, since its function implementation encapsulates remote procedure calls to the registry server.

The summarization graph from the example system is shown in Figure 3.

4. System-Wide CPU Consumption Impact Estimation

It is beneficial to estimate the change of the system-wide CPU consumption behavior before the actual change is carried out. For a large distributed and multithreaded system, it requires a great effort to perform source code implementation change, to recompile and re-link the system, and then to re-deploy the system, before another round of CPU consumption measurement can be carried out. In terms of change of target processors, the advantage of conducting the impact estimation is much more significant in the situation where the target processors are not currently available.

The exclusive CPU consumption can be decreased or increased by certain user-specified percentage for function invocations that meet a user-specified constraint. Such a constraint can be specified for a particular

set of interface functions. Processor groups might also be involved to confine the change to certain computing platforms only. For instance, the function implementation change may only happen on Windows and VxWorks platforms, but not Unix platform. An example of such constraints is shown as below:

reduction on processor group { PROCESSOR_GROUP_1, PROCESSOR_GROUP_1**}** **by 10% to**

function { Module_1::Interface_1::function_1, Module_1::Interface_1::function_2 };

Users have to rely on other platform and application knowledge in order to come up with a percentage change specification.

With the DSCG and the computed CPU consumption, to evaluate the impact of change is rather straightforward. It covers the following steps to perform the impact estimation:

(1) Compute the new exclusive CPU consumption for the relevant function invocations.
(2) Reevaluate the self and descent CPU consumption for the affected function invocation nodes and thread instance nodes in the DSCG, corresponding to the propagation phase of the CCSG construction;
(3) Reevaluate the self and descent CPU consumption for the affected function nodes and thread nodes in the CCSG, corresponding to the aggregation phase of the CCSG construction;
(4) *The CPU Consumption Delta Graph* (**CCDG**) is formed by making a copy of the graph nodes that have been reevaluated, along with the duplication of the involved graph edges. Each new graph nodes records the differences before and after the change for the attributes: the self CPU consumption and the descendent CPU consumption.
(5) By superimposing the CCDG to the original CCSG, the resulted CCSG represents the result of change to the system-wide CPU consumption.

Similar to the CCSG, the CCDG can also be displayed as a hyperbolic tree.

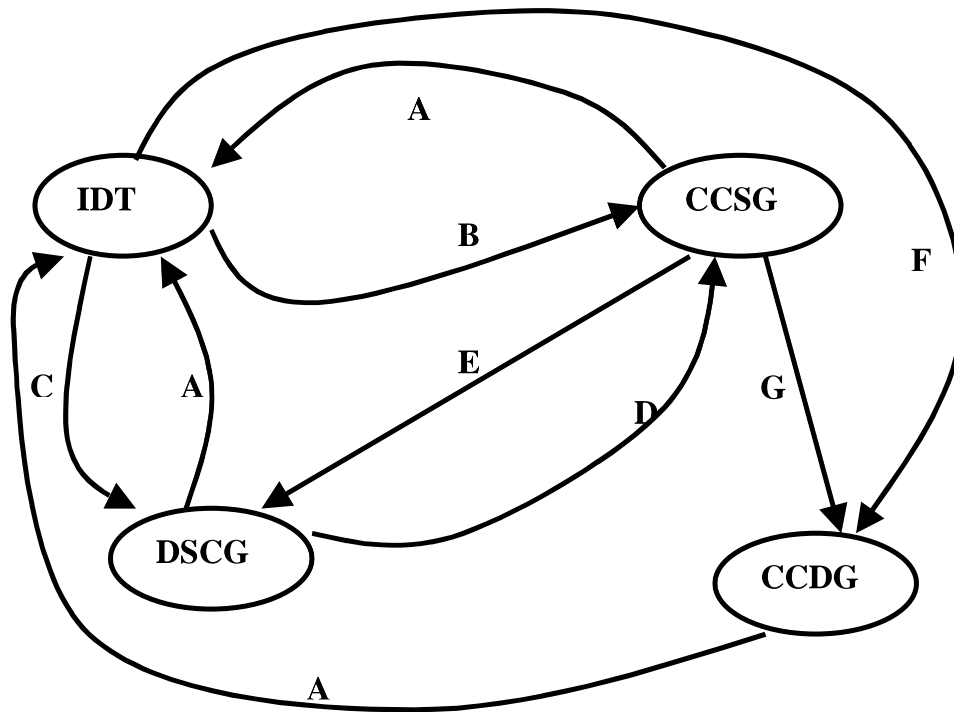
5. Coordinated Hyperbolic Trees for System Behavior Visualization

So far we have described three different types of graphs to represent different system behaviors. The DSCG shows how semantic causality is propagated system-wide. The CCSG presents how the CPU resource is consumed in the system. And the CCDG portrays the impact propagation in the system due to the CPU resource modification in some components or subsystems. In this section, we introduce a multi-hyperbolic tree navigation scheme to allow users to navigate seamlessly within a hyperbolic tree and between different hyperbolic trees to understand system behaviors in a comprehensive and coherent manner.

Besides the three graphs just mentioned, we introduce the fourth hyperbolic tree for the interface definitions associated with the application. The interface definitions are structured in a hierarchical manner. It consists of a set of modules. A module consists of a set of interfaces, data types, and exceptions. Each interface is further comprised of a set of function interfaces. Such hierarchical representation can be described in an *Interface Definition Tree* (IDT). By browsing the IDT, users can understand what services the components provide, and the correct way to request the services. Each function interface node in the IDT can be appropriately annotated with statistical behavior information relevant to the implementation of this function interface, e.g., timing latency, CPU consumption, and memory consumption. Sophisticated user-interface facilities may be necessary if such behavior information can not be represented in some concise formats like a scalar number or a string with modest length. For example, the CPU consumption can be displayed in a form to illustrate the average and the standard deviation of the CPU consumption, distinguished by component object instances located in different processors.

Each of the four hyperbolic trees identified provides different behavior aspects. The effective and easy navigation within each individual tree is powered by hyperbolic tree viewers. A scheme is devised herein to facilitate the seamless navigation between these hyperbolic trees. Figure 4 shows the scenarios for users to switch their current interest from one tree to the others.

Navigation between hyperbolic trees requires the involvement of the backend analyzer [9]. Every tree node in the front-end display is assigned with a unique identifier by the backend analyzer. At the current interested tree node P, the user can use mouse right click or pull-down menu to select his or her desired destination tree. Such request will be forwarded to the backend analyzer. The backend analyzer then decides what are the candidate tree nodes $\{N_i\}$ in the destination tree corresponding to the requested node. Note that $\{N_i\}$ might be empty. The candidate trees nodes can be displayed in a simple form if the size is



A: To inspect the corresponding function interface definition and its function implementation's timing latency, CPU/memory usage

B: To see how function implementation behaves actually in terms of CPU consumption and its propagation

C: To see how the function invocations are carried out and how each invocation is propagated

D: To see what is the result of CPU consumption given the defined semantic causality propagation

E: To see what is the actual semantic causality propagation

F: To see whether the corresponding function implementation is impacted or not

G: To see whether the CPU consumption associated with the function implementation is affected by the change

Figure 4: Scenarios for Users to Switch The Interest Focus from One Hyperbolic Tree to The Others

small, or can be represented by yet another dynamically created hyperbolic tree. For example, all function invocation nodes in a DSCG corresponding to the same function interface can be structured according to their execution locations, and locations in turn are structured in the following hierarchy: processor, process, and thread. Such forms or hyperbolic trees to bridge between different hyperbolic trees are called *navigation linking forms* or *navigation linking trees*. Nodes in either linking forms or linking trees inherit the unique identifiers of the original trees. When the user further selects one of the nodes in the linking forms or linking trees, the front-end display then brings the destination tree node into the current interest focus. Therefore, with the backend analyzer, along with the linking forms or linking trees dynamically created in the front-end display, users can navigate back and forth between different hyperbolic trees to inspect different behavior aspects of a distributed and multithreaded application.

6. Related Work

The Unix utility software Prof presents the number of invocations a function has been called and the associated average execution time. Such execution summarization facility is also provided by tools like Sun's Thread Event Analyzer [18] and Paradyn [11]. Threaded-Paradyn [20] allows function CPU consumption measurement performed at the thread level by per-thread virtual timers, such that different threads' contribution can be identified. The main disadvantage of these tools is that propagation of CPU consumption along the function call chain is not considered.

Gprof [3] and Quantify [15] are the well-known tools to profile the CPU consumption on single-processed and single-threaded applications. Function CPU consumption reported in such tools consists of two parts: the self portion contributed from the execution of the function itself, and the descendent portion contributed from the execution of the child functions being invoked during the function execution. Therefore, such profilers capture the CPU consumption propagation from function callees to their corresponding function callers. The main disadvantage of this approach is that the tools do not trace CPU consumption along the distributed function call chains (i.e., function chains that span threads or processes).

No prior solutions exist for distributed applications to report the system-wide CPU consumption and the associated propagation along multiple threads, multiple processes and multiple processor, let alone to estimate impact of change on system-wide CPU consumption due to a change to such distributed environments. According to our best knowledge, impact estimation in software systems either relies on source code implementation at the statement level, e.g., program slicing based on static code analysis [17, 5] and dynamic program slicing [1] with actual application input testing cases taken into account; or based on certain semantic reasoning framework on user-specification, which is not tightly coupled with the actual system implementation, e.g., timing sensitivity analysis based on Rate Monotonic Scheduling [16], and the software architectural level impact estimation based on the system architectural specification and its underlying language semantics [8].

In terms of hyperbolic tree display for visualization system behavior, the 3-D hyperbolic sphere described in [13] is employed to visualize source code structure of applications at the procedure call level [10], and HyperProf [7] applies 2-D hyperbolic tree (similar to [6]) to display the static Java source code artifact overlaid with the dynamic call behavior captured at runtime in one single tree. The hierarchical source code artifact in HyperProf includes Java packages, the classes inside the packages, and the functions defined in each class. The HyperProf's dynamic behavior capturing is reminiscent of the Gprof-like profilers, i.e., it does not capture function caller/callee chains. In HyperProf, when a function node in the hyperbolic tree is selected, additional tree links will be displayed to show which functions have been called by this selected function, and which functions have ever called this selected function.

7. Conclusions and Future Work

A tool to characterize the CPU resource consumption and to estimate the impact change of the CPU resource consumption change in distributed and multithreaded applications is presented in this report. The tool relies on the runtime monitoring framework for component-based systems that we have developed to capture CPU consumption related monitoring data. Moreover, a coordinated multi-hyperbolic-tree visualization system is devised to help inspect different aspects of system behaviors, which include the CPU consumption and its impact estimation.

In terms of tool implementation, the characterization tool currently works for the CORBA-based systems running on HPUX 11.0. The CPU characterization result is currently presented by a XML viewer.

We are planning to complete the tool implementation so that the CPU characterization and impact estimation is fully functional to CORBA-based systems running on HPUX 11.0, Windows NT and VxWorks, as well as to implement the proposed multi-hyperbolic-tree based user interface. The Gprof-like profilers are effective for single-processed applications at the local procedure call level. It will be useful to incorporate such local procedure level profiling capability into our component-level characterization tool to achieve characterization hierarchy. Such hierarchical view enables users to envision resource consumption upward propagation to the component-level from detailed source code implementation. To explore our tool, distributed system scheduling to maximize the CPU utilization while meeting timing constraints, and distributed system testing to ensure quality assurance for CPU utilization associated non-functional system behavior, are two of the many research directions that are worthwhile to pursue.

8. Acknowledgements

Thanks to Keith Moore from Imaging Systems Lab for his suggestions for building the multithreaded Gprof, Ming Hao from Software Technology Lab for providing the hyperbolic tree viewer, and Scott Marovich from Information Access Lab for the discussion of using on-chip high-resolution performance counters to estimate CPU measurement accuracy.

9. References

[1] H. Agrawal and J. R. Horgan, "Dynamic program slicing,", *Proc. SIGPLAN'90 Conference on Programming Language Design and Implementation*, White Plains, New York, June 1990. ACM Press. SIGPLAN Notices, 25, (6), 246-56, 1990.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to algorithms*, published by the MIT press and McCraw-Hill, 1990.

[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] M. C. Hao, M. Hsu, U. Dayal, and A. Krug, "Web-based visualization of large hierarchical graphs using invisible links in a hyperbolic space," HPL Technical Report HPL-2000-2.

[5] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. Programming Languages and Systems*, vol. 12, no. 1, 1990, pp. 26-60.

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

[7] HyperProf, http://www.physics.orst.edu/~bulatov/HyperProf/

[8] Jun Li, "Dependency tracking for real-time distributed systems," Ph. D. Dissertation, Department of Electrical and Computer Engineering, Carnegie Mellon University, 2000.

[9] Jun Li, "Run-time monitoring of component-based systems," HPL Report HPL-2002-25, Jan. 2002.

[10] S.-W Liao, A. Diwan, R. P. Bosch, Jr. and A. Ghuloum, M. S. Lam, "SUIF Explorer: An Interactive and Interprocedural Parallelizer,", *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPOPP'99), May, 1999.

[11] 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.

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

[13] T. Munzner, and P. Burchard "Visualizing the structure of the World Wide Web in 3D hyperbolic space," *Proceedings of VRML '95, (San Diego, California, December 14-15, 1995), special issue of Computer Graphics, ACM SIGGRAPH*, New York, 1995, pp. 33-38.

[14] C. Ponder, R. J. Fateman, "Inaccuracies in program profilers," *Software –Practice and Experience*, vol. 18, no. 5, pp. 459-67, 1988.

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

[16] S. Vestal, "Fixed-Priority sensitivity analysis for linear compute time models," *IEEE Trans. on Software Engineering*, vol. 20, no. 4, 1994, pp. 308-317.

[17] M. Weiser, "Program slicing," *IEEE Trans. on Software Engineering*, vol. SE-10, no. 4, 1984, pp. 352-357.

[18] P-T Wu and P. Narayan, "Multithreaded performance analysis with Sun Workshop Thread Event Analyzer," Sun Microsystems Technical White Paper. (April, 1998), Revision 03.

[19] XML viewer, http://www.alphaworks.ibm.com/tech/xmlviewer.

[20] 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–Algorithm to Construct The CPU Consumption Summarization Graph (CCSG)

The algorithm to perform the CCSG construction based on the constructed dynamic system call graph G_{DSC} is sketched in Figure 5.

```
procedure CONSTRUCT_CCSG ( G_DSC ):
  begin
        V ← TOPOLOGICAL_SORT( G_DSC );
        for each node v in V
          begin
              CALCULATE_SELF_CONSUMPTION(v);
              CALCULATE_DESCENDENT_CONSUMPTION(v);
              s ← REPRESENTATION_IN_CCSG(v);
              UPDATE_CCSG(s, v);
          End

        GROUP_TOP_FUNCTION_INVOCATION_NODE( G_DSC );

  end
```

Figure 5: Procedure to construct the CPU consumption summarization graph

Procedure *TOPOLOGICAL_SORT* sorts the constructed dynamic system call graph into a vector in which the leaf nodes of the graph are with the lowest index, and the topmost node is with the highest index.

Procedure *CALCULATE_SELF_CONSUMPTION* and *CALCULATE_DESCENDENT_CONSUMPTION* evaluates respectively the self and descendent CPU consumption associated with each function invocation node and thread instance node. Since nodes are sorted, the CPU consumption associated with the child nodes is guaranteed to be available for the current node's computation based on Equations 1–4.

Procedure *REPRESENTATION_IN_CCSG* queries the correspondent function node or thread node in the CCSG (under construction) for the specified function invocation node or thread node. All tree nodes in the CCSG are stored in a hash table for quick searching. The correspondent node found gets returned. Otherwise, the correspondent node is created and then returned. The searching associated with a function invocation node is rather straightforward through hashing. The searching associated with a thread instance node needs to propagate up following the thread-spawn-thread relationship, until it hits the ceiling thread instance node N_T that has a function invocation node N_F as the parent. If N_F does not have the corresponding function node R_F, then create R_F and its child thread node R_T, and then return R_T. If R_F exists, R_T will be created if R_T does not exist, and then R_T gets returned.

Procedure *UPDATE_CCSG* updates the self and descendent CPU consumption information of the function invocation node (thread instance node) to the corresponding function (thread) node, and then updates the caller/callee (function-thread) relationship information into the link between the two function nodes (between the function node and the thread node).

Finally, procedure *GROUP_TOP_FUNCTION_INVOCATION_NODE* is to create the top global virtual main node in the CCSG and make all the function nodes in the CCSG that have no parent function nodes to be this top node's child nodes.