Model Based Estimation and Verification of Mobile Device Performance

Gopal Raghavan
Nokia Research Center
5 Wayside Road
Burlington, MA 01803, USA
Gopal.Raghavan@nokia.com

Ari Salomaki
Nokia Technology Platforms
Visiokatu 3
Tampere, Finland
Ari.Salomaki@nokia.com

Raimondas Lencevicius
Nokia Research Center
5 Wayside Road
Burlington, MA 01803, USA
Raimondas.Lencevicius@nokia.com

ABSTRACT

Performance is an important quality attribute that needs to be planned and managed proactively. Abstract models of the system are not very useful if they do not produce reasonably accurate metrics. Detailed models are time consuming and expensive to build as well as to simulate. In order to strike a right balance, a framework is proposed in this paper that takes advantage of the flexibility of abstract modeling and intricacies of detailed modeling. Performance is modeled and verified per use case using a hierarchical queuing model of the system. Each component job is represented through characterization functions and service requests. Characterization functions may be parametric regression models derived from job measurements on system level model. A co-design framework is used to simulate and measure the performance of software components. The use case simulator analyzes the performance and verifies the use case requirements.

Categories and Subject Descriptors

D.4.8 [**Performance**]: Measurements, modeling and prediction, queuing theory, simulation

I.6.4 [**Simulation and Modeling**]: Model validation and analysis, model development, simulation output analysis

General Terms

Measurement, performance and verification.

Keywords

Performance analysis, system level modeling, use case verification.

1. INTRODUCTION

The overall performance of a system is dependent on both the hardware and the software architecture. The processor speed, bus

speed, cache configuration, number of processors, type of processors etc., determine the hardware configuration and in turn contribute to the system performance. The software component is like a workload generator to the hardware elements. Algorithmic complexity of the software component, interdependencies between components, task structure, inter-task communication, event handling etc, contribute to the performance of software. Software tasks that need faster processing can probably meet their performance needs when they are embedded in hardware. For example, some complex video/audio encoding and decoding algorithms are processed in hardware in order to enhance performance. To accommodate changes in hardware configurations, the software is built in layers. Therefore, higher-level applications layers are unaffected by minor hardware changes. This makes the software more portable across platforms. For example, on a mobile phone the core software for handling call origination and terminations are not frequently modified since the protocol requirements are seldom changed. Whereas the hardware platform changes significantly over a period of time to provide improved performance and to accommodate advanced features such as support for color display, image processing for camera phones, video streaming and voice recording.

It is a challenge for system designers to decide the architectural enhancements that are required to support a future application. For example: how to plan the system architecture of a mobile device that support about 10 times higher data rate in the next four years. This kind of decision-making involves in-depth analysis of various system level elements arranged in different configuration. There are several interesting situations. First, the software that is running on existing platform should run at least as well on the new platform. Second, the hardware platform should support the requirements of future applications. For example, the 3GPP quality of service requirement for audio streaming downlink needs at least a guaranteed downlink bit rate of 72 Kbps. The maximum bit rate is product specific and can range from 128 Kbps to 384 Kbps. If a product needs to support such high bit rates in the future, then its hardware should be capable of handling these requirements. And finally, the application demands might become higher due to which there might be changes in software as well as hardware. Performance of a system needs to be planned ahead of time based on several uncertainties and yet the estimates should be as close to reality as possible.

This paper proposes a model based performance estimation approach that uses modular, composable, and reusable component job models. Model characterizations are derived based on measurements obtained from hardware system models.

Section 2 presents a brief description of performance use cases. The modeling framework is explained in section 3. Use case model is outlined in section 4 with components and jobs further discussed in section 5. Section 6 describes job characterization and its dependencies on the execution paths, which is elaborated in sections 7 and 8. Section 9 shows how information for modeling is gathered and section 10 touches upon specifics of component context setup. Section 11 describes how characterizations are built from measurements. All these pieces are used in section 12 to analyze and verify the use cases.

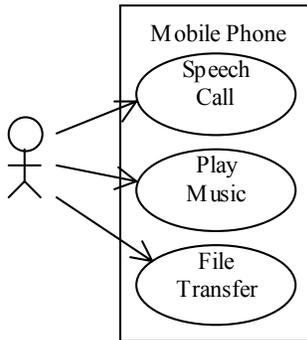## 2. PERFORMANCE CRITICAL USE CASES

Figure 1. A sample mobile phone performance use case

There are several use cases in a system, a subset of which is performance critical [1]. Performance critical use cases are those that produce heavy demand for system resources and have tight timing constraints. Such use cases may partially be supported by existing systems or may be in the road map for future systems. These use cases are driven by user requirements and contribute to the success of the product. In order to satisfy these requirements, performance of the system needs to be improved both in terms of software and hardware. Often performance solutions are not very generic but are tied to specific use cases. Choosing the right use case and concentrating on the requirement will facilitate an efficient problem analysis.

A simple use case on a mobile phone is illustrated in Figure 1. The user may invoke several system operations individually or simultaneously as shown in Figure 2. For example, while the user has an active speech call he may also be trying to backup some pictures that were captured using a mobile phone during a recent trip. We have at least two simultaneous scenarios: the speech call and the file transfer. While the file transfer is considered a background process, the speech call is an active foreground process that usually has a higher priority. The speech call will have load on both the application processor and the DSP. On the other hand, the file transfer scenario will have activity only on the application processor. In order to have good end user performance the system should support both scenarios without any distortion in voice or any delay in file transfers. Also, the file can be transferred through different physical medium, like IR, Bluetooth, serial cable, USB, etc. Depending on the medium, different performance constraints may be attached to the scenario. The goal of use case verification is to ensure that the model of

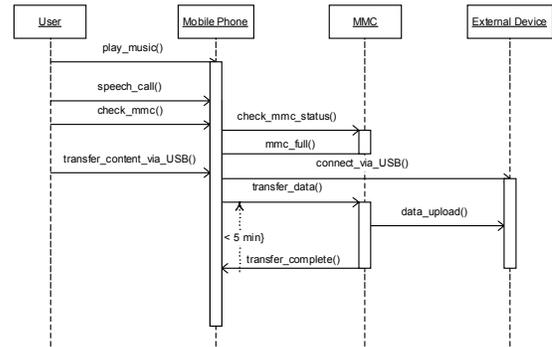system satisfies the use case requirements and any associated constraints.

Figure 2. A scenario illustrating file transfer

3. MODELING FRAMEWORK

Product visionaries foresee the features supported by future products and outline certain product requirements. System designers use these requirements to plan the platform architecture. Such architectural planning could happen at least three to four years before the product program starts developing the software and proceeds to manufacture the final product. There is usually a big gap between the time an idea is born and the time when the product hits the market. Any performance defects found on the target are very expensive to fix. The model based performance estimation provides a co-design framework through which performance of the system can be estimated earlier on in the life cycle [2]. Design alternatives may be analyzed and fixed to evade any costly mistakes. Models may be verified based on the use case with reasonable accuracy to ascertain that important performance requirements are met. The level of abstraction is very important in order to obtain practical results. If we use very abstract models then the estimates may not be very realistic. On the other hand, if we try to perform detailed simulation it might be overwhelming and expensive. The proposed approach is an intermediate solution that uses an abstract modeling framework, like LQN [3], but obtains parametric service time metrics through measurements on system models. The traditional co-design approach of executing the software on top of system model involves time-consuming simulation.

Our goal is to analyze performance characteristics, such as throughput and response latencies, for various use cases on the platforms that currently do not exist in hardware. However, we assume that software for these platforms exists and is executable on the hardware simulators. In such a situation, the straightforward solution would be to execute the software on the simulator and get the performance results from the execution. There are several issues with this approach. The most important one is that the hardware simulators are slow. For example, ARM926 processor with 211.2 MHz clock frequency and effective CPI of 3.94 (with caches) provides an effective MIPS of 53.63. The system level simulated MIPS on commercial tools that run cycle-accurate simulation ranges between 0.05 to 0.2 based on the level of detail, number of modules in the system and the processing capacity of the host system. For a reasonably small system model that includes an ARM9 CPU, memory system and AMBA bus the simulation running on a Pentium III 1 GHz machine has approximate slowdown factor in the range of 250 to

1000 compared to the actual target. This means that in the worst case the simulator can take approximately 24 hours to complete a use case that only takes about 86.4 s (1.44 min) on real hardware. This is unacceptable for mobile device systems, since the initialization process itself could take between 40-80 s and the actual use case may be even longer. Extended simulation lengths will delay the system validation process and may hinder the capabilities to experiment variations within reasonable time [4,5].

A related issue is that we need to collect and analyze performance metrics (throughput and latencies) not just for a single execution but also for a large number of diverse executions that may differ in input parameters, concurrently executing use cases, scheduling mechanisms, hardware characteristics and so on. It is a laborious and time-consuming task to gather all this information. Finally, without a model it is impossible to understand the performance metrics of software components and hardware resources that are parts of the use case.

Therefore, we were looking for approaches to lower the execution time requirements and yet to gather large amounts of reliable information. For this we decided to apply a well-known principle of "divide and conquer", modularity and reuse. First of all, we decided to execute only single use cases on a system level simulator and analyze multiple parallel use cases on LQN-like use-case simulator. Such simulator would take as an input use case models and provide capability to schedule single or parallel use case execution generating performance metrics (throughput and latencies) as an output.

Furthermore, we decided to modularize the use cases themselves. While the whole use case sometimes can be parameterized and reused, it is usually composed from various software components. Executions of these software components – these executions we call *jobs* - can be parameterized and reused more naturally. For example, the use case of file transfer via Bluetooth could be parameterized by the file size. However, a more precise approach would be to split it into the file reading component, whose jobs can be parameterized by the file size and the drive properties. Similarly, the Bluetooth transfer component is parameterized by the file size and transfer speed. It is possible to look at even finer grain at initialization of Bluetooth, initialization of file server, file read, file transfer and the connection teardown jobs. These jobs would be likely to be reused in other use cases, such as file transfer over USB or audio listening over Bluetooth. Also, from software engineering point of view file server and Bluetooth components are independently reusable and changeable. Separating use case into components requires minimal software reexecution on hardware simulator if certain component behavior changes.

4. USE CASE MODEL

As mentioned in Section 3, use case is composed from reusable component jobs. Here we provide some more details about this composition.

Component job is an execution of a software component. A concrete job can be characterized by its input, output, new system state, service requests to other components and resource consumption. However, we are interested not in a single job of a component, but in a collection of same component jobs with different inputs, initial and final states. Such job collection can be

represented as a *parameterized job* that is characterized by a state transfer function. This function produces a new global state based on a job's input and global state, resource usage function that produces the resource usage of the job, and service request function that produces service requests to other jobs (Figure 3).
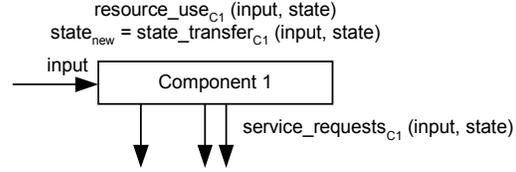
$$resource\_use_{C1} (input, state)$$
$$state_{new} = state\_transfer_{C1} (input, state)$$

input → Component 1

$$service\_requests_{C1} (input, state)$$

Figure 3: Component job and its functions

This is a rather abstract representation, since, for example, resource usage function can be a multi-function that produces execution time, memory consumption, energy consumption, and so on. The functions themselves could be described in various ways. A function could be represented as a table, where each row corresponds to specific input and shows what is the output of the function. A function could be a regression model – a linear or non-linear dependency of the output on the input parameters. Finally, a function could be a simple program mapping input to the output. Software components may differ a lot in their transfer functions. Some software, such as simple filters, protocols and codecs may have simple linear transfer functions. Other software such as complex graphical converter may have only complex transfer functions that are impossible to simplify without unacceptable loss of precision and may be difficult to extract during reverse engineering.

Service request multi-function produces the service requests to other jobs including the timing of such requests, input parameters, synchronous or asynchronous nature and so on. If we look at the jobs as nodes and service requests as arcs, we obtain job dependency graph that represents causal and temporal dependencies of jobs in a use case (Figure 4 shows jobs of two components C_1 and C_2).

Time

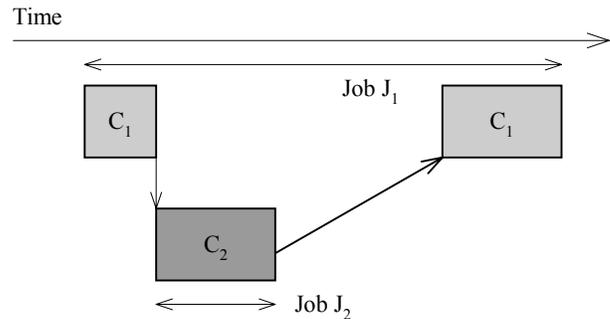C_1 Job J_1 C_1

C_2

Job J_2

Figure 4. Job dependency graph

In reality, job multi functions are probably best represented as an abstract algorithm in pseudocode program indicating resource consumption and service requests to other jobs. However, other forms of representation and visualization of multi functions are possible and useful.

5. COMPONENTS AND JOBS

As discussed in section 4, use case is composed from component jobs. However, component identification is a complex problem. We consciously delay our definition of a component, since components can be defined and identified in various ways. Each such way would lead to different components and their models. If components are not identified at a correct granularity, it may be difficult to discover the job transfer functions or there may not be a good transfer function at all.

Job transfer functions depend on the modeling approximation. Totally exact representation of any job for given input and state is the component execution itself, i.e. transfer function of a job is the execution of the component. If we want to obtain a simpler transfer function, we have to abstract and lose the precision. This loss of precision – and attempts to achieve the highest precision with the "simplest" representation – is intertwined with the identification of components leading to complex multidimensional optimization problem that could be informally expressed as follows:

Problem 1. Given a software system, find decomposition into components $C_1 \ldots C_n$ such that the approximation of component job transfer function is minimized using some measure (min total$_{i=1 \ldots n}$ (ApproximationMeasure(C_i, transfer$_{Ci}$()))) and total transfer function complexity is minimized using some measure (min total$_{i=1, \ldots n}$ Complexity (transfer$_{Ci}$())).

This problem is practically unsolvable, since the exploration space for decompositions and transfer functions are enormous.

As we mentioned, the components should be software components from the software engineering point of view, i.e. they should be separable, reusable software entities with clear semantics. Finer granularity components can improve the job approximation precision, however, they can lead to high complexity and very low reusability of components. Therefore decisions on component identification have to be balanced.

Component identification is easier for the software designers, since they usually know the "componentization" of their software and even possibly relevant job transfer functions. On the other hand, component view of performance engineers may not be the same as the component view of software designers. Component identification after system creation by someone performing system analysis is related to software reverse engineering, since components need to be created without having the knowledge of the original designer and architect. This can be achieved through the software reverse engineering techniques including static and dynamic system analysis, static call tree analysis, dynamic tracing and trace analysis [6,7,8].

To achieve simple and yet useful division into components, we propose to use as components the smallest schedulable entities. In our case, these entities correspond to *RunL()* method executions in active objects in Symbian operating system [9].

Most embedded operating systems support execution of multiple tasks and also provide mechanisms to switch between tasks. Such kernels usually operate in pre-emptive multitasking mode. Which means higher priority tasks can preempt lower priority tasks and switch the context. Symbian OS, one of the most popular operating systems on mobile devices, supports co-operative multitasking in addition to the conventional pre-emptive multitasking. This is implemented at the object level. Multiple

objects can remain active and be scheduled to execute based on some event occurrence. This mechanism facilitates scheduling multiple objects and asynchronous event communication without the context-switching overhead. Cooperative multitasking between active objects a1, a2, a3 etc., is shown in Figure 5. In this illustration t1, t2 and t3 are three different tasks. All the active objects except a5 run to completion. Active object a5 is preempted due to task switch from t2 to t3. The active object a5 resumes operation after completion of active object a6 in task t3.
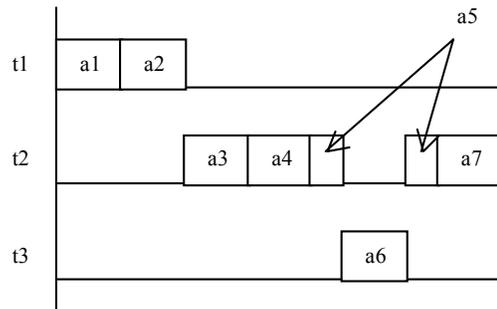

Figure 5. Active object and task scheduling

When an event is dispatched to an active object, a special Symbian active object method called *RunL()* is executed. This method executes to completion without being rescheduled by the active object scheduler. This method may, however, be interrupted during task switching. We define component job as a *RunL()* method execution. The *RunL()* method may contain other function calls. These calls belong to the same job as the *RunL()* itself.

Active object in one task may communicate with active object in another task using a *SendReceive()* method.

```
void CMyAO::RunL()
{
    foo1();
    foo2();
    foo3();
}

void foo2()
{
    bar1();
    SendReceive();
    bar2();
}
```

SendReceive() is a Symbian API defined in RSessionBase class and it may be used to send asynchronous or synchronous messages. When this function is invoked using the TRequestStatus parameter, it sends a message to the server and waits asynchronously for a reply. If this parameter is not used, the message is passed synchronously. When a *SendReceive()* happens synchronously and the message is sent to another thread, the originating thread is put to a wait state until the operation is completed. The function resumes when the called thread completes the function and releases control. Message passing happens through a pointer to the message array, which has four 32-bit parameters on the client address space into which the server can read or write data. The execution path in the client function after the *SendReceive()* call may depend on the data received from the server. *SendReceive()* calls correspond to

service requests in our job description. The called entities are themselves component jobs.

Currently we use active object execution within thread context as a boundary condition for a component job. In general, there can be several other conditions that mark the boundary, such as invocation or loading of methods from dynamically linked library, switch to processing in hardware or peripherals, input/output, and interrupts.

6. JOB CHARACTERIZATION

Although the proposed correspondence between *RunL()* methods and jobs seems to be straightforward and intuitive, it is not as simple as it appears.

```
void CSimpleAO::RunL()
{
    x = obj->simpleCall();
    Status = x;
}
```

In simple *RunL()* methods the execution is linear and the job characterization in terms of resources is trivial. In general though, there may be many execution paths through the *RunL()*:

```
void CStateMachineAO::RunL()
{
    switch(event)
    {
        case 1: callFoo1();break;
        case 2: callFoo2();break;
        default: callInvalid();
    }
}
```

An active object may implement a state machine within *RunL()* that maintains various states of the object and uses the events as triggers to transition from one state to another. A switch statement on the event is sufficient to structure the states. Because of the conditional constructs, state machines have multiple execution paths. In this case, the job characterization may be quite complex and should represent the whole state machine.

When *RunL()* calls *SendReceive()* methods, these methods may return information affecting further control flow in *RunL()*. Such dependencies have to be reflected in the job transfer functions.

In general, the characterization of jobs depends on the number of execution paths in a component.

7. EXECUTION PATHS

The *execution path* is defined as a sequence or control path encountered during a component execution. Component may contain assignment operations, control operations and function calls. Component may have numerous execution paths each uniquely distinguished by system state or input parameters.

Some components may have very few and very simple execution paths; others may have very many and complicated paths. The complexity of a *RunL()* method varies from one active object to another. For example, a simple *RunL()* may be as illustrated below:

```
void CHelloWorld::RunL()
{
    iEnv->InfoMsg(
    R_ACTIVEHELLO_HELLO_WORLD);
}
```

On the other hand, the server that handles all window management events involves sophisticated processing that invokes many functions as shown in the call stack below:

```
CCoeEnv::RunL()
CQikAppUi::HandleWsEventL(
    const TWsEvent &, CCoeControl *)
CEikAppUi::HandleWsEventL(
    const TWsEvent &, CCoeControl *)
CCoAppUi::HandleWsEventL(
    const TWsEvent &, CCoeControl *)
CCoeControl::ProcessPointerEventL(
    const TpointerEvent &)
CEikMenuPane::HandlePointerEventL(
    const TpointerEvent &)
CEikMenuPane::ReportSelectionMadeL()
CEikMenuPane::ProcessCommandToAllObserversL
    (int)
CQikAppUi::ProcessCommandL(int)
CQHelloGuiAppUi::HandleCommand(int)
```

7.1 Simple Components

For simple components it is easy to find all execution paths. If a function does not call other functions and if the number of alternative paths in the function is easily determinable using the conditional statements in the function, then it is simple to count the number of the execution paths.

```
void setX(int y)
{
    class_attribute_x = y;
}
```

The above function for example, is a basic helper function with some assignment statements. It may also include some basic arithmetic calculations. These patterns are commonly encountered in getter/setter functions of a class or some conversion functions. There is only one execution path in this function and it is pretty straightforward to capture this. These functions are considered to be end nodes as they do not invoke other functions and the data flow ends here. Transfer function of such component jobs is also simple, since they usually do not depend on external parameters.

$$d = 1$$

where

$$d \qquad \text{is the number of execution paths}$$

Components with conditional statements have multiple alternative paths. One or more parameters may be responsible for separate paths during program execution.

```
void setMenu(int selection)
{
    switch(selection)
    {
    case 1: choice = voiceCall; break;
    case 2: choice = sms; break;
    case 3: choice = calendar; break;
    default: choice = browser;
    }
}
```

Based on the function parameter there are four possible paths in the above function. Each path does some operation and terminates. In general, the number of execution paths is equal to the number of conditional branches that exists in the function. Conditional branches are due to language constructs like *for, while, if, switch, #ifdef* etc.

$$d = b_c$$

where

b_c is the number of conditional branches

Loop structures with fixed iterations to perform some operation are deterministic. Even if the loop has variable iterations, it is still executed in a single execution path. However, if the iterator is not initialized within the loop statement there can be at least two distinct execution paths based on the value of the iterator.

```
int multiPathLoop(int iterator)
{
    int i = iterator;
    for(;i< 1000; i++)
    {
        x = x*i;
    }
    return x;
}
```

Clearly there are two paths, in the above function, based on the value of the iterator. One path is covered within the loop if the iterator is less than 1000 and the loop is skipped if the iterator is greater than or equal to 1000. Nested loops are also similar. If the iterators are initialized then we will have only one execution path, otherwise each nesting level will result in branches and hence we will have multiple execution paths. In general, a loop structure will result in one execution path and may occasionally be two if the iterators are not initialized.

$$d = \sum_{i=0..n}(1 + l_{ui})$$

where

$l_{ui} = 1$ if the ith loop has un-initialized iterator

$l_{ui} = 0$ otherwise

n is the number of loops

7.2 Complex Components

Some components are more complex than the components above. The number of execution paths can get really large and difficult to determine as the dependency of a component on the program state and parameters increase. In general, the execution path is complex if the number of branch choices is not deterministic.

```
int loopWithConditions()
{
    int x = 1;
    for(int i = 1 ;i< 1000; i++)
    {
        y = generateNewValue();
        if(y< 50)
        {
            x = x+i;
        }
        else
        {
            x = x - i;
        }
    }
    return x;
}
```

The function above has both loop structure and conditional statements. Execution of the section of code inside the loop during each iteration is dependent on the value returned by the function generateNewValue(). Therefore, during each iteration one of the possible b_{ci} paths within the loop will be covered.

$$d = \sum_{i=0..n}(l_{ui} + y_i)$$

$$y_i = (b_{ci})^{x_i} \text{ when } b_{ci} > 0$$

$$y_i = 1 \text{ when } b_{ci} = 0$$

where

$l_{ui} = 1$ if the ith loop has un-initialized iterator

$l_{ui} = 0$ otherwise

x_i is the number of iterations in i[th] loop

b_{ci} is the number of conditional branches within i[th] loop

n is the number of loops

This is a polynomial equation and the number of the execution paths could get very large with larger values of x_i. It is therefore not practical to find all execution paths for this component. This situation may happen under the following conditions:

- the variables used in the conditional statement is modified due to function returns and have the possibility of varying during each iteration.

- the variables used in the conditional statement are shared variable that can be modified by other concurrent threads.

```
int loopWithBreaks()
{
    int x = 1;
    for(int i = 1 ;i< 1000; i++)
    {
        y = generateNewValue();
        if(y< 50)
        {
            x = x+i;
        }
        else if(y == 75)
        {
            break;
        }
    }
    return x;
}
```

Break within a loop can increase the complexity of the execution path. As illustrated in the function above, during each iteration of the loop one out of all the possible conditional branches is executed. A break statement may occur in one or more of the conditional branches, which causes an exit from the loop. If there are b_{ci} conditional branches and b_{bi} branches with breaks in the ith loop then:

$$p_{bi} = b_{bi} / b_{ci}$$

where p_{bi} is the probability of traversing the branches with breaks in ith loop

If the variables on conditional statements are changing during each iteration, then the situation becomes complex and can only be dealt per use case.

8. JOB CHARACTERIZATION REVISITED

As shown in Section 7, component jobs may be very complicated and the number of component execution paths could be unknown beforehand. How can we then characterize component jobs?

First of all, we can look only at a subset $d_p \subset d$

where d_p is the *performance execution paths*

The *performance execution paths* are a subset of all the execution paths that are interesting from performance point of view. Certain execution paths are not very interesting from performance perspective since they only perform few basic operations and their resource consumption is not very significant. On the other hand, some execution paths that are parts of the performance use case involve intense processing. Only the sections of the component code that generate significant resource load during the use case are important to consider in performance modeling with

reasonable approximation. The relevant code sections could be extracted from code coverage information during execution.

Also if the number of component execution paths is large and difficult to calculate, an approximate characterization functions could be used. Models are not intended to capture all the implementation details. Instead we attempt to capture only the characteristics of the component. The characteristics may be represented as a simple linear regression equation or may be derived from a complex vector table indexed by some input parameter. This method of associating a characteristic function that generates workload to resources and input to other components is more refined than specifying numeric values to service time parameter, as in LQN models.

9. INFORMATION GATHERING

A large amount of information can be gathered for component job characterization from software executions on a system level simulator. Component execution paths, their service requests and resource requirements can be gathered by executing varied use cases.

Measurement is done by loading the executable component image on system level environment that comprises of various system elements like ARM, AMBA, memory, peripherals etc., assembled to form the target platform [10,11].
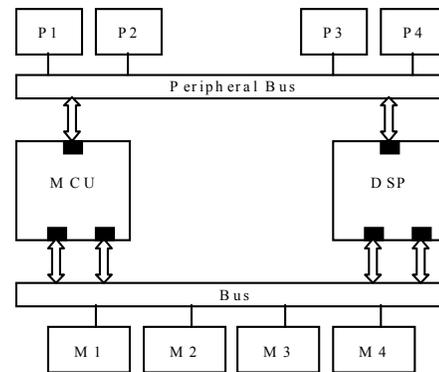

Figure 6. Platform model of mobile device

The software build is prepared for execution on a simulated environment. During the porting process software changes are done to adapt the implementation to system level environment and to ensure successful execution of software on the system level platform model. Software compiled for a specific processor may then be loaded on the processor model. The system level model is composed of various SystemC modules representing hardware elements. The model is a hierarchical composition of various basic system elements. As shown in Figure 6, a typical mobile device platform has an MCU for application processing and a DSP for signal processing. There are interfaces to RAM, ROM, Flash memory etc., through high-speed bus. Peripherals are attached to peripheral bus. Using SystemC, a library of various system elements is created with reasonable behavioral accuracy. However, for complex elements, like the processor, IP models may be purchased and integrated into the design. For example, ARM provides cycle-accurate SystemC models of ARM9 processor family that are very close approximations to actual device behavior. Transaction level models are high-level SystemC simulation models that are used to describe hardware.

At this level, data transfers are modeled as transactions. System level modeling activity includes architecture modeling, workload generation, execution of software on process models, and flow of data to other system elements. Appropriate monitors are inserted to collect performance data during simulation.

System level models are developed in order to facilitate execution of the software on a platform that may not be yet available as hardware implementation. The system simulation environment aids in monitoring the service request and resource usage. By executing the components on such models the actual resource usage may be captured by logging the simulation cycles. Monitoring the input and output parameters of a component during execution assists in characterizing the component. The process of using measured performance data from simulated target environment to characterize component offers realistic and practical parameter values for components.

Information gathering can be done by executing the whole use case. Such approach may take a lot of simulation time on SystemC simulator. We have also considered an approach that would extract components out of the software system and execute them in separation.

Components are extracted by monitoring their communication with other components, via traces collected during use case execution, determining component boundaries and then compiling the extracted components separately to execute on the system model.

The process of component extraction is currently manual. We are developing techniques to automate this process.

10. CONTEXT OF COMPONENT

For a software component to execute in isolation, certain pre-conditions need to be met. These pre-conditions are referred to as *context* of the component. It can also be viewed as the state of the hardware and software required for execution of the software component. It is required to capture the context in order to facilitate execution of single component or a set of related components in isolation.

The context of the component can be captured by using the trace information obtained during scenario execution and by checkpointing techniques.

```
void NewObject::myContext()
{
    StateType x;
    x = iMyObj->GetState();
    switch(x)
    {
    case 1: anotherFunction(); break;
    case 2: …
    default:
    }
}
```

While executing a software component, the section of code may assume that some of the objects that are involved are already created and initialized. For example, the function above assumes that attribute iMyObj is already created and initialized. In order to execute this function an object of type NewObject needs to be created and the attribute should be initialized properly. Also, functions that are called from this component could expect certain variable initializations that need to be satisfied in order to successfully execute. When isolating the component, the object initializations need to be programmed in order to obtain an executable component.

In addition to the data and control logic of the function, the value returned by a function depends on the parameters that are supplied to the function, the attribute initialization, values returned from other functions etc. During a use case execution these values are captured via instrumentation and provided as input during component execution. If it is not possible to extract these values then estimated values are used during simulation.

The event scheduler queues all the events and processes them one at a time. In Symbian, the active scheduler processes one event at a time based on its priority. If the component under consideration is part of a state machine, then based on the events that occurred in the past the system will advance to a specific state. In the source code, the state information is maintained in a state variable. To recreate the context, the component should be brought to appropriate state by initializing the respective state variables.

If a software segment is communicating with other threads then there will be context switches between threads and some information may be exchanged between threads. There can be message exchanges of type *Send()* or *SendReceive()* that may be synchronous or asynchronous. However, if a component is executing in isolation, the delays due to inter-thread communication will not be accounted for. This is addressed by the scheduling mechanism in the use case simulator.

Every software component has some dependency on certain hardware elements. The state of the bus, memory and other peripherals during the execution of the component is initialized as per the bootstrap and hardware abstraction layer specifications. Cache initializations are not trivial and can only be done with approximation by capturing average behavior over several simulation replications.

Capturing component context and execution of component in isolation is a complex task that we have not solved yet. Some work on this has been done for Java in testing community [12,13]. Context is specific to a use case and a components execution state. It is therefore not reusable across multiple use cases.

11. FROM MEASUREMENTS TO CHARACTERIZATIONS

The measurements of various use case scenarios on the system level model provide information for constructing or validating the characterizations of component jobs. One such characterization could be a parameterized regression model [4,14]. For example: the processor load due to window management thread during an image loading process was instrumented by inserting appropriate monitor points and the experiment was replicated multiple times. Table 1, illustrates the data that was captured during the experiment. In table 1, X represents image size in kilobytes and Y represents service time in seconds.

Table 1. Measured service time for image loading

X	Y	SQR(X)	X*Y
1.413	0.074485	1.996569	0.105247
4.079	0.074396	16.63824	0.303461
28.411	0.075843	807.1849	2.154775
113.385	0.076501	12856.16	8.674066
480.058	0.07887	230455.7	37.86217
627.346	**0.380**	**244137.7**	**49.09972**

The regression parameters are estimated as follows:

$$X' = \sum_{i=1..n}(x_i)/n = 125.47$$

$$Y' = \sum_{i=1..n}(y_i)/n = 0.076$$

$$SS_{xx} = \sum_{i=1..n}(x_i)^2 - (\sum_{i=1..n}x_i)^2/n = 165425.1$$

$$SS_{xy} = \sum_{i=1..n}x_i y_i - (\sum_{i=1..n}x_i)(\sum_{i=1..n}y_i)/n = 1.409$$

$$m = 8.52E{-}06, \quad c = 0.0749$$

The derived regression equation is:
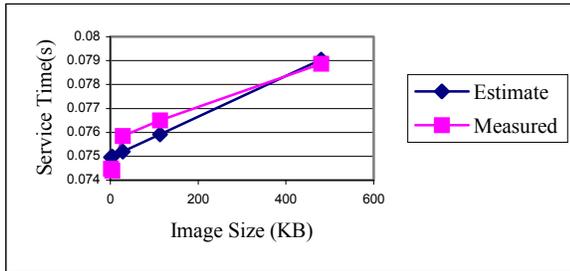
$$\hat{y} = 0.00000852x + 0.0749$$

Figure 7. Service time estimates for image loading

The above Figure 7 shows the measured service time of a window management component job and its corresponding regression estimate. The measured values show linear characteristics and the estimated curve is a very close approximation of realistic values.

12. MODEL ANALYSIS

Once we have models for all jobs in a use case, it is possible to analyze the system performance by executing the use case model on the use case simulator. In the use case simulator, the performance use case is initiated by invoking a specific component job with certain parameters. The simulator executes the rest of the use case, since the service requests to other components are embedded in the job characterization. The simulator also contains the operating system scheduling policy to make the appropriate scheduling decisions.

The use case simulator is used to interpret the component dependency graph and characterization in order to simulate the use case execution. In addition to the corner cases that most formal verification methods use to do schedulability analysis [15], we analyze general cases based on the underlying OS scheduling paradigm and our model of component jobs. The use case simulator executes the queuing model of the system.
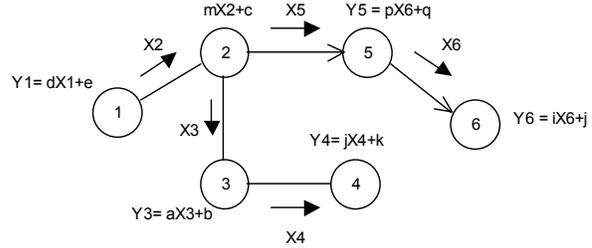

Figure 8. Queuing model in use case simulator

The use case simulator is an interpreter of a hierarchical queuing model of the system [3,4]. In Figure 8, each node of the model represents a component job and the links between nodes represents dependencies and flow of workload from one node to another. Each node's characterization function produces appropriate workload to resources based on input parameters and state. Additional characterization functions produce service requests – workload - to appropriate components in the dependency graph. The use case simulator supports underlying operating system scheduling discipline to schedule the operating system tasks in the model. Tasks are schedulable entities that perform the work associated with the use case. Each task maintains a queue of events and traverses through various component jobs in the system as the events are processed. Multiple tasks may be scheduled in the system. For example: task 1 follows the route 1,2,3 and 4. Similarly, task 2 follows the route 1,2, 5 and 6. Tasks are scheduled using algorithms similar to the operating systems task scheduling mechanism [16,17,18]. This model captures the resources consumed by each component job. Performance metrics of the use case are determined by executing the simulation on the system.

The end user of this model is an architect or developer who need not have performance expertise. The user provides use case as an input to the simulator and obtains the performance characteristics of the entire use case. From the user perspective the initiation of a use case is as simple as feeding an input parameter to the first component in the use case. The simulator handles traversal of the rest of the use case. Since the simulator captures the system behavior, it facilitates the execution of the use case, performance metrics collection and verification against its requirements. The simulator estimates resource consumption for a single or parallel use cases.

We currently have an initial implementation of the use case simulator and are working on the full implementation.

13. CONCLUSIONS

Formal verification is an activity that ensures that the specification satisfies system properties. In the simplest form, verification can be conducted by design walkthrough and code

inspections. In a more elaborate form it involves rigorous testing and simulation. At the highest level, formal verification involves application of mathematical deduction for proving system properties. We have used simulation, state exploration techniques and scheduling analysis to verify that the requirements specified in the use case are satisfied by the system. We take advantage of the cycle-accurate models of the processor available at the system-level to obtain measurements that support characterization of software components and their jobs. The component job models along with the dependency graph serve as a basis for use case analysis of the system. This approach facilitates system performance analysis. An important distinction between our approach and other queuing techniques, like LQN, is that we associate a characteristic function with a component to generate service request and model resource usage. Another unique feature of our approach is the integration of characterization with system level measurement. As a result of these modeling framework enhancements, the estimated values may be more realistic and may be practically used during component planning and early stage system analysis.

The challenging aspect of this approach is the execution path identification and context capture. For a given scenario we can manually extract this information by studying the code. However, to make it more efficient we are developing process and tool support based on source parsers like Source Navigator [19] to automatically capture this information.

Accuracy of results is directly proportional to detailed modeling. However, simulation speed is inversely proportional to detailed modeling. For early stage performance estimation, it is advisable to work with abstract models that provide reasonable accuracy without delving into intricacies.

Performance analysis approach proposed in this paper is still in the initial stages of development, however, we expect it to be useful for performance researchers working on improving performance analysis tools and methods. Challenges presented in the paper could provide a foundation for new exciting results in the model creation and analysis.

14. ACKNOWLEDGEMENTS

15. REFERENCES

[1] C. U. Smith and L. G. Williams, Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software, Addison-Wesley, 2002.

[2] K. Richter, M. Jarsak, Rolf Ernst, A Formal Approach to MpSoC Performance Verification, *Computer*, Vol 38, no 4, April 2003

[3] M. Woodside, Tutorial Introduction to Layered Modeling of Software Performance, Edition 3.0, Carleton University, http://sce.carlton.ca/rads

[4] K. Kant, Introduction to Computer System Performance Evaluation, McGraw-Hill, Inc., 1992.

[5] R. Jain, The Art of Computer Systems Performance Analysis, John Wiley & Sons, Inc., 1991.

[6] D. Jerding, S. Rugaber, "Using Visualization for Architectural Localization and Extraction", *Proceedings of the Fourth Working Conference on Reverse Engineering*, IEEE Computer Society Press, 1997.

[7] N. C. Mendonça, J. Kramer, "Developing an Approach for the Recovery of Distributed Software Architectures", *Proceedings of the 6th IEEE International Workshop on Program Comprehension*, IEEE Society Press, 1998.

[8] E. Stroulia, T. Systa, "Dynamic Analysis for Reverse Engineering and Program Understanding", *Applied Computing Review*, ACM, vol 10, No. 1, 2002.

[9] R. Harrison, Symbian OS C++ for Mobile Phones, John Wiley and Sons Ltd, England, 2003.

[10] T. Grotker, S. Liao, G. Martin, S. Swan, System Design with SystemC, Kluwer Academic Publishers, May 2002.

[11] ARM technical reference manuals, http://www.arm.com

[12] A. Orso, Improving Dynamic Analysis through Partial Replay of User's Executions, *Dagstuhl Seminar* N0 03491, 30.11.-05.12.2003. http://www.dagstuhl.de/03491/Talks/

[13] D. Saff and M. D. Ernst, Automatic mock object creation for test factoring, *In proceedings of Workshop on Program Analysis for Software Tools and Engineering (PASTE'04)*, Washington, DC, USA, June 2004.

[14] D.J. Lilja, Measuring computer performance: A practitioner's guide, Cambridge University Press, 2000.

[15] W. Wolf, A Decade of Hardware/Software Codesign, *Computer*, Vol 38, no 4, April 2003

[16] N. C. Audsley, A. Burns, M. F. Richardson and A. J. Wellings, Hard Real-Time Scheduling: The Deadline-Monotonic Approach, *In Proceedings, Eighth IEEE Workshop on Real-Time Operating Systems and Software*, pp. 133-137, 1991.

[17] C. L. Liu, J. W. Layland, Scheduling algorithms for multiprogramming in a hard real time environment, *Journal of the ACM*, vol. 20, no 1, pp. 46-61, 1973.

[18] RapidRMA tool, Tri Pacific Software, Inc. http://www.tripac.com/

[19] Source Navigator, http://sourcenav.sourceforge.net/