How Much CPU Time?

Expressing Meaningful Processing Requirements among Heterogeneous Nodes in an Active Network

Virginie Galtier, Craig Hunt, Stefan Leigh, Kevin L. Mills, Doug Montgomery, Mudumbai Ranganathan, Andrew Rukhin, Debra Tang

National Institute of Standards and Technology
Gaithersburg, Maryland

Abstract. Active Network technology envisions deployment of virtual execution environments within network elements, such as switches and routers, so that nonhomogeneous processing can be applied to network traffic associated with services, flows, or even individual packets. To use such a technology safely and efficiently, individual nodes must provide mechanisms to enforce resource limits associated with specific network traffic. In order to provide enforcement mechanisms, each node must have a meaningful understanding of the resource requirements for specific network traffic. In Active Network nodes, resource requirements typically come in three categories: bandwidth, memory, and processing. Well-accepted metrics exist for expressing bandwidth (bits per second) and memory (bytes) in units independent of the capabilities of particular nodes. Unfortunately, no well-accepted metric exists for meaningfully expressing processing (i.e., CPU time) requirements in a platform-independent form. This paper proposes a method to express the CPU time requirements of Active Network applications in a form that can be meaningfully interpreted among heterogeneous nodes in an Active Network.

Keywords: Active Networks; Benchmarks; Markov Models; Processor Performance Metrics; Resource Management; Self-Calibration

Send comments to: kmills@nist.gov

I. Introduction

Active Network technology envisions deployment of virtual execution environments within network elements, such as switches and routers, so that nonhomogeneous processing can be applied to network traffic associated with services, flows, or even individual packets. To use such a technology safely and efficiently, individual nodes must provide mechanisms to enforce resource limits associated with specific network traffic. In order to provide enforcement mechanisms, each node must have a meaningful understanding of the resource requirements for specific network traffic. In Active Network nodes, resource requirements typically come in three categories: bandwidth, memory, and processing. Well-accepted metrics exist for expressing bandwidth (bits per second) and memory (bytes) in units independent of the capabilities of particular nodes. Unfortunately, no well-accepted metric exists for meaningfully expressing processing (i.e., CPU time) requirements in a platform-independent form. This paper proposes a method to express the CPU time requirements of Active Network applications in a form that can be meaningfully interpreted among heterogeneous nodes in an Active Network. The ideas proposed here might also be applied more generally, for example to heterogeneous distributed systems, where processes execute on processing nodes that exhibit a wide range of computational capabilities. Such heterogeneous systems could include systems based on mobile code, such as mobile agent applications, or systems based on code loaded from disk, such as distributed parallel processors.

The paper is organized into eight main sections. We begin in Section II by considering the sources of variability in CPU time usage in an Active Network node. To the degree feasible any model we propose must account for all sources of variability. In Section III, we review the state-of-the-art with regard to computer system performance benchmarks. Section IV provides a brief summary of other work related to the problem addressed in this paper. References in this section point to a body of work related to graph theory and Markov models that can be applied to model computer programs. In addition, we identify some techniques to model a computer program using automated code analysis. In Section V, we propose a model to represent Active Network nodes and applications. Using hypothetical examples, we show how our proposed model can be used to express the CPU time required by an Active Network application and to interpret those requirements among a heterogeneous set of nodes in an Active Network. In Section VI we discuss how we might automate the calibration of Active Network nodes, and we describe an approach to automate the generation of Markov models from execution traces of Active Network applications. Section VII reports the results from a modest proof-of-concept prototype we developed to explore our ideas. In Section VIII we enumerate some of the benefits that should result from carrying our ideas forward to a successful implementation. Section IX outlines the future work required to further evaluate and then to implement the ideas outlined in this paper. The paper closes in Section X with our conclusions, which is followed by acknowledgments in Section XI and then by Section XII, which lists references cited throughout the paper.

II. Sources of Variability in CPU Time Usage in an Active Network Node

Any meaningful metric for an application's CPU time requirements must account for the major sources of variability affecting the application. In this section we identify and discuss the major sources of variability likely to affect the CPU time requirements of an Active Network application. A proposed architecture for an Active Network node [1] identifies several components and the relationships among them. The main components consist of a node operating system (or Node OS) and one or more execution environments (or EEs). Each EE provides a virtual execution environment (similar for example to a Java Virtual Machine [2]) in which Active Network applications can execute. Several EEs have been defined and implemented within the Active Networks research community [e.g., see 3, 4, 5, 6, and 7]. In addition, several implementations of a Node OS are being developed [e.g., see 8, 9, and 10]. To enable any EE to run over any implementation of a Node OS, a standard application-programming interface, in the form of system calls, is defined within a separate Node OS specification [11]. Any operating system that claims to be a conforming Node OS must implement the specified system calls. Any Active Network EE can then count on the availability of the specific system calls provided by any implementation of a Node OS. Figure 1 gives a conceptual overview of the major components and relationships within an Active Network node. (Note that all figures have been placed together at the end of the paper.)

Figure 1 divides an Active Network node into four major components: (1) Node Hardware, (2) Execution Environment Layer, (3) Node OS Interface Layer, and (4) Node Operating System Layer. Clearly, the Node Hardware has a large affect on the CPU time requirements of Active Network applications, as well as on the requirements for executing EE and Node OS code. An application that requires 100 CPU seconds to execute on a 10 million instructions per second (MIPS) processor might require only 10 CPU seconds on a 100 MIPS processor. Unfortunately, considering the raw processor and memory speed alone accounts insufficiently for some major sources of variability experienced by an Active Network application.

Consider the Execution Environment Layer in Figure 1. An Active Network application is implemented to execute within a virtual environment provided by a specific EE. This arrangement introduces two main sources of variability. First, each type of EE probably provides an application with different sets of functions. Where functions are the same, different types of EEs are likely to implement the functions differently. This implies that the CPU time required for an Active Network application varies based on the specific EE that hosts the application. For example, an auction application implemented for an ANTS EE could require a different number of CPU seconds than the same application implemented for a PLAN EE. Second, the virtual machine that animates an EE must be mapped onto a specific computing platform. The instruction sets of computing platforms differ, so mappings of virtual machines often differ. This indicates that the same EE might execute with different CPU time requirements when mapped to two different computing platforms with similar MIPS ratings. For example, the same byte codes for the virtual machine within an ANTS EE might require different amounts of CPU time when the virtual machine is mapped to a 300 MHz Pentium II instead of a 300 MHz PowerPC.

Of course, a virtual machine does not provide all the functions within an EE. Many EE functions are mapped to system calls provided by a Node Operating System. In the architecture of an Active Network node, an EE accesses system calls through a standard Node OS API, which is in turn mapped onto real system calls within a Node Operating System. This arrangement introduces several additional sources of variability in the CPU time required to execute an Active Network application. Variability introduced by the Node OS Interface Layer results primarily from the specific mapping chosen, which is often a function of the actual system calls available within a Node Operating System. For example, some operating systems come equipped with a native, multithreading package that permits EE threads to be mapped rather straightforwardly onto underlying threads in the operating system. Other operating systems might schedule execution only at the level of entire processes: EE threads must be mapped onto a multithreading library that is linked into the EE itself. Such differences in mapping will cause differences in the performance that an EE provides to an Active Network application. Even where the mapping to system calls is very similar between two Node Operating Systems, variations in CPU time requirements can still occur based on the fact that these operating systems might implement a similar system call using starkly different techniques. The implementation of similar system calls can differ based on differences in the CPU instruction set or in the programming language used to implement the system calls, or simply based on different algorithms selected by the programmer.

Beyond the mapping of Node OS system calls to native operating system calls, the Node OS specification itself introduces variable CPU time requirements based on the definition of channel, an architectural concept through which active data packets can be sent and received. The abstract concept of a Node OS channel is implemented by underlying protocol modules within an operating system. A path name, similar to a directory and file name, is associated with each channel when the channel is created. The path name identifies a series of protocol modules that will be used to implement the channel. For example, the path name /ANEP/UDP/IP/ETH indicates that packets on a channel will cross four network protocol modules between the send system call and the network device. A separate channel might have a different associated path name; for example, /ANEP/TCP/IP/ATM. For these two example channels, the same Node OS system call, channel-send, might well require different amounts of CPU time because the system call implements different code for each channel. This implies that in some cases identical Node OS system calls can require different CPU time, even when executed on the same computer platform, EE, and Node OS.

One final, important source of variability in CPU time must be considered. Active Network applications are computer programs. A computer program exhibits a range of execution behavior from run to run because many paths exist through the code. The CPU time required by an application will depend quite directly on the specific paths taken during each execution. For this reason, an effective statement of CPU time requirements for a specific application must provide a characterization of its likely behaviors.

To recap, we have identified five main sources of variability affecting the CPU time requirements of an Active Network application. These sources include the following:

(1) the raw performance of the node hardware,
(2) the specific EE in which the application executes, along with the mapping of the EE virtual machine to the node hardware,
(3) the mapping of Node OS system calls to real system calls in the host operating system,
(4) the implementation of real system calls within the host operating system, including the selection of specific protocol modules to implement each instance of a Node OS channel, and
(5) the behavior of an Active Network application itself.

An effective metric for CPU time usage in an Active Network node must account for all of these sources of variability. In Section V, we propose a model designed to respond to this challenge. First, though, we present a brief survey of some existing computer system performance benchmarks used by industry, followed by references to other related work.

III. Some Existing Computer System Performance Benchmarks

Over the years, the computer industry has developed many benchmarks to evaluate the performance of computer systems. The most successful among these benchmarks have evolved over time to account for changes in underlying hardware performance and for shifts in user interest. While these industry benchmarks have different characteristics, they share a common purpose: to enable users to compare performance among computer systems. None of these benchmarks was developed to express the CPU time requirements for specific applications. Still, measurements of performance differences among computer systems provide one essential ingredient in any design that attempts to meaningfully express an application's CPU time requirements among heterogeneous nodes. For this reason, examining existing industry benchmarks proved valuable to us. In addition, we considered other related research work, covered in Section IV, in order to inform our thinking. Readers uninterested in this background can move directly to Section V.

Figure 2 presents a taxonomy we devised to organize the specific benchmarks reviewed. As a main division, some benchmarks are synthetic and some are real, while a few hybrid benchmarks mix both elements. Synthetic benchmarks contain artificial programs designed to mimic characteristics that the designers believe would be exhibited by real programs from a population of interest. The programs are artificial in that they produce no useful results outside the benchmark. Real benchmarks contain programs from a population that the designers either: (1) know is the population of interest or (2) believe behave similarly to programs within the population of interest. The programs used in real benchmarks produce useful results when used outside the context of the benchmark.

The third-level decomposition in Figure 2 divides each class of benchmark into static or dynamic. Primarily benchmarks reviewed here involve actual program execution: classified as dynamic benchmarks. In the case of static benchmarks, selected and previously measured characteristics of a computer system are combined using static formulae to compute a set of relative performance metrics. We reviewed only one prominent static benchmark: the composite theoretical performance calculations used to classify the performance of a computer system for purposes of export control.

The fourth-level decomposition in Figure 2 partitions each benchmark into single-threaded or multi-threaded. A single-threaded benchmark either executes only one program, or executes a series of programs in sequence. A multi-threaded benchmark executes concurrently a mix of programs intended to represent a typical workload for a computer system. We reviewed one benchmark that actually adapts itself from single-threaded to multi-threaded whenever it detects the presence of a multiprocessor. The designers of this benchmark reasoned that the characteristics they were measuring would only see significant multi-threading effects when more that one processor was present.

Below we discuss some significant characteristics of each of the benchmarks we reviewed. The presentation is organized based on the top-level classification in our taxonomy: synthetic, real, and hybrid benchmarks.

A. Synthetic Benchmarks

We reviewed two of the most well known synthetic, dynamic, single-threaded benchmarks: Dhyrstone [12, 13] and Whetstone [14]. Dhyrstone is a short synthetic benchmark program intended to measure the ability of a computer system to handle integer operations. The benchmark consists of about 100 lines of C code that compiles to around 1.5 Kbytes of object code. Due to this small size, memory accesses beyond the L2 cache are not exercised for most modern CPUs. Some argue that the Dhyrstone includes too many string manipulation operations. In addition, compilers can easily optimize the code produced to maximize the performance of a computer on this benchmark. When a processor rating is quoted in MIPS, or millions of instructions per second, the metric typically refers to performance on the Dhyrstone, with the performance of a VAX 11-780 designated as one MIPS. The Whetstone, originally published in 1976, provided the first major synthetic benchmark program intended to assess the ability of a computer to perform floating-point applications. The Whetstone has limitations similar to the Dhyrstone, except that mathematical library functions are over-represented instead of string operations. In many cases, the Whetstone is used as a basis to assign processors with a MFLOPS, or millions of floating-point operations per second, rating. In general, while useful to compare relative computing capabilities of various processors, the Dhyrstone and Whetstone are not representative of any typical applications. For that reason, other benchmark techniques have been devised to represent application mixes.

The most notable synthetic, dynamic, multi-threaded benchmarks we surveyed include four industry transaction-processing benchmarks: TPC-A (on-line transaction processing queries) [15], TPC-B (update-intensive database applications) [16], TPC-C (a mixture of queries and updates) [17], and TPC-D (decision support) [18]. As with other industry benchmarks, these benchmarks continue to evolve [19]. TPC-A and TPC-B were replaced by TPC-C in June of 1995, and TPC-D became obsolete in April of 1999, splitting into TPC-H [20] and TPC-R [21]. A new benchmark, TPC-W [22], is under development to exercise web-based transaction processing systems. The construction of the TPC family of benchmarks is open to industry participants with a stake in the outcome. Aside from the technical content of the benchmarks, members of the industry also agree to a set of rules for running the benchmarks and for reporting results, including submitting the results to an independent body to certify that the benchmark rules were followed correctly.

We found one dynamic benchmark, Wintune98, which adapts its behavior between single threading and multi-threading [23]. Wintune98 contains a set of benchmarks that evaluate a computer system along several dimensions: Dhyrstone, Whetstone, memory access (read, write, and copy), and graphics performance (2-D, 3-D, and OPEN/GL). Interestingly, if the benchmark detects on start up that the computer has multiple processors, then the CPU tests (Dhyrstone and Whetstone) are run using multiple threads. This approach is intended to evaluate the processing boost available from multiprocessor systems.

We also reviewed the synthetic, static benchmark used to compute the composite theoretical performance (CTP) of computer systems in order to apply an export control regime [24]. CTP for a computer system is computed using a static formula relating various characteristics of the system architecture, such as word length, instruction cycle time, number of arithmetic operations per cycle, register-to-register transfer time, and register-to-memory copy time. CTP computations are expressed in terms of MTOPS, or millions of theoretical operations per second. While such a static and theoretical metric might provide an adequate basis for comparing machines against a politically decided limit for purposes of export control, the use of MTOPS, which omits consideration of program execution and workload, does not seem suitable for calibration of processing nodes. Conceivably though, MTOPS, or some similar metric, might be used to predict how various machines would perform relative to a reference machine against a real benchmark. Experimental evaluation would be required to determine the viability of such an approach.

B. Real Benchmarks

We surveyed two prominent industry standard benchmarks, SPEC95 [25] and SYSmark98 [26], which we classified as real, dynamic, and single-threaded. SPEC95 comprises a suite of programs intended to represent classes of applications typically executed on computer systems. The suite divides into two categories: CINT95 (integer-based programs written in C) and CFP95 (floating-point-based programs written in FORTRAN). CINT95 includes an artificial intelligence program, a chip simulator, a C compiler, a pair of compression and decompression programs, a Lisp interpreter, a string manipulation program, and a database program. CFP95 contains ten programs that perform scientific and mathematical computation in a range of domains: hydrodynamics, quantum physics, astrophysics, matrix and differential equation solvers, aerodynamics, quantum chemistry, and plasma physics. SPEC95 also defines several metrics that measure the time taken to complete each of the benchmarks and the number of SPECint and SPECfp that a system can compute per second. Because computer systems and applications continue to evolve, the SPEC committee is working toward a replacement for SPEC95; this replacement is tentatively known as SPEC99.

SYSmark98 also consists of a benchmark made for real programs; however, unlike the SPEC95 benchmark, SYSmark98 draws its programs from actual applications that a typical business user might run on a computer system. For example, SYSmark98 includes office automation software (word processing, spreadsheets, drawing packages, content viewers), graphics software (for creating images and videos), pattern recognition software (speech and optical-character recognition), and web browsing software from a number of commercial vendors. SYSmark98 defines a calibration platform against which can be compared the performance of other computer systems on the benchmark workload. The calibration platform is assigned a rating of 100, so computers performing twice as fast as the calibration platform would be rated at 200. Systems are rated in two categories, office productivity and content creation, and on an overall basis.

We reviewed four industry benchmarks that use real, multi-threaded workloads: WinStone99 [27], NetBench 6.0 [28], WebBench 3.0 [29], and SPECweb96 [30]. Similarly to SYSmark98, WinStone, NetBench, and WebBench use benchmarks constructed from programs that computer users typically execute on computer systems. For example, WinStone99 uses a mix of seven programs: Adobe Photoshop 4.01, Adobe Premiere 4.2, AVS/Express 3.4, Microsoft FrontPage 98, Microsoft Visual C++ 5.0, MicroStation SE, and Sound Forge 4.0. Each of the seven programs receives equal weight in assigning an overall score for performance against the WinStone99 benchmark. Three of the seven applications include a multiprocessor version so that they can be used to assess the increased power available on a multiprocessor workstation. The performance of a computer on the WinStone99 benchmark is compared against the performance of a reference platform, which is given a score of 10.0 by convention. Scores are also computed for performance on individual high-end applications, except in these cases the performance of the reference is assigned a value of 1.0. Unlike WinStone99, the NetBench 6.0 benchmark aims to evaluate the ability of a file server to handle file input and output requests from across a network. This benchmark measures system response time for each user. WebBench 3.0 focuses more specifically on the performance of a computer system as a web server. The standard metrics computed by WebBench encompass two throughput scores: requests per second and bytes per second. Similarly to WebBench 3.0, SPECWeb96 provides a benchmark for comparing the performance of web servers. The SPECWeb96 workload is artificially constructed from an analysis of logs from several web sites. Based on the statistical distributions of file request inter-arrival times and return sizes, files are placed on the web server in four size classes and a workload generator makes get requests for these files on the system under test. Because the benchmark leaves the choice of web server software up to the system under test, the benchmark will generally assess both the computer system platform and the web server software being used. Throughput is the main metric used in SPECWeb96.

C. Hybrid Benchmarks

We examined only one hybrid benchmark, WinBench99 [31], which uses synthetic elements, such as CPUmark99 [32], to assess the performance of a system's CPU and memory. WinBench99 is a system level benchmark that measures the performance of the graphics, disk, processor, and video subsystems of a computer platform, but only for systems running some version of the Microsoft Windows operating system. WinBench uses a mix of real application programs and synthetic programs to evaluate the characteristics of a computer system. Two of the synthetic programs, CPUmark99 and FPU Winmark99, assess the integer and floating point performance of a system. Other component programs, labeled as WinMark, assess a system's performance on business and high-end graphics and business and on high-end disk access. The composition of the WinBench test suite and the large variety of metrics do not lend themselves to calculating a single figure of merit for a particular system; instead, the performance of a system is reported as a table of results on each of the individual subsystems evaluated.

IV. Other Related Work

Computer science researchers and practitioners have explored the use of graph theory and Markov models to represent computer systems and programs and to predict performance characteristics at both micro and macro levels [35-42, 45]. These previous explorations have guided our current deliberations. Other researchers have explored additional techniques to analyze program code in order to predict the performance of programs on various computer systems [33-34, 43-44, 46]. At present we intend to base our approach on Markov modeling techniques; however, if this does not yield the expected results, then we will attempt to exploit some of these other techniques. For example, Saavedra and Smith [33, 34] developed a machine-independent model of program execution that formed the basis for characterizing both machine performance and program profile. Their goal was to predict the execution time for arbitrary combinations of computer systems and programs. Their approach involves defining a computer system in terms of abstract operations. Then a computer system can be benchmarked to determine the number of these abstract operations per second that can be executed. A given program (typically a standard industry benchmark) can be analyzed statically to identify and mark the abstract operations included in the program, and then monitored during execution to determine the number of times each abstract operation is executed. The total workload presented by a program is then simply the number of abstract operations that must be executed. Knowing the number of abstract operations per second that a given computer system can perform, the predicted execution time for the workload is then simply a linear vector multiplication of the workload and the machine performance for each abstract operation. While this is an appealing approach, the workload characterization, and thus the resulting performance prediction, is valid for only a particular instance of execution behavior, that is, a single run of a program. In our case, we will need to capture many possible executions of a program. The fundamental approach of Saavedra and Smith might possibly be extended to capture the stochastic behavior of a program.

V. Modeling Active Network Nodes and Applications

In this section, we define two related models, one for Active Network nodes and one for Active Network applications. Taken together, this pair of models can be used to meaningfully express the CPU time requirements of an Active Network application in a form that can be interpreted by heterogeneous nodes in an Active Network. After defining these models, we also define two transformations. One transformation converts an Active Network application model from a local CPU time scale to a reference CPU time scale, while the other transformation inverts this conversion. We begin by defining a model for Active Network nodes.

A. An Active Network Node Model

An Active Network node consists of a set of one or more execution environments (EEs) mapped onto a node operating system through a standard set of systems calls, outlined in a Node OS Interface Specification. Refer back to Figure 1 for an illustration of such a node. The software composing the EEs and the Node OS operates on node hardware. Our model for an Active Network node must account for variations in CPU time requirements that can be introduced by all of these elements. (A separate model, as discussed below, accounts for variations introduced by specific Active Network applications.) We begin by examining the Execution Environment Layer from Figure 1.

An Active Network node must support at least one EE, but may also support multiple EEs (for example, EE_1 through EE_n in Figure 1). On a given node, each EE will exhibit independent performance characteristics on workloads typical of Active Network applications. For that reason, we represent the performance of each EE on a node as the time taken by the EE to execute a representative workload. The performance for all EEs on a node can be represented as a vector, EE_{node}, where the time taken for EE_n to execute the representative workload on a node is given by element $EE_{node}[n]$. To account for differing characteristics among the node platforms, we select a specific node as a reference node. The performance of EEs on the reference node is distributed to all Active Network nodes as a vector, $EE_{reference}$, where the time taken for EE_n to execute the representative workload on the reference node is given by element $EE_{reference}[n]$. Using these two vectors, EE_{node} and $EE_{reference}$, the performance of EE_n on a specific node platform can be expressed relative to the performance of EE_n on the reference platform by using the ratio: $EE_{reference}[n]/EE_{node}[n]$. A concrete example follows.

Imagine a hypothetical Active Network consisting of three EEs, and two nodes, A and B. Suppose we execute a representative Active Networks application workload for each EE on a reference node and on the other two nodes, producing the results shown in Table 1. In this hypothetical example, node A executes the representative workload for each EE approximately twice as fast as does the reference node; the ratios range from about 1.86 to 2.01. Node B executes the representative workload for each EE nearly ten times faster than the reference node; the ratios range from 8.77 to 9.88.

Table 1. Measuring Relative Node Performance for Execution Environments

	$\mathbf{EE_1}$	$\mathbf{EE_2}$	$\mathbf{EE_3}$
$\mathbf{EE_{reference}}$	.456 s	.758 s	.326 s
$\mathbf{EE_{nodeA}}$	.228 s	.378 s	.175 s
$\mathbf{EE_{nodeB}}$	.052 s	.084 s	.033 s
$\mathbf{EE_{reference}/EE_{nodeA}}$	2	2.005291	1.86285714
$\mathbf{EE_{reference}/EE_{nodeB}}$	8.77	9.02	9.88

With a few adjustments, we can use a similar approach to model the Node OS Interface Layer and the Node Operating System Layer illustrated in Figure 1. Each Active Network node provides a set of Node OS system calls (for example, S_1 through S_m in Figure 1). This set of system calls will be implemented differently for specific node operating systems. Even when implemented for the same node operating system, each system call may execute with differing CPU time requirements on various node hardware platforms. For these reasons, we must consider the performance of the node operating system on each Active Network node. Assume that a benchmarking program can repeatedly execute the Node OS system calls, and then compute the average time taken to execute each system call. To account for the fact that channels can be composed from various protocol module graphs, treat each relevant combination of module graphs as a distinct system call. So, for example, a single channel-send call might be transformed into ten channel-send calls, one for each combination of protocol modules supported by a specific node operating system. The performance for all system calls on a node can be represented as a vector, S_{node}, where the average time taken for the node to execute system call S_m is given by element $S_{node}[m]$. As before, to account for variations among the node platforms, we select a specific node as a reference node. The expected performance for system calls on the reference node is distributed to all Active Network nodes as a vector, $S_{reference}$, where the average time taken to execute system call S_m on the reference node is given by element $S_{reference}[m]$. Using the vectors S_{node} and $S_{reference}$, the expected performance for S_m on a specific node platform can be expressed relative to the expected performance for S_m on the reference platform by using the ratio: $S_{reference}[m]/S_{node}[m]$. An illustrative example follows.

Imagine a hypothetical Node OS consisting of four system calls. Suppose we execute a benchmark program that repeatedly invokes system calls on a reference node, and that subsequently computes the average time taken to execute each system call. Further, assume we repeat this process on the other nodes, A and B, composing our hypothetical Active Network. Table 2 shows some results that might be obtained in such an exercise. In this hypothetical example, the reference node executes Node OS system calls significantly faster than node A. But the differences in performance vary notably from call to call. For example, the reference executes S_1 twice as fast as node A, but executes S_3 three times as fast. In the other case, node B performs better than the reference node on the four system calls, although the difference is not great, ranging from 10% to 33% faster.

Table 2. Measuring Relative Node Operating System Performance for Node OS System Calls

	S_1	S_2	S_3	S_4
$S_{reference}$	.0054 ms	.0109 ms	.0012 ms	.0075 ms
S_{nodeA}	.0108 ms	.0179 ms	.0036 ms	.0167 ms
S_{nodeB}	.0045 ms	.0099 ms	.0009 ms	.0069 ms
$S_{reference}/S_{nodeA}$	.5	.61	$.\underline{33}$	.45
$S_{reference}/S_{nodeB}$	1.2	1.1	$1.\underline{33}$	1.09

As presented, our Active Network node model can account for variability among nodes due to the EE used (and its mapping to an underlying platform) and the system calls invoked (as well as their implementation within a specific operating system mapped to a particular computer instruction set). By using the concept of a reference node, the model can also account for these variations in a form that adjusts for differences in the raw performance of node hardware. The behavior of individual Active Network applications provides the remaining source of variability that must be addressed. For this purpose, we conceived an Active Network application model, explained next.

B. An Active Network Application Model

In order for an Active Network node operating system to provide effective enforcement of CPU resource limits, each Active Network application must declare the CPU time required by the application. In a simple approach, an application might declare the average CPU time required for its execution. This single characteristic does not provide the node operating system with much information about the behavior of an application. A significant improvement would add a second metric, such as the variance in CPU time required by an application. This second metric provides more information to the node operating system. However, since many potential paths exist through an application, the average and variance in CPU time required by an application would prove more meaningful if the application could also indicate the statistical distribution from which these two statistics were drawn. Unfortunately, many potential statistical distributions exist, and differing distributions might apply to various Active Network applications. The potential range of variability in classes of statistical distributions could significantly complicate the resource enforcement mechanisms implemented in a node operating system. We propose a more general model that should capture a significant range of application behavior in a form suitable for CPU usage enforcement decisions within a node operating system.

We propose to express the CPU time requirements of an Active Network application using a continuous-time, finite-state Markov model [47]. Recall from Figure 1 that an Active Network application executes within an execution environment (EE), but requests periodically service from the node operating system. Each service request is issued through a specific system call (e.g., S_1 through S_m in Figure 1) provided by a Node OS Interface. The available Node OS system calls are mapped onto real system calls in the node operating system. Assuming that a monitor is placed at the boundary between an EE and the Node OS system calls, the behavior of an Active Network application within the EE can be viewed as a series of transitions between specific states, where each state is a Node OS system call. During each transition, an application executes for some amount of CPU time within its EE. While in each state, an application executes for some amount of CPU time within the corresponding system call. (An alternative view simply adds the CPU time spent in a state to each outgoing transition from the state.)

Figure 3 provides an abstract formulation of our proposed model as a state-transition diagram. We assume that each Active Network application can identify its beginning point, as well as its exit points. These correspond to state S_0, the Idle State, in Figure 3. Each of the other states (S_1 through S_m) corresponds to a Node OS system call. Assuming that an Active Network application has made a transition into a specific state, say S_n, then the application will transition to a next state, S_{n+1}, with some probability. Each transition in Figure 3 is labeled with $P_{Sn-Sn+1}$, which represents the probability of transitioning from S_n to S_{n+1}. While an application is in a particular state, S_n, the application will use some amount of CPU time, T_{Sn}; each state in Figure 3 is annotated with this value. During a transition between S_n and S_{n+1}, an application will use some CPU time, $T_{Sn-Sn+1}$. Each transition in Figure 3 is labeled with this value. Figure 4 provides an alternate representation of this model as a vector, SC_{vector}, which contains the average CPU time used by the application when executing each system call, and a matrix, EE_{matrix}. Each cell in the EE_{matrix} contains a two-element vector, which contains the probability of transitioning to a specific state and the average CPU time used by the application during the transition. This model represents each execution path through an application as a series of transitions that begins in the idle state, S_0, and then returns there. This expressive power comes at a minimal cost in size. Since the matrix and vector representations can be sparse, the information they contain can be expressed in a reasonably compact form that includes only those Node OS system calls actually used by an application.

When an Active Network application uses our proposed model to represent its CPU time requirements, a node operating system will possess a significant amount of information about the likely behavior of the application. Under appropriate assumptions, this information can be used to estimate the distribution of likely execution times that an application will exhibit on a node. We can pool all states beyond S_0 into one large state, say S', creating a two-state Markov chain. If we assume that the chain is stationary (meaning that transition probabilities are homogeneous in time) , then the distribution of the dwell time in each state will be exponential. (Even if the chain is not stationary, the distribution of measured dwell times might prove to be exponential, in which case the following analysis also holds.[1]) Given these assumptions, then the time to leave state S_0 (equation 1) and S' (equation 2) can be written as follows.

$$P(\varepsilon > t) = e^{-\lambda_0 t} \qquad \text{(equation 1)}$$

$$P(T > t) = e^{-\lambda t} \qquad \text{(equation 2)}$$

Using this approach, the dwell time in state S', given by equation 2, represents the first return time to state S_0. The distribution of average dwell times in state S' can be represented as an exponential distribution with parameter $\lambda* = 1/T$, where T can be computed using the following equation.

$$\frac{\left(\sum_{j=1}^{n} Xj\right)}{n} \qquad \text{(equation 3)}$$

In this equation, each Xj represents the observed average dwell time in one of the component states aggregated together to form S'. We can exploit this information to partition the distribution of average dwell times in state S' into two parts, α and $1-\alpha$, where each part represents a region

[1] If the measured dwell times prove not to be exponentially distributed, then we will seek other results from the literature that attempt to model the distribution of first return times for Markov chains with other statistical distributions, such as gamma or log normal.

of the distribution in which some percentage of dwell times in S' can be found. For example, if α is .05, then five percent of the average dwell times will have values greater than t_α, the value corresponding to α, while 95% of the average dwell times will have values smaller than t_α. For an exponential distribution, a given α can be computed using the following equation.

$$\alpha = e^{-\lambda * t\alpha} \qquad \text{(equation 4)}$$

From this equation, t_α can be found as follows.

$$\lambda * t_\alpha = -\log \alpha \qquad \text{(equation 5)}$$

$$t_\alpha = -(1/\lambda *) \log \alpha \qquad \text{(equation 6)}$$

Substituting 1/T for λ*: $t_\alpha = -T \log \alpha$ (equation 7)

Given equation 7 and a Markov model representation for an Active Network application, the expected return time, t_α, corresponding to any selected α can be computed easily. Using this technique the Node OS for an Active Network node can determine the expected CPU time that will be needed by an arbitrary proportion of executions of an application by selecting α to correspond to the percentage of interest. For example, computing the t_α for α = .05 will give the value that partitions the distribution of expected execution times into 95% and 5%. In other words, for 95% of the executions of an application, the CPU time required would fall below t_α, while for 5% of the executions the required CPU time will fall above t_α. Consider the following example.

Imagine a hypothetical Active Network node, A, that comprises a single execution environment, EE_1, and that executes on a node operating system providing only four Node OS system calls, S_1 through S_4. Further imagine an Active Network application, A_1. Table 3 shows an Active Network model representing A_1. Such a model can be computed from measurements taken during iterations of A_1 on node A. (See Section VI. B. for an example how this might be done.)

Table 3. Representing an Active Network Application Model for Node A

SC_{vector} System Call	CPU Time	EE_{matrix}	To S_0	To S_1	To S_2	To S_3	To S_4
S_0	0.0000	**From S_0**	0|0	.8|1234	.2|457	0|0	0|0
S_1	0.0114	**From S_1**	.05|2345	.6|347	.25|423	.1|256	0|0
S_2	0.0165	**From S_2**	.25|337	.15|1115	.2|313	.2|109	.2|92
S_3	0.0280	**From S_3**	.01|1632	.55|756	.04|577	.3|188	.1|89
S_4	0.0157	**From S_4**	.6|3521	.1|982	.2|345	.05|345	.05|107

In Table 3, SC_{vector}, contains the average CPU time used by application A_1 during each invocation of a system call, S_0 through S_4. These times might be similar but not identical to the times shown in the earlier benchmark of Node OS system calls for node A (see Table 2). These times represent the experience of A_1 with each system call. For simplicity, Table 3 shows all CPU times in milliseconds. Each of the remaining cells in Table 3, EE_{matrix}, show the probability that A_1 will transition between two system calls, followed by the average CPU time used in EE_1 by A_1 during the transition. The probabilities in each row, that is, from each state, must sum to unity. From the model shown in Table 3, what can be said about application A_1 when executing in EE_1 on node A using system calls S_0 through S_4?

Using the Active Network application model, presented in Table 3, and equation 3 and equation 7, a Node OS can compute the expected CPU time required for an arbitrary proportion of executions of the application. For example, let's determine the CPU time that should suffice for 95% of the executions of the application represented in Table 3. In other words, let $\alpha = .05$. First, the value of T must be found using equation 3. Table 3 contains 21 observations, so n is 21. The sum of Xj can be computed by adding the time value contained in each cell in Table 3. Actually the time taken by the system calls, S_n, should be distributed across each corresponding row of Table 3 before Xj is computed; however, the values of the S_n are so small that we have omitted them from the current computation. Summing over each cell we find that Xj is 15,570. By dividing Xj by n, T is computed as 741.428571. Now, using equation 7, we can determine the value of t_α to be approximately 965 milliseconds. This means that 95% of the executions of the application encoded in Table 3 can be expected to execute in fewer than 965 milliseconds of CPU time on node A. This computation can be scaled quickly for any other threshold of interest. For example, selecting α of .01 and reevaluating equation 7, we find that 99% of all executions of the application can be expected to execute in fewer than about 1,483 milliseconds of CPU time on node A. More generally, assuming an exponential distribution, Figure 5 graphs the probability density function and cumulative distribution function of expected CPU times, derived from Table 3 using equation 3 and equation 7, for the application executing on node A. The figure indicates the 95^{th} and 99^{th} percentiles.

C. Application Model Transforms

The preceding section described how to model the behavior of an Active Network application on a specific node. Once it exists, such a model must be sent to Active Network nodes that will execute the application, and that will enforce the CPU time requirements expressed within the model. Unfortunately, the CPU time values included in the model will have no meaning on Active Network nodes that differ from the node on which the model was generated. To overcome this problem, we define two transformations: (1) node-to-reference (NR) and (2) reference-to-node (RN). Prior to transferring an Active Network application model between two nodes, A and B, the model is subjected to an N_AR transform. The model, with its CPU time requirements expressed in terms of a reference node, is then transmitted across the network. Upon arrival at node B, the model is subjected to an RN_B transform. The combination of these two transforms will convert the CPU times within an application model from a form meaningful on node A into a form meaningful on node B. Refer to Figures 6 and 7 to understand how these transforms work.

Figure 6 gives the algorithm for the NR transform. Given the index for the EE used by the application and the number of system calls supported by the node operating system, the transformation algorithm iterates twice over the range of system calls. The exterior iteration converts the CPU times in SC_{vector} from the local node basis to a reference basis. The conversion multiples each CPU time in SC_{vector} by a corresponding factor, $S_{reference}[i]/S_{node}[i]$, which represents the relationship between the average performance of the local node and the reference node on a specific system call. The interior loop iterates over the cells in EE_{matrix}. For each cell, the CPU time value, T, is multiplied by a factor, $EE_{reference}[n]/EE_{node}[n]$, which represents the relationship between the average performance of the local node and the reference node on the benchmark workload for EE_n.

Figure 7 gives the algorithm for the RN transform. This transform follows the outlines of the NR transform, except that the conversion process must be inverted; so, the SC_{vector} conversion factor is now $S_{node}[i]/S_{reference}[i]$ and the EE_{matrix} conversion factor is now $EE_{node}[n]/EE_{reference}[n]$. A concrete example follows, using the previously discussed nodes A and B.

Earlier, in Table 3, we presented a model for a hypothetical Active Network application, A_1, executing on node A. Table 4 gives that same model after applying the NR transform to convert the CPU times into a reference form. Table 5 then shows the model after applying the RN transform to convert Table 4 from the reference form to a form understood by node B.

Based on the fact that node B executes the benchmark workload for EE_1 about 4.4 times faster than node A, the EE_{matrix} in Table 5 should predict that application A_1 will execute with similar improvement on node B. This proves to be the case. Similarly, since node B executes system calls on average between two and four times as fast as node A, depending on the system call, the SC_{vector} in Table 5 should reflect this in the predictions for the average CPU time used by application A_1 for each system call. As you can see, this is also the case.

Table 4. Reference Model for Application A_1 after Applying the NR Transform to the Node A Model

SC_{vector}			EE_{matrix}				
System Call	**CPU Time**		**To S_0**	**To S_1**	**To S_2**	**To S_3**	**To S_4**
S_0	0.0000	**From S_0**	0|0	.8|2468	.2|914	0|0	0|0
S_1	0.0057	**From S_1**	.05|4690	.6|694	.25|846	.1|512	0|0
S_2	0.0101	**From S_2**	.25|674	.15|2230	.2|626	.2|218	.2|184
S_3	0.0092	**From S_3**	.01|3264	.55|1512	.04|1154	.3|376	.1|178
S_4	0.0071	**From S_4**	.6|7042	.1|1964	.2|690	.05|690	.05|214

Table 5. Node B Model for Application A_1 after Applying the RN Transform to the Reference Model

SC_{vector}			EE_{matrix}				
System Call	**CPU Time**		**To S_0**	**To S_1**	**To S_2**	**To S_3**	**To S_4**
S_0	0.0000	**From S_0**	0|0	.8|281	.2|104	0|0	0|0
S_1	0.0047	**From S_1**	.05|535	.6|79	.25|96	.1|58	0|0
S_2	0.0092	**From S_2**	.25|77	.15|254	.2|71	.2|25	.2|21
S_3	0.0069	**From S_3**	.01|372	.55|172	.04|132	.3|43	.1|20
S_4	0.0065	**From S_4**	.6|803	.1|224	.2|79	.05|79	.05|24

Once Table 5 is obtained, equation 3 and equation 7 can be used by the Node OS on node B to compute the expected CPU time required for a threshold of interest. For example, let's determine the CPU time required on node B that should suffice for 95% of the executions of the application represented in Table 5. As before, let $\alpha = .05$. From equation 3, the value of T can be found. As before, n is 21. The sum of Xj can be computed by adding the time value contained in each cell in Table 5. Again, we omit the time taken by the system calls, S_n. Summing over each cell, we find that Xj is 3,549. By dividing Xj by n, T is computed as 169. Now, using equation 7, we can determine the value of t_α to be approximately 220 milliseconds. This means that 95% of the executions of the application can be expected to execute in fewer than 220 milliseconds of CPU time on node B. Selecting α of .01 and reevaluating equation 7, we find that 99% of all executions of the application can be expected to execute in fewer than about 338 milliseconds of CPU time on node B. More generally, Figure 8 graphs the probability density function and the cumulative distribution function of expected CPU times, derived from Table 5 using equation 3 and equation 7, for the application executing on node B. The figure indicates the 95th and 99th percentiles.

VI. Calibrating Active Network Nodes and Generating Application Models

To implement the model for Active Network nodes, as presented in section V.A, we must devise a process to set values for the various EE_n and S_n measures on every node, including the reference node. We use the term node calibration to denote this process. Further, we must provide a mechanism to generate Active Network application models (SC_{vector} and EE_{matrix}), in the form presented in section V.B, for specific Active Network applications. We use the term model generation to denote this mechanism. Finally, we must augment the Node OS interface with functions needed to support node calibration, model generation, and model transformations. The following paragraphs discuss these topics, beginning with node calibration.

A. *Approaches to a Self-Calibrating Active Network (SCAN) Node*

For each Active Network node, including the reference node, values must be established that represent node performance on specific EEs and on specific system calls. Perhaps the most intuitively appealing approach to calibrate node performance is to execute a dynamic workload that includes a realistic mix of actual Active Network applications implemented for each EE. Unfortunately, Active Network technology is new, and the few applications that have been developed are intended for experimental purposes. No reasonable possibility exists at this stage to define a realistic mix of actual Active Network applications. We could simply use a mix of the experimental applications that currently exist, ensuring that we include a versioning capability into the calibration scheme so that the benchmark workload can change over time as the pool of applications grows. (No matter what approach we select, inclusion of a versioning mechanism will be necessary.) As an alternative, one might attempt to define a workload of artificial applications whose behavior mimics major classes of Active Network applications, as envisioned for a future time. Here again, too little knowledge exists at present about the likely classes of Active Network applications to make a credible proposal for an artificial, representative workload. One could examine the experimental applications, identify major classes among those applications, and then select, or implement, a representative of each class for inclusion in the workload.

While appealing, selecting representative workloads, whether real or artificial, does have a specific drawback. Given that an EE provides a wide range of functions to Active Network applications, it seems likely that few applications will use all functions. In fact, among any representative set of applications, it might turn out that not all functions in an EE will be exercised. Omitting rarely used functions could provide an inadequate characterization of the performance of an EE on an Active Network node. To overcome this flaw, a synthetic benchmark workload might be designed to execute all the functions provided by an EE. Such an approach would simply ignore the usage patterns of specific Active Network applications, or classes of applications. Using this approach, versioning of the benchmark workload would be used to distinguish among versions of each EE. In this way, as functions evolve in an EE, the synthetic benchmark can evolve in concert.

To recap, we could use real Active Network applications to construct a benchmark workload for each EE, we could use representative applications that behave as would major classes of Active Network applications, or we could use a synthetic benchmark that repeatedly exercises all the functions within an EE. Of course, we could also use a hybrid of these approaches. No matter what course we adopt, each benchmark must have an associated version number so that benchmarks can evolve over time.

The approaches considered to this point require that each node execute a benchmark program for each EE on the node in order to achieve calibration. Running such benchmarks can be costly in terms of CPU time (more about this in the next paragraph). As an alternative approach, we might consider executing EE benchmarks only on the reference node, and then using some static calculation, such as the ratio of a node's MTOPS rating against the MTOPS rating of the reference node, to quickly compute a projected performance of the node on the benchmarks actually executed on the reference. While quick, this approach will not account for variability in mapping the various EEs to specific operating systems and CPU instruction sets; thus, this approach should prove less accurate then would executing benchmarks on each node. We might consider verifying this supposition with actual experiments.

Beyond calibrating a node with respect to EE performance, we must also calibrate each node with regard to performance on each Node OS call. Here the choice seems fairly clear: we must develop a synthetic benchmark program that repeatedly exercises each system call, and then computes the average CPU time used when each call is invoked. Given the wide range of operating systems and methods and languages to implement system calls, an approach involving MTOPS ratings seems even less likely to apply here than was the case for EE calibration.

Since calibration is likely to involve substantial computation on a node, we must consider appropriate means to perform the calibration. Several approaches should be explored. We could perform the calibration off-line, and then store the results as parameters within a Node OS. This approach has the merit of requiring no resources from a node during operational execution. Of course, whenever a system configuration changes, such as a new version of the operating system, an added protocol implementation (implying an additional Node OS channel and system call), or a replacement networking interface card or device driver, the previously computed off-line calibration may no longer prove accurate. As a second alternative, we could consider boot-time calibration. Here, the calibration programs would execute automatically as part of the boot process in the operating system. The approach has two advantages. First, since most operating systems must be rebooted after significant configuration changes, calibration at system boot is likely to account for the variability introduced by system alterations. Second, since calibration is completed prior to system execution, the calibration process will require no resources during node execution. One downside is that boot calibration could lengthen the boot time considerably. Additionally, future operating systems seem destined to include dynamic configuration through components downloaded during execution. Boot time calibration could not account for such dynamic run-time changes in an operating system.

A third alternative is to execute an off-line calibration, and then to perform run-time calibration adjustments. Here boot time would not be lengthened due to calibration requirements. In addition, configuration changes that affect the calibration can be accounted for during the run-time calibration adjustments. One might even consider altering automatically the frequency of run-time calibration adjustments depending on the variance computed between successive calibrations. As the variance diminishes between successive calibrations, the calibration adjustment interval could be lengthened. Conversely, increasing variance would stimulate more frequent calibration adjustments. The approach has two drawbacks. First, run-time calibration adjustments would subtract resources from operational uses of a node. Second, it might prove difficult to design and implement an effective run-time calibration adjustment mechanism.

B. *Trace-based Generation of Application Models*

To help designers of Active Network applications generate Markov models corresponding to specific applications, we envision providing a software tool, specifically an application model generator, which consumes execution traces and then produces tabular models in the form shown earlier in Figure 4. In this section, we discuss the type of trace that a system will need to generate in order to inform such a tool. For purposes of trace generation, we will assume that an application executes alone within a virtual machine, wrapped in a process, and that all threads in the process belong to the application. Under this assumption, accounting for the CPU time used by a process should prove sufficient; accounting on a per thread basis will not be necessary. Given these assumptions, a process uses CPU time to execute within a virtual machine and to execute within system calls on behalf of a virtual machine; thus, we will need to track the CPU time used by a process both between and during system calls. A sufficient execution trace will provide information as a set of 4-tuples, as follows.

<process id> <call id> <entry | exit> <cumulative CPU time>

The <process id> element identifies the process wrapping the virtual machine. The <call id> element selects the system call that the application invokes, and each trace line represents either an entry to or exit from a system call (as denoted by the third element). The fourth element contains the amount of CPU time used cumulatively by the associated process. From a trace of these 4-tuples, a program can be written to build a Markovian application model.

C. *Implications for the Node OS Interface and Active Network Protocols*

To permit an Active Network node to use the services outlined in this paper, the Node operating system application-programming interface must be extended to include several new operations. Add, delete, and modify operations must be defined to manage the following information: (1) EE and System Call benchmark times obtained from calibrating the reference node and (2) EE and System Call benchmark times obtained from calibrating the local node. In addition, interfaces must be defined for application model transformations, including both the NR and RN transforms. An interface must also be provided to compute appropriate values for the CPU time requirements contained within an application model. Some care will be needed to properly connect the information in our models to the other information maintained by a node operating system. Depending upon the approach used for calibration, interfaces might be needed to start and stop self-calibration on a node.

Beyond extensions to the Node operating system interface, formats must be devised to allow Active Network protocols to represent, encode, transport, and decode application models. These representations must be crafted to carry the necessary information with the least amount of overhead. Similar mechanisms might be developed to allow remote update of reference times.

VII. Proof-of-Concept Results

To gain confidence in our proposed approach to provide Active Network application (ANA) developers with an automated tool that can generate ANA models in the form similar to the one we outlined earlier in Section V. B., and to evaluate the distribution of CPU time usage in an ANA, we developed a modest proof-of-concept prototype. For this proof-of-concept, we considered only two EEs: the OCaml version of PLAN and ANTS, both running on Linux. We generated a trace from the Active Networks version of the ping application running in ANTS. The main objective of our prototype is to obtain an execution trace of an Active Network application. The execution trace will have the following format.

$<SC_i>$ $<SC_j>$ $<SCT_i>$ $<EET_{i,j}>$ $<CPU_{i,j}>$

Where each line represents a transition between two Node OS system calls, and

$<SC_i>$ is a unique integer number assigned to identify the "from" NodeOS system call,
$<SC_j>$ is a unique integer number assigned to identify the "to" NodeOS system call,
$<SCT_i>$ is the CPU time spent while executing the "from" system call,
$<EET_{i,j}>$ is the CPU time spent while executing in the EE between $<SC_i>$ and $<SC_j>$, and
$<CPU_{i,j}>$ is total of $<SCT_i>$ + $<EET_{i,j}>$.

Consider the following example scenario in the life of a process, P, executing on behalf of an EE and invoking Node OS system calls. P begins by performing some operations in EE mode, outside of the kernel, and then it invokes an operating system call S1, which executes on behalf of P, but in kernel mode. Upon return from S1, P performs additional operations in EE mode, and then invokes system call S2. Specifically, let's say P is chosen by the scheduler and begins to execute at time 10 (this is wall clock time, not CPU time). Suppose that at time 12, P had finished its operations and called S1, which enters the kernel; then at time 16 P exits the kernel and begins its next set of operations. But at time 20 the scheduler suspends P and allows other processes to run. At time 40, the scheduler again selects P to continue execution. Then P finishes the set of operations underway before being suspended, and enters the kernel at time 48 to execute system call S2. At time 53, P exits the kernel and finishes its execution at time 57.

Assuming that system call S0 corresponds to begin and end execution of process P, the trace corresponding to the life of process P in this scenario would contain the following three elements.

$<\mathbf{SC}_i>$	$<\mathbf{SC}_j>$	$<\mathbf{SCT}_i>$	$<\mathbf{EET}_{i,j}>$	$<\mathbf{CPU}_{i,j}>$
SC_{S0}	SC_{S1}	0	2	2
SC_{S1}	SC_{S2}	4	12	16
SC_{S2}	SC_{S0}	5	4	9

Note that the quiet time spent in the kernel by a sleeping process must not be included in the CPU times attributed to process P in the trace.

As surprising as it may sound, we did not find an existing tool providing the trace we need. The candidate trace tools that we considered are discussed in subsection A. Then subsection B explains how we built our trace tool for the prototype. Subsection C presents the limitations of the approach used in our prototype.

A. Tracing Tools Surveyed

In order to speed our work, we began by surveying available execution tracing tools that appeared likely to produce the output we needed. Specifically, we examined four different approaches based on existing tools. Each of these approaches is discussed below.

1. *GPROF*. Gprof is a Unix tool that displays a call-graph execution profile for an application (see http://www.gnu.org/manual/gprof-2.9.1/gprof.html for details). Unfortunately, the results provide by Gprof proved too highly aggregated for our purpose. Using Gprof, it is impossible to make distinctions between two occurrences of a system call. In addition, the program must exit before a trace is written; this is a problem since the active network daemons are running as long as they are not killed by an exterior command.

2. *Java Profiling tools*. Since ANTS is based on Java, the use of the -Xrunhprof option available with the latest versions of Java launcher may have been a possible approach (see http://www.java.sun.com/products/jdk/1.2/docs/tooldocs/win32/java.html for more details). But the CPU profile is generated on a per-Java-method basis, and not a on a per-system-call basis. The same problem exists with the JVMPI and JVMDI (JVM Profiler/Debugger Interface).

3. *Adding Instrumentation to glibc*. Since the Java Virtual Machine (JVM) makes system calls to the kernel through glibc, the C library wrapper, a solution may be to augment

this wrapper with a tracing capability. The source code for the GNU version of the C library (glibc) is freely available, and it is possible to recompile the JVM using glibc. Unfortunately, both glibc and the JVM source code are poorly documented and as a result they are difficult to modify. Still, this approach might prove viable.

4. *Strace and Truss*. A Linux command, "strace", can intercept and record the system calls made by traced processes and the signals that these processes receive. The name of each system call, its arguments and other information can be printed. As a tool, strace is interesting because there is no need to recompile a program in order to trace it. The Solaris equivalent of strace, called "truss", does not provide the timing information we need. Unfortunately, strace produces wall clock times, not CPU times. This introduces two problems. First, since Linux is a multi-tasking, multi-user system, some tasks will be running in the background and will influence the benchmark results; the framework of an application will also register some noise. Second, an application, A, sleeping for 5 seconds will have the same model as an application, B, which uses the processor for 5 seconds but without making any system call. Thus, the information provided by strace cannot constitute a good measure of CPU time used by a process. However, a combination of the wall clock information and average load information (see the uptime command) may be better than no information at all. Despite these shortcomings, strace provides an example illustrating how to trace system calls for a process.

B. Generating Traces in Our Prototype

Inspired by what is done in strace, we used the system call ptrace to control the process to be traced. Once a process is attached, we start a loop in which we use the request PTRACE_SYSCALL to have execution of the traced process continued until the beginning of the next system call and also stopped at the end of the system call. (Note that this approach can also be used for Solaris.) When the process is stopped before and after a system call, we use TRACE_PEEKUSR to access the user structure, which gives us a pointer to the values of the registers of the processor, where we can read the number of the future system call.

We also need to trace the CPU time spent by the attached process both in user mode and kernel mode. The comparison with the previous values of those variables allows us to determine the CPU time required by the last system call, as well as the CPU time spent between the last system call and the next. Our first solution was to look at the values of utime and stime in /proc/<pid>/stat to obtain the number of jiffies that the process has been scheduled in user mode and kernel mode (See UNIX man proc to interpret this jargon). Unfortunately, the initial value of 100 jiffies per second was not accurate enough to log short events, and we did not manage to increase this value over 1000 jiffies per second, which was still insufficient. For Linux, the HZ value in /include/asm/param.h establishes the timer interrupt frequency. When we put HZ=10000, the interrupt processing cost was so high that there was no time left for any other computation.

Another solution is to record clock cycles, a very accurate value. On Pentium processors, a special instruction provides access to the number of clock cycles since the last reboot (see http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM for more detail). But recording this information for a process requires some kernel modifications. We first needed to add some fields to the process structure and then two additional system calls to retrieve their values from user space. Then we modified the scheduler in order to have these fields updated each time the process is scheduled. We also modified the fields at entry to and exit from the kernel. This solution is still under development.

Even using this approach, some hurdles must be overcome in order to isolate the system calls of interest. First, to trace an ANTS application on a node, the activity of the node should be traced as soon as the daemon starts because the application may be active and making system calls immediately. But the part of the trace corresponding to the start-up of the node must be deleted because we do not want to charge the application for that overhead. Since the JVM is running as a single Linux process, there is no way to make a distinction (based on process identifier, for instance) between system calls made for the daemon and system calls made on behalf of the application. One solution is for a human to examine several successive traces in order to discover where the daemon startup ends. This may vary according to how many nodes are to be read in the configuration file, and other parameters such as the length of the name of the application to load or the version of the JDK used. In the case of ANTS in our configuration, the first "pause" seems to indicate the end of the initialization. Perhaps we could modify the EE to signal the end of startup. A second solution is to start the node, use the "ps" command to locate the associated process identifier, and then to start a tracing process using the process identifier as a parameter. This second solution is much easier and avoids human examination of a trace; however, the application must not start before the tracing process. The second hurdle results from the fact that the daemon periodically makes system calls, for the purpose of its listening function. These system calls must be excluded from the trace. We expect that these system calls form a sequence easy to isolate and delete from the trace. But we note that the repeating sequence of the ANTS daemon is of course different from the repeating sequence of the PLAN OCaml daemon. For this reason, the trace cleanup task must be automated. Once the generated trace has been cleaned up, it is easy to compute the average time spent in each system call and to generate the Markov matrix with its labeled transitions.

C. Limitations with Our Prototype

In addition to the problems already raised (difficulties in "purifying" the trace for instance), our approach is not perfect. Four limitations must be reported. First, there exists a problematic slow down of processes. A traced process runs slowly because it is stopped before and after each system call, and we have seen an example of a complex program where this affects the behavior of the application. This problem must be taken into consideration when generating traces for the actual system. We might be able to measure the measurement overhead and then factor it from the trace. Or, since we modified the kernel, we could modify it even more to be able to implement our own tracing mechanism without using ptrace. This should at least improve the speed of the traced process. Or we might be able to use a hardware measurement technique, such as the NIST Multikron chip, to minimize measurement overhead. Second, measurement requires us to place some restrictions on applications. For example, as explained earlier, the current tracing procedure implies that the daemon must be started completely before starting the trace process, and that the daemon have time to do at least one routine sequence before stopping the trace. In addition, only the protocols we want to monitor should be running at the time of the measurement. Indeed, for now, it is impossible to have various applications on a single ANTS node. But capsules of different protocols may execute on a node, and it is impossible to make the distinction between two different protocols in the trace of system calls. These difficulties are largely procedural, and can be handled by asking the EE developer to observe certain conventions and by giving a proper set of guidelines to application developers. Third, the approach we used requires modifications to the operating system kernel. Using a modified kernel may be acceptable during experiments, but it does not seem realistic for deployment, especially since the modified part cannot be released as a module. Fourth, the approach we used is not widely portable because we used TRACE_PEEKUSER, which points to an OS-specific data structure, and we relied on special CPU instructions available only on Pentium chips, and we modified on the kernel portions related to single-processor systems. To overcome these last two limitations, we must investigate other approaches to trace generation.

VIII. Potential Benefits of Success

Enabling Resource Management Systems to Address CPU Time. Current network nodes provide a relatively fixed set of processing functions for particular classes of arriving packets. Since the processing per packet is well known, and since a node understands its own computing capacity, the amount of CPU time needed to process a packet on a node can be estimated fairly straightforwardly. This proposition does not hold for an Active Network node. In Active Networks, new packet classes, as well as the processing associated with each class, can be injected into a node. In such cases, the node does not know the amount of CPU time needed to process packets in the new class. The model we outlined in this paper enables an Active Network node to estimate the CPU time likely to be needed for packets in a new class. This information can be used to inform resource management decisions on a node. For example, a node might reject a new packet class if the processing requirements cannot be supported. Or, having accepted a new packet class, a Node operating system can monitor the CPU time used by each arriving packet of the new class in order to halt the execution of packets that exceed the expected CPU resource requirements.

Opening New Territory in Admission Control and Path Routing Decisions. Given a means to express CPU time requirements among heterogeneous network nodes, some interesting new possibilities arise. Consider for example, admission control policies. Currently, admission control regimes are used in connection-oriented networks, such as Asynchronous Transfer Mode (ATM) networks and Integrated Services Digital Networks (ISDN), and are being considered for use in the connectionless Internet, once quality of service mechanisms, such as the resource reservation protocol (RSVP) and multi-protocol label switching (MPLS), are widely deployed. As currently used, admission control decisions for a connection, or flow, are made based on a comparison of bandwidth, delay, and jitter requirements for the connection against the ability of the network to meet the aggregate demands currently being placed on the network. In reaching an admission decision, the network management system currently considers two main resources: the capacity of network links and the capacity of queues in network switches. These resources suffice because each packet sent through each switch and link on the network receives identical processing. Once programmable networks, such as Active Networks, are deployed, the situation changes because each packet can receive nonhomogeneous processing. Under this new situation, network management systems will need to take into account the CPU time requirements for new classes of packets injected into the network to serve data flowing along a path. Admission control decisions will then have to account for link capacity, memory capacity, and CPU capacity available in the network along the path that packets are expected to flow. This adds a new dimension to admission control research.

Beyond admission control, new questions can be imagined relating to path routing decisions in an Active Network. Current networks focus on allocating a path from source to destination that will meet throughput, delay, and jitter requirements. Given the ability to express CPU time requirements for injected packet classes, one can imagine new path routing research questions. For example, can a path be found through the network that can provide the CPU time required by an application, while also meeting the application's throughput, delay, and jitter requirements? How about combining application performance constraints and CPU time, memory, and bandwidth requirements together with cost? Can an Active Network application express such requirements and then query the Active Network for multiple paths, each with an associated cost? The resource management questions in nodes become very interesting under these circumstances because taking on new processing loads can have a negative effect on the ability of nodes to meet the requirements promised for other applications. It might be the case that nodes will need to condition such promises on a certain level of CPU utilization dedicated to an application. Then, resource management algorithms must police CPU utilization for each application, in addition to total CPU time used. The main point of this discussion is to illustrate that dynamically programmable networks will provide a range of interesting research questions that cannot be addressed unless CPU time can be interpreted among heterogeneous nodes.

IX. Future Work

The ideas outlined in this paper provide a plausible approach to express estimated CPU time requirements for Active Network applications in a form that can be interpreted among heterogeneous nodes in an Active Network. We developed this approach over the first six months of a planned 30-month project. In order to fully evaluate the approach, eight main tasks must be completed. Our intent is to complete the first four tasks over the next 12 months, while completing the remaining four tasks during the final 12 months of the project. Failure on task five will indicate a need to revisit the proposed approach; thus, the tasks 6-8 (to be performed in the final year of the project) might need to be redefined. Below, we outline each task, as currently foreseen.

1. ***Develop and Evaluate an Active Network Application (ANA) Model.*** In this paper we proposed the simplest ANA model that can reasonably account for the sources of variability among active network nodes. We also proposed interpreting the model based on assumptions of exponential dwell times in model states. These assumptions must be tested against measured observations from several Active Network applications. Previous work by others suggests that CPU time usage in programs tends toward geometric or hyper-exponential distributions [35]. After determining the observed dwell times in our model states, we might need to revise the model to use a different statistical distribution, and to include additional parameters. These decisions can only be taken based on sufficient measurements and observations of Active Network applications. Should the statistical properties of the observed behavior render our current or revised model intractable, we might have to develop a different model based on code analysis, coupled with run-time measurement. Others have proposed more complex programs models based on code analysis and profiling [33, 34, 43, 44, 46]. If necessary, we could build on this work.
2. ***Design and Develop a Self-Calibrating Active Node (SCAN).*** This task involves three subtasks: (1) designing and implementing calibration workloads for EEs, (2) designing and implementing calibration workloads for Node OS system calls, and (3) implementing a self-calibration mechanism. Each subtask is addressed in turn.
    a. *Designing and Implementing Calibration Workloads for EEs*. One could examine the existing experimental Active Network applications, identify major classes among those applications, and then select, or implement, a representative of each class for inclusion in a calibration workload. In this paper, we suggest that insufficient experience exists with Active Network applications to support node calibration based on a representative workload. To overcome this current situation we propose to design and develop a synthetic benchmark workload to execute all the functions provided by two specific EEs. Currently, we intend to use ANTS and PLAN; however, we are receptive to other choices. In order to establish the execution probability for each function we intend to measure the existing experimental Active Network applications for each EE we select, and then to parameterize the workload so that the most

likely used functions have highest probability. We will also ensure that the workload for each EE executes all functions to at least a minimum degree. Each workload will be associated with a version number so that benchmarks can evolve over time.

   b. *Designing and Implementing a Calibration Workload for Node OS System Calls*. We must develop a workload for calibrating Node OS system calls. Here we intend to produce a synthetic workload that simply cycles through an execution pattern covering all system calls. Each channel configured with different protocol modules will be treated as a distinct system call.
   c. *Implementing a Self-calibration Mechanism*. The remaining subtask requires us to select and implement a mechanism for executing the synthetic workloads, and thus to calibrate a node. In this paper we identified several options: (a) off-line calibration, (b) boot-time calibration, (c) run-time calibration, and (d) off-line calibration with run-time adjustments. We intend to implement off-line calibration with run-time adjustments. While we think that variable execution cycles for run-time calibration adjustments might be possible and beneficial, for the present we plan to have run-time adjustments execute with a fixed periodicity.

3. ***Design and Implement an Automated ANA Model Generator***. To assist Active Network application (ANA) designers to construct an ANA model that corresponds to their application, we propose to design and implement an automated ANA model generator. Earlier in this paper, we discussed the outlines of such a generator. We intend the generator to consume execution traces, as outlined in Section VI. B., and to produce an ANA model in the form of a Markov transition matrix. In Section VII we discussed a proof-of-concept prototype that we developed to generate an ANA model from an execution trace of an Active Networks ping application running under Linux on the Ocaml PLAN EE and the ANTS EE. The purpose of the proof-of-concept was four-fold: (1) to establish that automated model generation is feasible, (2) to understand the trace-generation capabilities that must be provided by a Node OS, (3) to assess the degree that measurement artifact will interfere with accurate model generation, and (4) to provide measurement data that can be fitted against statistical distributions. The results from the prototype appear promising; however, we must still find a means to create traces that will be portable across various operating systems and CPU chips, and that can be obtained with little, or at least well known, measurement overhead. After overcoming these issues, which do not appear insurmountable, we need to design and implement a real model generator, we need to establish trace-generation capabilities that can be supported by a typical Node OS, and we need to develop guidelines for an application developer to follow when generating traces. We also intend to explore real-time trace consumption, as an alternative to consumption of off-line traces.
4. ***Specify, Design, and Implement Additional Node OS System Calls***. A number of the components we identified in this paper require support through Node OS system calls that do not currently exist. For example, system calls will be needed to add reference and local calibration data to a node, to manage ANA model information, to transform ANA models from local-to-reference and from reference-to-local forms, to start and stop run-time calibration adjustment, and to estimate CPU usage requirements from ANA models. We need to specify the

application programming interfaces to these added Node OS calls, to design the logic necessary to implement the calls, and then to implement the calls for the Node operating systems that we intend to use in the next task. Our current thinking is that we might well go beyond the strict requirements of this task by implementing a portable Node OS interface layer that provides the currently specified Node OS system calls, as well as the added calls that we will introduce. This approach has some advantages. First, a portable layer could be modified to support the trace-generation capabilities we expect a Node OS to provide. Second, a portable layer would facilitate our need to implement our ideas in several different Node operating systems. Finally, we would be able to provide detailed comments and constructive suggestions for improving the Node OS interface specification proposed as part of the DARPA Active Networks program.

5. ***Prototype and Evaluate Components as a System***. Once the myriad components have been designed, implemented, and evaluated, the entire ensemble must be integrated and evaluated as a system. We expect to perform five tests. Each test is outlined below. We are also open to other tests. Passing these tests should provide ample confidence for moving ahead to the next set of three tasks. Failure on these tests suggests either that adjustments to the approach are needed, or that the approach must be re-examined.
    a. *Evaluate SCAN and ANA Modeling.* As a first test, we intend to ensure that our calibration method leads to ANA models that produce the expected results on Active Network nodes. We will implement SCAN for a single EE and the Node OS system calls on three Active Network nodes with very different processing capabilities. We will generate an ANA model for a single Active Network application. We will run the application on each of the nodes and compare the measured results with the results predicted by the ANA model. We will characterize the performance of our approach. Successful results on this test will demonstrate that our approach applies across Active Network nodes that possess disparate CPU processing capabilities. Success on this test is critical to the viability of our approach.
    b. *Evaluate Multiple EE Calibration.* We will rerun the test in subtask (a), but we will use a different EE. We will compare the results from subtask (a) with the results obtained from subtask (b). Successful results on this test will demonstrate that our approach can be applied across different EEs and node processing capabilities.
    c. *Evaluate Multiple ANA Calibration.* We will rerun the tests in subtasks (a) and (b) but with a different Active Network application. We will compare the results from subtasks (a) and (b) with the results from subtask (c). Successful results on this test will demonstrate that our approach works across different Active Network applications, in addition to different EEs and node processing capabilities.
    d. *Evaluate Multiple Node OS Calibration.* We will rerun the tests in subtask (c), but we will use three different Active Network applications (adding a third application to the two considered on previous tests). Rather than using nodes with different processing capabilities, we will use Active Network nodes with different Node operating systems, but with identical or similar processing capabilities. Successful results on this test will demonstrate that our approach works for multiple applications, multiple EEs, and multiple Node operating systems.

e. *Evaluate System Generally*. We will repeat the tests in subtask (d), except we will use active nodes with three different processing capabilities and with three different Node operating systems. Successful results on this test will demonstrate that our approach works for three different node operating systems, nodes with three different classes of processing capabilities, three Active Network applications, and two different EEs.

6. ***Update the Prototype.*** After successfully completing the rigorous battery of tests outlined in task 5, the prototype should be updated to improve its usability based on experiences during the testing. The main intent of this task is to produce self-contained code that can be distributed widely in the research community, and that can be quickly and easily incorporated into an Active Network node.
7. ***Integrate the Prototype with an Active Network Resource Manager.*** The intent of this task is to integrate SCAN, the ANA model generator, and the added node OS system calls into an Active Network Node OS that implements a resource manager. Our current plans call for using the AMP Node OS being developed by Trusted Information Systems. We are open to select other node operating systems.
8. ***Demonstrate the Ability to Enforce CPU Resource Usage Policy.*** In the final task, we will demonstrate that a node resource manager can use our model to estimate the likely CPU usage of an ANA injected into a node. Further, we will demonstrate that the node resource manager can terminate executing Active Network applications that exceed their stated CPU resource needs.

X. Conclusions

Among its main contributions, this paper identifies the major sources of variability in CPU time usage in an Active Network node, and then proposes a model for CPU time usage by Active Network applications (ANA). The proposed ANA model is among the simplest possible models that can account for all the major sources of variability identified in the paper. The model uses a Markov state-transition matrix to represent the execution probabilities and CPU times used by an ANA as it transitions between Node OS system calls and other processing within an Execution Entity. The model also includes a means to calibrate Active Network nodes with respect to a reference node, as well as algorithms to transform a local ANA model to a reference model and a reference ANA model to a local model. Further, assuming that the ANA CPU time usage patterns exhibit an exponential distribution when observed during execution traces, the paper shows how the ANA model can be used by an Active Network Node OS to compute quickly and easily the expected CPU time usage for an arbitrary proportion of executions of the modeled ANA. The exponential assumption has yet to be tested. Should measured times exhibit other statistical properties, then the associated computations must be reformulated, and additional parameters might need to be included in the ANA model. The paper also proposes several approaches to build workloads for calibrating EEs and for calibrating Node OS system calls. Different mechanisms are identified for performing node calibrations. Further, the paper discusses an approach to generate automatically an ANA model from an execution trace of an ANA. A modest proof-of-concept prototype was implemented and discussed in the paper. The prototype suggests that automatic model generation is feasible, once some trace-generation issues are resolved.

The paper also makes some subsidiary contributions. First, the paper identifies some benefits that can accrue when heterogeneous Active Network nodes are able to interpret estimated CPU time requirements for an ANA. The obvious benefit is that node resource managers can begin to monitor the CPU time usage of an ANA for conformance with the promised profile.

Perhaps more importantly, new research will be enabled regarding admission control and resource allocation among programmable network nodes. Second, the paper provides a reasonable survey of the current state-of-the-art in benchmarking for computer systems. Third, the paper provides a brief discussion of some existing execution tracing tools available in the public domain. In the section on future work, the paper lays out a clear research agenda for implementing, testing, and evaluating the ideas discussed throughout the paper.

XI. Acknowledgments

The work discussed in this paper was conducted under joint funding from the Defense Advanced Research Projects Agency (DARPA) and the National Institute for Standards and Technology (NIST). The authors thank Hilary Orman for her insightful comments during discussions that led to the beginning of this project. In addition, the authors thank Doug Maughan for his continued support of the work.

XII. References

[1] K. Calvert (ed.), Architectural Framework for Active Networks, Version 0.9, Active Networks Working Group, August 31, 1998.

[2] T. Lindholm and F. Yelling, The Java Virtual Machine Specification, Addison-Wesley, Reading, Mass., 1997.

[3] D. Wetherall, J. Guttag, and D. Tennenhouse, "ANTS: Network Services Without the Red Tape", *IEEE Computer*, April 1999, pp. 42-48.

[4] D. S. Alexander, W. A. Arbaugh, M. W. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith, "The SwitchWare Active Network Architecture", *IEEE Network Special Issue on Active and Controllable Networks*, vol. 12 no. 3, pp. 29 - 36.

[5] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, D. Rockwell and C. Partridge, "Smart Packets for Active Networks", *Proceedings of OpenArch 99*, March 1999.

[6] Samrat Bhattacharjee, Kenneth L. Calvert and Ellen W. Zegura. "An Architecture for Active Networking", *Proceedings High Performance Networking (HPN'97)*, White Plains, NY, April 1997.

[7] D. Mosberger and L. L. Perterson, "Making Paths Explicit in the Scout OS", *Proceedings of the Second Symposium on Operating System Design and Implementation*, ACM Press, New York, 1997, pp. 153-168.

[8] F. Kaashoek at al., "Application Performance and Flexibility on Exokernel Systems", *16th Symposium on Operating System Principles*, ACM Press, New York, 1997, pp. 52-65.

[9] D. Decasper, G. Parulkar, S. Choi, J. DeHart, T. Wolf, and B. Plattner, "A Scalable, High Performance Active Network Node", *IEEE Network*, January/February 1999.

[10] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, O. Shivers, "The Flux OSKit: A Substrate for OS and Language Research", *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, ACM Press, October 1997.

[11] L. Peterson (ed.), NodeOS Interface Specification, Active Networks Node OS Working Group, February 2, 1999.

[12] R. P. Weicker, "A detailed look at some popular benchmarks", *Parallel Computing*, No. 17, 1991, pp. 1153-1172.

[13] R. P. Weicker, "Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules", *SIGPLAN Notices*, 23, 8, August 1988, pp. 49-62.

[14] H. J. Curnow and B. A. Wichmann, "A Synthetic Benchmark", *The Computer Journal*, 19, 1, 1976, pp. 43-49.

[15] TPC Benchmark A Standard Specification, Revision 2.0, Transaction Processing Performance Council, June 7, 1994.
[16] TPC Benchmark B Standard Specification, Revision 2.0, Transaction Processing Performance Council, June 7, 1994.
[17] TPC Benchmark C Standard Specification, Revision 3.4, Transaction Processing Performance Council, August 25, 1998.
[18] TPC Benchmark D Standard Specification, (Decision Support) Revision 2.1, Transaction Processing Performance Council.
[19] K. Shanely, History and Overview of the TPC, Transaction Processing Performance Council, February 1998.
[20] TPC Benchmark H Standard Specification, (Decision Support), Revision 1.1.0, Transaction Processing Performance Council.
[21] TPC Benchmark R Standard Specification, (Decision Support), Revision 1.0.1, Transaction Processing Performance Council.
[22] TPC Benchmark W (Web Commerce), Public Review Draft Specification, Revision D 5.0, July 12, 1999.
[23] WinTune98, Windows Magazine, Version 1.0.39, June 24, 1999, 1.75 Mbyte download.
[24] Export Administration Regulations: Technical Note to Category 4, Computers: Supplement No.1 to Part 774.
[25] SPEC CPU95 Benchmarks, Standard Performance Evaluation Corporation, June 24, 1999.
[26] "BAPCo Debuts First Benchmarking Software for Computers Running Windows*98" press release from Business Applications Performance Corporation, Santa Clara, CA, August 26, 1998.
[27] "PC Magazine Labs Benchmarks Tests: Winstone99", *PC Magazine On-line*, ZDNet, December 1, 1998.
[28] "NetBench 6.0", ZDNet, 1999.
[29] "WebBench 3.0" ZDNet, 1999.
[30] "SPEC Announces SPECweb96, Industry's First Standardized Benchmark for Measuring Web Server Performance", press release from Standard Performance Evaluation Corporation, July 22, 1996.
[31] "WinBench 99", ZDNet, 1999.
[32] "PC Magazine Labs Benchmark Tests: CPUmark99", *PC Magazine On-line*, ZDNet, December 1, 1998.
[33] R. H. Saavedra-Barrera, A. J. Smith, and E. Miya, "Machine Characterization Based on an Abstract High-Level Language Machine", *IEEE Transactions on Computers*, Vol. 38, No. 12, December 1989, pp. 1659-1679.
[34] R. H. Saavedra and A. J. Smith, "Analysis of Benchmark Characteristics and Benchmark Performance Prediction", *ACM Transactions on Computer Systems*, Vol. 14, No. 4, November 1996, pp. 344-384.
[35] Boris Beizer, Micro-Analysis of Computer System Performance, Van Nostrand Reinhold Company, 1978.
[36] William S. Bowie, "Applications of Graph Theory in Computer Systems", *International Journal of Computer and Information Sciences*, Vol. 5, No. 1, 1976, pp. 9- 31.
[37] R. M. Karp, "A note on the application of graph theory to digital computer programming", *Information and Control*, Vol. 3, No. 6, 1960, pp. 179-190.
[38] C. V. Ramamoorthy, "Discrete Markov analysis of computer programs", in *Proceedings of the 20th ACM National Conference*, 1965, pp. 386-392.
[39] J. D. Foley, "A Markovian model of the University of Michigan execution system", *Communications of the ACM*, Vol. 10, No. 9, 1967, pp. 584-589.
[40] W. S. Bowie, "Towards a distributed architecture for OS/360", PhD Thesis, Department of Applied Analysis and Computer Science, University of Wateroo, 1974.

[41] M. A. Franklin and R.K. Gupta, "Computation of Page Fault Probability from Program Transition Diagram", *Communications of the ACM*, Vol. 17, No. 4, 1974, pp. 186-191.
[42] P. Hoschka, "Compact and Efficient Presentation of Conversion Code", *IEEE Transactions on Networking*, Vol. 6, No. 4, 1998, pp. 389-396.
[43] B. Wiegbreit, "Mechanical Program Analysis", *Communications of the ACM*, Vol. 18, No. 9, 1975, pp. 528-539.
[44] T. Hickey and J. Cohen, "Automating Program Analysis", *Journal of the ACM*, Vol. 35, No. 1, 1988, pp. 185-220.
[45] G. Ramalingam, "Data Flow Frequency Analysis", *ACM SIGPLAN NOTICES*, Vol. 31, No. 5, 1996, pp. 267-277.
[46] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, "A Static Performance Estimator to Guide Data Partitioning Decisions", *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 1991, pp. 267-277.
[47] M. Kijima, Markov Processes for Stochastic Modeling, Chapman and Hall, London, 1997, especially pages 168-229.

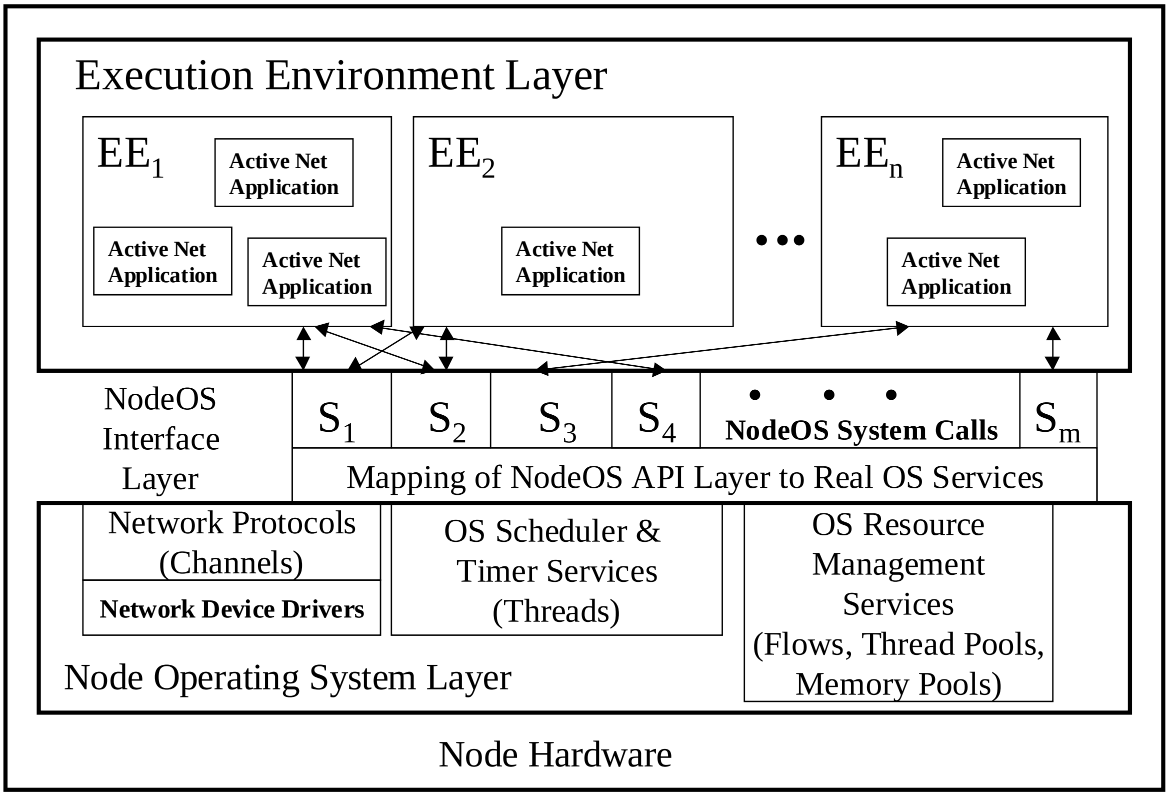

Figure 1. Conceptual Architecture of an Active Network Node

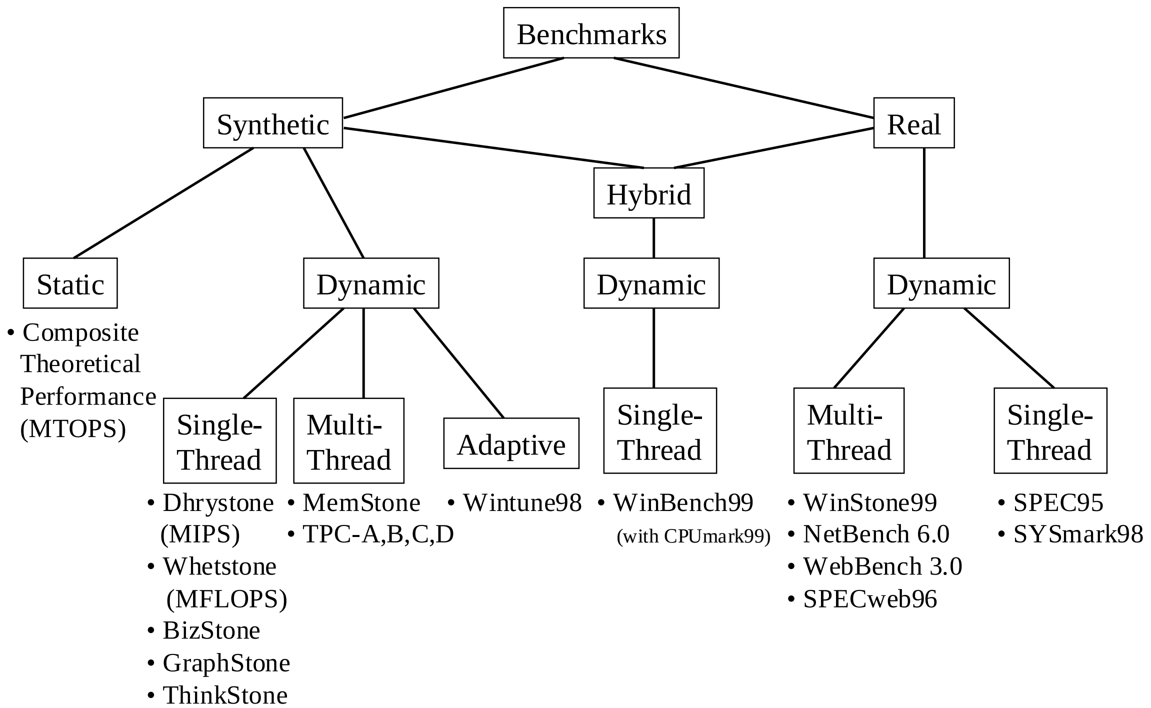

Figure 2. A Taxonomy of Selected Computer System Benchmarks

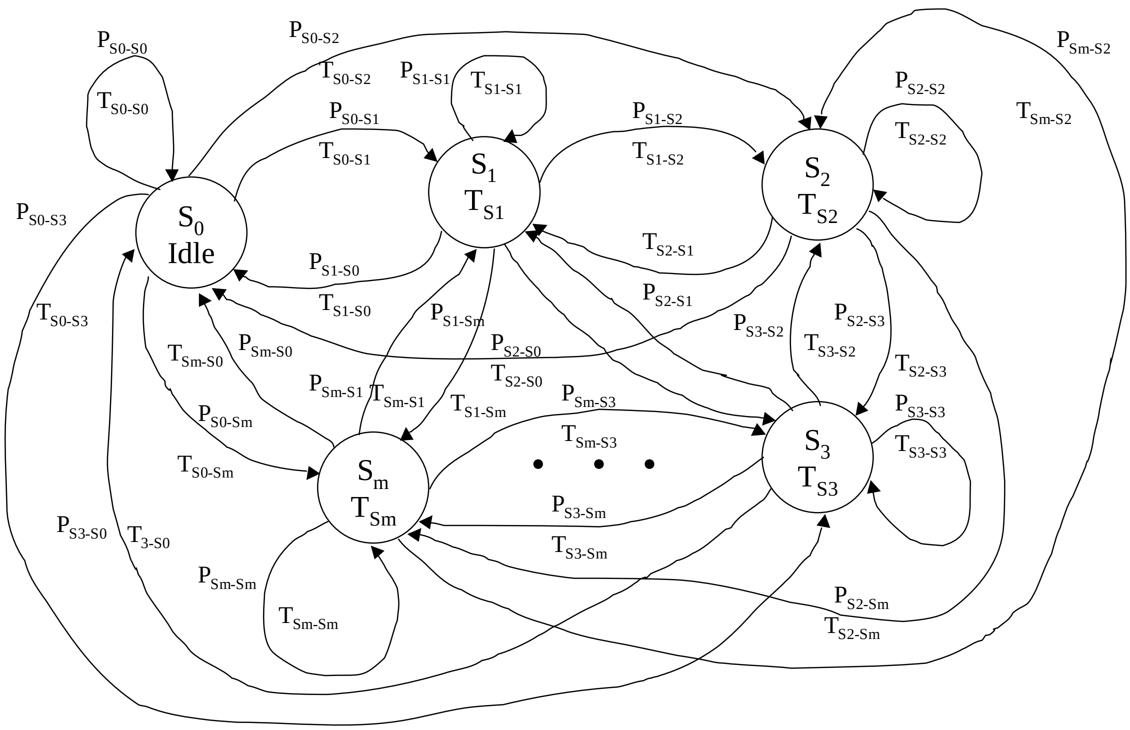

Figure 3. Continuous-Time, Finite State Markov Model of an Active Network Application

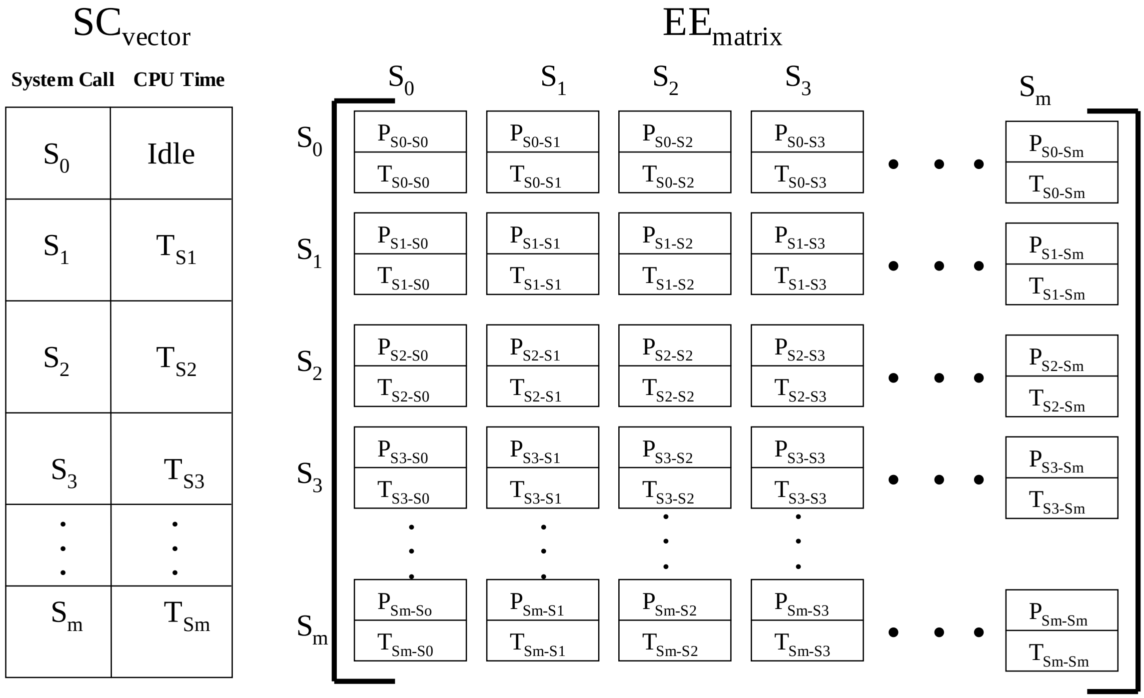

Figure 4. A Vector and Matrix Representation of the Active Network Application Model

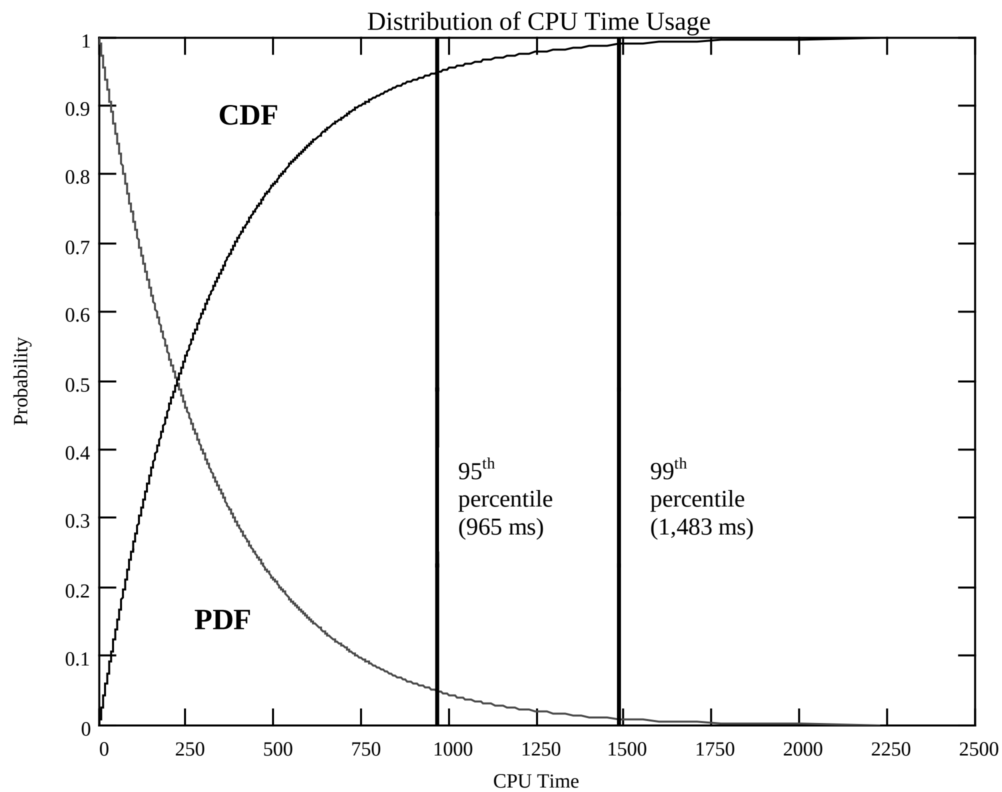

Figure 5. Estimated Distribution of Average Execution Times Corresponding to Table 3.

Node to Reference Transformation

n := the index for the specific execution environment used for the application
m := the number of system calls supported by a NodeOS

for i from 0 to m
    $SC_{vector}[i] := S_{reference}[i]/S_{node}[i] * SC_{vector}[i]$
    for j from 0 to m
        $EE_{matrix}[i,j].T := EE_{reference}[n]/EE_{node}[n] * EE_{matrix}[i,j].T$
    end for
end for

Figure 6. Algorithm to Transform an Active Network Application Model from a Node Basis to a Reference Basis

Reference to Node Transformation

n := the index for the specific execution environment used for the application
m := the number of system calls supported by a NodeOS

for i from 0 to m
    $SC_{vector}[i] := S_{node}[i]/S_{reference}[i] * SC_{vector}[i]$
    for j from 0 to m
        $EE_{matrix}[i,j].T := EE_{node}[n]/EE_{reference}[n] * EE_{matrix}[i,j].T$
    end for
end for

Figure 7. Algorithm to Transform an Active Network Application Model from a Reference Basis to a Node Basis

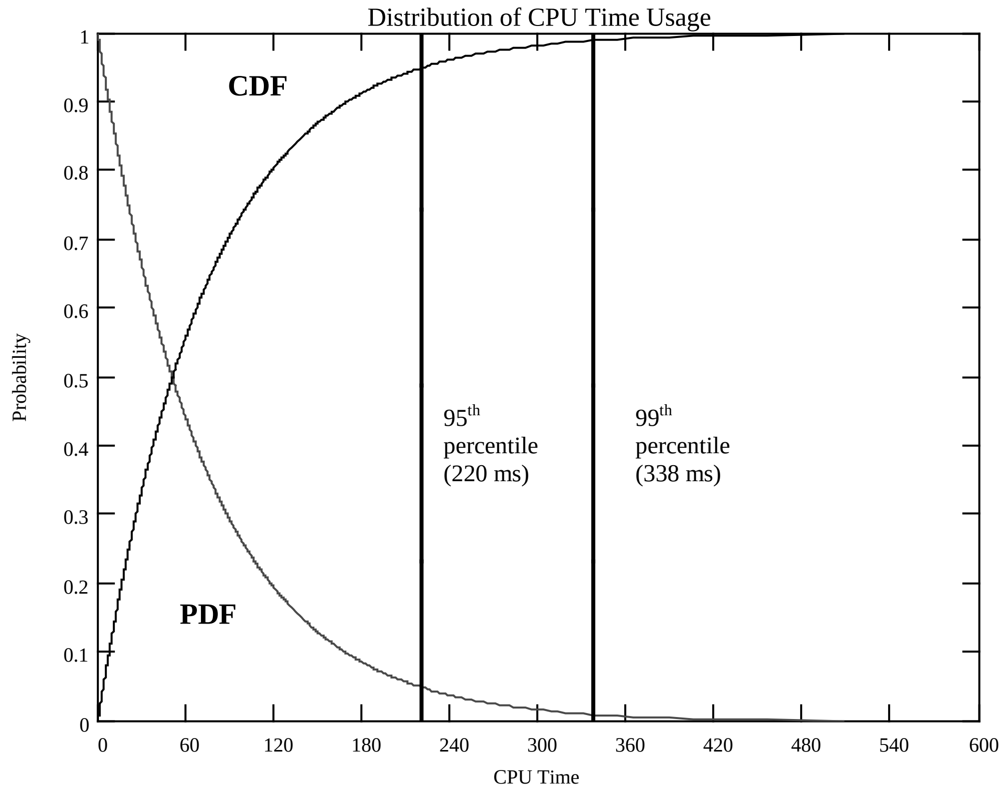

Figure 8. Estimated Distribution of Average Execution Times Corresponding to Table 5.