Modeling the Performance of E-Commerce Sites

Jonathan C. Hardwick[1], Efstathios Papaefstathiou[1], and David Guimbellot[2]
[1] Microsoft Research Limited, Cambridge, UK
[2] Microsoft Corporation, Redmond WA, USA

Indy is a new performance modeling framework for the creation of tools for many different classes of performance problems, including capacity planning, bottleneck analysis, etc. Users can plug in their own workload and hardware models while exploiting core shared services such as resource tracking and evaluation engines. We used Indy to create EMOD, a performance analysis tool for database-backed web sites. We validate EMOD using the predicted and observed performance of SVT, a sample e-commerce site.

INTRODUCTION

As the software industry moves to supplying services over the internet, the problem of predicting and modeling the performance of these services becomes even more acute. Instead of running on a single computer, services rely on a distributed collection of servers. These can range from a simple two-tier e-commerce site to a geographically distributed collection of portal services. Performance, reliability, management cost, and scalability are all critical to the success of these services [SPE00]. However, their distributed nature makes it difficult to predict or understand the ramifications of changes to the system. Monitoring can reveal the impact of a change, but only after the fact. Furthermore, because of the 24/7 nature of these services, it is important for operations managers and developers to be able to anticipate the performance implications of internal changes (e.g. system topology, software modification) and external factors (e.g. load spikes).

What is needed is a range of performance modeling tools that allow software developers, planning staff, and operations managers to ask a wide variety of what-if questions before they apply changes to the service itself. Currently, there are a limited number of modeling tools that can be used for this purpose. This lack of general purpose tools can be attributed to the complex nature of the modeling process. A modeling tool has to choose:

- A basic modeling technique, e.g. simulation, statistical, or analytical.
- A role for the tool, e.g. capacity planning or performance debugging.
- A level of abstraction, which can range from treating servers as black boxes to modeling individual lines of code.
- A target audience, which will affect output methods, e.g. system administrators or performance analysts.

Existing tools tend to provide a single solution in this multi-dimensional problem space. That is, they choose one combination of the above range of parameters to produce a specialized solution. As a result, there is little sharing of expertise or code between tools, and the tools themselves are not widely adopted.

The rest of this paper is organized as follows. First, we propose the idea of modeling infrastructures as a general solution to the problems outlined above, and describe Indy, a particular infrastructure that we have developed. Next we describe how we used Indy to create EMOD, a tool for modeling the performance of e-commerce sites. Then we show how EMOD can be used to model a particular site, and validate its predictions. Finally, we discuss further extensions to the Indy infrastructure and possibilities for future work.

MODELING INFRASTRUCTURES

As a solution to this problem, we propose the use of a *modeling infrastructure* [PAP00]. This is not a single tool, because as we have seen above a single tool cannot handle all of the possible modeling requirements. Rather, it is a general-purpose toolkit that can be used to create any number of specialized tools for individual modeling purposes. Furthermore, it is not limited to a fixed set of components – users can contribute new components to extend the toolkit's capabilities and the range of tools that can be produced using it.

We now differentiate between *tool developers* and *end users*. A tool developer interacts directly with the modeling infrastructure, choosing from the sets of

parameters discussed previously (modeling technique, tool role, abstraction level, output method, etc) to produce a tool for a particular role. The developer may choose to plug together preexisting components supplied in a library as part of the infrastructure, or to develop new components to further extend the capabilities of the infrastructure. An end user then uses the resulting tool to solve a particular performance problem. They can modify problem parameters, workloads, and configurations within the limits set by the developer of the tool, but cannot further extend it.

Given this outline, we can now define the requirements of such an infrastructure. It should support:

1. A component-based plug-and-play structure, where each component has a particular role.
2. Well-defined interfaces between the individual components.
3. A core engine or kernel that controls the flow of information between components
4. A development environment to simplify the creation of new tools using the infrastructure.

Our Infrastructure: Indy

As a proof of concept of a modeling infrastructure, we have developed a particular implementation, called Indy. It meets all of the requirements listed above. Individual components are Win32 DLLs that can be dynamically loaded into a running process. The components communicate either via XML according to standardized schemas, or via well-defined APIs. The kernel is a linkable library containing the algorithms and data structures necessary for the evaluation of performance models, and for the coordination of the whole system. Any required user interface can be put on top of this kernel. One example is Indyview, a front-end to the Indy system which can be used both as a development environment for the creation of new tools, and a production environment to run them in.

A diagram of information flow in the Indy architecture is shown in Figure 1. As shown, a tool created with the Indy system adds the following user-contributed components to the kernel:

- Configuration parameters
- System topology
- Workload model
- Hardware models

Configuration parameters are simple name/value variables that are defined by the creator of a performance study, and can then be adjusted by a user to modify the behavior of a model. For example, the hardware model for a CPU might take a "CPU speed" configuration parameter, and adjust its behavior accordingly. Configuration parameters are read from an XML file and stored in a kernel *metadirectory* that holds user-supplied information.

The *system topology* lists the devices in the system being modeled, their interconnections, and the hardware models that correspond to these devices. Again, the system topology is stored in an XML file. The user can easily browse, modify, and save any of this XML information using tree-based editors from within the Indyview interface, as shown in Figure 2. Other approaches to generating this information can easily be imagined: for example, a drag-and-drop interface for constructing and editing topologies.

A *hardware model* describes the behavior of a device listed in the topology. Note that "device" is used here in a fairly loose sense, in that it may be a virtual entity (such as a thread) instead of a physical one (such as a CPU or disk). A device represents an active entity that provides a certain capability – for example, a disk can process disk operations, and a network interface can process incoming and outgoing messages. The Indy kernel calls hardware models to determine how much time is required to perform these actions on each device. The complexity and level of detail of a hardware model is entirely up to the user: a simple model of a network might be implemented by a one-line function that computes "time = latency + message size /

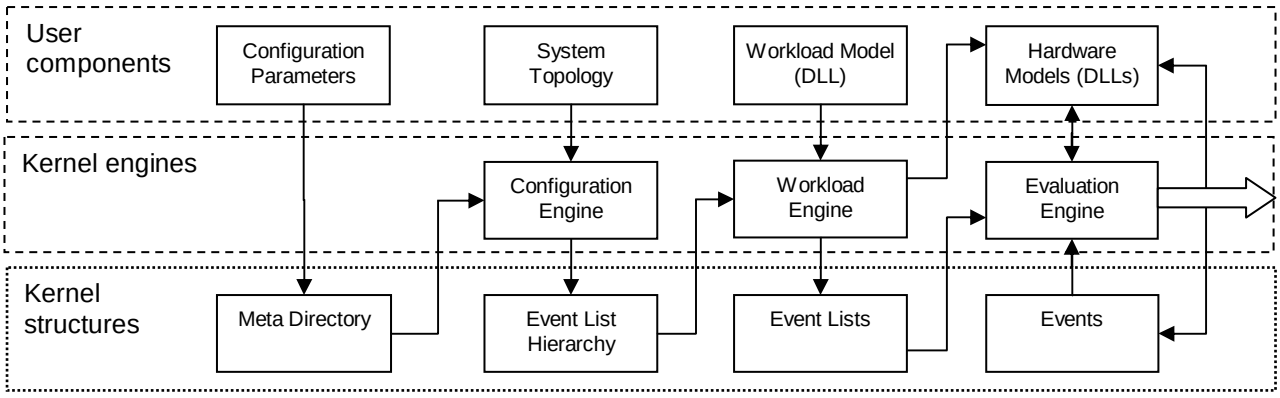


Figure 1: Information flow in the Indy architecture

bandwidth", while a more complicated model might take contention into account, or model the individual packets traversing the network. The user can then choose which model to use based on their requirements of evaluation time, accuracy, and level of abstraction. For example, the user might choose to use a slow but detailed model for the component under study, and simple fast models for the rest of the system. Hardware models are implemented as Win32 DLLs that implement a standard set of interfaces called by the Indy kernel.

The *workload model* is the most complicated part of a performance study. It defines the flow of events through the system being modeled. The workload model therefore combines the tasks of generating the simulated input, and describing the causal relationships between events that take place on system devices as a result of that input. Again, workload models are implemented as Win32 DLLs, and interact with the Indy kernel through standard APIs. It is up to the writer of a workload model whether to generate a synthetic input workload according to some statistical profile, or to simply replay a captured workload. The output of the workload model is a series of *timelines* containing *workload requests* for each of the actions that should take place in the system, and specifying what resource they require. A timeline may contain forks and joins, representing actions that can take place in parallel.

The Indy Kernel

At the next level down, the Indy kernel itself includes three engines that hide the complexity of the internal modeling algorithms by providing well-defined abstractions.

The system configuration engine processes the system topology script and creates an *event list* for each of the active devices modeled in the system. An event is the internal representation of an action being performed, such as computation, communication, or disk I/O. The event lists are then populated by the workload engine, which determines which events are run on which hardware devices, taking into account the incoming timelines from the workload generator, the system topology, and the underlying hardware models.

Finally, the evaluation engine coordinates the evaluation of each of the events using their assigned hardware models, and combines individual event timings to determine the overall performance of the system. The current evaluation engine uses event-based simulation together with novel scheduling and resource tracking algorithms (described later) that can efficiently model the contention happening on a real system.

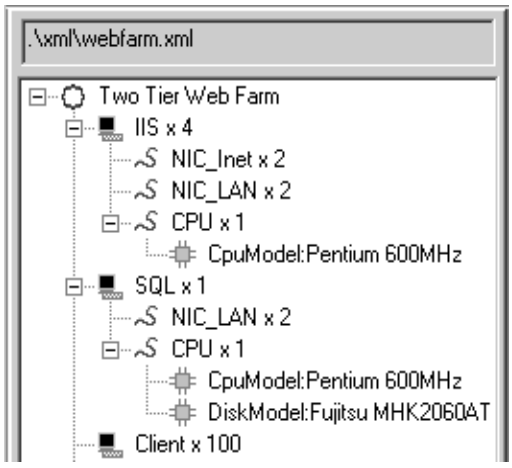
The result of the evaluation is an output trace showing the interactions between the components of the system as it processes the workload. This can then be visualized directly using a tool such as Indyview, or it can be post-processed for use by other tools.

BUILDING THE EMOD MODEL

We'll now describe the creation of EMOD, an Indy workload model for use in the Indyview system that models e-commerce sites. It is designed for capacity planning purposes, and therefore uses a simple model with a high level of abstraction in order to obtain fast turnaround from simulations. This simplicity also makes it a good example.

EMOD is designed to model multi-tier web sites which consist of front-end web servers, possibly a business logic layer, and finally a database layer. Such systems service queries from an internet or intranet.

EMOD must therefore be flexible enough to handle




```
<active_device type="computer" name="IIS" count="4">
  <active_device type="thread" name="NIC_Inet" count="2"/>
  <active_device type="thread" name="NIC_LAN" count="2"/>
  <active_device type="thread" name="CPU" count="1">
        <use_template name="CpuModel:Pentium 600MHz"/>
  </active_device>
</active_device>
<active_device type="computer" name="SQL" count="1">
  <active_device type="thread" name="NIC_LAN" count="2"/>
  <active_device type="thread" name="CPU" count="1">
    <use_template name="CpuModel:Pentium 600MHz"/>
    <use_template name="DiskModel:Fujitsu MHK2060AT"/>
  </active_device>
</active_device>
<active_device type="computer" name="Client" count="100"/>
```

Figure 2: Two views of a system topology, in Indyview and as the corresponding XML file

most conceivable types of e-commerce sites, which can have:

- Different numbers and types of servers
- Different types of backbone network
- Different basic actions
- Different transaction mixes and client loads

The first choice to make is the level of abstraction to use. This affects the devices that must be modeled and the actions that can be performed on them. For EMOD we use a high level of abstraction, modeling servers with just three types of active device: CPUs, disks, and network interfaces. Each of these devices can have a basic named action performed on it, as follows:

Device	Action	Measurement unit
CPU	Compute	CPU cycles (Mcycles)
Disk	Diskop	Disk operations (seeks)
NIC	Comm	Message size (Kbytes)

To use EMOD to model a particular e-commerce site, a user must therefore break down the transactions of that site into individual CPU, disk, and network actions, and compute their costs. This is easy to do with existing performance monitoring tools, as described in the validation section. The relative ease of capturing the data required for a particular model is an important consideration in choosing the level of abstraction.

Note that memory is not being modeled here – for the purposes of the EMOD example we assume that servers will always have adequate memory to handle the load. It would be easiest to model memory as a resource that has a limit on its use and is temporarily consumed by an event, just as a network model has an upper bandwidth limit and costs for individual messages. Measuring memory requirements of individual applications is trickier, however, and hence we have left it out this simple model.

Modeling Threads of Control

The next decision in creating the model is how to represent these devices, and the threads of control that they use to process actions, in the topology. There are two basic alternatives. The first is to represent reality as closely as possible. For example, each CPU on a web server might be assigned twenty threads of control, representing the listening threads of the web server process. All these threads contend for a single resource (the CPU), and the kernel must therefore schedule them appropriately. An Indy representation to achieve this is described in the Extensions section.

A simpler alternative is to model a single thread of control that services the resource under contention. For example, we can model a single thread of control per web server that has the entire CPU to itself. All client requests are then directed to this single thread, where they are queued and executed in order by the Indy kernel scheduler. Effectively we are modeling the flow of control on the CPU itself, rather than in the multiple threads contending for it. The overall effect in terms of throughput is the same, in that each request waits its turn for the CPU to become available, uses the appropriate amount of CPU time, and then leaves the system. However, it may not accurately reflect the latency of individual requests.

We will use this single-thread queuing technique for all of the resources under contention in the EMOD model: CPUs, network interfaces, and disks on web servers, business logic servers, and database servers. For network interfaces we will use two threads, one for sending messages and one for receiving them, representing the full-duplex nature of the interface. This explains the two threads shown for each NIC in Figure 2.

Creating a Workload

Having chosen a level of abstraction for our model in terms of basic actions and the threading model that it should support, we must now define the possible workloads of the EMOD model. That is, we must choose how a user can express the transactions supported by a particular e-commerce site, such as viewing the home page, or adding an item to a shopping basket, in terms of these basic actions. The simplest approach would be to hardcode a set of supported transactions into the model, where each transaction would be composed of a series of actions, and the user would only be able to specify the relative frequency of transactions and the costs of their individual actions. While this is suitable for a special-purpose model aimed at a known application with a known workload, it is not viable for EMOD, which is intended to be more general-purpose.

Instead, we have chosen to let the user describe the transactions of a particular e-commerce site using a *transaction script*. This is an XML file that defines a number of transactions, each of which is composed of one or more actions, which represent the behavior of the system during the transaction. The type and composition of transactions is completely up to the end-user – they can supply a different transaction script for each e-commerce site being modeled, with a different range of transactions. Only the possible actions (CPU computation, disk operations, and network communication) are fixed by the model.

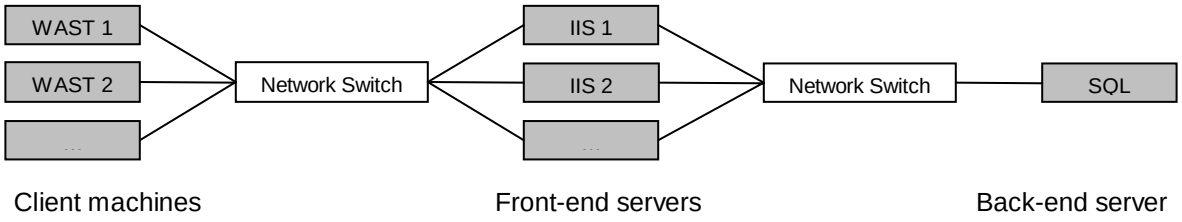


Figure 3: Experimental setup used to benchmark SVT site for validation of EMOD model.

The workload generator for EMOD thus has two phases. In the first phase, it reads a transaction script (whose name is found from a configuration parameter), and parses the information into internal data structures representing each of the transactions. In the second phase, it generates a synthetic workload, assigning transactions to clients using a statistical model to generate them at appropriate frequencies based on workload parameters. The workload generator creates and populates a timeline for each of the clients.

USING EMOD TO MODEL THE SVT SITE

Having created EMOD, we now want to use it to model SVT, a sample e-commerce site. This is a simple two-tier database-backed website, with a typical range of page actions: viewing the home page, browsing details of an item for sale, running a search over the site, viewing ads, logging in as a customer, adding an item to the shopping cart, viewing the cart, and checking out (page actions are shown in Figure 4). Note that SVT assumes a two-tier hierarchy where the business logic is executed on the web servers themselves.

We will create a separate EMOD transaction for each of the page actions in SVT, and therefore need to benchmark the website in order to derive costs for the various actions in a transaction. For this we used a cluster of 16 machines, which could be configured into different numbers of clients and servers, as shown in Figure 3. All machines were dual PII-300s with 384MB of RAM running Windows 2000 Advanced Server SP2, connected by two private switched Fast Ethernet networks to simulate the internet and intranet networks. Specific roles were configured as follows: the client machines ran Microsoft's Web Application Stress tool [WAS01]; the front-end servers ran Microsoft IIS 5.0; and the back-end server ran Microsoft SQL Server 2000. All servers also ran Microsoft Commerce Server 2000.

To test each page, we used WAST to record our actions as we visited it. This script was then replayed using multiple clients against a single web server. Clients were added until the performance of the system (as measured by its thoughput in transactions per second) leveled off. The following Windows performance counters were recorded for all servers:

Device	Performance counter
Processor	%Processor Time
Network Interface	Bytes Sent/Sec
Network Interface	Bytes Received/Sec
Physical Disk	Disk Transfers/Sec

The resulting figures per transaction are shown in Figure 4. Note that the transactions from ViewCart onwards require the user to have logged in first. The WAST script therefore contains both a Login transaction and the transaction being tested, and the costs of the Login transaction are subtracted to get the

Name	Internet Request (kbytes)	IIS CPU Load (Mcycles)	LAN Request (kbytes)	SQL CPU Load (Mcycles)	SQL Disk Transfers	LAN Response (kbytes)	Internet Response (kbytes)
Home	2.9	3.2	0	0	0	0	18.4
Browse	0.8	0.8	0	0	0	0	7.4
SearchASP	0.7	5.8	0	0	0	0	4.0
Ads	1.0	13.8	0	0	0	0	5.6
Login	8.1	7.2	0.9	1.6	0	0.1	26.2
ViewCart	1.0	15.2	0.3	1.0	0	0.1	5.9
AddItem	2.8	36.8	1.9	4.2	0	0.3	7.1
Checkout	2.9	59.3	6.9	10.6	2	0.9	8.2

Figure 4: Measured transaction costs for the SVT sample site.

final results.

Hardware Models and Parameters

For the purposes of testing we wrote very simple hardware models for the CPU, network, and disk models. These use a linear function to map the size of a request (number of CPU cycles, message size in kilobytes, or number of disk operations) to the time required. The behavior of these models can be adjusted via hardware parameters that represent CPU speed, network bandwidth and latency, and disk seek time. As noted previously, these can be replaced by more complex hardware models, as long as they support the same basic actions.

The workload parameters used in the study are shown in Figure 5. Note that all of the parameter names ending in Freq represent transaction frequencies – they are referred to in the transaction script created for the SVT site.

Parameters:

Name	Description	Type
ClientName	Name of client layer	string
Layer1Name	Name of first server layer (...	string
Layer2Name	Name of second server la...	string
NumTrans	Number of transactions to...	uint
MarkTrans	Number of transactions b...	uint
TransactionFile	Name of file containing tra...	string
HomeFreq	Frequency of home page ...	float
BrowseFreq	Frequency of browse oper...	float
SearchFreq	Frequency of search oper...	float
SearchASPFreq	Frequency of alternate se...	float
AdsFreq	Frequency of ads page o...	float
LoginFreq	Frequency of login operati...	float
CartFreq	Frequency of view cart op...	float
AdditemFreq	Frequency of add item to ...	float
CheckoutFreq	Frequency of checkout o...	float
HomeDetailFreq	Frequency of home page ...	float
StudyTime	Time to run for	uint

Figure 5: Workload parameters for SVT

Modeling Network Traffic

There are at least two different approaches to modeling the internet and backbone network traffic generated by WAST scripts. The simple approach is to use aggregate numbers for network traffic, lumping together all the network actions in a given transaction. For example, a web page consisting of six separate elements is transferred in real life using six separate HTTP request/response pairs, of different sizes. We can represent these with a single large HTTP request/response pair, deriving the sizes very easily from the aggregate WAST traffic numbers. The end result will be a model that accurately represents network throughput in terms of total bandwidth, but not latency. The advantages of this approach are that the data can be collected very easily, and the resulting script is very simple. This approach can also be used for modeling traffic on the backbone network, using performance monitor counters of bytes/sec for the various network interfaces to get the data.

The more accurate approach is to model each of the network actions individually. For the front-end machines this is fairly straightforward, using the content length figures from the page data section of the WAST report to get HTTP response sizes. HTTP request sizes can be approximated to be identical, and can be derived from the aggregate WAST traffic divided by the number of requests. For back end machines, WAST does not capture the number of messages sent, only their size. We therefore have to use extra tools and techniques to monitor network traffic during a single test transaction. This approach can give full modeling of traffic and latency effects between the client and front end machines, at the cost of some extra complexity in capturing the data, and much lengthier XML scripts.

We have chosen to use the simple approach to model the SVT site. A fragment of the resulting transaction script is shown in Figure 6: this models a Checkout transaction as a single request from the client to the IIS server, computation on the IIS server, a single request to the SQL server, computation and disk operations on the SQL server, and finally communication chained back to the client via the IIS server.

```
<transaction name="Checkout"
             frequency="CheckoutFreq">
  <action name="Inet:icomm" device="Client">
    <config name="target" device="IIS" />
    <config name="msgsize" value="2.9" />
  </action>
  <action name="compute" device="IIS">
    <config name="cpuops" value="59.3" />
  </action>
  <action name="LAN:lancomm" device="IIS">
    <config name="target" device="SQL" />
    <config name="msgsize" value="6.9" />
  </action>
  <action name="diskop" device="SQL">
    <config name="DiskOp" value="2" />
  </action>
  <action name="compute" device="SQL">
    <config name="cpuops" value="10.6" />
  </action>
  <action name="LAN:lancomm" device="SQL">
    <config name="target" device="IIS" />
    <config name="msgsize" value="0.92" />
  </action>
  <action name="Inet:icomm" device="IIS">
    <config name="target" device="Client" />
    <config name="msgsize" value="8.2" />
  </action>
</transaction>
```

Figure 6: XML representation of the SVT Checkout transaction for EMOD.

Note the use of CheckoutFreq as a named variable in the second line of the script – this is looked up in the workload parameters shown in Figure 5. By contrast,

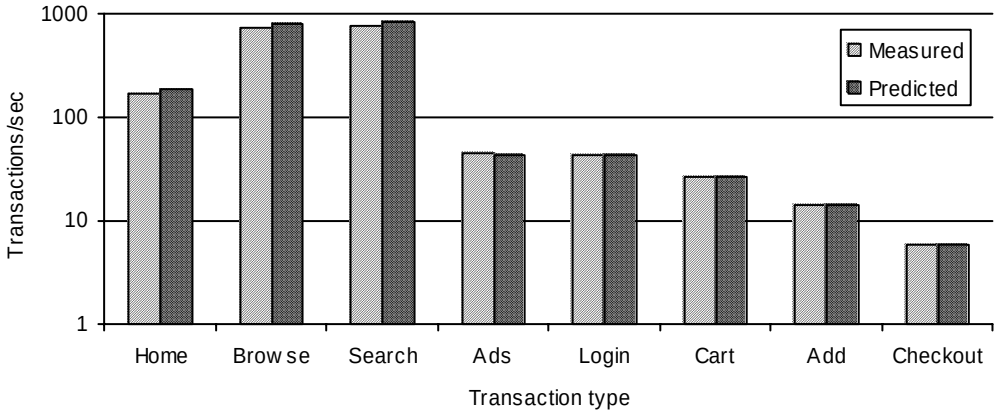


Figure 7: Measured versus predicted transaction rates for SVT site

the costs of the individual actions are hard coded into the script, which must be edited in order to change them. These could also be named variables, which would make it easier to adjust them but at the cost of a much longer list of workload parameters.

RESULTS

We now show that Indy can give accurate results. At the simplest level, we can replicate the experiments from which the SVT model data were collected. Figure 7 shows the measured and predicted results for the maximum transaction rates achieved by one IIS server in our experimental setup. The predictions are accurate to within 5%. This level of accuracy could also be achieved using simple regression techniques and transaction cost analysis [MSS99].

We can also validate the effect of increasing the load on the system by increasing the number of clients making simultaneous transactions. The measured and predicted results for the SVT home page transaction running on a single IIS server are shown in Figure 8. Note that although the SVT model underestimates the overall system throughput for small input loads, and overestimates it for higher loads, it accurately models the relationship between throughput and CPU load on the IIS server (that is, the corresponding shapes of the TPS and CPU lines). It also conservatively predicts system saturation at 8 simultaneous clients, versus an observed saturation point of 10 clients – this behavior cannot be obtained using TCA techniques. The SVT model could obviously be further refined to match the observed shape of the performance curve.

Indy also offers the ability to easily ask "what-if" performance questions, and further analyze the results. For example, Figure 9 shows bottleneck analysis of a hypothetical server farm with a single SQL server and multiple IIS servers, running sequences of "login, add product to cart, checkout" transactions. For a fixed input load, the system shows no performance improvement beyond seven IIS servers. When we look at the Indy kernel variables that report average queuing delays in each of the active components, we see that the SQL server has reached saturation point. After this point the system throughput will remain the same until we increase the number of SQL servers or their

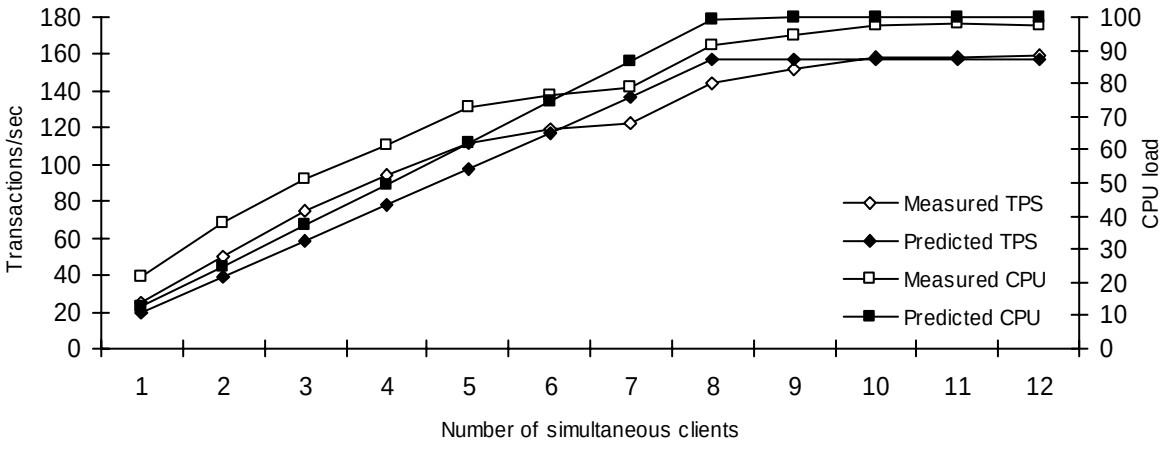


Figure 8: Measured and predicted transaction rates and CPU load on IIS server for a single transaction type as workload is increased

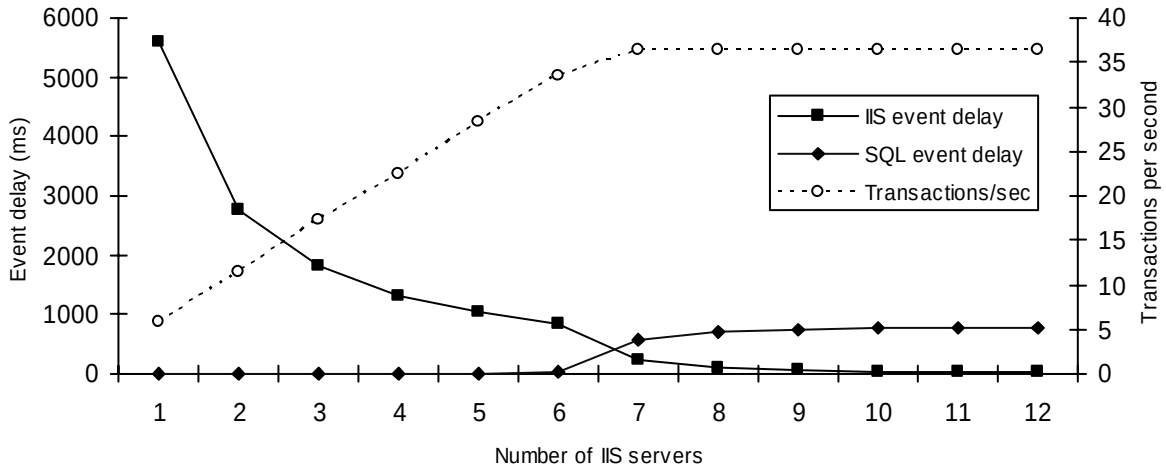


Figure 9: Transaction rate for fixed load plateaus after seven IIS servers: EMOD bottleneck analysis shows that the SQL server has become the limiting factor

performance.

Finally, we can use Indyview to examine events in more detail. For example, Figure 10 shows the components of an SVT individual transaction being run on the threads of IIS and SQL servers, using a time-space diagram.

EXTENSIONS

As described, the EMOD tool does not utilize two important features of the Indy kernel: schedulers and resource contention timelines. These can be used to further improve its realism and accuracy.

Schedulers

EMOD uses a simple round-robin mapping of events to devices. This is sufficient for the uniformly random workloads and identical hardware configurations tested so far. However, it will produce incorrect results if the expected load on each server is not the same. A real-life example of this would be a partial upgrade of the web servers on an e-commerce site, increasing the CPU speed of half of them. Assuming that a dynamic load-balancing package is being used on the site, the

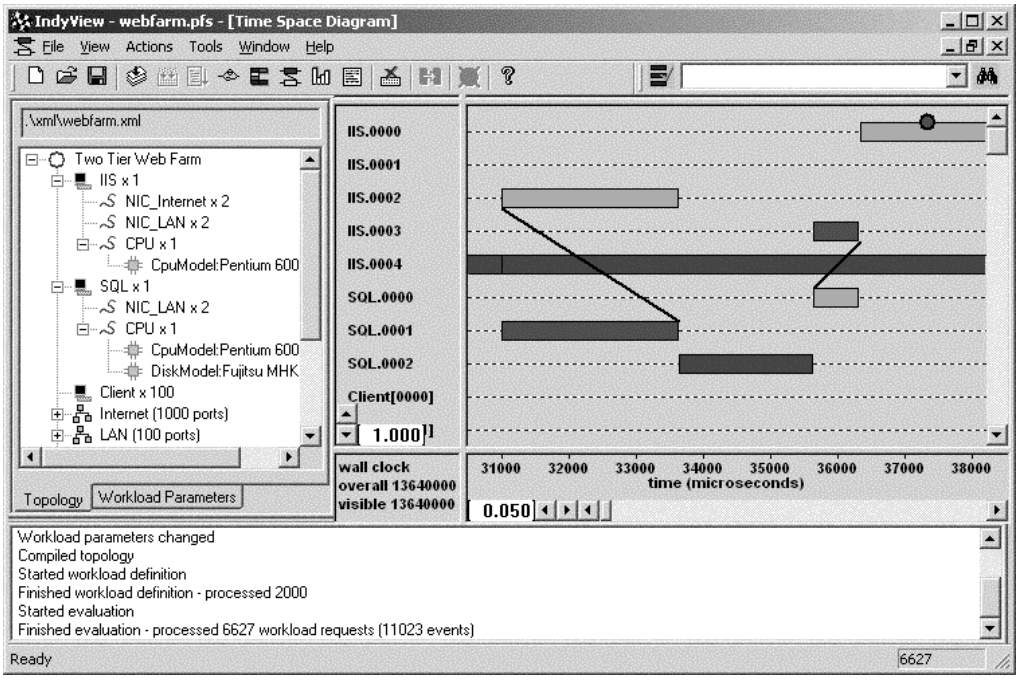


Figure 10: EMOD model running in Indyview. An SVT transaction is being inspected using a time-space diagram to show the activity of individual threads of control.

faster servers will then be assigned proportionately more of the load. Although EMOD would correctly model the throughput of such a site, it would overestimate the total time required to process a given workload (or, equivalently, the delay experienced by users). Another example would be dynamic load-balancing of computation requests amongst multiple CPUs in an SMP server. If the requests are not uniform, EMOD's simple round-robin mapping would result in queues rapidly developing on those CPUs that happen to get several large requests in succession, while other CPUs that happen to get several small requests would quickly complete them and then sit idle. Again, while total throughput would be accurately modeled, overall time taken and individual response time would not be.

To solve this problem, we must delay assigning events to devices until evaluation time, when system status is known. This is a decoupling of causality (which events cause which other events) from evaluation (where events are processed and how long they take).

We have therefore extended the Indy kernel to support dynamic scheduling of resources. A group of resources can now be controlled by a single scheduler, which chooses at evaluation time which of the resources each incoming action should be assigned to. The algorithm used by each scheduler is of course customizable, and can directly correspond to the equivalent algorithm being used in real life. Thus, a load-balancing web server scheduler might allocate requests based on the queue length of each web server. Actions can then be assigned to a scheduler, rather than to the event list of a particular device.

Note that existing performance modeling tools typically do not suffer from this problem, because they tightly couple the generation of the workload with its evaluation. That is, generation and evaluation proceed in lockstep. This simplifies the process of generating new events, because they have full access to the current state of the system. However, this tight coupling is unsuitable for a component-based toolkit such as Indy, where we want to be able to cleanly separate the functions of the different components. Adding schedulers to Indy restores the lost information.

Resource contention timelines

As described, the Indy kernel models contention between devices for shared resources. For example, the bandwidth of a network is shared amongst the messages currently traversing it. This is a first-order effect: sharing a resource directly reduces the amount of it available to any one request. In real life there can be additional second-order effects, related to the number of simultaneous requests. For example, as the traffic on a shared Ethernet network approaches the theoretical maximum capacity of the network, contention between messages (and the resulting backoff and retransmit actions) effectively reduces the total bandwidth available. Thus, message transmit times become longer than a simple linear model would suggest.

In order to model these second-order effects, we have added resource contention timelines to the Indy kernel. These track the instantaneous usage of every resource being modeled, and feed this information back to the appropriate hardware models. The hardware model can then impose additional costs on any events currently taking place. For example, in a model of a shared Ethernet, the resource tracked could be the number of messages contending for the network, and the hardware model would map this number into an extra delay for each message.

As well as increasing the accuracy of the modeling process by capturing second-order performance effects, resource contention timelines also enable the user to be shown a much more intuitive view of the system. For example, Indyview can simultaneously display a graph of the contention for a particular resource above a time-space diagram showing events taking place on that resource.

Figure 11 shows both dynamic scheduling and resource contention timelines being used in an Indy performance study of the IBuySpy ASP.NET site.

RELATED WORK

The Microsoft Commerce Server [MSS99] uses a methodology that is based on Transaction Cost Analysis (TCA) aiming to characterize the performance of the commerce site, determine bottlenecks, and perform capacity planning. A web stress application tool is initially employed to measure the transaction rates and the resource utilization varying the client load. A usage profile aiming to capture the anticipated user behavior is then defined. TCA is used to measure the cost of individual transaction costs. The capacity of the site is determined by dividing the cost of operation into the total CPU capacity available for the server.

[MEN00] includes a detailed methodology for modeling commerce sites, evaluating infrastructure and services, and perform capacity planning. Workload formalisms are described for commerce sites including a Customer Behavior Model Graph (CBMG) and a Customer Visit Model (CVM). Other formalisms are also introduced to describe the software control flow and interactions. Analytical models for various aspects of the commerce site operation are also discussed, such as authentication protocols, secure transactions, and payment systems. Queuing models are employed to represent the more complex aspects of the commerce architecture.

In [LOO00] a framework is presented for enumerating the components of the response time of the transaction of a Commerce Site. Each transaction type is analyzed into stages of processing and communication that take place during its lifecycle. The authors claim that the framework can form the foundation of a systematic review process that has the ability to expose performance problems and reveal possibilities for improving response times. The data gathered and organized into this framework can be further utilized as inputs to performance models for identifying architectural alternatives.

CONCLUSIONS

In this paper we have described how to use Indy, a new performance modeling infrastructure, to create EMOD, a simple performance model for e-commerce sites. We have then validated this model using the SVT sample site. We have shown that:

- Using the Indy kernel, a simple model can make reasonable performance predictions.
- The Indyview interface can be used to answer "what if" performance questions
- A model can be enhanced to use additional Indy features to improve its accuracy.

We hope to release a revised version of the EMOD tool on MSDN (http://msdn.microsoft.com) by early 2002.

REFERENCES

[LOO00] C. Loosley, R. Gimarc, and A.C. Spellmann, "E-Commerce Response Time: A Reference Model", in Proceedings of the Computer Measurement Group's 2000 International Conference (CMG 2000), Dec 2000.

[MEN00] D.A. Menasce and V.A.F. Almeida, "Scaling for E-Business", Prentice Hall, 2000.

[MSS99] Microsoft Site Server Commerce, "Using Transaction Cost Analysis for Site Capacity Planning", http://www.microsoft.com/technet/default.asp

[PAP00] E. Papaefstathiou, "Design of a Performance Technology Infrastructure to Support the Construction of Responsive Software," in Proceedings of the 2nd International Workshop on Software and Performance (WOSP 2000), Ottawa, Canada, Sep. 2000, pp. 96-104.

[SPE00] A. Spellmann and R.L. Girmarc, "eBusiness Performance: Risk Mitigation in Zero Time", in Proceedings of the Computer Measurement Group's 2000 International Conference (CMG 2000), Dec 2000.

[WAS01] Microsoft Web Application Stress web site, http://webtool.rte.microsoft.com/.

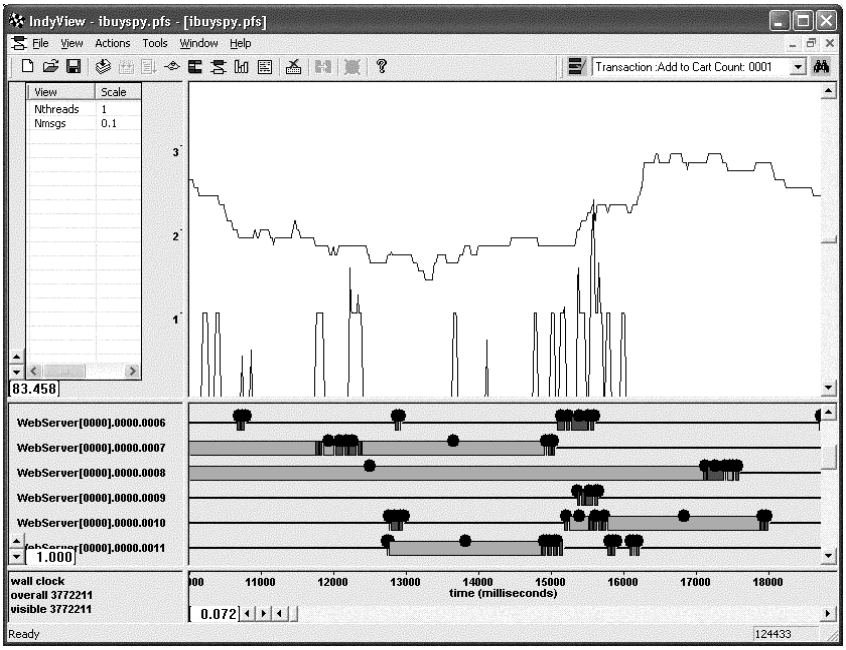


Figure 11: View of resource contention timelines and dynamic scheduling in the Indyview interface. The lower panel shows events taking place on multiple dynamically-scheduled threads on a single CPU of a web server. The upper panel shows resource contention timelines tracking threads on a CPU and messages on a LAN.