A unifying approach to performance analysis in the Java environment

by W. P. Alexander
R. F. Berry
F. E. Levine
R. J. Urquhart

In general, performance analysis tools deal with large volumes of highly complex data of varying types and at varying levels of granularity. The result is that it is common for there to be many different tools and components that implement performance data collection, recording, and reporting in an analysis environment. This variety complicates communication within a group and makes cross-group communication about specific performance findings even more difficult. The analysis of the performance of Java™ virtual machines and Java applications introduces additional complexity. We describe an approach that unifies the recording and reporting components of performance analysis into a single data model and standard set of reports. We have employed this model with significant success in the analysis of IBM's Developer Kits for the Java virtual machine.

The performance analyst's job is straightforward: measure performance, find constraints to the level of performance achieved, eliminate or reduce their effects, and then start again; stop when measured performance achieves a previously agreed-to target. The challenges are enormous. Software performance can be degraded by many factors, e.g., by a particular hardware configuration, by the way the hardware is used by the software, by poor programming practices in the underlying operating system, by unexpected interactions between software modules, by inappropriate use of system resources by application or middleware software, and by poor programming or data-structuring technique in the application. The analyst's objective is to isolate the primary cause and deal with it as quickly as possible. Even for small software applications it can be difficult; for highly complex environments it can be daunting.

The Java** virtual machine (Jvm)[1] presents an even more complex challenge to the performance analyst. In many cases, the Jvm operates as a process in a traditional operating system (OS) environment. (Even when more tightly embedded, there is a small but significant microkernel below the Jvm layer that provides basic OS functionality.) Java applications run within the Jvm process. The Java programming language is an object-oriented environment affording a very rich library of classes on which programmers can base their applications. The net result is that applications, and the Jvm itself, consist of an assembly of many relatively small pieces of software organized in a highly layered structure. Jvm code is reused extensively, so that the same functions are called from many places. The environment can be very dynamic; classes can be loaded at run time; the user can modify the function of many system classes through extension. Add to this the multitude of operating modes for Jvm and application code: interpreted Java bytecode, just-in-time (JIT) compiled code, native Jvm library code, native user code, and core Jvm

0018-8670/00/$5.00 © 2000 IBM

internal code, and the analysis alternatives are significant.

Many tools have been developed to assist analysts in dealing with these challenges. These tools include system and application profilers, e.g., AIX* (Advanced Interactive Executive) tprof,[2] gprof,[3] Intel's VTune**,[4] application and system tracing facilities, e.g., AIX Trace,[2] application and system memory use profilers, e.g., svmon,[2] and system performance monitors, e.g., Windows NT** Performance Monitor.[5] Implementations of these and similar tools exist on many (though not all) platforms. Unfortunately, their implementations are not consistent, their output formats not readily comparable, and their models for computation and resource consumption not equivalent. We view the benefits of a unifying approach for performance measurement, reporting, and analysis as profound: analysts working on a single system could readily switch from one type of analysis to another (e.g., from memory analysis to delay analysis); analysts working on quite distinct systems could more readily share performance data and results (e.g., path length analysis comparisons for a key operation between the OS/2* [Operating System/2*] Jvm and AIX Jvm implementations).

In this paper we describe our efforts toward developing a unifying, general model for recording and reporting resource consumption that supports a broad range of performance data and a broad range of analysis questions. In the next section, we provide motivation for the model. We then describe the technique, which we call "arcflow," in more detail, and we present a detailed example. The model is similar to work described by Hall et al.[6] and Ammons et al.[7] In the fourth section, we discuss how this technology fits into the context of an instrumented Jvm. Also, natural extensions to Jvm and system instrumentation that derive from the adoption of this approach are introduced in that section. We conclude with a discussion of further work.

Motivation and contribution

Computer operating systems (e.g., Microsoft's Windows NT, IBM's AIX and OS/390*, or Operating System/390) are resource managers: they remove the burden of correct, efficient, and fair resource allocation and management from the application programmer's concern. The scope of the resource allocation function begins at a very low level with the allocation of the CPU instruction and data fetch and execution units. It extends through system layers to include the allocation of physical and virtual memory, the creation and assignment of system buffers, the realization of files or blocks or pages in a file system, and the creation and life cycle of data structures representing threads of execution. Middleware, such as Netscape Application Server**, the IBM WebSphere*, Sapphire/Web** from Bluestone Software Inc., and application programming frameworks such as the IBM SanFrancisco* project, build on the system foundation and present layer upon layer of further resource concepts (e.g., socket, connection) to the programmer. The Jvm introduces additional key resource types, including interpreter cycles (e.g., as consumed in interpreting bytecode), heap objects allocated, heap bytes allocated, "JITed" code instructions executed, and objects of different classes.

These resources are then consumed by various actors, or agents, in the process of achieving some desirable programming objective. A thread allocates a buffer and passes control to a function to fill that buffer with data from a file. A process creates a pool of threads, each of which opens a socket to communicate work to a corresponding thread within another process. Program X executes 1000 instructions, calls program Y 100 times, which subsequently executes 1000 instructions before returning to program X. A list of key consumers for the Jvm and Java application environment would include functions, basic blocks, threads, objects, transactions, interpreted methods, system calls, and JITed methods.

Measured performance is a combination of the efficiency with which the system and middleware layers provide resources to consumers and the manner in which these resources are consumed. In general, system and application performance problems are most quickly identified by viewing the relationship between resources and consumers of those resources. Most performance tools report data that reflect and quantify this basic relationship. For example, consider the simple report obtained from the execution of a simple program shown in Figure 1.

Of the total 10 seconds, eight were consumed by Thread 3. If it is assumed that a reduction in the total time spent is the appropriate objective, this report suggests that it would be sensible to concentrate future tuning efforts on Thread 3 and what it is doing with 80 percent of the total CPU time.

Thus, the universe of system concepts is organized along two somewhat natural and mostly orthogonal dimensions: *resources* and *consumers* of resources.

Figure 1 Simple report on a simple program

```
CPU TIME = 10 SECONDS
CPU TIME BY THREAD:
  THREAD 1: 1 SECOND
  THREAD 2: 1 SECOND
  THREAD 3: 8 SECONDS
```

Table 1 Performance analysis steps

Step 1	Step 2	Step 3
Collect data	Record data	Generate report

Table 2 Performance analysis steps

Step 1	Step 2	Step 3
Collect data	Record into standard model	Generate standard reports from the standard model

As suggested above, this is the classic, traditional organization that is reflected in most performance tools today. As noted in Reference 6, this organization is inadequate. It is useful for answering simple "what or which" questions, e.g., which function consumes the most CPU time? However, deeper analysis questions, e.g., the "where" and "why" questions, require additional context.

A unifying approach to data recording and reporting

Table 1 lists in very general terms the basic steps of performance analysis. First, performance data are collected. The data could be event data, sampled data, counter data, or some combination of these different types. The data are then recorded in some form (e.g., a trace file). Finally, reports are produced that are made available to analysts for their use in identifying problems and discovering opportunities for improvement.

The contribution of the present paper is depicted by the steps in Table 2. We introduce a generic model that allows this activity to be automated in a manner that is largely independent of the actual data being collected. This model is employed primarily at Steps 2 and 3 of the process—at data recording and at report generation time.

The model is open in that it embraces data collected in a wide manner of ways. The model also admits different implementations, enabling steps to be performed in either real time or postprocessing modes. What these implementations mean is that data can be collected and analyzed off line, or collected and analyzed (reported on) live. In both cases the model is the same, and the reports are the same. (We motivate and discuss live recording of data later in the paper.)

We now describe the arcflow model in detail.

The arcflow model. In order to capture the key relationships between consumers and resources necessary to help analysts answer performance analysis "what," "where," and "why" questions, we provide the three definitions that are described next.

Definition of *consumer_context*: A consumer_context is an abstraction of the state of a consumer *at the point of* resource consumption. This mapping reflects the state of the consumer at the time we record the consumption of the resource in question.

We define a consumer_context as a set of consumer identifiers and an optional time stamp:

$$\{c_1, c_2, \ldots c_n, t\}$$

where c_i, $1 \leq i \leq n$, is the ID (identifier) of the ith consumer and t is the time stamp. There is an explicit hierarchy, i.e., c_i is the parent of c_{i+1}. Each c_i corresponds to a consumer, with c_n being the consumer most immediately responsible for the resource consumption, c_{n-1} is the consumer responsible for the actions of c_n, and so on. This hierarchy reflects the desired context in the level of detail appropriate for the analysis, e.g., one natural consumer_context hierarchy is a consumer calling sequence.

The rationale behind this definition is that resource consumption occurs in context, and it is important to understand that context when exploring performance improvements.

For example, a "callstack" A-B-C, reflecting that function A called function B called function C, is a valid consumer_context. Any resources consumed by C when B and A are on its invocation stack are

considered distinct from resources consumed under a different invocation stack, *even if* consumed by (within) C. A more detailed variant on the above might record the callstack in more detail, e.g., A:F0001330-B:F0002992-C:F0010122, where the hex addresses following the function name indicate the actual virtual address of the invocation. (The value lies in discriminating between multiple call sites in a function.) Another valid consumer_context would record both the callstack and time stamp at the time of the resource consumption.

Although the most common consumer_context reflects a function/method invocation hierarchy, it need not. Other variants include:

- Object containment hierarchy (where each consumer ID, c_i, represents a unique object). The hierarchy reflects that object c_i contains c_{i+1}.
- Object allocation/creation hierarchy (where each c_i again represents a unique object). The hierarchy reflects that object c_i created c_{i+1}.
- Module invocation hierarchy (where each c_i indicates a module). This hierarchy is equivalent to the calling sequence hierarchy but with a different level of granularity for the consumer identifiers.
- Basic block invocation hierarchy (where each c_i indicates an instruction-level basic block). Again, this is an invocation hierarchy but, in this case, one that employs a finer granularity.

Definition of *resource_consumption_list*: A resource_consumption_list is a list of resources and an indication of the total amount of resources consumed. When coupled with a consumer_context, this represents the total amount of each resource consumed within a specific context.

For example, <"object allocations" 100, "heap bytes allocated" 12450> is a resource_consumption_list indicating that a total of 100 object allocations have been made and that 12450 bytes have been allocated from the heap.

Definition of *arcflow model*: The arcflow model couples consumer_context data and resource_consumption_list data in order to provide a useful indication of resources used in some context. Thus, our general model of resource consumption consists of a set of triples and a descriptive vector:

AF0 $\equiv$ {(X, n, Y) such that

X is a consumer_context,
n is the number of times this consumer_context has been encountered, and
Y is a resource_consumption_list}

and

Z, a vector of descriptive bindings

where **Z** provides:

- A label describing the consumer identifiers (c_i) in X (e.g., "functions")
- A label describing the units employed in the resource_consumption_list (e.g., "CPU_time," "instructions")
- A label describing the meaning of n—the number of times a consumer_context is encountered (e.g., "calls," "entries")

Note that, depending on the granularity of the abstraction employed, this definition allows us to discriminate between executions of consumer A resulting from being called by consumer B and those resulting from being called by consumer C. Note that if the consumer_context includes a time stamp, the model can record an explicit log of all events, in effect a trace of system behavior with respect to all interesting consumer states and resources they have consumed.

Note further that the model is unitless; this characteristic is a powerful abstraction that allows it to be employed in a broad range of analyses. Indeed, it is the key to the success of the model in unifying data recording and reporting. However, unitless reports are not very helpful, so **Z** is introduced to allow for the labeling of fields produced by the tools that implement the model and derivative reports.

Although the above is quite general, we have found that a relatively storage- and computationally-efficient choice for the consumer_context is to eliminate the explicit time stamp. Similarly, we have found that restricting the resource_consumption_list to a single resource (thus providing only a univariate view of resource consumption) is highly effective for focused analysis where the key resource of interest is known in advance. Therefore, for the remainder of the paper the term *arcflow* refers to the restricted arcflow model:

Figure 2 Sample resource use trace

```
0 pidtid AC_test
0 > Main
0 > A
1 > B
2 < B
2 > B
3 > C
4 < C
4 < B
4 < A
4 > B
5 > A
6 > C
7 < C
7 > X
7 > E
8 < E
8 > F
9 < F
9 > G
10 < G
10 < X
10 < A
10 < B
10 < Main
```

Table 3 Arcflow model data for AC_test.java

X consumer_context "Method names"	*n* number of occurrences "Number of calls"	*Y* resource_ consumption "cpu-seconds"
Main	1	0
Main A	1	1
Main A B	2	2
Main A B C	1	1
Main B	1	1
Main B A	1	1
Main B A C	1	1
Main B A X	1	0
Main B A X E	1	1
Main B A X F	1	1
Main B A X G	1	1

AF1 ≡ {(X, n, Y) such that
X is a consumer_context having no time stamp,
n is the number of times this consumer_context has been encountered, and
Y is a resource_consumption_list consisting of a single resource}

and

Z, a vector of descriptive bindings

The model described in the foregoing, AF1, forms the basis for a powerful set of standard performance reports. It is these reports that analysts use to answer the questions above. We first introduce an example. This example is used first to illustrate the data model and then to introduce the core reports, or views, that the arcflow model facilitates.

Suppose we have collected the raw trace data in Figure 2 indicating resource acquisition and release events during the execution of a sample application, AC_test. Two types of records exist in the example. The first, and first record in the example, indicates a process/thread switch. In this case, it indicates that all subsequent records (up to the next process/thread switch) reflect activity performed by the process/thread: AC_test. (The name is only illustrative. In general, process/thread identifiers have a different appearance.) The next type of record indicates either the start of a resource use (">") or the end of a resource use ("<"). The name of the consumer is also indicated (in this case, the name of a method in the Java program). Each record is marked with a resource consumption level. As with Reference 6, we focus on resources whose consumption is measured against a monotonically increasing metric (e.g., system CPU time, wall clock time, and instructions executed). In this example, the metric is CPU time in seconds, and each trace record begins with a CPU time stamp.

When recorded into the arcflow model, the resulting representation for the performance data in Figure 2 is shown in Table 3.

The differences in the time stamps of successive trace records have been used to compute the total resource_consumption for each consumer_context, and the time stamps themselves are no longer needed.

Note the explicit representation of the consumer_context, in this case, the callstack. It is these contextual data that provide the additional information required to gain insight into the "where" and "why"

analysis questions. In particular, the context provides more information about application structure, and explicitly surfaces leverage points for performance improvement.

The reader will notice significant redundancy in the model shown in Table 3. In particular, many of the consumer context entries share common prefixes (e.g., "Main B A"). Fortunately, the structure and semantics of the consumer context allow for very efficient storage of the arcflow model. Recall that the consumer context represents a hierarchy, with each consumer identifier in a superior position to its immediate successor. This hierarchy has many possible interpretations; the specific interpretation depends on the particular bindings associated with a specific application of the model (i.e., a specific set of data collected and recorded in the model). The most common interpretation is that of a callstack, e.g., for consumer context $\{c_1\ c_2\ c_3\}$, the interpretation is that c_1 called c_2 and c_2 then called c_3. Another interpretation is containment, e.g., object c_1 contains c_2 and c_2 contains c_3.

We employ a tree structure for recording the arcflow model. This tree structure (which we have named the *call-tree*, in deference to the most frequent use of this methodology) consists of a set of nodes, with each node containing the following information:

{
parent consumer ID,
consumer ID,
number of occurrences,
total resource consumption,
list of children consumer IDs
}

There is a single root node having no parent.

This approach reduces the storage costs of the model to being roughly proportional to the total number of unique contexts in which consumer IDs are encountered in the measured data. In the case of the present example, there are 11 such unique contexts. Thus, the call-tree contains 11 nodes. The call-tree for our example is shown graphically in Figure 3. Each node is shown with the consumer ID, number of occurrences, and total resource consumption. The parents and children are indicated graphically. (In fact, there is one more node corresponding to the process/thread consumer under which the methods Main, A, B, C, X, E, F, and G consumed the CPU resource. We have left this out of Table 3 and Figure 3 as a simplification.)

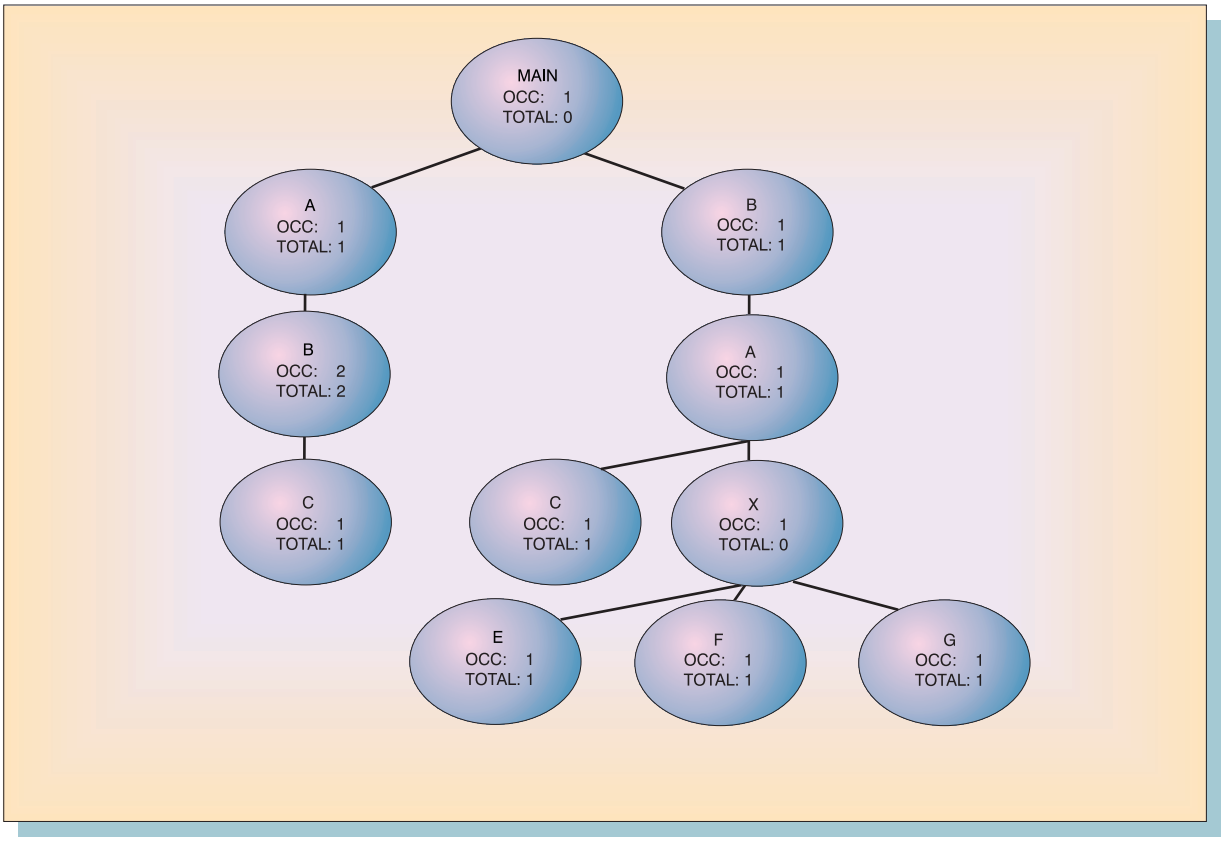
As noted in Reference 7, the storage cost reduction in employing this technique is very significant. For example, when the application method entries and exits for the portable business object benchmark (pBOB)[8] were instrumented and a two-minute event trace for the single-threaded case was collected, it resulted in the generation of approximately 2.8 million entry-exit and thread switch events, requiring approximately 40 MB of disk (or memory) storage. Transforming the raw event stream into the call-tree form of the arcflow model reduced the storage requirement by approximately 200-fold to 200 KB, while retaining the important resource consumption information. The reduction has broader implications than simply saving space and speeding data analysis. Since many applications typically reach steady state behavior with respect to program flow (that is, after a relatively short number of entry-exit events, most common call-stacks and consumer contexts are realized as elements in a call-tree), it becomes feasible to consider building the call-tree live, as resource consumption events are recorded. This way obviates the need to write the events at all, saving both storage and a significant number of processor cycles. Indeed, we have adopted this approach in some cases. We discuss these in the fourth section.

Arcflow reports. Several standard reports form the basis of the arcflow methodology.

The xarc report. This report is the one from which the methodology derives its name. Analysts are interested in the consumption of resources, but they are generally more interested in why they are consumed. The xarc report helps to answer these questions. This report, although textual in format, is actually a graphical rendition of application structure. Individual consumers (e.g., methods in our running example) are presented along with their resource (e.g., CPU time) usage, but the context for the consumption is made explicit by linking the consumer with its parents (e.g., its callers) and its children (e.g., the methods it calls). In this way it is possible to discern the magnitude of the resource usage of one method, the drivers of that use, and the implications of changing the behavior of this method.

The xarc report is organized in paragraphs, or stanzas. Stanzas are separated by dashed (==) lines. Each stanza includes a record for "self," a set of "parent" records, and a set of "child" records. Every unique

Figure 3 Arcflow model for example data



consumer is represented by a stanza in which the self record identifies that consumer by name in the "function" column. Three key types of metrics are reported for each consumer: "calls," "base," and "cum" (short for cumulative). Calls indicates the number of times that a consumption event (e.g., a function call) has been recorded on behalf of this consumer. Base and cum both pertain to the quantity of resource consumed. Base indicates the amount consumed directly by this consumer in the context defined by its set of parents. Cum indicates the amount consumed both by the consumer directly and all of its children. (The examples illustrate a variant of the arcflow reports that normalize the resource consumption to percentages rather than show the absolute counts.) Also, there is special treatment of recursion, which will not be discussed here.

Each stanza focuses an analyst's attention on resource consumption from the perspective of a single consumer (the one named in the self record for that stanza). We refer to this consumer in the following discussion as the self-consumer. Within each stanza, the interpretation of base and cum depends on which records are being considered. For the self record, base and cum apply directly to all resource consumption associated with that function, irrespective of context. For the parent records, the base value reports resources consumed by the self-consumer in the context of (e.g., as a result of) the indicated parent. For the parent records, the cum value indicates the resources consumed directly by the self-consumer and any of its children.

These relationships are formalized in a set of invariants. Understanding these invariants is important to navigating the report:

- Sum(parent(calls)) = self(calls): The total number of calls recorded for the self-consumer are broken out by which parent made the calls.
- Sum(parent(base)) = self(base): The total re-

source directly consumed by the self-consumer can be decomposed into resource consumed on behalf of each of the parents of the self-consumer.
- Sum(parent(cum)) = self(cum): The total resource consumed by the self-consumer (which includes all of its children's consumption) can be decomposed into total resources consumed on behalf of each of the parents of the self-consumer.
- Sum(child(cum)) = self(cum) − self(base): The resources consumed by the self-consumer can be decomposed into those directly consumed by the self-consumer and those consumed by its children.

As an illustration, Figure 4 shows the complete set of xarc report stanzas for our running example. If we locate the stanza for method main, main is shown as having one parent, named AC_test_pidtid. (All methods are executed on some thread of control; the instrumentation in our implementations associates activity with the thread that executes it. In effect, the thread becomes the highest-level consumer or driver of work.) Main is responsible for 100 percent of CPU time consumed (although none of that is within the body of method main; the children, B and A, are wholly responsible). The children of main are shown as B and A. B is responsible for 60 percent of the total CPU time consumption, whereas A is shown as being responsible for 40 percent.

A benefit of the xarc report is in reflecting the performance impacts of concepts such as reuse. The analyst can immediately see the relationships between key methods and thus can immediately begin to posit more interesting questions, such as, "What if X were eliminated or improved substantially?"

The xtree report. The xtree report most closely reflects the contents of the underlying model. Indeed, it is produced by navigating the call-tree directly. The value of the xtree report lies in the completeness of its depiction of application structure coupled with measured performance data. It communicates more information about program structure than the xarc report (e.g., discriminating between all unique calling sequences, whereas xarc will aggregate across calling sequences from the perspective of a single method). For example, consider the xarc stanza for method A and compare it with the xtree information reported below that includes A in the calling sequences. The xarc report is unable to discriminate between execution paths through A, e.g., A is called by main; A calls B, C, and X. But do all calls from main to A result in a call to C? Indeed no. Only the xtree report retains that level of structure. The xtree report for the running example is shown in Figure 5.

In Figure 5 we see that A calls B, which then calls C only one time. The other call from A to B does not result in subsequent calls to C or to any other (instrumented) method. Only the xtree report retains this level of detail regarding application structure.

The xprof report. The xprof report is the report with the least structural information retained, but it is also the report most closely associated with classic performance profiling. It provides a nearly flat profile of resource consumption from a consumer (e.g., method) perspective. Structural relationships are not reflected. Only the base and cum times indicate that one particular consumer may drive more consumption than is actually reflected in the direct actions of that consumer. (For example, consider the entry for X shown in Figure 6. Base time of 0 percent indicates no work done in this method, whereas cum time of 30 percent suggests that X is central to the performance of the application.)

The common structure revealed in the xarc report is not present here. Note in Figure 6 that although we can see the role that X plays in overall performance, we cannot determine which methods X calls. Even more telling, the complex interactions between B and A are not reflected at all. Nevertheless, this report has value; it is concise and provides a first-order assessment of performance improvement opportunity.

Building the arcflow model. Data for the model can come from a variety of sources. Most typically, and perhaps most naturally, the data result from instrumentation placed at the start and end of the consumption of a resource, e.g., at the entry and exit of a method, at the beginning and end of a basic block, and at the entry to a function and exit from that function. Other types of instrumentation data include an object containment hierarchy extracted from a detailed dump of the Java heap (e.g., such data are valuable for discerning heap-residency causation for a particular set of objects). Another example is callstack data obtained from C-heap memory allocation instrumentation (i.e., the callstack provides insights into which parts of the environment made the calls to *malloc()* and *free()*, and therefore may be useful in diagnosing memory leaks). All of these data have direct representation in the arcflow model.

Figure 4 xarc report stanzas (Units :: CPU time; Total :: 10)

```
Source        Calls      %Base       %Cum       Function
======        ===        ====        ====       ========

    Self          1         0.00      100.00     [0]  AC_test_pidtid

   Child          1         0.00      100.00     Main
 ======        ===        ====        ====       ========
  Parent          1         0.00      100.00     AC_test_pidtid

    Self          1         0.00      100.00     [1]  Main

   Child          1        10.00       60.00     B
   Child          1        10.00       40.00     A
============  ============
  Parent          2        20.00       30.00     A
  Parent          1        10.00       60.00     Main

    Self          3        30.00       90.00     [2]  B

   Child          1        10.00       50.00     A
   Child          1        10.00       10.00     C
============  ============
  Parent          1        10.00       40.00     Main
  Parent          1        10.00       50.00     B

    Self          2        20.00       90.00     [3]  A

   Child          2        20.00       30.00     B
   Child          1         0.00       30.00     X
   Child          1        10.00       10.00     C
 ======        ===        ====        ====       ========
  Parent          1         0.00       30.00     A

    Self          1         0.00       30.00     [4]  X

   Child          1        10.00       10.00     E
   Child          1        10.00       10.00     G
   Child          1        10.00       10.00     F
 ======        ===        ====        ====       ========
  Parent          1        10.00       10.00     A
  Parent          1        10.00       10.00     B

    Self          2        20.00       20.00     [5]  C

 ======        ===        ====        ====       ========
  Parent          1        10.00       10.00     X

    Self          1        10.00       10.00     [6]  E

 ======        ===        ====        ====       ========
  Parent          1        10.00       10.00     X

    Self          1        10.00       10.00     [7]  F

 ======        ===        ====        ====       ========
  Parent          1        10.00       10.00     X

    Self          1        10.00       10.00     [8]  G
```

Figure 5 xtree report (Units :: CPU time; Total :: 10)

```
Lv  RL  Calls    %Base    %Cum    Indent HkKey_HkName
 0   1      1     0.00  100.00     AC_test_pidtid
 1   1      1     0.00  100.00    - Main
 2   1      1    10.00   40.00    -- A
 3   1      2    20.00   30.00    --- B
 4   1      1    10.00   10.00    ---- C
 2   1      1    10.00   60.00    -- B
 3   1      1    10.00   50.00    --- A
 4   1      1    10.00   10.00    ---- C
 4   1      1     0.00   30.00    ---- X
 5   1      1    10.00   10.00    -----+ E
 5   1      1    10.00   10.00    -----+ F
 5   1      1    10.00   10.00    -----+ G
```

Figure 6 xprof report (Units :: CPU time; Total :: 10)

```
Calls      %Base    %Cum    Ind   Name
===        ====     ====    ==    ====
    1       0.00  100.00      0   AC_test_pidtid
    1       0.00  100.00      1   Main
    3      30.00   90.00      2   B
    2      20.00   90.00      3   A
    1       0.00   30.00      4   X
    2      20.00   20.00      5   C
    1      10.00   10.00      6   E
    1      10.00   10.00      7   F
    1      10.00   10.00      8   G
===
   13
```

For maximum flexibility, we have designed a basic set of input representations from which the arcflow model can be readily constructed:

- *Start and end events*—These events explicitly indicate the beginning and ending of a resource consumption by a consumer. They are typically associated with individual event-trace collection.
- *Stack-based events*—The events record the full consumer_context of a resource consumption. They are typically associated with sampling.
- *Complete call-trees*—The entire call-tree, as defined above, is input, and the arcflow model is constructed directly from the call-tree. (The arcflow model and the call-tree representation are isomorphic; the net result is that once a call-tree is provided, all other arcflow reports can be quickly generated.)

Depending on the application, data conforming to one of these input types are passed from the collection step (see Table 2) into the recording step, and the recording step constructs the arcflow model. As an illustration of model construction, the Appendix provides a listing of a Java implementation of a mech-

Table 4 Jvm metrics of interest

Where Collected	Data Collected	Type of Data	How Used
At method entry/exit	Method name, count of interpreted bytecodes	Event + counter	To identify methods with heavy bytecode content
At method entry/exit	Method name, CPU time	Event + counter	To identify methods consuming most CPU time
At method entry/exit	Method name, instructions executed (hardware counter)	Event + counter	To profile the JIT compiler for path length of generated code
At method entry/exit	Method name, other hardware counters, e.g., pipeline stalls, etc.	Event + counter	To identify methods responsible for hardware-related performance problems
At monitor request, acquire, release	Monitor identification, CPU time	Event + counter	To quantify lock use and contention
Periodically (e.g., every 0.01 secs)	Program counter	Sample	To identify methods consuming most CPU time
At C function (e.g., in the Jvm) entry/exit	Function address or name, CPU time	Event + counter	To identify run-time functions consuming most CPU time
At method entry/exit	Method name, count of objects allocated	Event + counter	To identify those methods most responsible for heap pressure (object allocation)
At completion of a garbage collection cycle	Object identification, object containment hierarchy for all live objects	Sample	To identify those objects responsible for heap utilization
Periodically (e.g., every second)	A trace of method entry/exits with CPU time	Sample + event	To identify the methods consuming most CPU time together with the added benefit of obtaining application structure

anism to build the model from a raw trace of start and end events.

The arcflow model implementation issues and experience

We collect many types of data in our analysis of the Java virtual machine. The most common type of data is event-oriented, collected at method entry and exit. However, many other types of data are collected to satisfy different analysis needs. Table 4 contains a summary of the most common.

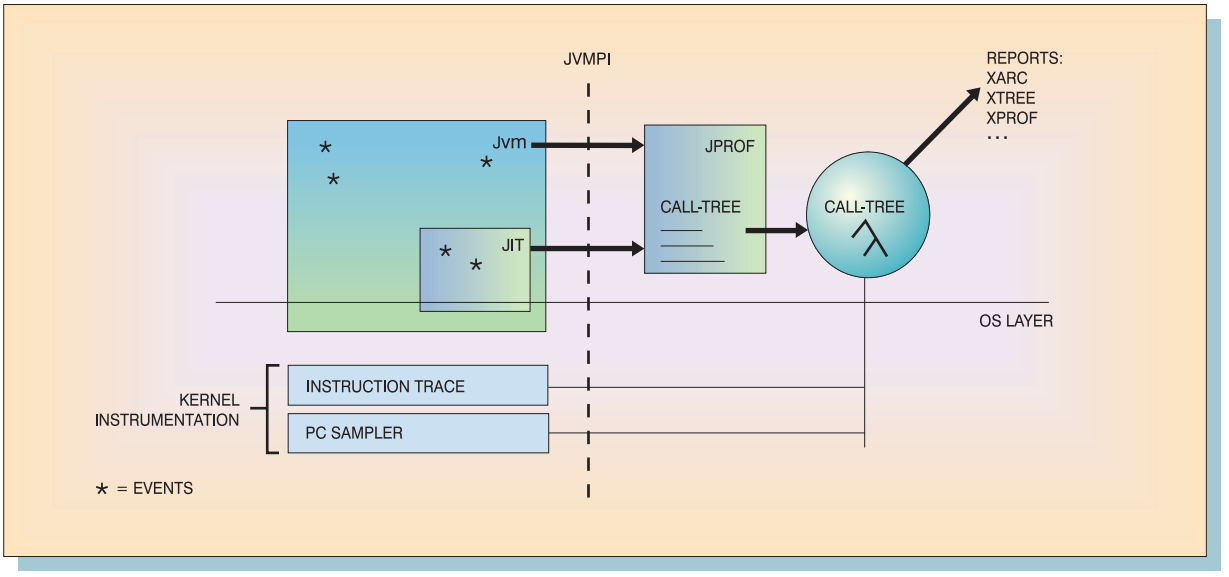
We were motivated to develop this model by the need to contain the number of reports and reporting tools for Jvm analysis. We were further motivated by a desire to develop a methodology and language that would apply across IBM platforms, thus enhancing internal communications about Jvm performance and reducing overall development costs.

Our primary development platform for this effort has been the IBM Developer Kit for Windows**, Java Technology Edition, Version 1.1.7. The IBM Developer Kit includes an implementation of a subset of the Java Virtual Machine Profiler Interface[9] (JVMPI). This interface provides a standard means for reporting events from the Jvm and its components, including the JIT compiler. With the JVMPI infrastructure in place, and with a fully instrumented Jvm for reporting method entries and exits, we have a very powerful demonstration vehicle for the value of this approach.

Figure 7 illustrates some key elements of our deployment of this technology for Jvm and Java application performance analysis. Depicted are several independent applications of the arcflow methodology here described. These implementations have been employed at various times in the effort to enhance various aspects of the performance of the IBM Jvms. Although not an exhaustive depiction, the variety of uses indicated in this figure provides compelling evidence of the power and success of the model in addressing our requirements.

In the first case, note that the Jvm and JIT compiler have been instrumented with events (represented by "*" in the figure). These events correspond directly with JVMPI events to support method entry and exit

Figure 7 Deployment of arcflow for Jvm and Java application analysis



notification to a profiler implemented as a dynamically loaded library (in the figure, jprof). The jprof library receives notifications of each method entry and exit (among other events). Each event is time-stamped, and the event is recorded dynamically into an arcflow model. Note that a different time-stamping mechanism can be employed in the profiler to effect quite different views of Jvm performance. For example, if the time stamp is CPU time, then the mechanism measures CPU time by method. If the time stamp is configured to be a count of hardware instructions completed (available to programmers on many platforms, including Intel as a feature on Pentium** and follow-on hardware), then instructions executed in a method forms the metric recorded and reported by the infrastructure. Using hardware instruction counts as a time-stamping mechanism is useful as a measure of JIT compiler effectiveness and may facilitate meaningful cross-platform JIT compiler comparisons. In contrast, to obtain a meaningful measure of overall performance, the preferred time-stamp metric is CPU time. As indicated in Table 4, other metrics are available—each providing a new, unique insight into Jvm and application behavior.

Figure 7 also indicates that for method entry and exit profiling, the profiler builds the model dynamically as the profiled Java application is executing. This implementation choice is a direct consequence of the storage and execution improvements that the call-tree affords. When the application completes, or on some other trigger, the call-tree form of the model is output, and then the standard arcflow reports are generated.

Other applications are illustrated in the figure. A classic profiling technique based on program counter sampling is available on many platforms. (We have developed this technique for AIX, OS/2, and Windows NT. The AIX version, called tprof, is described in Reference 2.) Each data item recorded by this instrumentation technique corresponds to a program counter value (a virtual address). This classic approach typically obtains a sample of the program counter every 0.01 second or so. Thus, hundreds of samples are obtained when an application is executed. Note that such program counter samples fit directly into the arcflow model. The consumer_context recorded at each sample is simply the program counter. The data recorded is a unit of one (to reflect one sample). The result is a relatively flat arcflow model, but one that presents samples organized by the program counter address in the xprof report. (Postprocessing is required to resolve these addresses into function names.)

A simple extension to the above profiling approach produces profound results and serves to reinforce the unifying power of this methodology. Motivation for this approach derives from examination of the arcflow model, AF1 (or AF0). Each entry in the model consists of a consumer context and a resource accounting. In the case of profiling function/method calls, the desired consumer context is precisely the callstack. Rather than obtain the callstack through explicit entry and exit instrumentation, we can obtain it by navigating the stack frames that exist in the environment. In the case of the program counter sampling profiling technique described in the previous paragraph, the natural extension is to record the callstack of the interrupted thread at the time of the interrupt in addition to its current program counter value. Note that many operating systems and application programming environments employ well-defined linkage conventions resulting in readily navigable stack frames. (Indeed, they must do so in order to function correctly and to support system and application debuggers.) This extends Table 4 with a new kind of data that uses sampling to achieve in-depth visibility to application structure; see Table 5. The net result is a quantified understanding of the call structure of the Jvm (and of the application) without the overhead of instrumenting every method/function entry and exit. Both collection time and storage costs are reduced dramatically. In a sense, it is an ideal approach in that it couples the low overhead of sampling with the high information content of a complete view of the context in which resources are consumed. Thus the consumer_context is realized in this application as a full (or even partial, depending on environmental limitations[10]) callstack. This approach is similar to the stack sampling described in Reference 6.

The entire range of arcflow reports now has direct and valid interpretation. This approach samples the entire program structure—not just where time is spent, but why it is spent there.

Yet another application is depicted in Figure 7. Instruction tracing technology exists for many operating systems; this technology takes advantage of hardware mechanisms to trace individual instructions executed by the operating system and its hosted middleware and applications. Such traces are extremely useful when analyzing the performance of the Jvm, especially when the output of the JIT compiler is considered. An instruction trace provides direct measurement of precisely the code that the JIT compiler is producing. As JIT compilers become more complex and more dynamic, the value of this instruction tracing technology increases (e.g., consider the very dynamic nature of the compiler in Sun's Java 2 HotSpot** technology;[11] from one run to the next the compiler might generate different code, alternately making different inlining decisions based on measured behavior). Although powerful, instruction traces are very lengthy, and in many cases it is not necessary to actually look at each and every instruction executed. It suffices to consider the trace at the basic block level, for example, and just count the number of instructions executed within the block. Even with this simplification, the volumes of data are very high. The arcflow model provides a means to reduce these data in a common way and yet retain the information that is most important.

Table 5 Callstack sampling

Where Collected	Data Collected	Type of Data	How Used
Periodically (e.g., every .01 secs)	Complete thread callstack	Sampled	To identify methods consuming most CPU time

Performance. The run-time cost of implementing Step 1 of the arcflow model, collecting data, is directly proportional to the density of events recorded. Generating an event and notifying the profiler consume a significant number of processor cycles, so if events occur frequently, the time to run the application being analyzed can be greatly distorted. In our experience, Java methods tend to be small, consuming on average far fewer cycles than event generation and recording. Creating a method entry and exit trace can dilate the run time of a Java program by 20 times or even more. We also have to keep in mind that this distortion is not uniform. Because the cost of each entry and exit event is approximately the same, the apparent increase in time spent in small methods is greater than the increase in large methods. We can partially compensate for this dilation by measuring the average time required for each event and by subtracting this time from the apparent time between each method entry and matching exit, but this compensation is inexact. However, even with the worst distortion of time, the structure of the call-tree is unaffected, and the frequency of events is accurate. These alone are often of great value to analysts.

Compensating for the dilation can be more accurate if the metric of interest is machine instructions instead of cycles, because the number of instructions to generate and record an event is more nearly constant than time. If the metric is Java bytecode interpreted, no dilation occurs, and no compensation is needed. The metric of time spent between events represents a worst case for the accuracy of our implementation.

Note that nearly all the run-time cost of our implementation is in Step 1, generating an event and notifying the profiler. The cost of recording the event in an in-memory call-tree is negligible in comparison. The postprocessing step, generating the x-file reports, usually takes a minute or less.

Scalability. Because all our experience so far is with small- and medium-sized applications and benchmarks, it is natural to ask if the arcflow model will scale to very large programs. We are confident that it will for several reasons.

As mentioned earlier, most programs reach a steady state with respect to their call-tree in which they have executed all or nearly all their distinct callstack configurations and after which their call-tree grows slowly if at all. If the call-tree of an application does continue to grow, or is simply too large, we can resort to sampling callstacks as described previously. Sampling works best on long-running programs, which is exactly the class for which scalability is most likely to be an issue.

To make reading tree reports easier, we have developed and routinely use tools that prune trees and other arcflow reports. By the definition of our cum metric, all the nodes between a given node and the root are guaranteed to have a cum value greater than or equal to that of the given node. Therefore, eliminating all nodes with cum less than any value N will trim only leaves and branches from a tree, leaving a coherent well-structured tree. A performance analyst can vary N until he or she has a tree small enough to be tractable while retaining the interesting, performance-intensive parts.

Future work

We are exploring many extensions. The current model deals very well with a single process and its (potentially) many threads. However, we are increasingly interested in the analysis of distributed Java applications (e.g., those that communicate with the Java remote method interface, or RMI). Such applications require that instrumentation be applied in multiple Jvms. See Reference 12 for recent work in this area with RMI for client/server work. Given the value of the arcflow model to single-system analysis, we believe it will extend well into helping provide organization and insight into the performance of multiple distributed Jvms.

The general model introduced earlier in the paper (AF0) provides for consumption data for multiple resources to be recorded. At present, our implementations, based on AF1, do not explore this aspect of the model; instead we conduct our measurements with one metric enabled, then repeat with another metric that may be of interest. Unfortunately, this method loses any information about correlated behavior between consumers and multiple resources. As the general model indicates, it need not be the case. More work is required in developing simple, yet effective, reporting extensions to deal with multivariate data from the arcflow model.

The current model deals cleanly with homogeneous data, e.g., data collected from all entry and exit events or data collected from samples. Because of the tradeoffs associated with different types of data, and because experience suggests that a middle ground is sometimes the best approach, we feel it is important to ensure that the arcflow model deals well with data of mixed type. For example, a set of samples of method entry and exit events, each one having, say, 10000 events would provide the benefits of deep insight into program structure—yet not cost much more than a flat sample providing no such structural information. However, integrating such data into the arcflow model will require care and some clever searching and pattern matching.

Regular users of arcflow output soon become adept at reading xtree and xarc files, especially pruned ones. Still, there is no denying the value of graphical views and navigational aids. We believe that interactive graphical interfaces that allow users to view callstack trees graphically, navigate them, and cross-reference the various x-files would enhance the value of arcflow. We are considering various designs for such tools.

Conclusion

We have presented arcflow, a general model for the recording and reporting of performance data. This model has proven extremely powerful in simplifying

Figure 8 Java implementation of arcflow

```
class Node
/*
 * Implements simple, non thread-safe, call-stack based on Start/Stop consumption events.
 *
 */
{
 /* These are the call-tree variables for each node */
 public int occurences;
 public int total;
 public String consumer_id;
 private Node parent;
 private Hashtable children;

 /* an instance variable used to determine resource consumption deltas from Start-to-end */
 private int entry_stats;

 /* The "current" node; a cursor indicating where we are in the call-tree */
 private static Node current;

 /* the root of the callstack tree */
 private static Node base;

 /*
  * we start with a base root node.
  */
 static  {
        current = new Node("***Root***");
        base = current;
 }

 /*
  *constructor for a consumer id node.
  */
 public Node(String name)  {
          consumer_id = name;
          children = new Hashtable( );
 }

 /*
  * Determines if a node with the consumer id (childname) already exists as the child of
  * the current node. If it does not, then a new node is created and adopted as a child
  * of current. This new node is then returned.
  *
  * If an appropriately named child node already exists, then the node is returned
  */
 public Node adoptChild (String childname)  {
          if (!children.containsKey(childname))  {
                    Node child = new Node(childname);
                    child.parent = this;
                    children.put(child.consumer_id,child);
                    return (child);
          }
          else return( (Node)children.get(childname) );
 }

 /* non-thread aware mechanism to build call-tree.
  *
  *Called on start of a resource consumption.
  *Arguments are: method name and value of metric at entry.
  */
 public static void Enter(String id, int entry_val)  {
          current = current.adoptChild(id);
          current.entry_stats = entry_val;
 }

 /*
  * Called on end of a resource consumption.
  * Arguments are: method name and value of metric at exit.
  */
 public static void Exit(String id, int exit_val)  {
          int delta = exit_val-current.entry_stats;
          current.occurrences++;
          current.total+=delta;
          current = current.parent;

 }

 }
```

the way in which we look at the highly varied performance data from the Java virtual machine implementations on several IBM platforms. Like many good things, it is simple. It has served to extend our vocabulary and enhance our internal communications about performance. Finally, it has served to facilitate cross-platform comparisons and leverage cross-platform enhancements that heretofore would not have been feasible. We believe that there are many valuable applications of this approach. Although we realize we are not unique in viewing data in this way, we feel that the broad applicability with which we have applied these techniques and the broad benefits we have received from their application are novel.

Acknowledgments

We are grateful to Honesty Young at the IBM Almaden Research Center for first introducing us to the xarc report concept. Arcflow has been implemented in many different contexts by many individuals, including Will Cain and Ron Edmark. We wish to express our thanks to all of our colleagues who have contributed to the arcflow concept by using it and providing feedback. We are particularly grateful to P. J. Kilpatrick for providing us with a supportive environment rich with challenges, exciting ideas, superlative technical talent, and the resources to make a difference.

Appendix: Constructing the arcflow model from entry and exit event data

Assume that events are reported at the start and end of a resource consumption. The implementation shown in Figure 8 records these events directly into a call-tree structure. This implementation is in Java, though an actual implementation need not be. In this implementation, there is a single class, Node. Objects of class Node are created for each unique consumer ID in context. (To simplify the presentation, the code is not thread-safe; our actual implementation runs correctly in a multithreaded environment.)

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc., Intel Corporation, Microsoft Corporation, Netscape Communications Corporation, or Bluestone Software, Inc.

Cited references and note

1. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley Publishing Co., Reading, MA (1997).
2. F. Waters, *AIX Performance Tuning Guide*, Prentice Hall PTR, Upper Saddle River, NJ (1995), (c) 1995 IBM Corporation.
3. S. L. Graham, P. B. Kessler, and M. K. McKusick, "Gprof: A Call Graph Execution Profiler," *ACM SIGPLAN Notices* **17**, No. 6, 120–126 (June 1982).
4. VTune Performance Analyzer, Intel Corporation, Santa Clara, CA, http://developer.intel.com/VTune/analyzer/index.htm.
5. Microsoft Windows NT Workstation Resource Kit, Microsoft Corporation, Redmond, WA (October 1996).
6. R. J. Hall and A. J. Goldberg, "Call Path Profiling of Monotonic Program Resources in UNIX," *1993 Summer USENIX* (June 1993), pp. 1–13.
7. G. Ammons, T. Ball, and J. R. Larus, "Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling," *Proceedings of ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI)*, Las Vegas, NV (June 15–18, 1997), pp. 85–96.
8. S. J. Baylor, M. Devarakonda, S. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, R. Arora, R. Dimpsey, and S. J. Munroe, "Java Server Benchmarks," *IBM Systems Journal* **39**, No. 1, 57–81 (2000, this issue).
9. D. Viswanathan and S. Liang, "Java Virtual Machine Profiler Interface," *IBM Systems Journal* **39**, No. 1, 82–95 (2000, this issue).
10. In the Jvm environment there are some complexities in implementing this approach. Callstacks are a combination of native, JIT-compiled, and interpreted code. It is beyond the scope of this paper to discuss resolution to this in detail. Note that in general, sampling profilers operate on a system-wide basis. It may not be possible for the interrupting thread to interrogate the interrupted thread and obtain its complete callstack without additional Jvm support.
11. *The Java HotSpot Performance Engine Architecture*, White Paper, Sun Microsystems, Inc., Palo Alto, CA (April 1999), http://java.sun.com/products/hotspot/whitepaper.html.
12. I. H. Kazi, D. P. Jose, B. Ben-Hamida, C. J. Hescott, C. Kwok, J. Konstan, D. J. Lilja, and P.-C. Yew, "JaViz: A Client/Server Java Profiling Tool," *IBM Systems Journal* **39**, No. 1, 96–117 (2000, this issue).

Accepted for publication September 20, 1999.

William P. Alexander *IBM Network Computing Software Division, 11400 Burnet Road, Austin, Texas 78758 (electronic mail: balexand@us.ibm.com).* Dr. Alexander has a B.A. in philosophy from Rice University and a Ph.D. in computer science from the University of Texas at Austin. He has taught computer science at Boston University, worked as a performance analyst in the computing center at Los Alamos National Laboratory, and helped design parallel database and distributed transaction systems at the Microelectronics and Computer Technology Corp. (MCC) research consortium. He joined IBM in 1991 where he has worked on the performance of numerous hardware and software platforms.

Robert F. Berry *IBM Network Computing Software Division, 11400 Burnet Road, Austin, Texas 78758 (electronic mail: brobert@us.ibm.com).* Dr. Berry joined IBM in the Research Division at the Thomas J. Watson Research Center in 1987 where

he worked in the systems performance management area. In 1992 he transferred to the RS/6000 Division in Austin where he led the team developing performance tools for the AIX operating system. Subsequently, he joined the Personal Systems Products Division and worked on performance instrumentation, tools, and analysis for the OS/2 and Warp Server products. Most recently he has led the team responsible for developing performance instrumentation and tools for the Intel platforms for IBM's Java Developer Kits, including OS/2, Windows NT, JavaOS, and Linux. Dr. Berry received his Ph.D. in computer sciences from the University of Texas at Austin in 1983. He was elected to the IBM Academy of Technology in 1999.

Frank E. Levine *IBM Network Computing Software Division, 11400 Burnet Road, Austin, Texas 78758 (electronic mail: levinef@us.ibm.com).* Mr. Levine received a B.S. in mathematics from Tufts University in 1970 and an M.S. in mathematics from Purdue University in 1972. He continued taking graduate courses in mathematics and computer science until 1974, when he joined the former IBM Federal Systems Division where he was a programmer for various software components for the ground support system for the Space Shuttle at Cape Kennedy, Florida. In 1979, he moved to Austin, Texas, where he was a lead programmer on various software development projects, including IBM DisplayWriter, DisplayWrite, and OS/2 EE Data Base Manager. In 1989, Mr. Levine became a software development program manager for AIX RS/6000® development projects. In 1992, he joined the PowerPC™ System Architecture Department and coauthored *RISC System/6000 PowerPC System Architecture*, published by Morgan Kaufmann Publishers, Inc. In 1995, he developed the Performance Monitor application programming interface, which was completed in 1997 and made available for a short time to users outside of IBM. In 1997, he joined the Performance Tools and Instrumentation team in NCSD, where he is currently the lead member of a tools team for development of performance tools for various hardware and software platforms with an emphasis on Java.

Robert J. Urquhart *IBM Network Computing Software Division, 11400 Burnet Road, Austin, Texas 78758 (email: urquhart@us.ibm.com).* Dr. Urquhart joined IBM in the former Federal Systems Division in 1956. In the 1950s he designed and implemented the first real-time I/O processor used in FSD's B58 Digital Navigation System. In the 1960s he was the lead programmer for the IBM Gemini Spacecraft Rendezvous and Reentry Software and made key contributions to IBM's first space application, the Orbital Astronomical Observatory (OAO). In the 1970s he was lead analyst on the U.S. Navy's Verdin ULF & VLF Nuclear Submarine Communications System. His signal processing tools package was used widely within IBM and the San Diego Naval Lab. In the 1980s he was a lead programmer for the DisplayWriter Spelling Verification System and participated in development of the Lexis Spelling Verification and Correction technology used by PROFS® on the VM operating system. In the 1990s he worked on performance and tools for the RS/6000 AIX system and invented the "tprof" profiler that is widely used within IBM and by IBM customers. He also developed the AIX mtrace postprocessor, another corporate wide-trace reduction tool. Dr. Urquhart also developed a wide range of performance tools for OS/2, Windows NT, and the Java language, receiving an IBM Corporate Award for that work, which included several versions of the arcflow tool. His degrees include a B.S. in electrical engineering from Lawrence Institute of Technology, an M.S. in electrical engineering from Syracuse University, an M.S. in mathematics from the University of Michigan, a Ph.D. in electrical engineering from the University of Michigan, and an M.S.S.W. from the University of Texas. He has earned 12 invention levels and is currently developing Java memory analysis tools as well as being a consultant in the performance area.

Reprint Order No. G321-5719.