Performance Analysis of Idle Programs

Erik Altman  Matthew Arnold  Stephen Fink  Nick Mitchell
IBM T.J. Watson Research Center
{ealtman, marnold, sfink, nickm}@us.ibm.com

Abstract
This paper presents an approach for performance analysis of modern enterprise-class server applications. In our experience, performance bottlenecks in these applications differ qualitatively from bottlenecks in smaller, stand-alone systems. Small applications and benchmarks often suffer from CPU-intensive hot spots. In contrast, enterprise-class multi-tier applications often suffer from problems that manifest not as hot spots, but as idle time indicating a lack of forward motion. Many factors can contribute to undesirable idle time, including locking problems, excessive system-level activities like garbage collection, various resource constraints, and problems driving load.

We present the design and methodology for WAIT, a tool to diagnosis the root cause of idle time in server applications. Given lightweight samples of Java activity on a single tier, the tool can often pinpoint the primary bottleneck on a multi-tier system. The methodology centers on an informative abstraction of the states of idleness observed in a running program. This abstraction allows the tool to distinguish, for example, between hold-ups on a database machine, insufficient load, lock contention in application code, and a conventional bottleneck due to a hot method. To compute the abstraction, we present a simple expert system based on an extensible set of declarative rules.

WAIT can be deployed on the fly, without modifying or even restarting the application. Many groups in IBM have applied the tool to diagnosis performance problems in commercial systems, and we present a number of examples as case studies.

Categories and Subject Descriptors Software [Software Engineering]: Metrics, Performance Measures

General Terms Performance

Keywords bottlenecks, performance analysis, multi-tier server applications, idle time

1. Introduction

Shun idleness. It is a rust that attaches itself to the most brilliant metals. Voltaire.

This paper addresses the challenges of performance analysis for modern enterprise-class server applications. These systems run across multiple physical tiers, and their software comprises many components from different vendors and middleware stacks. Many of these applications support a high degree of concurrency, serving thousands or even millions of concurrent user requests. They support rich and frequent interactions with other systems, with no intervening human think time. Many server applications manipulate large data sets, requiring substantial network and disk infrastructure to support bandwidth requirements.

With these requirements and complexities, such applications face untold difficulties when attempting to scale for heavy production loads. In our experience with dozens of industrial applications, every individual deployment introduces a unique set of challenges, due to issues specific to a particular configuration. Any change to key configuration parameters, such as machine topology, application parameters, code versions, and load characteristics, can cause severe performance problems due to unanticipated interactions.

Part of the challenge arises from the sheer diversity of potential pitfalls. Even a single process can suffer from any number of bottlenecks, including concurrency issues from thread locking behavior, excessive garbage collection load due to temporary object churn [9, 21, 24], and saturating the machine’s memory bandwidth [24]. Any of these problems may appear as a serialization bottleneck in that the application fails to use multiple threads effectively; however, one must drill down further to find the root cause. Other problems can arise from limited capacity of physical resources including disk I/O and network links. A load balancer may not effectively distribute load to application clones. When performance testing, testers often encounter problems generating load effectively. In such cases, the primary bottleneck may be processor or memory saturation on a remote node, outside the system-under-test.

Furthermore, many profiling and performance understanding tools are inappropriate for commercial server environments. Many tools [2, 3, 5, 6, 11, 13, 14, 16, 18, 19] rely on restarting or instrumenting an application, which is often forbidden in commercial deployment environments. Similarly, many organizations will not deploy any unapproved monitoring agents, nor tolerate any significant perturbation of the running system. In practice, diagnosing performance problems under such constraints resembles detective work, where the analyst must piece together clues from incomplete information.

Addressing performance analysis under these constraints, we present the design and methodology of a tool called WAIT. WAIT’s methodology centers on Idle Time Analysis: rather than analyze what an application is doing, the analysis focuses on explaining idle time. Specifically, the tool tries to determine the root cause that leads to under-utilized processors. Additionally, the tool must present information in a way that is easily consumable, and operate under restrictions typical of production deployment scenarios.

The main contributions of this paper include:

• a hierarchical abstraction of execution state: We present a novel abstraction of concrete execution states, which provides a hier-
arch of abstract states corresponding to different sources of idle time. In contrast to traditional profiling tools, we present a high-level characterization of application behavior computed by an expert system. The expert system is codified by a set of declarative rules, which a practitioner can easily customize.

- a methodology to infer behavior based on ubiquitous sampling mechanisms. We show that a tool can analyze performance effectively based on lightweight, non-intrusive sampling information available by default from standard Java Virtual Machines and operating systems. Notably, the end user need not restart, recompile, or otherwise modify the running application. In our experience, this feature is crucial for wide-scale adoption of a tool in this space, and rules out most previous approaches described in the literature.

We also present a number of case studies from real deployments illustrating how the methodology applies in practice for a number of different types of performance problems.

The remainder of this paper proceeds as follows. Section 2 presents a high-level overview of the methodology. Section 3 describes details of how the system collects monitoring data. Sections 4 and 5 describe the abstraction of idle states and the analysis that computes this abstraction. Section 6 discusses the user interface and implementation. Section 7 reviews case studies illustrating various examples from the field. Section 8 reviews related work, and Section 9 concludes.

2. Overview

We consider, as a motivating example, performance diagnosis for a Java Enterprise Edition (JEE) application. Figure 1 illustrates the typical structure of a JEE application. A JEE application server, running Java code, sits in the middle of several communicating tiers of machines; these tiers include clients, relational databases, and directory and caching services.

To understand performance of this application, WAIT uses a Hub Sampling approach. With the approach, we demonstrate how the analysis of only the Java tier can provide insight into bottlenecks in the system as a whole.

Hub Sampling To identify primary bottlenecks in Java-hub applications, we collect samples of processor utilization and samples of the state of the Java threads.

Production environments impose severe constraints on the types of monitoring and tools deemed acceptable. For example, code instrumentation is often a non-starter: many organizations simply will not rebuild an application with instrumentation, deploy a non-standard runtime system, or enable any non-trivial monitoring agent. Many organizations will not tolerate any observable performance overhead, except perhaps under limited and carefully controlled guidance. Additionally, many organizations will not tolerate large trace files, and will not allow any interactive access to the monitored systems.

To work within these constraints, we must rely on ubiquitous monitoring technology, without requiring instrumentation or non-trivial agents. We must also make do with a relatively small corpus sample-based monitor data, collected during a small window and processed offline.

Fortunately, most production JVMs provide built-in sampling mechanisms, whereby the JVM will respond to signals and dump relatively small core (“javacore”) files with data representing the current JVM state. Figure 2 illustrates the data that is readily available from most commercial JVMs, without requiring any changes to an application’s deployed configuration: the monitor graph, which specifies the ownership and queuing relationships between threads and monitors; thread stack samples; the conventional run state of each thread (i.e. Runnable, CondWait, Blocked, Parked); and a window of garbage collection events.

We have found that even infrequent samples, acquired once or twice per minute, can yield surprising insight into the primary system bottlenecks.

Idle Time Analysis Even with relatively infrequent samples, java-core dumps of JEE server applications can carry a tremendous volume of information. Consider simple stack samples: often stacks in a JEE application extend to several hundred stack frames spanning dozens of logical components from different vendors, as illustrated in Figure 4. Understanding the relevant information from even a single such stack requires a lot of work. Now consider that an application will typically have many dozens of threads performing various activities, and that to understand performance changes over time, we must inspect samples from at least several points in time.

In this common scenario, a tool can easily overwhelm a human with too much information.

The fundamental problem is that the profile data lacks abstraction: there are too many distinct concrete methods in play, and a user cannot easily digest profile information spanning thousands of methods. While an expert with experience and intuition can probably navigate the raw data and diagnosis a problem, this task is usually too difficult for mere mortals.

To address this problem, WAIT analyzes the sample data and produces an abstract model of the application behavior, designed to illuminate bottlenecks. The analysis uses a set of expert rules to infer a hierarchical categorization of the state of threads across time (c.f. [6, 7, 10, 23]). The rules depend on the participation of a thread in the monitor graph, and the names of methods on its call stack. The analysis machinery is exceedingly simple and runs quickly, relying primarily on simple pattern-matching and decision trees. However, the expert rules embody sophisticated understanding of various Java frameworks.

At its coarsest level, the analysis assigns each thread an abstract state called a Wait State. A thread’s Wait State specifies whether it is able to make forward progress, and if not, the nature of the hold up. Each state, such as “Blocked”, “Disk”, “GC”, and “Network”, represents a general class of delays, independent of application-
Figure 3. From information that includes monitor states and stack sample context inferences, WAIT infers a Wait State for every sample.

level details. In this way, the Wait States serve the same purpose as the conventional run states, but provide a richer semantics that helps identify bottlenecks. Figure 3 gives example inferences of Wait States. For the stack in Figure 4, the analysis would infer the Wait State “Network”, as the stack matches the pattern shown in the middle example of Figure 3.

For each stack frame, we compute an abstraction called a Category that represents the code or activity being performed by that method invocation. For example, a method invocation could indicate application-level activity such as Database Query, Client Communication, or JDBC Overhead. We further label each stack sample with a Primary Category which best characterizes the activity being performed by an entire stack at the sampled moment in time.

The abstract model of activity forms a hierarchy; a Wait State gives a coarse but meaningful abstraction of a class of behaviors. For more information, one can “drill down” through the model to see finer distinctions based on Primary Categories, stacks of Categories, and finally concrete stack samples without abstraction. This hierarchy provides a natural model for an intuitive user interface, which provides a high-level overview and the ability to drill down through layers of abstraction to pinpoint relevant details.

Tool We have implemented a tool based on the above abstraction, which is deployed as a service within IBM. The tool computes the abstraction described above based on a simple set of rules, defined declaratively by an expert based on knowledge of common methods in standard library and middleware stacks. We present some statistics and case studies that indicate the methodology is practical, and successfully identifies diverse sources of idle time.

3. Hub Sampling
WAIT relies on samples of processor activity and of the state of threads in a JVM. The system typically takes samples from the hub process (e.g., application server) of a multi-tier application, but can also collect data from any standard Java environment. Despite collecting no data from the other tiers, information from a hub process frequently illuminates multi-tier bottlenecks.

We next describe how WAIT collects information from a Java hub, and discuss challenges due to limitations in available data.

3.1 Sampling Mechanisms
We have found a low barrier to entry to be a first-order requirement for tools in this space. Many, if not most, potential users will reject any changes to deployment scripts, root permissions, kernel changes, specific software versions, or specialized monitoring agents. Instead, WAIT collects samples of processor utilization, process utilization, and snapshots of Java activity using built-in mechanisms that are available on nearly every deployed Java system. Table 1 summarizes the mechanisms by which the system collects data.

Figure 4. Part of a 105-deep call stack, from the DayTrader benchmark (a configuration of which is now part of the DaCapo Suite).
Upon receiving this, the JVM stops running threads, and then writes out the in-
formation specified in the filesystem implementation.

The system can also produce meaningful results with partial data. In practice, data sometimes arrives corrupted or prematurely terminated, due to a myriad of problems. For example, the target machine may run out of disk space while writing out data, the target JVM may have bugs in its data collection, or there may be simple user errors. If any of the sources of data described are incomplete, the system will produce the best possible analysis based on the data available.

**Processor Utilization** Most operating systems support non-intrusive processor utilization sampling. We attempt to collect time series of processor utilization at three levels of granularity: (1) for the whole machine, (2) for the process being monitored, and (3) for the individual threads within the process. For example, on UNIX platforms we use `vmstat` and `ps` to collect this data.

**Java Thread Activity** To monitor the state of Java threads, we rely on the support built into most commercial JVMs to dump “javacore” files. Our implementation currently supports (parses) the javacore format produced by IBM JVMs, and contains preliminary support for the HotSpot JVM. This data, originally intended to help diagnose process failures and deadlock, can be sampled by issuing a signal to a running JVM process. Upon receiving this signal, the JVM stops running threads, and then writes out the information specified in Figure 2. The JVM forces threads to quiesce using the same “safepoint” mechanism that is used by other standard JVM mechanisms, such as the garbage collector.

IBM JVMs can produce javacore samples with fairly low perturbation. For a large application with several hundred threads with deep call stacks, writing out a javacore file can pause the application for several hundred milliseconds. As long as samples occur infrequently, writing javacos has a small effect on throughput, but an unavoidable hit on latency for requests in flight at the moment of sample. Table 2 shows that samples taken once every 30 seconds result in a 2% slowdown. For server applications, where operations repeat indefinitely, perturbation can be dialed down as low as needed, at least concerning throughput.

When the hub of a multi-tier application spans multiple processes, possibly running on multiple machines, we currently choose one at random. In the future, we will tackle multi-process environ-
ments more generally, including cloud environments. However, we note that in many enterprise workloads, the bottlenecks facing all JVMs are similar, and thus our approach provides substantial benefit even now.

3.2 Difficulties with Thread States

The run states provided by the JVM and operating system are often inconsistent or imprecise, due to several complications. The first problem is that many JVM implementations quiesce threads at safe-points before dumping the javacore. Threads that are already quiesced (e.g., waiting to acquire a monitor) will be reported correctly as having a conventional run state of Blocked. However, any thread that was Runnable before triggering the dump will be reported to have a false run state of CondWait, since the thread was stopped by the JVM before writing the javacore file.

The boundary between the JVM and the operating system introduces further difficulties with thread run states. The JVM and OS each track the run state of a thread. The JVM may think a thread is Blocked, while the OS reports the same thread Runnable, in the midst of executing a spinlock. Spinning is sometimes a detail outside the JVM’s jurisdiction, implemented in a native library called by the JVM. Similarly, the JVM may report a thread in a CondWait state, even though the thread is executing system code such as copying data out of network buffers or traversing directory entries in the filesystem implementation.

Even if conventional run states were perfectly accurate, they often help little in diagnosing the nature of a bottleneck. Consider the conventional CondWait run state. One such thread may be waiting at a join point, in a fork-join style of parallelism. Another thread, with the same CondWait run state, may be waiting for data from a remote source, such as a database. A third such thread may be a worker thread, idle only for want of work.

For these reasons, we instead compute on a richer thread state abstraction that distinguishes between these different types of states.

4. A Hierarchical Abstraction of Execution State

WAIT’s analysis maps concrete program execution states into an abstract model, designed to illuminate root causes of idle time. In this Section, we describe the concepts in the abstraction. Section 5 later describes how the analyzer computes the abstraction, and how details of the abstraction hierarchy arise from a declarative specification.

The analysis maps each sampled thread into an abstract state, which consists of a pair of two elements called the **Wait State** and a stack of **Categories**. A **Wait State** encapsulates the status of a thread regarding its potential to make forward progress, while the **a Category** represents the code or activity being performed by a particular method invocation.

We next describe the hierarchical abstraction in more detail.

4.1 The Wait State Abstraction

The **Wait State** abstraction groups thread samples, assigning each sample a label representing common cases of forward progress (or the lack thereof). Figure 5(a) shows the the hierarchy of Wait States which cover all possible concrete thread states. The analysis maps each concrete thread sample into exactly one node in the tree.

The hierarchy has proven to be stable over time; we have not needed to add additional states beyond those shown in the hierarchy of Figure 5(a), even as usage of the WAIT tool has expanded to included diverse workloads.

At the coarsest level, the Wait State of a sampled thread indicates whether that thread is currently held up or making forward progress: Java threads may be either Waiting or Runnable. A third

<table>
<thead>
<tr>
<th>data</th>
<th>UNIX</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>machine utilization</td>
<td>vmstat</td>
<td>typeperf</td>
</tr>
<tr>
<td>process utilization</td>
<td>ps</td>
<td>tasklist</td>
</tr>
<tr>
<td>Java state</td>
<td>kill -3</td>
<td>sendsignal</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sampling interval (seconds)</th>
<th>throughput slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>59%</td>
</tr>
<tr>
<td>5</td>
<td>24%</td>
</tr>
<tr>
<td>10</td>
<td>13%</td>
</tr>
<tr>
<td>20</td>
<td>8%</td>
</tr>
<tr>
<td>30</td>
<td>2%</td>
</tr>
<tr>
<td>1000</td>
<td>unmeasurable</td>
</tr>
</tbody>
</table>

**Table 1.** The built-in mechanisms we use to sample the Java hub. Note that `kill -3` does not terminate the signaled process, and 3 is the numeric code for SIGQUIT.

**Table 2.** Throughput perturbation versus sampling interval, from a document processing system running IBM’s 1.5 JVM.

---

1 There are some nominal differences between this format and that provided by a Sun JVM. Also, Sun provides a finer granularity for data acquisition, via the `jstat`, `jstat`, and related family of commands.
Figure 5. Every concrete state, a sampled thread, maps to a pair of abstract states: a Wait State and a stack of Categories (one per frame). Every frame in a call stack has a default name based on the invoked package; e.g. `com.MyBank.login()` would be named `MyBank Code`.

For Java threads, the analysis partitions Waiting and Runnable into finer abstractions, which convey more information regarding sources of idle time. For example, a Waiting thread might be waiting for data from some source (Awaiting Data), blocked on lock contention (Contention), or have put itself to sleep (Sleeping). As shown in the Figure, finer distinctions are also possible. Consider a Sleeping thread: this could be part of a polling loop that contains a call to `Thread.sleep (Poll)`, or the join in a fork-join style of parallelism (Join); or it could be that the thread is a worker in a pool of threads, and is waiting for new work to arrive in the work queue (Awaiting Notification).

We claim that in many cases, distinctions in Wait States give a good first approximation of common sources of idle time in server applications. Furthermore, we claim that differences in Wait States indicate fundamentally different types of problems that lead to idle time. A server application suffering from low throughput due to insufficient load would have many threads in the Awaiting Notification state. The solution to this problem might, for example, be to tune the load balancer. A system that improperly uses `Thread.sleep()` suffers from a problem of a completely different nature. Similarly, having a preponderance of threads waiting for data from a database has radically different implications on the system health than many threads, also idle, suffering from lock contention.

Thus, the Wait State gives a high-level description of the root cause of idle time in an application. The second part of the abstraction, the Category stack, gives a finer abstraction for pinpointing root causes.

4.2 The Category Abstraction

The Category abstraction assigns each stack frame a label representing the code or activity being performed by that method invocation. Category names provide a convenient abstraction that summarizes nested method invocations that implement larger units of functionality [10].

Note that since each stack frame maps to a Category, each stack will contain representatives from several Categories. To understand behavior of many stacks at a glance, it is useful to assign each stack a primary Category, which represents the Category which provides the best high-level characterization of the activity of the entire stack. For example, in Figure 4, the JDBC Category would be chosen as the primary Category, based on priority logic that determines that the JDBC Category label conveys more information.
than other Categories in the stack, such as Networking or WebContainer.

Figure 5(b) shows a subset of the Categories that we currently model. As with Wait States, the Category abstract states form a tree. The primary distinction is drawn between activities, which name what a method is doing, and nicknames for common libraries and frameworks. Common activities include sorting and marshalling data, such as occurs in the handling of the XML data of SOAP requests. It is common for server applications to have dozens of administrative (Admin, in the figure) activities. These activities include background logging threads, cache eviction threads, and alarm threads that periodically probe for changes of files stored on disk.

Since the Category abstraction in Figure 5(b) reflects activities in well-known software components, the Category abstraction is bound to be incomplete with respect to non-framework application code. This incompleteness stands in contrast to the Wait State abstraction in Figure 5(a) which describes more general conditions of all systems. Since the Category abstraction is incomplete, it is crucial to have a fallback Category assignment, in the case of insufficient coverage of Category names. We have found that the method’s package fills this coverage hole effectively, and as such, we assign the package as the “Code Nickname”, shown at the upper right of Figure 5(b).

Section 5 illustrates how a practitioner can define new Category abstractions declaratively, in the specification of the analysis expert rules system.

Figure 6 gives an example call stack, and the corresponding Category stack. For example, a call to the SocketRead0 method belongs to the code that has been named Network. The com.mybank code has no nickname, nor known activity, and so is assigned its default name MyBank Code. The primary Category of this call stack is the highlighted Database Communication activity.

Declarative rules (described shortly) indicate priorities used to choose the primary label for a stack. Sometimes the appropriate choice of priorities varies depending on who views the report. For example in Figure 6, if a network analyst were the primary viewer of the output, Network might be a better choice as the primary Category.

### 4.3 Wait State Analysis Definition

Having informally introduced the abstraction, we can now state more precisely the analysis performed by WAIT:

**Definition 1 (Wait State Analysis).** Let \( k \) be the maximum stack depth of sampled threads, \( W \) be the tree of Wait States and \( C \) be the tree of Categories. We define a wait state analysis as a function that maps each sampled concrete thread state to an abstract state \((w,c) \in W \times C^k\). We say that \( w \) is the Wait State of a sampled thread, and \( c \), its Category Stack, is a tuple whose components correspond to frames in the sampled call stack.

**Definition 2 (Category Priorities and Primary Category).** Let the call stack of a sampled thread contain methods \( m_1, \ldots, m_k \), and the output of a wait state analysis be \((w,c)\), where the Categories are \( c = [c_1, \ldots, c_k] \). Each element of the Category mapping, \( m_i \rightarrow c_i \), has a priority \( p \). The primary Category used by the sampled thread is that \( c_i \) with maximum priority, and, in the case of ties, the one closest to the leaf invocation \( m_1 \).

The abstract model provides a backbone for progressive disclosure of details regarding thread activity, at a sampled moment in time. In practice, we have found that it is most useful to start by clustering stack samples according to Wait State. A user request to focus on particular Wait States results in the navigation to a view that clusters the stack samples in that Wait State according to their primary Category. In this way, navigation of a user interface corresponds directly to traversals of the tree-structured abstract model. Section 6.2 describes the user interface in more detail.

### 5. Analyzer

We now describe a relatively simple engine that computes the analysis just defined. The analysis engine has three steps: 1) parse and transform the raw data, 2) infer Categories, and then 3) infer Wait States.

#### 5.1 ETL Step

As described earlier, WAIT takes input that includes raw javacore samples and raw output of machine utilization utilities. A pre-pass to the analysis performs an Extract-Transform-Load (ETL) step that parses the raw data and transforms it to a canonical form, which abstracts away irrelevant details that vary from platform to platform.

Figure 7 provides a more detailed summary of the raw data available in a javacore thread sample, including the name of thread sampled, its native thread identifier (e.g. on UNIX, this turns out to be the address of the corresponding pthread data structure), the call stack as a list of method names, and information about how this thread interacts with the monitors.

The ETL step consumes this data and produces the data model shown in Figure 8. First, we compute equivalence classes of thread samples, where two thread samples are considered equivalent if they represent the same stack of methods and locking status. The Figure labels these equivalence classes as **call stack clusters**.

The output model represents the data in a tabular fashion, similar to a relational database. Viewing the **count row** of the **call stack**
Figure 8 reveals that cluster \( c_1 \) occurred 3592 times across all application samples. Viewing table blocked in Figure 8 indicates that in the first and last application samples, cluster \( c_1 \) was waiting in the critical section guarded by monitor \( m_2 \). Viewing the owned by table in turn reveals that \( m_2 \) was owned by cluster \( c_2 \). In other words, thread stack \( c_1 \) was blocked on a monitor held by thread stack \( c_2 \).

5.2 Category Analysis

WAIT relies on a simple pattern-matching system, driven by a set of rules characterizing well-known methods and packages, to determine the Category label for each stack frame. The system relies on a simple declarative specification of textual patterns that define Categories.

The declarative rules that define the Category analysis define two models. The first model is a Category Tree, such as the one shown in 5(b). A Category Tree provides the namespace, inheritance structure, and prioritization of the Category abstractions that are available as method labels. The second model is a set of rules. Each rule maps a regular expression over method names to a node in the Category Tree. For example, Figure 9 shows rules that, in part, define Database activity. The rules distinguish between five aspects of database activity: queries, batch queries, commits, rollbacks, iteration over result sets. This example illustrates how it is easy to define a Category Tree that is more precise than the one shown in 5(b).

Given these rules, the Category analysis is simple and straightforward. The analysis engine iterates over every frame of every call stack cluster, looking for the highest-priority rule that matches each frame. As described in Section 4, every method has an implicit Category, its package which is assigned to the Category's Code Nickname. Thus, if no Category rule applies to a frame, then we form an ad hoc Category for that frame: a method: P1/P2/P3/.../Class.Method receives the Code Nickname P2 Code.

5.3 Wait State Analysis

In addition to inferring Categories, WAIT infers a Wait State as illustrated in Figure 5(a). Our analysis to infer Wait States combines three sources of information: processor utilization, the concrete data model previously presented in Figure 8, and rules based on method names. The rules over method names, in some cases, require the inspection of multiple frames in stack. This differs from the Category analysis, where each frame’s Category is independent of other frames.

The main challenge in using method names to infer a Wait State concerns handling imperfect knowledge of an application’s state. As discussed in Section 3.2, the true Wait State of a sampled thread is, in many cases, not knowable. To fill this knowledge gap, we use expert knowledge about the meaning of activities, based on method names. Fortunately, many aspects of Wait States depend on the meaning of native methods, and the use of native methods does not vary greatly from application to application. The design of Java seems actively to discourage the use of native methods. Nevertheless our conclusions are always subject to the inevitable of Java seems actively to discourage the use of native methods.

Nevertheless our conclusions are always subject to the inevitable of Java seems actively to discourage the use of native methods. The design of Java seems actively to discourage the use of native methods.

Rules that define aspects of the Database Category.

\[
\text{com/mysql/jdbc/ConnectionImpl.commit} \Rightarrow \text{Database Commit} \\
\ldots/\text{ITransactionWrapper.rollback} \Rightarrow \text{Database Rollback} \\
\ldots/\text{SQLServerStatement.doExecuteStatement} \Rightarrow \text{Database Query} \\
\ldots/\text{WSJdbcStatement.executeBatch} \Rightarrow \text{Database Batch} \\
\ldots/\text{OracleResultSetImpl.next} \Rightarrow \text{Database Cursor}
\]

The Wait State of a given call stack cluster \( c \) at sample index \( i \) is the first match found when traversing the following conditions, in order:

1. Deadlock: if this stack cluster participates in a cycle of lock contention in the monitor graph; i.e., there is a cycle in the Blocked and Owned By relations.
2. Lock Contention: if the stack cluster, at the moment in time of sample \( i \), has an entry in the Blocked relation.
3. Awaiting Notification: if the stack cluster, at the moment in time of sample \( i \), has an entry in the Waiting relation.
4. Spinlock: if the Wait State rule set defines a method in \( c \) that with high certainty, implies the use of spinlocking. Many methods in the java.util.concurrent library fall in this high-certainty category.
5. Awaiting Data from Disk, Network: if the rule set matches \( c \) as a use of a filesystem or a network interface. Most such rules need only inspect the leaf invocation of the stack, e.g. a socketRead native invocation is a very good indication that this stack sample cluster is awaiting data from the network. In some cases, requests for data are dispatched to a “stub/tic” method, as is common in LDAP or ORB implementations.
6. Executing Java Code: if the method invoked by the top of the stack is not a native method, then \( c \) is assumed to be Runnable, executing Java code.
7. Executing Native Code: if the method invoked by the top of the stack is a native method, and the rule set asserts that this native method is truly running, then we infer that the native method is Runnable. We treat native and Java invocations asymmetrically, to increase robustness. A Java method, unless it participates in the monitor graph, is almost certain to be Runnable. The same cannot be said of native methods. In our experience, native methods more often than not, serve the role of fetching data, rather than executing code. Therefore, we require native methods to be whitelisted in order to be considered Runnable.
8. JVM Services: if \( c \) has no call stack, it is assumed to be executing native services. Any compilation and Garbage Collection threads, spawned by the JVM itself, fall into this category. Even though these call stack samples have no call stacks, and unreliable thread states, they participate in the monitor graph. Thus, unless they are observed to be in a Contention or Awaiting Notification state, we assume they are Runnable, executing JVM Services.
9. Pull, IOWait, Join Point: if there exists a rule that describes the native method at the top of the stack as one of these variants of Sleeping.
10. NativeUnknown: any call stack cluster with a native method at the top of the stack and not otherwise classified is placed into the NativeUnknown Wait State. This classification is in contrast to call stack clusters with Java leaf invocations, which are assumed to be Runnable. For robustness, the algorithm requires call stacks with native leaf invocations to be specified by rules to be in some particular Wait State. This allows tool users to quickly spot deficiencies in the rule set. In practice, we

This is likely, but not an absolute certainty. The JVM and operating system spend some time copying network buffers, etc., as part of fetching remote data. We are currently working to incorporate native stacks into the rule set, will allow for a more refined Wait State inference.
Figure 10. Some example Wait State rules. (a) Simple rules specify a Wait State (e.g., Network) or an auxiliary tag (e.g.,%CondVar) based on a single method frame. (b) Complex rules can use conjunctions of antecedents to match stacks which satisfy multiple conditions at different layers in the stack.

Table 3. The 76 Wait State rules, grouped according to Wait State. Only 5 of these rules use semantics from outside the Standard Library and Apache Commons.

<table>
<thead>
<tr>
<th>Wait State Rule</th>
<th># Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waiting on Condition Variable</td>
<td>26</td>
</tr>
<tr>
<td>Native Runnable</td>
<td>22</td>
</tr>
<tr>
<td>Awaiting Data from Disk,Network</td>
<td>16</td>
</tr>
<tr>
<td>Spinlock</td>
<td>12</td>
</tr>
</tbody>
</table>

Rules for Wait States The syntax for declaring Wait State rules is more general than that for Category rules, which depend on exactly one method name. In particular, rules can specify antecedents which depend on a conjunction of frame patterns appearing together in a single stack, as illustrated in Figure 10.

For convenience, the rules engine allows a declarative specification of tags, which are auxiliary labels which can be attached to a stack to encode information consumed by other rules which produce Wait States. Figure 10(b) shows an example that matches against two frames, and relies on an auxiliary tag (%CondVar) which labels various manifestations of waiting on a condition variable.

5.4 Rule Coverage

A key hypothesis in any rule-based system is that a stable, and hopefully small, set of rules can achieve good coverage on range of diverse inputs. This section evaluates this hypothesis for the WAIT rule-based analysis.

WAIT has been live within IBM for 10 months, and over that time has received over 1,200 submissions, 830 of which were submitted by 151 unique users who were not involved in any way with WAIT development. The submissions came from multiple divisions within IBM and cover a wide range of Java applications, including document processing systems, e-commerce systems, business analytics, software configuration and management, mashups, object caches, and collaboration services. To handle these reports we have encoded a total of 76 Wait State rules.

This set of rules has proven to be extremely stable. When our tool sees code from a new framework, these rules must change only to the extent that the framework interacts with the world outside of Java in new ways. In practice, we rarely see new flavors of crossings between Java application code and the native environment. In the case of a new cryptographic library, or a new in-Java caching framework, none of these Wait State rules need change.

For the Category analysis we have found that only a small number of rules are necessary to capture a wide range of Categories. Table 4a characterizes most of the 387 Category rules that we have currently defined. For example, our current rule set covers five common JDBC libraries, including IBM DB2 and Microsoft SqlServer, with only 72 rules. The number of rules specific to a particular JDBC implementation lies on the order of 10–20, as shown in Table 4b. We have also found the rules to be stable across versions of any one implementation. For example, the same set of rules covers all known versions and platforms of the DB2 JDBC driver that we have tested. This testing includes three versions of the code and four platforms.

It is difficult to prove that the existing rules are “sufficient” other than to observe the eventual success of the tool within IBM. One concrete metric to assess rule coverage is to observe how often a thread stack does not match any Category, thus defaults to a nickname based on the Java package.

Table 4. It typically takes only a small number of rules to cover the common Categories.

<table>
<thead>
<tr>
<th>Category</th>
<th># Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>72</td>
</tr>
<tr>
<td>Administrative</td>
<td>59</td>
</tr>
<tr>
<td>Client</td>
<td>41</td>
</tr>
<tr>
<td>Disk, Network</td>
<td>46</td>
</tr>
<tr>
<td>Waiting for Work</td>
<td>30</td>
</tr>
<tr>
<td>Marshalling</td>
<td>30</td>
</tr>
<tr>
<td>JEE</td>
<td>22</td>
</tr>
<tr>
<td>Classloader</td>
<td>13</td>
</tr>
<tr>
<td>Logging</td>
<td>12</td>
</tr>
<tr>
<td>LDAP</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 5. Rule coverage statistics on WAIT reports submitted. Data shows what percent of thread stacks are labeled explicitly by the Category rules, vs falling back to nickname based on the Java package.

<table>
<thead>
<tr>
<th></th>
<th># Reports</th>
<th># Thread</th>
<th>Category</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT team</td>
<td>378</td>
<td>605,460</td>
<td>87.7%</td>
<td>12.2%</td>
</tr>
<tr>
<td>External users</td>
<td>830</td>
<td>1,391,033</td>
<td>77.3%</td>
<td>22.7%</td>
</tr>
<tr>
<td>Total</td>
<td>1208</td>
<td>1,996,493</td>
<td>80.4%</td>
<td>19.5%</td>
</tr>
</tbody>
</table>

6. The WAIT Tool

Based on the abstractions and analyses presented throughout the paper, we have implemented WAIT as a software-as-a-service deployed in IBM. In this section, we describe the overall design of the tool, present the user interface, and discuss some implementation choices.
6.1 WAIT Tool Architecture

We designed WAIT with a primary utilization of a low barrier to entry: the tool must be simple and easy to use. Any complex install procedure or steep learning curve will inhibit adoption, particularly when targeting large commercial systems deployments.

To meet this goal, we implemented WAIT as a service. Using WAIT involves three steps:

1. Collect one or more javacores. This can be done manually, or by using a data collection script we provide that collects machine utilization and process utilization, as discussed in Section 3.1.
2. Upload the collected data to the WAIT server through a web interface.
3. View the report in a browser.

A service architecture offers the following advantages:

- **Zero-install.** The user can use WAIT without having to install any software.
- **Easy to collaborate.** A WAIT report can be shared and discussed by forwarding a single URL.
- **Incrementally refined knowledge base.** By having access to the data submitted to the WAIT service, the WAIT team can monitor the reports being generated and continually improve the knowledge base when existing rules prove insufficient. This incremental refinement approach has proved particularly beneficial during the early stages of development and allowed us to release the tool far earlier than would have been possible with a standalone tool.
- **Cross-report analysis.** Having access to a large number of reports allows looking for trends that may not stand out clearly in a single report.

The main disadvantage of a service-based tool is that it requires a network connection to the server, which is sometimes not available when diagnosing a problem at a customer site. There may also be privacy concerns with uploading the data to a central server, although these concerns are mitigated by a server behind a corporate firewall. One IBM organization has deployed a clone WAIT service on their own server, to satisfy more strict privacy requirements.

6.2 User Interface

Figure 11 shows a screenshot of a WAIT report being viewed in Mozilla Firefox. The report is intended to be scanned from top to bottom, as this order aligns with the logical progression of questions an expert would likely ask when diagnosing a performance problem.

**Activity Summary** The top portion of the report present a high-level view of the application’s behavior. The pie charts on the left present data averaged over the whole collection period, while timelines on the right show how the behavior changed over time.

The top row shows the machine utilization during the collection period, breaking down the activity into four possible categories: Your Application (the Java program being monitored), Garbage Collection, Other Processes, and Idle. This overview appears in the WAIT report first because it represents the first property one usually checks when debugging a performance problem. In this particular report, the CPU utilization drops to zero roughly 1/3 of the way through the collection period, a common occurrence when problems arise in a multi-tier application.

The second and third rows report the Wait State (described in Section 4.1) of all threads found running in the JVM. The second row shows threads that are Runnable, while the third row shows threads that are Waiting. Each bar in the timeline represents the data from one Javacore. This example shows as many as 65 Runnable threads for the first 8 javacores taken, at which point all runnable activity ceased, and the number of Waiting threads shot up to 140, all in Wait State **Delayed by Remote Request**.

Skimming the top portion of the WAIT report enables a user to quickly see that CPU idle time follows from threads backing up in a remote request to another tier.

**Category Viewer** The lower left hand pane of the WAIT report shows a breakdown of most active Categories (technically, primary Categories from Section 4.2) executing in the application. Clicking on a pie slice or bar in the above charts causes the Category pane to drill down, showing the Category breakdown for the Wait State that was clicked.

This report shows that all but one of the threads in Wait State **Delayed by Remote Request** were executing Category **Getting Data from Database**. This indicates that the source of this problem stems from the database becoming slow or unresponsive. WAIT does not currently support further detailed diagnosis of problems from the database tier itself; WAIT’s utility stems from the ease with which the user can narrow down the problem to the database, without having even looked at logs from the database machine.

**Stack Viewer** Glancing at the commonly occurring Wait States and Category activity often suffices to rapidly identify bottlenecks; however, WAIT provides one additional level of drilldown. Selecting a bar in the report opens a stack viewer pane to display all call stacks that match the selected Wait State and Category. Stacks are sorted by most common occurrence to help identify the most important bottlenecks. Having full stack samples available has proven valuable not only for understanding performance problems, but for fixing them. The stacks allow mapping back to source code with full context and exact lines of code where the backups are occurring. Passing this information on to the application developers is often sufficient for them to identify a fix.

The presence of thread stacks makes WAIT useful not only for analyzing waiting threads, but also for identifying program hot spots. Clicking on the Runnable Threads pie slice causes the Stack Viewer to display the most commonly occurring running threads. Browsing the top candidates often produces surprising results such as seeing “logging activity” or “date formatter” appear near the top, suggesting wasted cycles and easy opportunities for streamlining the code.

**Discussion** The WAIT user interface takes an intentionally minimalist approach, striving to present a small amount of semantically rich data to users rather than overloading them with mountains of raw data. Although the input to WAIT is significantly lighter weight and more sparse than of many other tools, particularly those based on tracing, we have found that WAIT is often more effective for quick analysis of performance problems.

The pairing of WAIT’s analyses together with drilldown to full stack traces has proven to be a powerful combination. The WAIT abstractions guide the user’s focus in the right direction, and presents a set of concrete thread stacks that can be used to confirm the hypotheses. If the WAIT analysis is incorrect, or is simply not trusted by a skeptical user, viewing the stacks quickly confirms or denies their suspicion.

---

3This label corresponds to the **Awaiting Data** node in Figure 5(a). The UI does not always present the same text as used to define the underlying model, but the mapping is straightforward. For the remainder of this paper, when discussing the UI, we simply refer to text labels as presented in the UI.
Figure 11. Example WAIT Report
6.3 Implementation

WAIT is coded in a combination of Java and Javascript. The ETL step of parsing the raw data and producing the data model (Section 5.1) runs in Java and executes on the server once, when a report is created. The remaining analyses (Wait State analysis and Category analysis) run in Javascript and execute in the browser each time a report is loaded. This somewhat controversial design allows users to modify the rules or select alternate rules configurations without a round trip to the server. This design also allows WAIT reports, once generated, to be viewed in headless mode without a server present; the browser can load the report off a local disk and maintain full functionality of the report.

This design presents two performance constraints on the WAIT analysis: (1) the data models must be sufficiently small to avoid excessive network transfer delays, and (2) the analysis written in Javascript must be fast enough to avoid intrusive pauses, and moreover avoid Javascript timeouts. For example, Firefox 2.5 warns the user that something might be wrong with the page after roughly five seconds of Javascript execution.

Table 6 reports data model sizes for six representative IBM customer applications, labeled A1–A6. The clustering and optimization steps described in Section 5.1 produce a significant reduction in space for the data model. For larger inputs, the clustered data model is multiple orders of magnitude smaller than the zipped raw data input files. The largest submission received to date contained 946 samples (Javacores), with a total of 127,536 stack samples. The data model for this report was still a manageable 499kB. More typically, we have found that the clustered output consumes fewer than 50kB.

The performance of the WAIT analyses is reasonable, even though executed within the browser. Table 7 shows the analysis runtime for the applications from Table 6. The table shows that, even with a large number of application samples (e.g. the 946 of application A6), the analysis completes in less than one second. As discussed later in Section 7, most typical uses of the tool have inputs with a smaller number of samples, more similar to applications A1 through A4. For these typical cases, the analysis usually completes in less than 100ms. In general, analysis takes less time than general widget-related rendering work.

7. Case Studies

We now show six additional case studies to demonstrate how WAIT illuminates various performance problems. All these case studies represent real problems discovered when analyzing IBM systems.

To conserve space, we show only the relevant portions of the UI for each case study.

<table>
<thead>
<tr>
<th>Application</th>
<th>Threads Sampled</th>
<th>Zipped Input</th>
<th>Clustered Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>228</td>
<td>156kB</td>
<td>50kB</td>
</tr>
<tr>
<td>A2</td>
<td>206</td>
<td>447kB</td>
<td>40kB</td>
</tr>
<tr>
<td>A3</td>
<td>90</td>
<td>112kB</td>
<td>22kB</td>
</tr>
<tr>
<td>A4</td>
<td>356</td>
<td>449kB</td>
<td>15kB</td>
</tr>
<tr>
<td>A5</td>
<td>27,638</td>
<td>27MB</td>
<td>56kB</td>
</tr>
<tr>
<td>A6</td>
<td>127,536</td>
<td>120MB</td>
<td>499kB</td>
</tr>
</tbody>
</table>

Table 6. Size of the data model communicated from server to the client's browser. This data is a representative sample from actual IBM customer data that was submitted, by support personnel, to the WAIT service.

<table>
<thead>
<tr>
<th>Analysis Time (milliseconds)</th>
<th>Firefox 3.6</th>
<th>Safari 4.05</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>140</td>
<td>23</td>
</tr>
<tr>
<td>A2</td>
<td>90</td>
<td>16</td>
</tr>
<tr>
<td>A3</td>
<td>53</td>
<td>13</td>
</tr>
<tr>
<td>A4</td>
<td>68</td>
<td>28</td>
</tr>
<tr>
<td>A5</td>
<td>560</td>
<td>204</td>
</tr>
<tr>
<td>A6</td>
<td>720</td>
<td>110</td>
</tr>
</tbody>
</table>

Table 7. Execution time for the inference engine, as Javascript on a 2.4GHz Intel Core2 Duo with 4GB of memory, on MacOS 10.6.2. The applications, A1–A6, are the same as those in Table 6.

7.1 Lock Contention and Deadlock

Figure 12 depicts a WAIT report on a 48-core system. The Waiting Threads timeline shows a sudden and sustained surge of Blocked on Monitor; i.e. threads seeking a lock and not receiving it, and thus being blocked from making progress. Looking at the Category breakdown suggests that lock contention comes from miscellaneous APACHE OPENJPA Code. The thread stacks from this category identify the location of the lock contention as line 364 of getMetaDataLocking().

Sometimes locking issues go beyond contention to the point of deadlock or livelock. When WAIT detects a cycle in the monitor graph, their Wait State is Blocked on Deadlock. This situation appears in Figure 13. Looking at the thread stacks at the lower right of the report suggests that the threads are waiting for a lock in the logging infrastructure method SystemOutStream.processEvent(). Looking at the code and try to determine the reason for the deadlock.

7.2 Not Enough Load

The WAIT report in Figure 14 shows that the machine was 67% idle, and the timeline confirms that utilization was consistently low during the entire monitoring period. The Waiting Threads piechart indicates that threads spend most of their time Delayed by Remote Request. Digging more deeply, the Category view indicates that the remote request on which the threads are delayed is client communication using an HTTP protocol. In this case, the performance problem is not with the server machine, but that the amount of data being received from the client machine is not sufficient to keep the server busy. Indeed it is possible that there is no problem in this situation, other than that the server may be over provisioned for this workload. WAIT cannot directly determine whether the client is generating a small amount data or if the network between the client and server is under-provisioned to handle the request traffic. To answer this question, a network utility such as netstat could be employed, or the CPU utilization of client machines could be investigated.

7.3 Memory Leak

WAIT can also detect memory leaks. As shown in Figure 15, a memory leak can be detected by looking at just the first timeline, which shows garbage collection activity over time. In this example, initially the non-GC work dominates, but over time the garbage collection activity increases until it eventually consuming most of the non-idle CPU cycles. The large increase in garbage collection as time passes is strong evidence that the heap is inadequate for the amount of live memory; either the heap size is not appropriate for the workload, or that the application has a memory leak.
7.4 Database Bottleneck

Figure 16 presents an example of a database bottleneck. Unlike Figure 11 where the database became completely unresponsive, in this case the database is simply struggling to keep up with the application server’s requests. Over time the server’s utilization varies between approximately 10% and 85%, and these dips in utilization correlate roughly with the spikes in the number of threads in Waiting state Delayed by Remote Request and Category Getting Data from Database, thus pointing to the likely source of the delay.

Clicking on the orange bar for Getting Data from Database reveals the thread stacks that a developer can analyze to determine key parts of the application delayed by the database, and try to reduce the load generated against the database. Alternatively, the database could be optimized or deployed on faster hardware.

7.5 Disk I/O Affecting Latency

Figure 17 shows a WAIT report where filesystem activity is limiting performance. The top two pie charts and timelines show that there is enough Java code activity to keep the CPUs well-utilized. However, the Waiting activity show a significant number of threads in Wait State Delayed by Disk I/O, and Category Filesystem Metadata Operations, suggesting room for improvement with faster disks, or by restructuring the code to perform fewer disk operations.

Reducing these delays would clearly improve latency, since each transaction would spend less time waiting on Disk I/O, but such improvement would have other benefits as well. Even though the four CPUs on this machine are currently well utilized, this application will likely scale poorly on larger machines. As the number of processors increases, the frequent disk access delays will eventually become a scaling bottleneck. WAIT can help identify these scaling limiters early in the development process.

8. Related Work

Most conventional gprof-style performance analysis tools (e.g., [19, 20]) provide a profile which indicates in which methods a program spends time. Often, these tools provide a hierarchical tree view, in which an analyst can recursively drill down to find time spent in subroutines. These tools also often provide a summary view which indicates the sum totals of execution time over a run. Since performance characteristics vary over time, a summary view is often less helpful than snapshot views of profile data over smaller windows. Many tools (e.g., [12, 15, 20]) provide snapshot or window views.

A few previous works have applied abstraction to profile information, in order to infer higher-level notions of bottlenecks. Hollingsworth [7] identified classes of performance bottlenecks in parallel programs, and introduced a methodology to dynamically insert instrumentation to test for bottlenecks. Srinivas and Srinivasan [10] present a tool design that allows the user to define “Components”, abstract names for code from particular packages or archives. This paper presents a new methodology to present lay-
ers of abstraction when monitoring performance, suitable for production enterprise deployment scenarios.

A few previous works have been developed to mine log information in large systems [22, 23]. However, unlike the approach proposed here, both require access to source code that is often unavailable in enterprise environments. In addition, [22] employs machine-learning to determine important characteristics to report as opposed to the expert rule structure employed here. [23] uses analysis of source code control paths to determine precise causes of error conditions.

There are heavier-weight alternatives to sampling that we chose to avoid. Performance tools exist that collect detailed information about method invocations [2, 3, 5, 6, 11, 13, 19], or that stitch together an end-to-end trace of call flows across a multi-tier system [1, 4, 14, 16, 18].

We also note that performance experts have been using Java core dumps for many years, but generally in an ad-hoc way using their eyes and brains to summarize the data instead of having a tool to do so. Though other tools like Thread Analyzer [17] take Java cores as input, the tools of which we are aware are interactive, and require an expert user for effective results. As a result, existing Java core tools are generally not suitable for use by less expert users for quick triage.

Others have attempted to use method names to help diagnose software problems [8]. However, unlike our approach which employs method names to diagnose functional issues in the system, [8] uses method name analysis to promote adherence to naming conventions and readable, maintainable software.

9. Conclusion

We have presented the design and methodology behind WAIT, a tool designed to pinpoint performance bottlenecks in server applications based on idle time analysis. WAIT has been used widely throughout IBM, in both development and production deployments. Through these deployments, we have learned how to exploit ubiquitous sample-based monitoring to infer rich semantic information about server application performance. Although much of the literature explores intrusive monitoring based on instrumentation and monitoring agents, we have shown effective performance analysis is possible with a lower barrier to entry.
References


[14] HP. HP Diagnostics for J2EE.


Figure 17. WAIT Report: Good Throughput but Filesystem Bottleneck

