

Measuring the Performance of ARM 3.0 for Java™

Mark W. Johnson
Tivoli Systems

Jason Crowe
Tivoli Systems

ARM (Application Response Measurement) is a standard of The Open Group for measuring the performance and availability of business transactions, and their sub-transactions. A new version of the standard for use in Java programs, ARM 3.0 for Java, has been specified and submitted for approval. This paper analyzes the performance of a reference implementation of the new version, and the additional latency incurred by applications that use it.

1 Introduction

Computers, networks, and the applications that run on them have become essential to the operation of the world's economy. In many cases end users don't think about the fact that they're using a computer. They want to do some transaction, such as check their bank balance, make a credit card purchase, or get the news delivered to their cell phone. They want the transaction to complete reliably and quickly. If it doesn't they will take their business elsewhere.

Companies providing services to end-users have an enormous stake in the correct and timely completion of these transactions. They need to measure the transactions to know if they are meeting their customers' expectations. They need to understand the relationships of the transactions to other transactions and to the underlying computing and network infrastructure. Following are some questions of interest.

- Are transactions succeeding?
- If a transaction fails, what is the cause of the failure?
- What is the response time of the transaction?
- If the response time is unsatisfactory, where is the excess delay occurring?
- How many transactions are being executed?
- How can the application and environment be tuned to be more robust and perform better?

ARM is a standard for measuring transactions that helps answer these questions (see [ARM97] and [JOHN97]). ARM was adopted by The Open Group [TOG98] in 1998. A new version of the standard for use with Java™

programs, ARM 3.0 for Java, has been proposed and is awaiting formal approval by The Open Group.

ARM differs from some techniques that measure application performance because it is a collaboration between the application and the measurement software. Applications imbed calls to an ARM instrumentation library when a transaction begins and/or ends. Because this requires changes to the application, developers need to know that using ARM is reliable and has minimal impact on performance.

1.1 Objective of the Paper

The objective of this paper is to provide application developers and system administrators with information helpful when deciding whether and how to use ARM 3.0 for Java. The performance of a reference implementation available with the software developer's kit (SDK) is assessed qualitatively and quantitatively. Some questions the paper addresses are:

1. Could a faulty consumer (such as a management agent) or a misbehaving application impact another application?
2. How much will using ARM impact the application response time?
3. How fast can ARM transaction records be collected and processed?

Performance analyses are always specific to a particular environment and configuration. The intent of the paper is

not to be a definitive reference or guarantee of performance. The intent is to provide a first-order assessment that is suggestive of how using ARM affects application latency, and how large a workload could be reasonably measured with ARM.

The reference implementation runs in a tracing mode in which a record for each and every transaction is passed to consumers to process. Section 6 will discuss why this is an unusual and highly stressful configuration for a production environment. ARM implementations intended to be used in production environments would typically not exhibit the degradation seen in these experiments at very high transaction rates (over 5000 per second).

The design of the reference implementation is reasonably streamlined, but the implementation has not been heavily optimized. Commercial or specialized implementations would be expected to be more optimized and more refined, and to perform at least as well. Assuming they do, the performance results reported in this paper would constitute a lower bound.

1.2 Organization of the Paper

Section 2 is a brief survey of different techniques for measuring transactions, including ARM. (Note that the term “transaction” is used loosely. Any unit of work with a response time and status can be measured. Transactional closure, such as provided through the use of Commit/Rollback protocols, is not required). Section 3 briefly describes the ARM 3.0 for Java interface used by business applications. Section 4 describes a reference implementation and discusses how its architecture may impact performance. Section 5 discusses the important performance criteria from two perspectives, the business application using ARM, and a management agent receiving ARM data. Section 6 describes how the reference implementation differs from a typical commercial implementation. Section 7 describes some experiments measuring the latency and throughput of the ARM implementation, and Section 8 presents the results. Section 9 presents some conclusions and Section 10 is a summary.

2 Measuring Application Performance

There are two general ways to measure the response time and status of transactions. The first way uses probes external to the application. The second way uses instrumentation within the application, or within a proxy for the application. ARM is used in the second way. Applications call ARM to provide measurements to management agents and applications.

2.1 External Probes

Four examples of external probes are network probes that “sniff” packets, GUI (Graphical User Interface) monitors that intercept and analyze operating system messages, probes that monitor API (Application Program Interface) calls to libraries, such as a database client access library, and probes that track the control flow within an application. In each case the objective is to learn signatures by which the beginning and end of a transaction can be recognized and matched up with each other. The elapsed time between the beginning and end is the response time of the transaction.

The major advantage of external probes is that no changes to the application are required.

Potential disadvantages include higher overhead to sort through many messages or packets, difficulty determining reliable and unambiguous signatures, coping with encrypted or compressed data streams, and potential instability if the probes inadvertently disrupt the system.

2.2 Instrumentation and ARM

Instrumentation uses calls from the business application to a software library when a transaction starts and stops, as shown in Figure 1. The box labeled “ARM” represents the libraries. The libraries act like a broker between the applications and management agents and applications. The elapsed time between the start and stop calls is the response time. Alternatively, the application can measure the response time itself and then call the library to report the results. ARM defines a standard interface between application instrumentation and a measurement library.

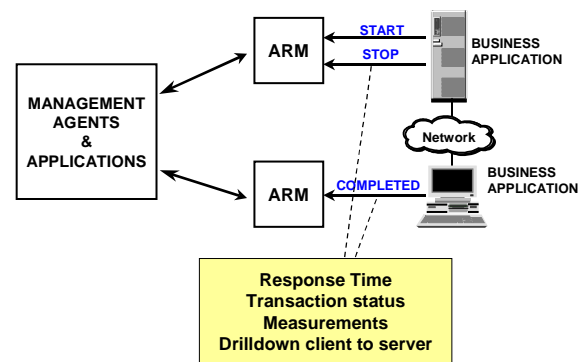


Figure 1. Overview of ARM

There are several potential advantages using instrumentation and ARM. Instead of sorting through potentially many messages, packets, or API calls per transaction, one or two calls per transaction are sufficient. There isn't a need to learn relevant and unambiguous signatures and to update them for each new release of

the application. Instead the calls to the library specify unambiguously which transaction is being executed. There is a well-defined interface for providing additional measurements and identity information about a transaction (“metrics” in the vocabulary of the ARM standard). There is also a well-defined interface for generating and passing tokens that uniquely identify each transaction, and the transaction that is the “parent” of the transaction. The parent/child data can be used to reconstruct the calling tree of nested transactions to isolate delays or failures.

The big potential disadvantage to using ARM is that if the application (or its proxy) can’t be instrumented to call ARM, which happens if the source code is not available, the standard is not useful. Developers also need satisfactory assurances that the reliability and performance of their applications will not be adversely affected.

3 Overview of ARM 3.0 for Java

A transaction in ARM 3.0 has the following properties. The entire purpose of ARM is to facilitate collecting this information about each instance of a transaction.

- A 16-byte universally unique identifier (UUID) that identifies the transaction type. It can be associated to the name of an application and a transaction. Examples would be the “Home Banker Version 1.2.4” application and the “Check Balance” transaction.
- An 8-byte transaction handle differentiates between different instances of the same transaction type in the same Java Virtual Machine (JVM).
- The response time of the transaction.
- The time of day the transaction completed (from which, when combined with the response time, the time of day the transaction started can be derived).
- The status (Good, Failed, Aborted).
- (Optionally) tokens of about 50 bytes that uniquely identify the type and instance of a transaction and/or the parent transaction. These tokens are named “correlators” in the ARM vocabulary.
- (Optionally) zero to seven values that further describe the transaction or the state of the environment. Examples are the number of records processed by this transaction, a part number being updated, a sense code describing details about a failure, and the number of jobs in a queue when this transaction was processed or received.

An application uses one of two ways to provide the measurements. In the first way, shown in Figure 2, an application creates instances of the ArmTransaction class. Just prior to beginning a transaction it calls the start() method. Just after the transaction completes it calls the stop() method. Optionally it sends any number of heartbeats using the update() method.

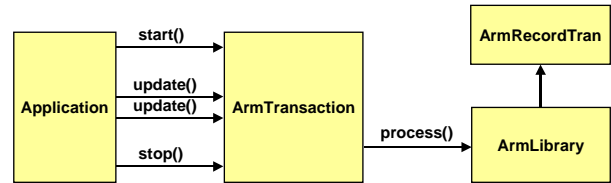


Figure 2. Measuring transactions synchronously with start/stop calls

ArmTransaction measures the elapsed time between the start and stop calls. This is the response time of the transaction. It also optionally gathers other values and calls ArmLibrary.process(). ArmLibrary populates a record (an ArmRecordTran instance) that contains all the data about the transaction and passes the data to the ARM implementation.

In the second way, shown in Figure 3, the application itself measures the response time. It creates and populates an ArmRecordTran instance, then calls ArmLibrary.process() to pass the data to the ARM implementation.

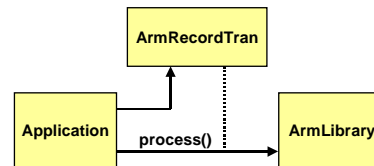


Figure 3. Reporting measurements asynchronously

Using ArmTransaction has two advantages. ArmTransaction is part of the ARM implementation and it can be designed to monitor for “hung” transactions, that is transactions with a start but without a corresponding stop within a specified time. For long running transactions the update heartbeats can provide useful information on the progress and health of the transaction.

Using ArmRecordTran has two advantages. First, the application can call ArmLibrary once anytime after the transaction completes, when it is most convenient, instead of making two synchronous inline calls. Second, the transaction can execute on one system (such as a client system) and be reported on a second system (such as the server system).

Error handling is done out-of-band and is implementation-dependent. If an application makes a programming error, such as providing invalid data, or calling stop() without

first calling `start()`, no exception or other notification is returned to the application. Notification of the error is optionally provided to the system administrator who configures the ARM environment. This simplifies the programming in the application, while still giving the developer and system administrator the information needed to test and manage the environment.

4 The ARM Reference Implementation

There are three actors in ARM, the application, the ARM implementation, and the consumer.

- The application calls the ARM implementation to measure transactions.
- The ARM implementation serves as a conduit between applications and consumers. The SDK contains a reference implementation.
- The consumer subscribes for transaction data from the ARM implementation. What the consumer does with the data is not a concern of the ARM implementation. Three examples are processing the data in-memory (such as monitoring thresholds), writing serialized records to a file, or writing serialized records across a network.

Figure 4 depicts the reference implementation in a common configuration. Multiple application threads, running in a JVM, pass measurement data to `ArmLibrary`. `ArmLibrary` puts one `ArmRecordTran` instance per completed transaction on a First-In-First-Out (FIFO) queue implemented in `ArmQueue`. The data from all the threads is multiplexed through a single `ArmQueue` instance, which passes it to a consumer, running in its own thread. The consumer then passes the data to analysis programs or writes serialized records to storage or across a network.

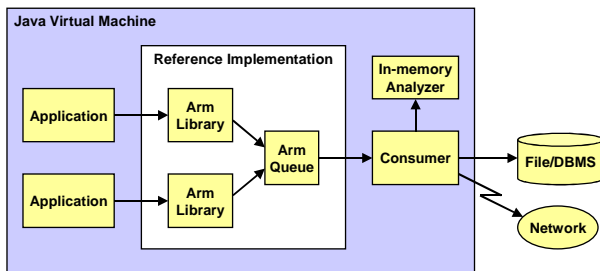


Figure 4. ARM SDK's reference implementation

4.1 Types of Data Records

There are five types of data records in the reference implementation. The first three are part of the standard.

The last two are specific to the reference implementation. A consumer can subscribe to any combination of the five. The administrator can limit the number of consumers that can subscribe to any record type.

- An `ArmRecordTran` instance represents one completed transaction. The type of transaction is identified by a 16-byte UUID.
- An `ArmRecordTranDefinition` instance associates any of an application name, transaction name, and zero to seven 16-byte metric identifiers with a 16-byte UUID. Its use is optional. When used it would typically be used only when an application initializes.
- An `ArmRecordMetricDefinition` instance associates a metric name and format with a 16-byte UUID. Its use is optional. When used it would typically be used only when an application initializes.
- An `ArmRecordError` instance represents a detected error. An example of an error is when an application makes an incorrect call, such as issuing `stop()` to an `ArmTransaction` instance that is not currently executing a transaction. It would be most heavily used during program testing and debug, but could provide useful diagnostic information in a production setting.
- An `ArmRecordStatistics` instance represents data about the reference implementation itself, and the amount of activity that has occurred. Examples are counts of application programming errors, counts of lost measurements due to overrun, the number of transactions processed per minute, etc. Consumer programs or administrators may find the data useful for tuning the configuration.

This paper analyzes only the `ArmRecordTran` instances because they will usually comprise almost all of the records in a healthy production environment. Of the other four types, only statistics records would be created regularly, and only a handful of those each hour.

4.2 Data Flows

To receive ARM data a consumer creates an `ArmQueue` instance, and one or more `ArmPool` instances (one per record type). It populates the `ArmPool` with instances of unused records. It uses `ArmQueue`'s `subscribe()` method for each type of record in which it has interest. If a consumer is subscribed to more than one record type, all the records are interleaved in the order they are received through the same `ArmQueue` FIFO queue.

Figure 5 shows how data flows from the application to a consumer through an ArmQueue instance. The flows are for ArmRecordTran instances. The flows do not change for other record types except that there is a separate pool of unused records for each record type.

1. The consumer creates several instances of ArmRecordTran and adds them to ArmPool.
2. If it hasn't done so already, it subscribes for ArmRecordTran records. Its thread blocks waiting for a record to process..
3. The application completes a transaction and passes the data to ArmLibrary.
4. ArmLibrary gets one of the consumer's records from ArmPool. If there isn't a record in the pool, it notes the fact (not shown in Figure 6) for later reporting and returns immediately to the application. The measurement will be lost but the application will not be delayed.
5. ArmLibrary populates the consumer's record with the transaction data and puts it on the ArmQueue FIFO queue. ArmLibrary returns to the application. The application is now free to reuse its ArmRecordTran (or ArmTransaction) instance.
6. The consumer's thread is dispatched. The record is removed from the FIFO queue and a reference to it passed to the consumer. The consumer processes the data. For example, it might serialize the record contents and write them to a file. When it is done with the data in the record, it returns to step (1), putting the record back in the pool to be reused.

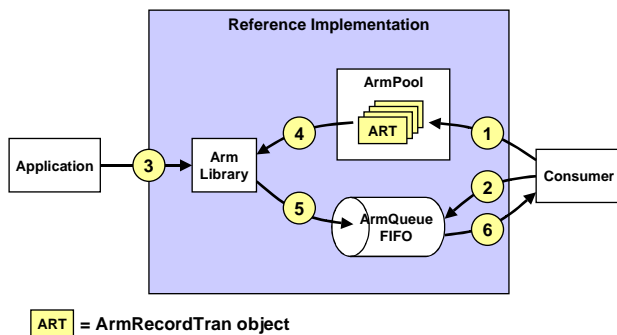


Figure 5. How data flows from applications to the consumer

4.3 Applications Insulated from Consumers

As shown in Figure 5, both the application and the consumer create their own ArmRecordTran instances (or the application uses ArmTransaction instances). Data

from the application's instance is copied to the consumer's instance. Why is this done? It would require fewer CPU cycles to pass a reference and have the consumer access the properties of the record directly. There are several reasons.

The single most important design objective for ARM is to avoid negative impacts to application reliability. If a consumer is given a reference to the application's data, a misbehaving consumer could inadvertently corrupt the application's data. This situation is avoided by maintaining a strict separation between the application and the consumer.

Almost as important is minimizing the impact on application response time from using ARM. If a reference to the application's instance would be passed to the consumer, the application thread would have to be blocked until the consumer finished reading the data in the instance. With many application threads and potentially multiple consumers, this delay would likely be unacceptable, and it would certainly be unpredictable. Instead, the ARM implementation combines the separation of data with a no-waiting philosophy. If an ArmRecordTran instance is not available when the application completes a transaction, the application is not delayed. This approach requires copying several attributes from the application's record to the consumer's record, but this is a fast operation.

4.4 The Role of the Consumer

The application is not dependent on what a consumer does or does not do. The application is not even aware if a consumer is running. Its only interface is through ArmLibrary and/or ArmTransaction.

As shown in Figure 5, the consumer is responsible to subscribe for records, to populate the appropriate ArmPool with a sufficient number of record instances, and to process the records in the ArmQueue FIFO queue in a timely manner. If there aren't enough records, the measurement is discarded, because the application is not delayed. By giving the consumer the responsibility and the capability to manage its own pool of records and FIFO queue, each consumer has maximum control over its environment.

4.5 A Multiple-Consumer Configuration

Generally ARM is used with a single consumer, that is, a single management agent. This agent manages all the applications on the system. An example is an agent that monitors response times versus thresholds and also logs summaries of 15-minute intervals for service level reporting. An administrator configures the agent to adapt

within certain bounds to the specific application and environment. For example, the administrator would set different response time thresholds for different transactions.

There are situations in which system administrators wish to manage the same application with multiple consumers.

- One example is using a specialized diagnostic agent that analyzes the transactions of one application but does not know how to analyze other applications. The administrator might wish to monitor all applications with the monitoring agent and also send the data for the one application to the specialized diagnostic agent.
- Another example is using load balancing software that adjusts system priorities based on observed response times. The administrator wishes to use the load balancing software, but without disabling the monitoring agent.
- Another example is an administrator who wishes to capture the ARM data for some in-house analysis that is not performed by the monitoring product.

The reference implementation has an alternative configuration that supports multiple consumers. Figure 6 depicts this configuration. There is no difference to the applications or the consumers between the single and multiple consumer configurations. The application interface is through ArmLibrary and/or ArmTransaction. The consumer interface is through ArmPool (not shown) and ArmQueue. The differences are in the heart of the implementation. Whether one or more consumers are supported is not part of the ARM standard.

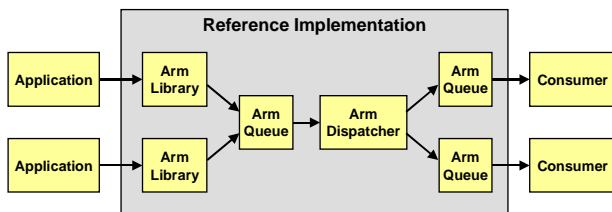


Figure 6. An alternative configuration with multiple consumers

As in single-consumer mode the data from multiple application threads are multiplexed together through a single ArmQueue instance. The difference is that instead of the records being put on the ArmQueue of the consumer, they are put on the ArmQueue of a dispatcher. The dispatcher runs in its own thread. It removes records from the single ArmQueue instance and puts copies of the records on the ArmQueue instances through which the consumer(s) have subscribed. Each consumer runs in its own thread.

The queuing and congestion phenomena are the same in single- and multiple-consumer configurations. When using the dispatcher the impact is compounded because each record passes through at least two ArmQueues. There is additional overhead due to the extra processing to copy the records multiple times.

5 Important Performance Criteria

This section discusses the architecture of the reference implementation and how it affects performance. Two perspectives are discussed: a business application using ARM and a management agent (consumer) receiving ARM data.

5.1 Business Application Perspective

A business application is primarily concerned with the additional latency from using ARM. There are two sources of latency. The first is the processing time to populate ARM records and pass the data. The second is the additional latency waiting for the ARM implementation to process the data.

5.1.1 Latency due to Processing

There is an overhead cost to using ARM, even if small. The methods used to populate an ArmRecordTran object consist mostly of simple set() operations. Two system calls are used to get the system clock value – one at the beginning of the transaction and one at the end – so the response time can be calculated. If the optional features are used – metrics and unit of work tokens – the processing time would be expected to increase in a predictable manner.

The interface enables reusing pools of ArmRecordTran and ArmTransaction objects, or allocating one object per thread and reusing that one object. Reusing the objects avoids the cost of repeatedly allocating and initializing the objects, plus the cost of garbage collection.

5.1.2 Latency due to Congestion and Queuing

Although not obvious from the high level schematic in Figure 5, there are three potential queuing delays that in a multi-threaded system could increase application latency above the cost to process the records. To understand why this is so, one needs to understand the system structure, where there are shared resources that must be accessed by only one thread at a time, and how cross-thread synchronization works in the Java language.

Critical sections are common in programs that share resources across independently executing threads or

processes. A critical section is any part of a program that, once entered by a thread, must execute to completion in that thread before any other thread can execute that part of the program. Critical sections need to be used when updates to shared data are not atomic operations. Failure to protect such shared data leads to data corruption and an unstable program state. A complete treatment of critical sections is beyond the scope of this paper. For an introduction that focuses on how to manage critical sections in Java programs, see [HELL99].

In Java access to critical sections is controlled with the keyword “synchronized”. Any block of code can be designated as a synchronized section. It is particularly convenient to designate entire methods as synchronized, and then adhere to good programming practices by keeping the code in the methods to the bare minimum needed to insure the integrity of the system.

A thread is said to “hold the lock” for a synchronized method when it enters the method. If a second thread attempts to execute the method while another thread holds the lock, the second thread is blocked until the first thread completes and releases the lock. Any number of threads can be blocked simultaneously waiting for the lock. One thing to note is that the Java language does not require or guarantee that threads will be released to run in the order that they were blocked.

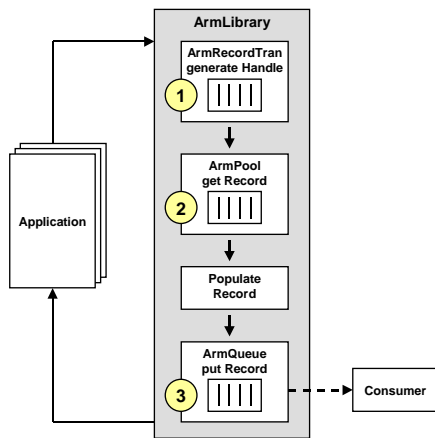


Figure 7. An application's view of the reference implementation as a queuing system

Figure 7 depicts the ARM reference implementation as a queuing system. The three rectangles numbered 1, 2, and 3 represent synchronized methods. Many threads may attempt to execute each method at the same time, but only one at a time can. From a queuing system perspective, each method is a server and the wait for the lock to enter each method is a queue. The processing time in each method is minimal.

The “**ArmRecordTran generate Handle**” rectangle is the `genNewTranHandle()` method of the `ArmRecordTran` class. It creates a 64-bit handle from a combination of the system time plus a 20-bit sequence counter. It is called once each time a transaction executes. The handle is used in the unit of work tokens (a.k.a. “correlators”) to differentiate instances. Both `ArmRecordTran` and `ArmTransaction` use this method. The sequence counter increments starting from zero each time a new time value is used. It will produce a unique value as long as no more than $2^{20} = 1,048,576$ transactions execute within one clock update period (typically about 10 milliseconds). The method is a critical section because it is essential that the time value and sequence counter be updated together in one atomic operation to avoid duplicates.

The “**ArmPool get Record**” rectangle is `ArmPool`'s `get()` method. It is used by `ArmLibrary` to get an unused record from the pool of available records (step 4 in Figure 5) prior to copying data from the application's `ArmRecordTran` or `ArmTransaction` instance.

The “**ArmQueue put Record**” rectangle is `ArmQueue`'s `put()` method. It is used by `ArmLibrary` to put a record onto the FIFO queue (step 5 in Figure 5) after the record has been populated with data.

Both `ArmPool` and `ArmQueue` are examples of using a shared data structure to facilitate communication between threads. Methods that update them are critical sections because several application and/or consumer threads may try to add or remove records simultaneously. The individual add and remove operations themselves are not atomic, so a critical section is needed to preserve data integrity.

5.2 Management Agent Perspective

The most common consumer of ARM data is expected to be a management agent. Four considerations for the management agent are: the robustness of the measurement delivery process, the throughput to deliver the measurements, the age of the data when it is delivered, and the overhead processing cost.

Figure 8 depicts the measurement delivery process as a queuing system. There are six queues.

- Queues 1-3 are shown in Figure 7 and described in detail in section 5.1.2.
- Queue 4 is the `ArmQueue` itself. The record will spend some time in the `ArmQueue` working its way through the FIFO queue, before it is the next record to be processed.

- Queue 5 is the delay due to contention for access to ArmQueue. The consumer is contending with ArmLibrary, acting on behalf of the business application threads (step 3 in Figures 7 and 8).
- Queue 6 is the delay due to contention for access to ArmPool. The consumer is contending with ArmLibrary, acting on behalf of the business application threads (step 2 in Figures 7 and 8).

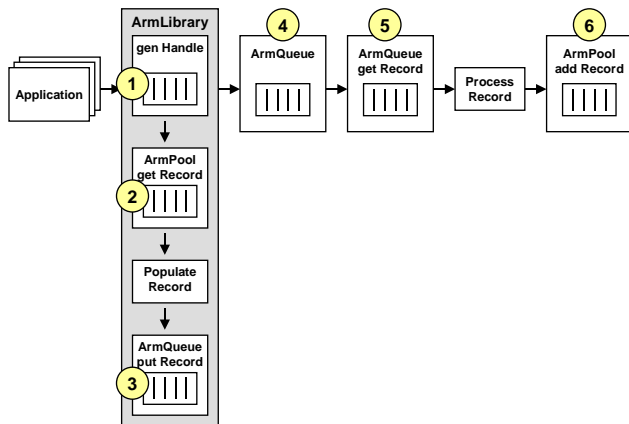


Figure 8. A consumer's view of the reference implementation as a queuing system

5.2.1 Robust Measurement Delivery

If the queuing system was loss-free the robustness of the delivery would be dependent entirely on the quality of the code, and the performance of the implementation would not be a factor. This is not the case. Because minimal impact to business applications is the single most important design criteria, the reference implementation is designed to explicitly drop data records when confronted with unacceptable congestion. Because of this, dropped records is an indicator of congestion.

Unacceptable congestion in the reference implementation is indicated by the non-availability of records in ArmPool (step 4 in Figure 5). If no record is available when ArmLibrary.process() or ArmTransaction.stop() are called, control returns immediately to the application without processing the data. Another implementation could use other indications of congestion, such as the number of records waiting to be processed.

5.2.2 Data Latency

The end-to-end latency is the time from when the application reports data until the data record is received and processed (all the steps in Figure 8). This time is generally not a concern to the consumer. The consumer expects to receive records that are delayed from when the transaction executed. Whether the delay is several

milliseconds or a few seconds is not material. The delay could be minutes or hours if the application is reporting data for transactions that executed much earlier. In this case the consumer needs to be aware of this and process the data in an appropriate context. For example, it would probably not be advisable to generate an alert notification for a missed response time threshold for a transaction that executed several hours earlier.

The consumer has one interest in the time that a record spends in ArmQueue. A commonly used theorem in queuing theory is Little's Law, which states that the mean number of jobs in a system is equal to the product of the arrival rate and the mean time a job spends in the system. If jobs (records in this case) spend a long time in ArmQueue, and the arrival rate of records is high, the consumer would have to put many records in ArmPool to avoid losing measurements.

5.2.3 Measurement Throughput

The throughput of the system is a measure of how many transactions per second can be processed by the ARM implementation. On a busy server system there could be many transactions per second to measure. If the ARM implementation cannot service transactions as fast as they are generated, congestion will build, leading to lost measurements.

The congestion due to queuing effects has been discussed. Another part of the system is the rectangle labeled "Process Record" in Figure 8. There are no queuing effects associated with this process because it executes entirely within the consumer's thread(s). However, it could be a non-trivial amount of processing. One example would be I/O to a network, file, or database. The delays could be significant enough that the consumer's ability to service ArmQueue is degraded, leading to congestion and lost records.

5.2.4 Resource Consumption

Another important factor is the amount of system resources required to process the data through the ARM implementation and within the consumer. This demand for resources competes with the application demands for resources because they are both executing in the same process on the same machine.

6 Common Commercial Implementations

The reference implementation is intended to serve the needs of two types of users. To serve their needs the reference implementation operates in a trace mode, producing a copy of each and every transaction record.

- Application developers calling ARM measurement objects need a way to test their instrumentation. They need detailed trace data.
- System administrators considering managing their applications with ARM want a way to quickly and easily collect data and see some reports. They would like to deploy a pilot configuration on a modest number of machines, typically ten to fifty machines. The SDK contains some simple formatting programs and the records can be written to a spreadsheet for simple reporting.

Trace-level processing can also be useful in production environments, and is needed when correlating parent and child transactions. A commercial ARM product might offer a tracing feature, but usually such a feature is used on a sampling basis, or occasionally for diagnostic purposes. Figure 9 shows a configuration that would be used by many commercial ARM products most of the time they are running. Instead of ArmLibrary making a copy of the data record and putting it on a queue for delivery to a consumer, as shown in Figure 4, the ArmLibrary implementation updates a statistical summary for the transaction type and returns control to the application. Access to the shared statistical summary would be synchronized across many threads, like the six queues are in Figure 8, to avoid data corruption. Periodically, such as once every one to ten minutes, the summaries are delivered to analysis and reporting programs.

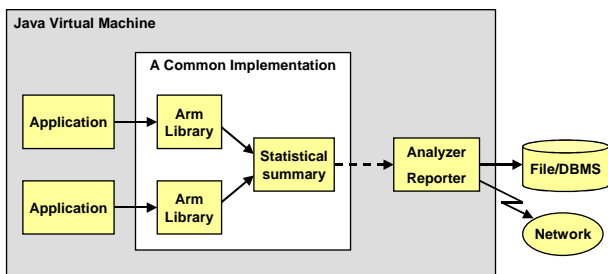


Figure 9. An ARM implementation without tracing

In this configuration the volume of statistical summaries is usually inconsequential when compared to the number of transaction records passed by the application. Referring to Figure 8, steps 1-6 would be replaced with one step to access a shared statistical summary and update its contents. A typical update would consist of summing data values from the transaction record to counters in the summary. There would be one summary per transaction type, identified by its 16-byte universally unique identifier. Typical counters would be a count of executed transactions by status (good, failed, aborted), the cumulative response time, and the sum of the squares of the response time (useful for calculating the standard

deviation of the response times). Some products keep counts of completed transactions with response times in defined ranges (0-500 milliseconds, 500-1000 milliseconds, 1000-2000 milliseconds, etc. This data can be used to create a histogram of the response time frequency distribution.

7 Description of Experiments

A series of experiments were run to measure key performance indicators. Most measurements were made using the reference configuration depicted in Figure 4. Two experiments were done using the configuration for multiple consumers, shown in Figure 6. These are both high-stress configurations because they operate in a trace mode (see the discussion in Section 6). Because they are high-stress configurations, and because the reference implementation has not been heavily optimized, the observed results probably represent a lower bound on what an application would experience when running in a production environment.

The typical configuration in production settings, which updates statistical summaries instead of processing trace-level records, is best approximated by the lighter loaded cases (e.g., under 1000 transactions/second).

7.1 System Configuration

The application was simulated by a driver program running in many threads (10-1000). All other processing not related to ARM was minimized. Executing transactions were simulated using the Java sleep() method. This is the type of behavior one would see in an application blocked and waiting for a remote system to process a child transaction before the application can continue. This combination of many threads waiting for transactions to complete on remote systems most closely emulates a mid-tier application server that makes remote calls to other (data or application) servers.

The consumer either did no processing except to get a record from the FIFO queue and put it back in the pool of available records, or it also wrote the serialized data from each record to a file. Both represent trace-level processing, with the difference between them being whether the individual trace records are processed in memory or serialized and written to a data store.

The same three configuration and three measurement values were reported in each experiment. The three configuration values are the number of threads, the mean response (sleep) time, and the theoretical number of transactions that would be attempted if processing cost was zero and timers 100% precise. For most

experiments, the number of threads was the only variable modified for the several runs in each configuration. This was a convenient way to vary the load on the system.

Section 4.1 describes the five record types and explained why almost all the workload in a production environment will be transaction records. All other records combined will be an insignificant part of the workload. Because of this, the measurements for this paper use only the transaction instance records.

The system used for all measurements was a Pentium® III 733 MHz desktop system, with 256MB of RAM. The software was Microsoft® Windows NT® 4.0, Service Pack 6a, and the IBM® JDK (Java Developers Kit) implementation, version 1.3. Other software on the system took less than 5% of CPU.

8 Presentation of Results

The experimental results are presented in the following sections. Section 8.1 describes a baseline for comparing the other experiments. Section 8.2 shows the impact of different choices an application developer could make. Section 8.3 shows the impact of different choices the provider of a consumer or an administrator could make.

8.1 Baseline Measurements

The baseline configuration was the configuration with one consumer shown in Figure 4. The number of application threads varied from 10 to 1000, with the mean simulated response time (sleep time) of each transaction at 10, 25, and 100 milliseconds, distributed exponentially. For all experiments each thread executed 1000 transactions. This resulted in attempted loads that ranged from 100 to 100,000 transactions/second.

The configuration was the following:

- The application used ArmRecordTran and the ArmLibrary.process() method as shown in Figure 3.
- The application had ArmRecordTran generate the handle, which uses Queue 1 in Figures 7 and 8.
- The application did not use metrics or correlators.
- The consumer put 2000 ArmRecordTran instances in the available pool (step 1 in Figure 5), expecting this to be ample to avoid making this a bottleneck.
- The consumer did minimal processing and no I/O.
- The thread priorities for the applications and the consumer were the same.

Tables 1-3 show the results of the baseline for the 10, 25, and 100 millisecond response time cases, respectively.

Threads	Resp Time (msec)	Theor. trans per sec	Actual trans per sec	Appl Latency (msec)	% meas dropped
10	10	1000	1003	0.013	0 %
25	10	2500	2417	0.012	0 %
50	10	5000	4751	0.015	0 %
100	10	10000	9402	0.018	0 %
250	10	25000	17768	0.330	0 %
500	10	50000	17974	1.543	2 %
1000	10	100000	11525	8.955	45 %

Table 1. Baseline measurements using 10 millisecond mean response time

Threads	Resp Time (msec)	Theor. trans per sec	Actual trans per sec	Appl Latency (msec)	% meas dropped
10	25	400	393	0.012	0 %
25	25	1000	946	0.008	0 %
50	25	2000	1868	0.007	0 %
100	25	4000	3733	0.007	0 %
250	25	10000	9236	0.032	0 %
500	25	20000	16704	0.914	0 %
1000	25	40000	11066	7.983	39 %

Table 2. Baseline measurements using 25 millisecond mean response time

Threads	Resp Time (msec)	Theor. trans per sec	Actual trans per sec	Appl Latency (msec)	% meas dropped
10	100	100	97	0.010	0 %
25	100	250	234	0.006	0 %
50	100	500	463	0.004	0 %
100	100	1000	926	0.004	0 %
250	100	2500	2310	0.003	0 %
500	100	5000	4614	0.019	0 %
1000	100	10000	8901	0.063	0 %

Table 3. Baseline measurements using 100 millisecond mean response time

8.1.1 Throughput in transactions/second

Transactions/second are a measure of the throughput of the combination of the driver program emulating applications, the ARM implementation, and the consumer.

The maximum transaction rate reached peaks of about 18,000 and 16,700 transactions/second in the 10 and 25 millisecond cases. At 1000 threads in the 100 millisecond case congestion had not yet occurred and presumably a peak had not yet been reached either. Informal observations of CPU utilization levels were about 95% when the maximum rate was reached, indicating that the system had become CPU-bound.

Comparing the theoretical attempted and actual transactions/second shows that the ARM reference implementation passed 90-100% of the offered transactions for rates up through about 10,000 transactions/second. Above that level the percent throughput began to fall (84% at 20,000/second in the 25 millisecond case and 71% at 25,000/second in the 10 millisecond case). This would be expected because all the threads were competing for CPU resource.

In all three cases the percent of theoretically attempted transactions that were actually completed and processed decreased slightly as the loads increased. This is probably due to the increased number of context switches and blocking of threads that occur as the number of threads increased.

8.1.2 Application Latency

The application latency is a measure of the additional delay incurred by an application when it uses ARM. It consists of the time the application takes to populate measurement objects plus the time the application is blocked waiting for the ARM implementation to process them (steps 1-3 in Figure 8). It does not include the time that a copy of the object spends in ArmQueue and being processed by the consumer (steps 4-6 in Figure 8), except indirectly because the consumer in steps 4-6 is using CPU cycles that otherwise would be available to the application.

To measure the latency, the system timer value was captured before preparation of the measurement object started and after it was processed, and the difference calculated. Because the resolution of the timer is 10 milliseconds, the reported results are the statistical mean over 10,000 to 1,000,000 transactions, depending on the number of threads running. For example, a reported mean of 0.010 milliseconds would be the result if one transaction was measured at 10 milliseconds and 999 were measured at 0 milliseconds (because the timer happened to roll over during one of the actual 0.010 millisecond periods). An empirical test comparing the sum of 1,000,000 intervals of about 10 microseconds each with the elapsed time over all 1,000,000 intervals demonstrated that this technique yields a correct result. An unfortunate side effect of measuring latency by making system timer calls is that this excess processing distorts measurements, especially when ArmTransaction is used (see section 8.2.3).

Up to 5000 transactions/second the latency observed by the application ranged from .003 to .019 milliseconds. As a percentage of the transaction response time, it varied from 0.003% to 0.2%. As the load increased through 10,000 transactions/second latency began to grow

substantially. Why the latencies in the 25 and 100 millisecond cases decreased as the transaction rate increased to about 3,000 transactions/second before increasing is not well understood; it could be related to not getting the system to a steady state before taking measurements.

The latency increased as the transaction load per thread increased, and when the number of threads increased. For example, at a throughput of 10,000 transactions/second, the latencies were 0.018, 0.032, and 0.063 milliseconds for corresponding thread counts of 100, 250, and 1000. Similarly, for 100 threads the latencies were 0.018, 0.007, and 0.004 milliseconds for corresponding transaction loads of 10000, 4000, and 1000 transactions/second. These results are not surprising because contention for the queues would be expected to increase with increases in threads and/or transaction rates.

At very high attempted transaction rates, congestion built up, and the application latency increased dramatically. This type of increase is acceptable in a reference implementation like the one tested because it is not intended to support extremely high transaction rates, and because it always runs in trace mode. In a commercial implementation substantial increases in application latency under congestion conditions would almost certainly not be acceptable. Such an implementation would need to regulate the amount of tracing that it attempted so the latency visible to applications remained consistently low.

8.1.3 Percent of Dropped Measurements due to Congestion

Section 5.2.1 explained that if transactions are completing faster than the ARM implementation and consumer together can process the transaction records, the application is not blocked. Instead, the measurement is discarded so the application is not delayed. Therefore, the number of discarded measurements is an indicator of congestion within the ARM implementation and/or consumers.

Up through 10,000 transactions/second there was no loss of data and at least 89% of the transactions that theoretically could have been processed were processed. As congestion built up and the maximum transaction rate could not keep up with the theoretical targets, dropped measurements can be seen (in the 10 and 25 millisecond cases). This behavior is consistent with the design philosophy that it is better to lose measurements than to delay the application (though in this implementation the application delays were also unacceptably high).

8.2 Business Application Choices

ARM has optional capabilities that applications may choose to use. This section discusses the impact on the baseline performance of three of those possibilities. Three of those possibilities are:

- Should correlators be used?
- Should metrics be used?
- Which is better to use, ArmTransaction or ArmRecordTran?

Each of these questions is discussed in a following section. Figures 10 and 11 summarize the impact of these choices on the measured performance of the reference implementation.

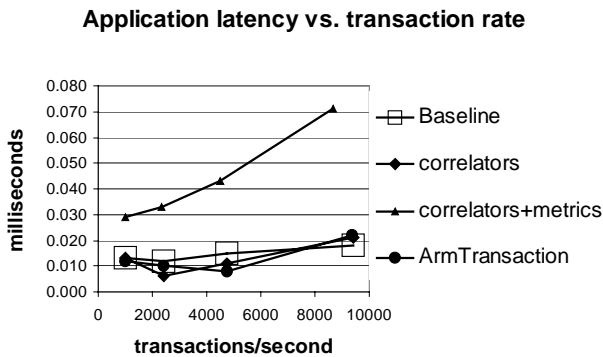


Figure 10. Measuring impacts on application latency

Figure 10 shows the impact on application latency from different implementation decisions. Generally, as the number of transactions that execute increases the observed latency also increases slightly (moving left to right along a curve). More interesting is the comparison between the curves. Three of the curves are equal for all practical purposes, and one has much higher latency. The one with higher latency occurs when the optional metric fields are being used, discussed in section 8.2.2.

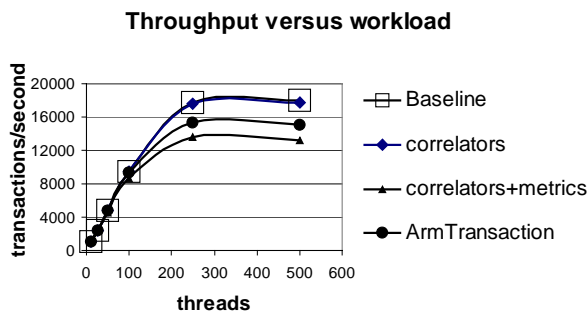


Figure 11. Measuring impacts on measurement throughput

Figure 11 shows how actual throughput of measurement records increases as the number of threads executing

transactions increases, until a peak rate is reached. As the attempted throughput increases past the peak, the achieved rate starts to decrease, presumably due to an overloaded CPU combined with many context switches and thread blockings from the many threads that are executing. Similar to Figure 10, the worst performance is seen when metrics are used. In this case there was a modest penalty for using ArmTransaction. This is discussed in section 8.2.3.

8.2.1 Using Correlators

A correlator is a unique token of work for each transaction instance. Correlators are used to relate parent and child transactions to each other. For example, a client transaction is the parent of a transaction it invokes on an application server. The transaction on the application server is a parent to transactions it invokes on a data server. Each transaction instance has two associated correlators – its own and its parent's. A correlator is typically about 50 bytes long, depending on the length of the system address or name, and is represented in the Java language as a byte array.

The additional processing cost when using correlators consists of generating the correlator for this transaction, plus copying the correlator and the parent correlator into the transaction record. The application also has to send its correlator to its child transactions, which was not measured in this experiment. Any processing of correlators by, for example, a consuming agent, was also not measured.

Figures 10 and 11 show that the delta cost from the baseline when using correlators is insignificant.

8.2.2 Using Correlators and Metrics

Metrics are optional data values that applications can provide about the transaction or the execution environment. The experiment used all seven metrics, including a mix of string and numeric values, setting the values for each transaction instance. Applications have the option of using zero to seven metrics.

Figures 10 and 11 show that there is a significant processing cost when using metrics. This can be explained by the additional processing required for storing values into the seven fields, plus setting a boolean flag for each field indicating that a value is valid. These fourteen additional set operations more than double the number of fields that must be processed. Presumably the cost would decrease if fewer than seven metrics were used.

8.2.3 Using ArmTransaction and start/stop

In the baseline case the application created instances of ArmRecordTran, measured the response time itself, and stored the response time and other values into the ArmRecordTran instance, then passed it to ArmLibrary. This is the mechanism shown in Figure 3.

The other way to provide the information is to create instances of ArmTransaction, and call start() and stop() just before and just after the transaction executes. This is the mechanism shown in Figure 2. Section 3 discussed the advantages and disadvantages of each approach. For the test application there was no functional difference between using one versus the other.

Figure 10 shows that there was not a substantial difference in application latency between using one versus the other. Figure 11 shows a decreased throughput of about 15%. This difference is not well understood, especially because ArmLibrary converts an ArmTransaction into an ArmRecordTran and the processing is identical after that. It may be an experimental error. As described in section 8.1.2, a system call to get the timer value is made before and after each call to the ARM implementation. When using ArmTransaction, this results in six system timer calls per transaction instance. The application makes four to measure application latency, one before and one after each start and stop method calls. ArmTransaction makes two to measure transaction response time, one during each start and stop call. When using ArmRecordTran in the baseline, three calls are made per instance. The extra three calls, on top of the extra call in ArmRecordTran to measure application latency, result in excess load, which may in turn result in congestion that impacts throughput. The lack of impact on application latency could be because the latency is measured with these calls. If time allowed this hypothesis would have been tested with another type of instrumentation, such as a profile interface in the JVM.

8.3 ARM Configuration Choices

ARM as a standard defines the interface between the business application and the ARM implementation. This was discussed in sections 3 and 8.2. There is no standard for an ARM implementation, or any other interfaces. The consumer interface in the reference implementation, using ArmQueue and ArmPool, is a potential standard for a cross-thread interface to pass measurement objects, such as ArmRecordTran.

This section explores some possible implementation configurations. The baseline at higher transaction rates

approximates an implementation in trace mode using in-memory analysis in the consumer.

- Section 8.3.1 explores the impact of a consumer writing every trace record to a file.
- Section 8.3.2 explores the impact of using a dispatcher running in its own thread to distribute transactions to one consumer (Figure 6 except using one consumer).
- Section 8.3.3 explores the impact of using a dispatcher with three consumers (Figure 6 except using three consumers).

Figure 12 contrasts the impact on application latency for these cases. Degradation occurs as the load on the system increases, and is most noticeable with three consumers and when writing every record to a file.

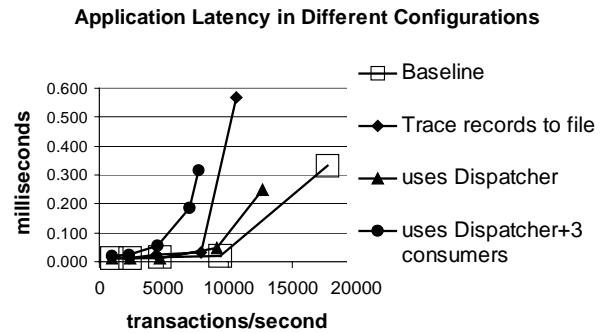


Figure 12. Application latency in different configurations

Figure 13 contrasts the impact on throughput for these cases. Each of these cases perform substantially worse than the baseline, and is most noticeable with three consumers and when writing every record to a file.

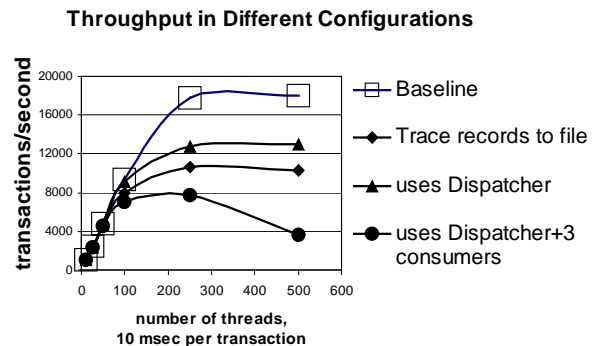


Figure 13. Comparing ARM implementation for throughput

The typical configuration of a production ARM implementation was not tested. The measurements in this section should be treated as an extreme lower bound (where high means better) compared to a production implementation. This is because a production ARM

implementation would do much less processing and have less contention for queues, as discussed in section 6. The latency figures for the 100-millisecond case (Table 3) at lower transaction rates (because these minimize queuing delays) are the closest approximation of a typical production ARM implementation.

8.3.1 Writing trace records to a file

Writing trace records to a file didn't have much impact at transaction rates below 10,000 transactions/second. Above 10,000/second the degradation was dramatic on both latency and throughput. The throughput levels out at about 55% of the baseline case.

This is not surprising given the nature of I/O operations. At lower workload levels these occur in parallel in when the CPU is otherwise not in demand. At higher rates they are competing with the application and other processing for the CPU, and drag down all processing on the system.

8.3.2 Using the Dispatcher

Using a dispatcher adds three additional queues to Figure 8. These would be the same as the queues labeled 4-6, except between the dispatcher and the consumer, rather than the application and the consumer. This processing is done in parallel with the application and ArmLibrary processing, so at lower loads it would not be expected to impact latency or throughput significantly. At higher loads it would consume CPU resources needed elsewhere.

The data bears out this expectation. Response times are close to the baseline up to 10,000 transactions/second. Above that there is some degradation, though it isn't overly dramatic. Throughput levels are consistently below the baseline case, tailing off at about 70% of the baseline case above 10,000 transactions/second. There are no sudden changes in the curves. This is consistent with an impact due to general loading, as opposed to the sharp elbows that may occur looking at service times in an overloaded queue.

8.3.3 Using the Dispatcher + Three Consumers

Using a dispatcher plus three consumers, all in a trace mode, is an extreme case. The typical deployment will rarely be in a trace mode. Many configurations would use only one consumer. Some would use a second consumer, but only for selected applications or at selected times. Perhaps the normal mode is a service level monitoring agent in a configuration like Figure 9, and occasionally a sampling of records are passed to a second consumer. Nevertheless, testing this configuration can confirm some hypotheses.

There is substantial degradation in both application latency and throughput. Although the processing is done in parallel with the application, the dispatcher and additional consumers consume substantial resources, and all programs suffer.

9 Conclusions

ARM 3.0 for Java is a good fit for measuring business transactions and significant child transactions. This conclusion is based on the ability to measure transactions for a cost of a few microseconds in additional latency under normal conditions, as shown in Tables 1-3 in section 8.1.

If an ARM implementation uses the system timer to measure transactions, which the reference implementation described in this paper does, the value of measuring transactions that have a response time much less than the granularity of the system timer (10 milliseconds in this configuration) needs to be considered carefully. As long as statistical summaries are being created, the response time mean (though not the standard deviation) of transactions with shorter response times will be accurate. Trace data of each measured transaction will appear erratic, with values at quantum levels, such as 0 milliseconds or 10 milliseconds. An advantage of using the system timer is that the resulting implementation can be fully portable to any JVM implementation.

Application developers should use correlators whenever reasonable. This conclusion is based on the fact that section 8.2.1 showed that using correlators resulted in insignificant changes in application latency and processing overhead. Given the advantages of using correlators to understand where transactions are hanging or being slowed down, the slight cost to using them is justified in almost any scenario involving distributed transactions.

Application developers should weigh the benefits of using metrics. They contribute to substantial increases in application latency and processing, though to be fair, an increase of, say, 20 microseconds for a transaction that has a response time of tens or hundreds of milliseconds would probably be considered insignificant.

Applications developers should choose between using ArmTransaction versus ArmRecordTran objects based on the operating environment and the type of transaction. The difference in performance is negligible. For example, for long-running transactions it would be useful to use ArmTransaction so heartbeats could be sent. If it is impractical to call ARM at the moment the transaction

starts and stops, the developer should use `ArmRecordTran`.

An ARM implementation can be built that processes several thousand measurements per second without undue loading of a system if it creates statistical summaries of transactions.

Tracing of individual transaction records, especially if they are written to a file, requires enough system resource that it should be used judiciously. Examples of judicious use are collecting trace data on a sampled basis and turning on tracing for specific transactions rather than for all transactions.

Using multiple consumers entails a significant cost and should only be used when there is a need. The comments about judicious use of tracing apply to using multiple consumers. An example of a judicious use would be to send data for only a certain transaction type to a second consumer.

In all the experiments a maximum throughput was observed on a system doing little except processing ARM transactions. This suggests that the throughput can provide a crude estimate of the percent of system CPU required to trace a transaction with ARM. (Collecting statistical data would require less). For example, in the baseline case in 8.1, the maximum throughput was about 18,000 transactions per second. The CPU was close to 100% busy, suggesting that tracing 100 transactions per second would require about 0.5% of the CPU on this class of system. This estimate is only valid for ARM implementations similar to the reference implementation described in this paper.

It's worth noting that the apparent "cost" of using ARM may actually be a savings from a systemic perspective when compared to using external probes. If a probe has to intercept many API calls or packets or operating system messages, filter them, analyze them to see if there are any matches with known signatures, and finally process the data if any is found, the total resource consumption may be substantially higher than calling ARM once or twice per transaction.

10 Summary

This paper described ARM 3.0 for Java and presented measurements of its impact on system overhead and application latency for a reference implementation. The creators of ARM have positioned it as a useful and reasonable way to measure moderately coarse-grained units of work, such as business transactions or server transactions. Based on the observed data in the tested configuration, we concur with this positioning. We also

believe that it is reasonable to provide an ARM implementation that is highly if not completely portable from one system platform to another, while still performing at a level that would be satisfactory to system administrators.

11 References

[ARM97] "Application Response Measurement 2.0 API Guide", available from "<http://www.cmg.org>".

[HELL99] P Heller and S. Roberts, "Java™ 2 Developer's Handbook, SYBEX Inc.; ISBN: 0-7821-2179-9 (1999), pages 283-354.

[JOHN97] M. Johnson, "The Application Response Measurement (ARM) API, Version 2", CMG97 Proceedings.

[TOG98] Open Group Technical Standard, C807 ISBN 1-85912-211-6 July 1998 78 pages.

12 Acknowledgements

IBM is a trademark or registered trademark of International Business Machines Corporation in the United States and other countries.

Java is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries.

Microsoft and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Pentium III is a registered trademark of Intel Corporation in the United States and other countries.