

# Session 3105

## Measuring the Performance of ARM 3.0 for Java™

Mark W Johnson

Tivoli Systems

# Agenda

- ◆ Objectives

- ◆ ARM Background

  - ARM = Application Response Measurement

  - ARM is a standard of The Open Group

- ◆ ARM Implementation Requirements

- ◆ ARM 3.0 for Java™ SDK Reference Implementation

- ◆ Performance Measurements

- ◆ Conclusions

# Objectives

- ◆ **Introduce ARM 3.0 for Java**
- ◆ **Describe a reference implementation and some factors that affect its performance**
- ◆ **Show how this implementation may compare to common commercial implementations**
- ◆ **Present measurements of the performance of the reference implementation**
  - Useful as a first-order assessment**
  - Your mileage may vary with a different implementation...**
- ◆ **Suggest insights for application developers and system administrators**

# **ARM Background**

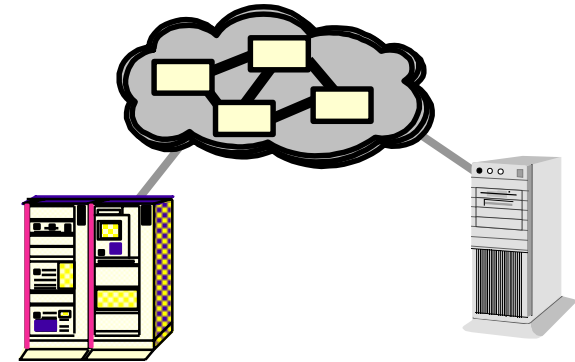
➤ **Overview**

➤ **History**

# ARM focuses on the user experience



**ARM**

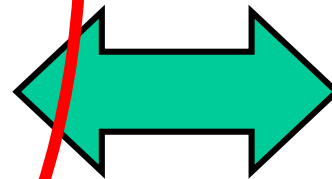


## User Transactions

Transactions  
completing?  
Response time?  
Transactions/sec?  
Which transactions?

## Infrastructure

Server up?  
Application running?  
Load on server?  
Load on network?  
Additional capacity?



# History

## ◆ ARM 1.0

Released June 1996 by Hewlett-Packard and Tivoli

## ◆ ARM 2.0

Released December 1997

Developed by the ARM Working Group (~15 companies, consisting of product vendors and end-users)

Adopted by The Open Group July 1998

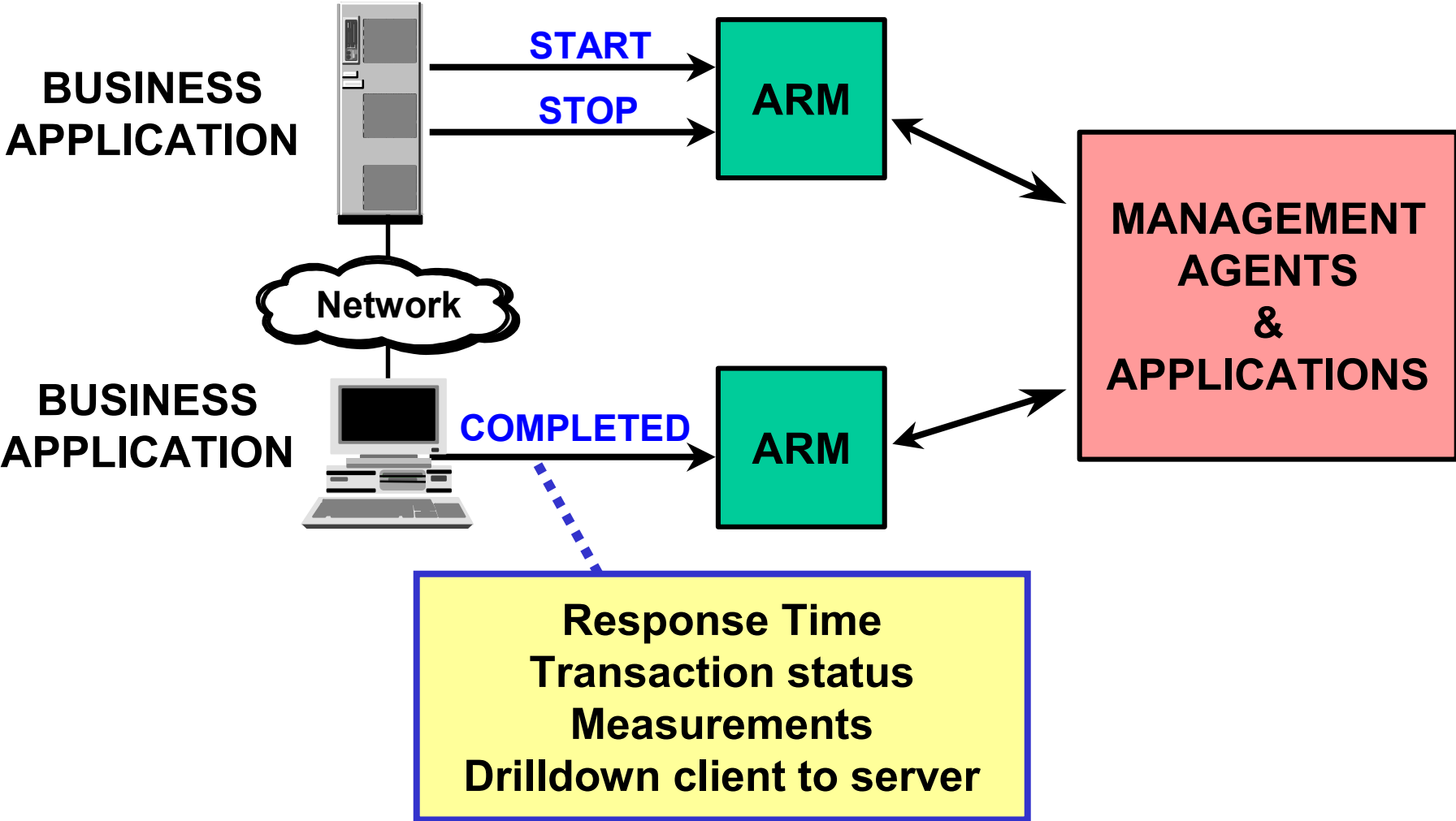
## ◆ ARM 3.0

C API drafted 1998-1999. Not finished or implemented.

Java interface proposed and implemented November 2000

Standardization will occur in The Open Group

# ARM measures business transactions

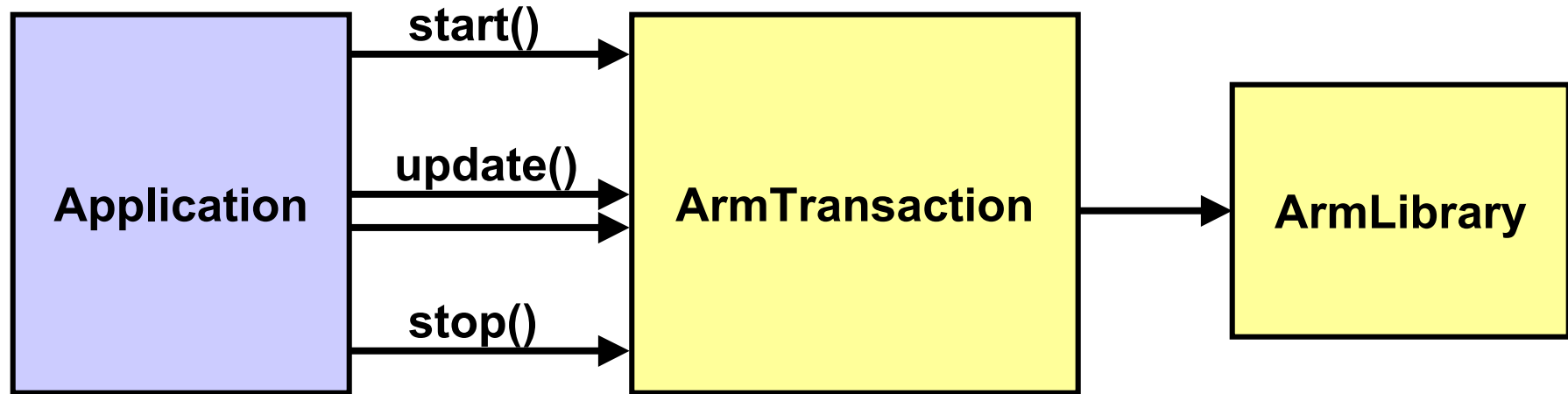


# What is an **ARM 3.0 transaction**?

- ◆ 16-byte UUID identifies the type
- ◆ 8-byte handle identifies instance in local namespace
- ◆ ~50-byte token (“correlator”) identifies instances in global namespace
- ◆ Measurements and other data
  - Status (good, failed, aborted)
  - Start time, stop time, response time
  - (optionally) zero to seven string or numeric metrics

**Uses DMTF’s CIM Distributed Application Performance/Metric data model**

# Measuring Synchronously

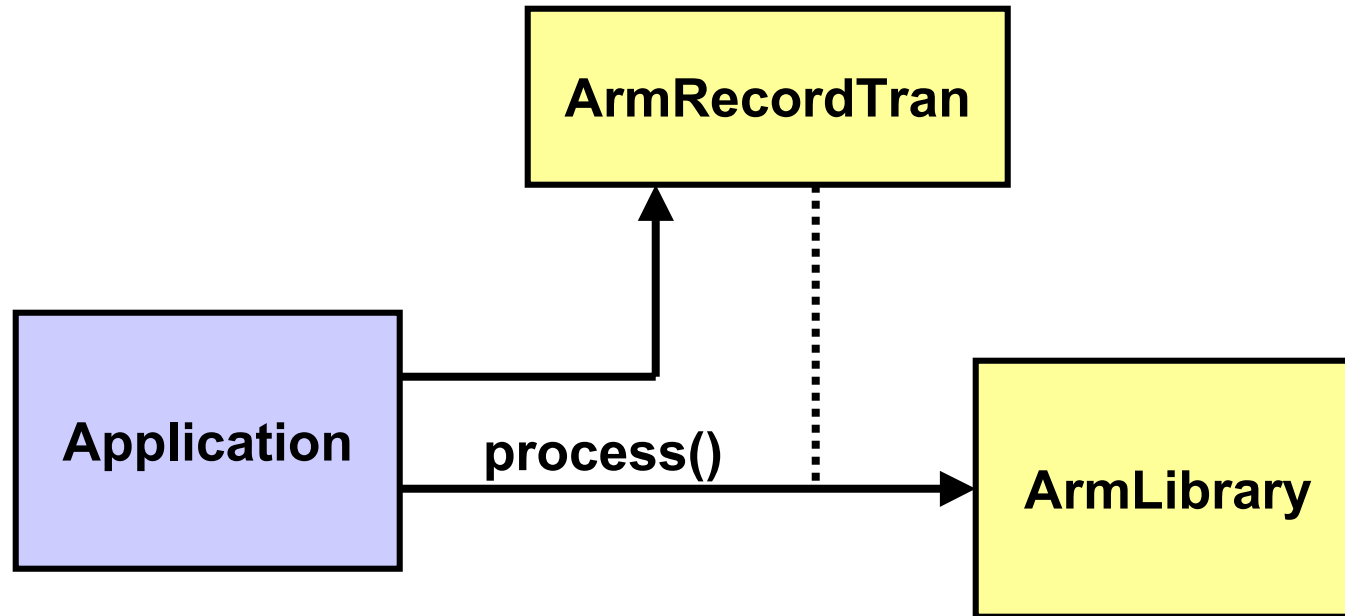


## Advantages:

**ArmTransaction can monitor for 'hung' transactions**

**Long running transactions can send heartbeats**

# Measuring Asynchronously (new in 3.0)



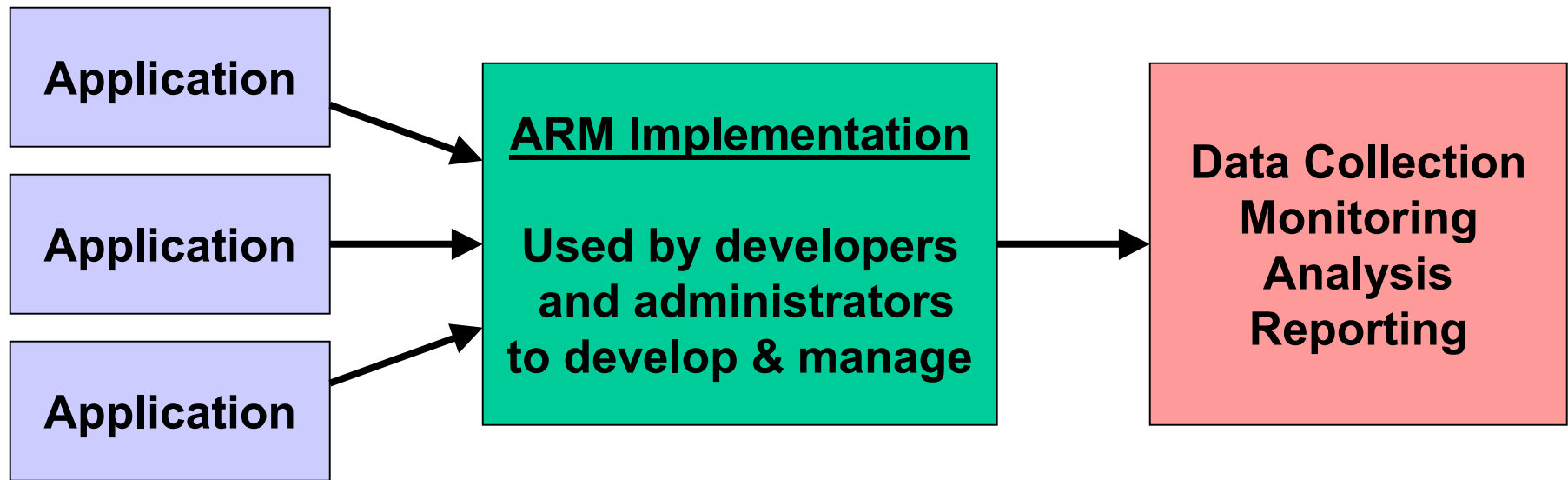
## Advantages:

**Call ARM when it suits the application (e.g., delayed)**

**Report transactions measured on other systems**

# **ARM Implementation Requirements**

# The Different Views of ARM



**Developer's  
View**

**Administrator's  
View**

# Requirements from App Developers

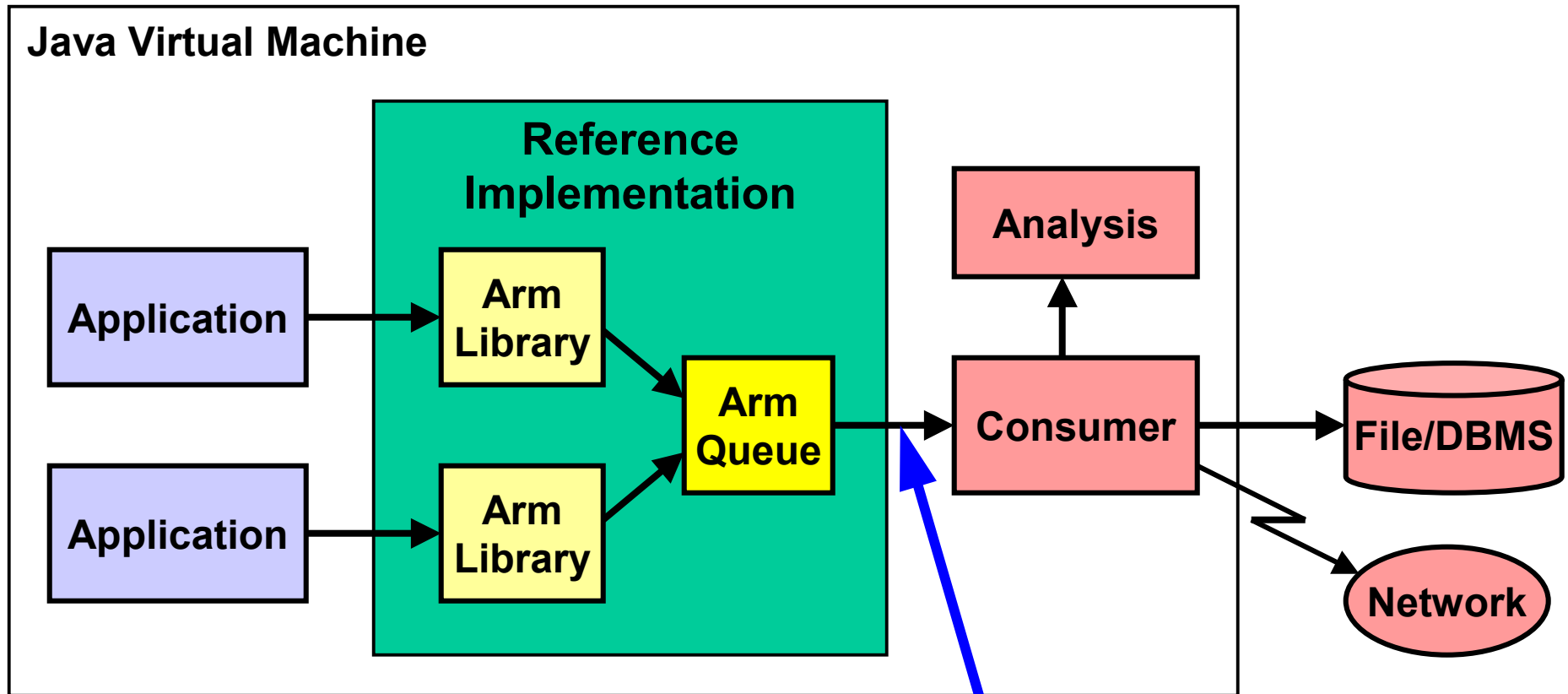
- ◆ **Simple programming model**
- ◆ **Independent of deployed management solution**
- ◆ **No impact to application reliability (from other applications or management products)**
- ◆ **Predictable and low latency for calling ARM**
- ◆ **Predictable resource consumption**

# Requirements from System Admins

- ◆ **No impact to application reliability (other applications, management products)**
- ◆ **Predictable and low latency for calling ARM**  
Drop measurements rather than delay application
- ◆ **Predictable resource consumption**
- ◆ **Maximize the number of transactions that can be measured**
- ◆ **Flexibility selecting how much management function to use**

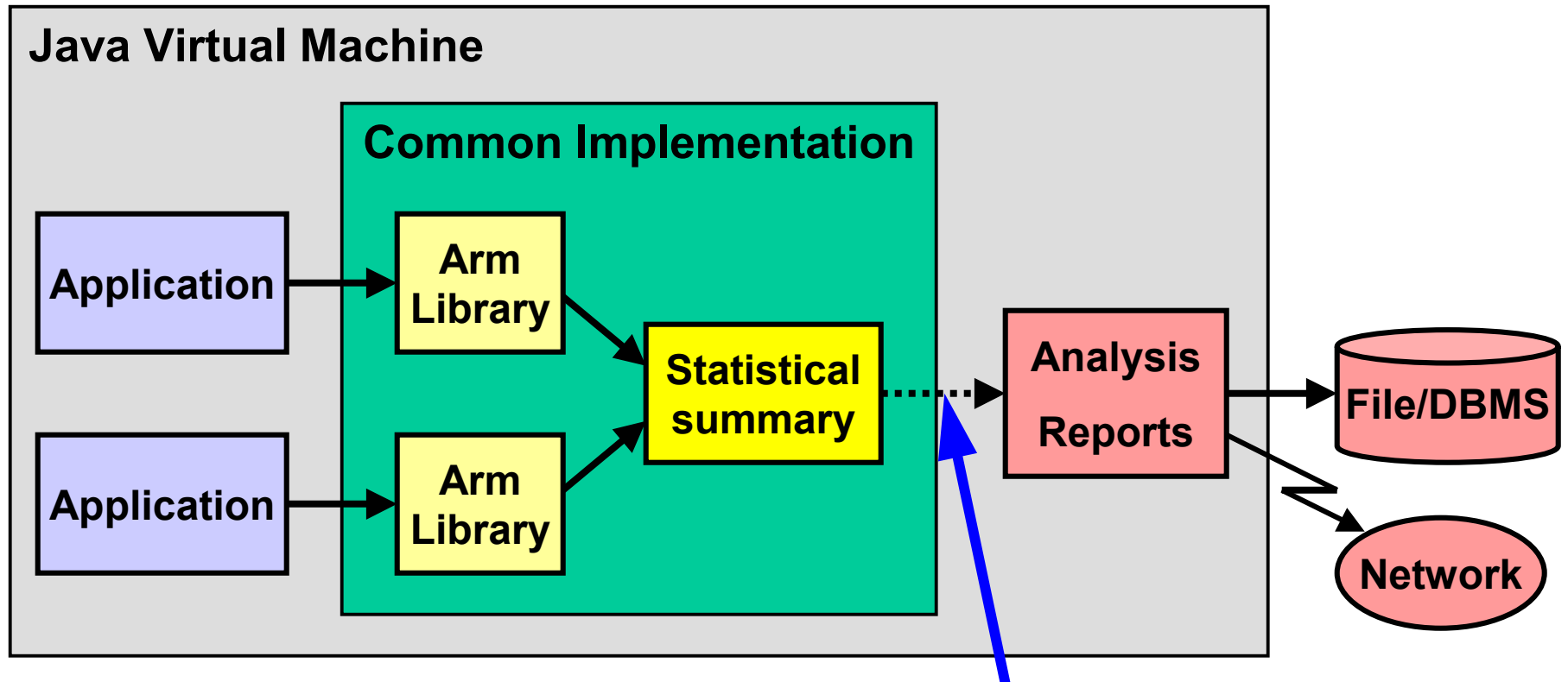
# **ARM SDK Reference Implementation**

# SDK Reference Implementation



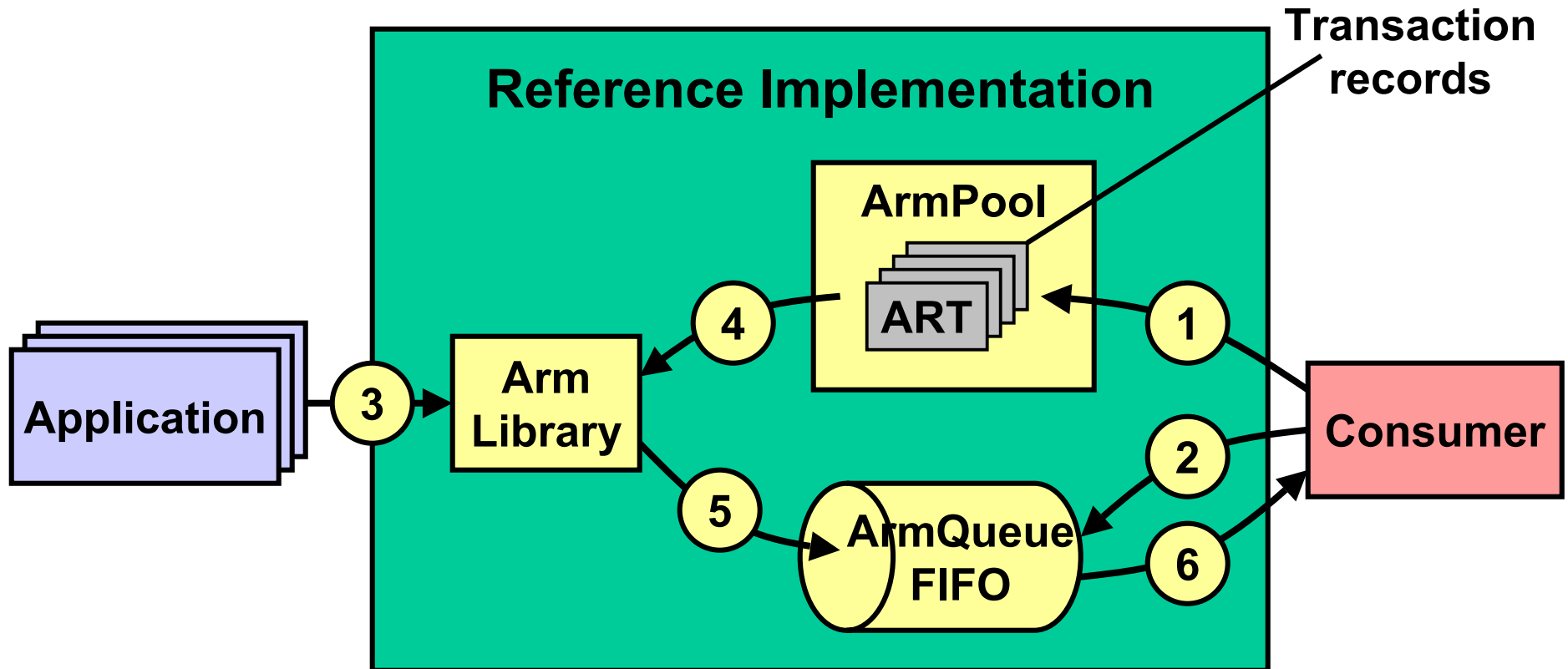
**Traces every record**

# Common Commercial Configuration



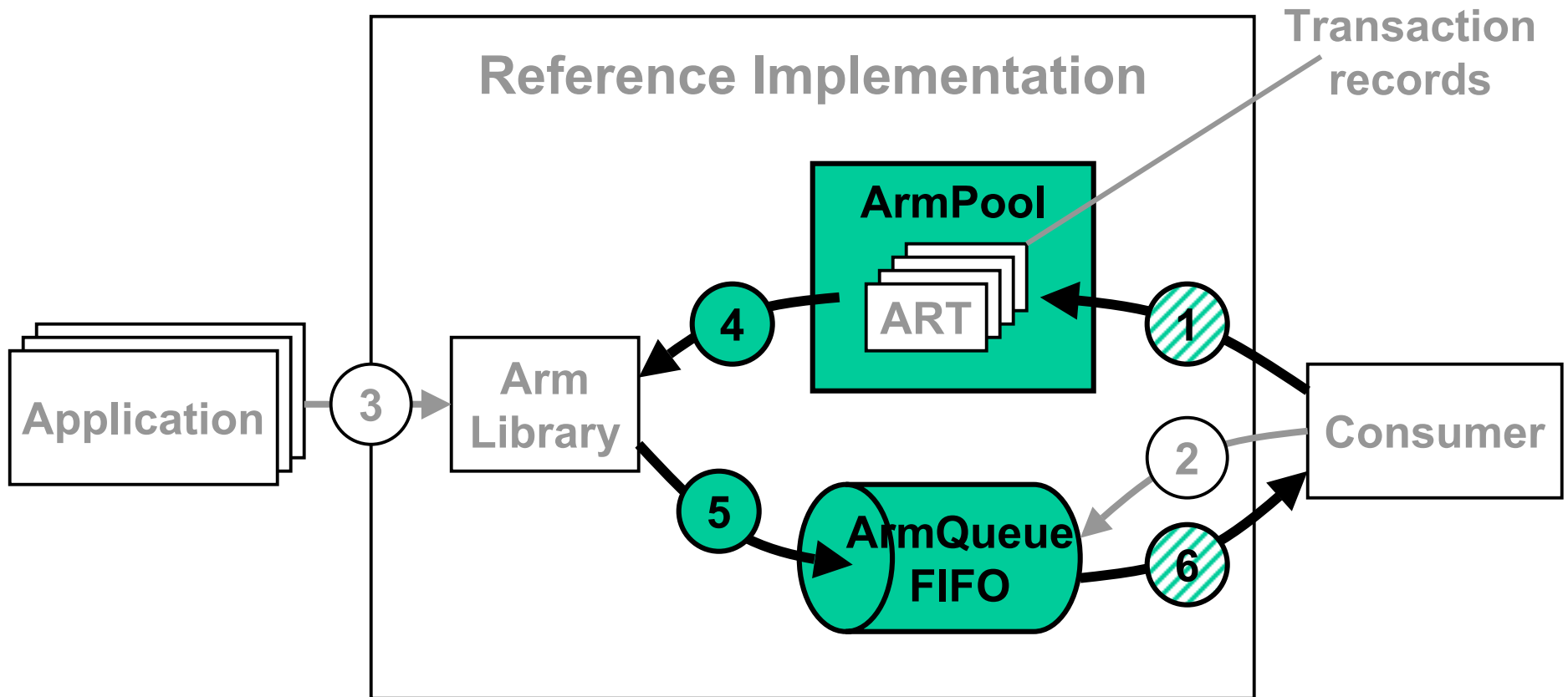
**Periodically sends summaries  
(Traces each record occasionally)**

# Data Flows in Reference Implementation

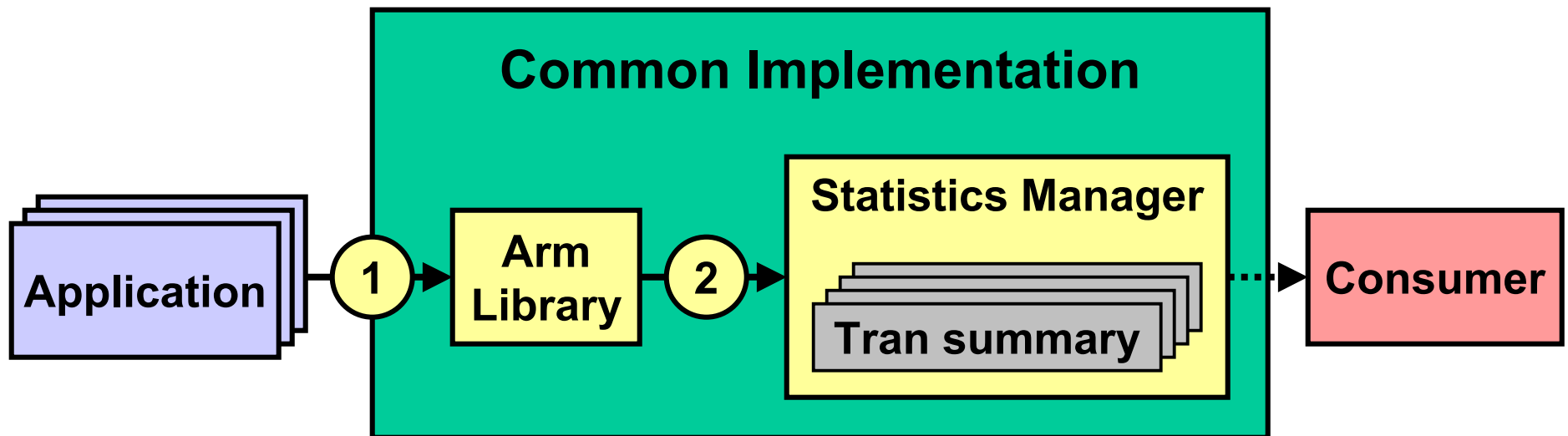


**If no record available in pool when requested, measurement is discarded**

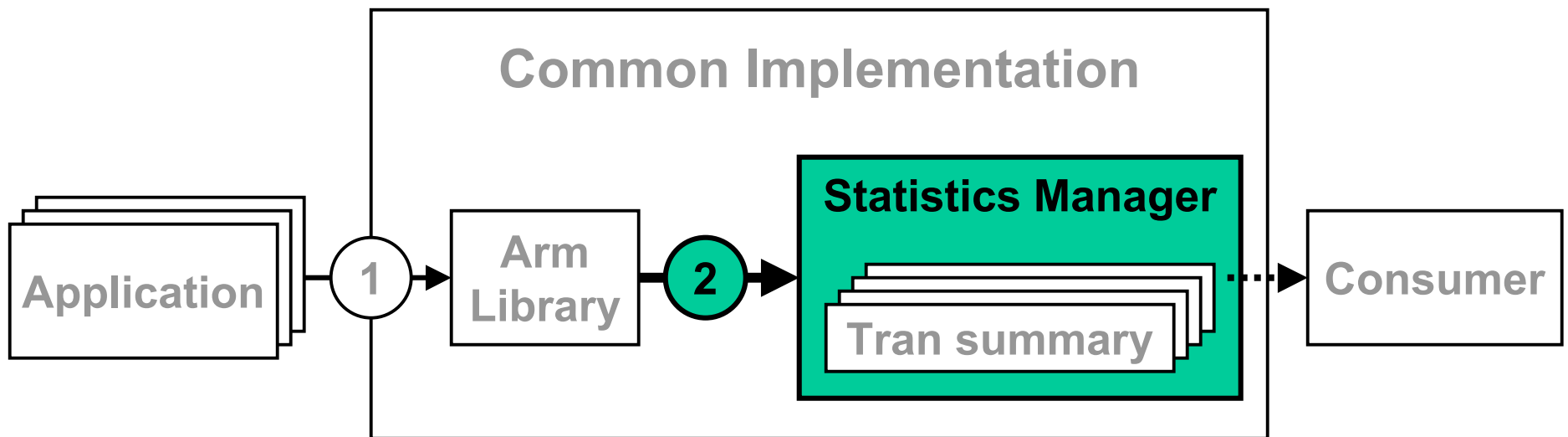
# Queuing that blocks applications



# Data Flows in Common Implementation

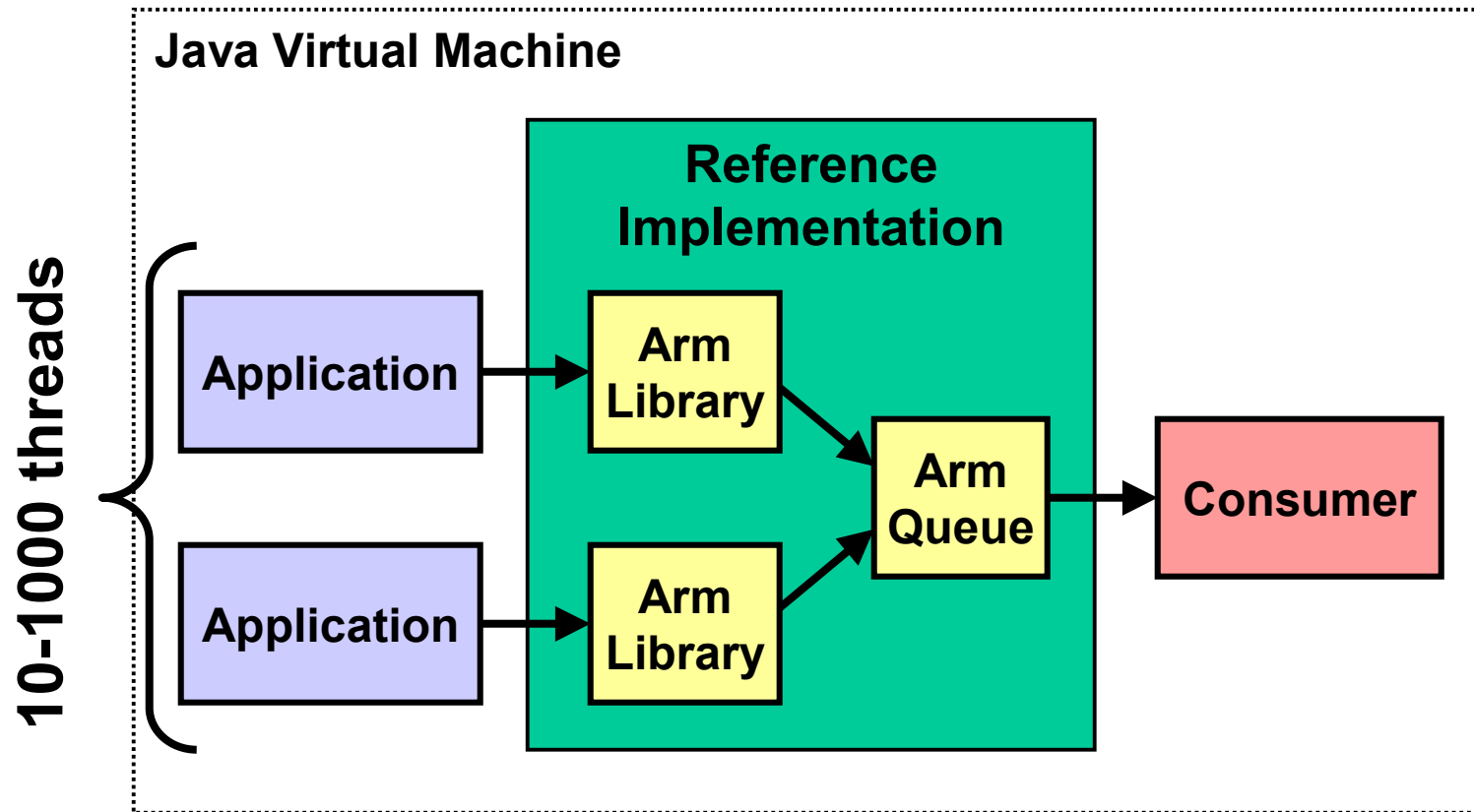


# Queues that block applications in a typical implementation



# **Performance Measurements**

# Baseline Experiment Configuration



# Baseline for each thread (10-1000)

Create transaction record

Repeat 1000 times:

Get transaction start time

Sleep (**mean=10/25/100**) msec to simulate running transaction

Get transaction stop time (& ARM processing start time)

Populate transaction record

Tran ID, Handle, stop time, resp time, user name

← Call ArmLibrary.process()

Get ARM processing stop time & sum counters

Store summaries in statistics object for report

*distributed exponentially*

*Experimental error*

# System Configuration

- ◆ **Pentium™ III desktop system**

  - 733 MHz CPU**

  - 256 MB RAM**

- ◆ **Microsoft Windows NT® 4.0, Service Pack 6a**

- ◆ **IBM® JDK (Java Developers Kit) 1.3**

  - Java programs use version 1.1**

# Some Baseline Measurements

Threads	Resp Time (msec)	Attempted trans/sec	Actual trans/sec	App Latency (msec)	% records dropped
10	25	400	393	.012	0%
25	25	1000	946	.008	0%
50	25	2000	1868	.007	0%
100	25	4000	3733	.007	0%
250	25	10000	9236	.032	0%
500	25	20000	16704	.914	0%
1000	25	40000	11066	7.983	39%

# Delta from baseline: correlators

**Create transaction record**

**Repeat 1000 times:**

**Get transaction start time**

**Sleep (mean=10/25/100) msec to simulate running transaction**

**Get transaction stop time (& ARM processing start time)**

**Populate transaction record**

**Tran ID, Handle, stop time, resp time, user name**

**correlator, parent correlator**

 **Call ArmLibrary.process()**

**Get ARM processing stop time & sum counters**

**Store summaries in statistics object for report**

# Delta from baseline: correlators+metrics

**Create transaction record**

**Repeat 1000 times:**

**Get transaction start time**

**Sleep (mean=10/25/100) msec to simulate running transaction**

**Get transaction stop time (& ARM processing start time)**

**Populate transaction record**

**Tran ID, Handle, stop time, resp time, user name**

**correlator, parent correlator  
metrics**

**← Call ArmLibrary.process()**

**Get ARM processing stop time & sum counters**

**Store summaries in statistics object for report**

# Delta from baseline: **ArmTransaction**

Create transaction record

Repeat 1000 times:

Get ARM processing start time

Populate transaction record

user name

← Call ArmTransaction.start()

Get ARM processing stop time & sum counters

Sleep (mean=10/25/100) msec to simulate running transaction

Get ARM processing start time

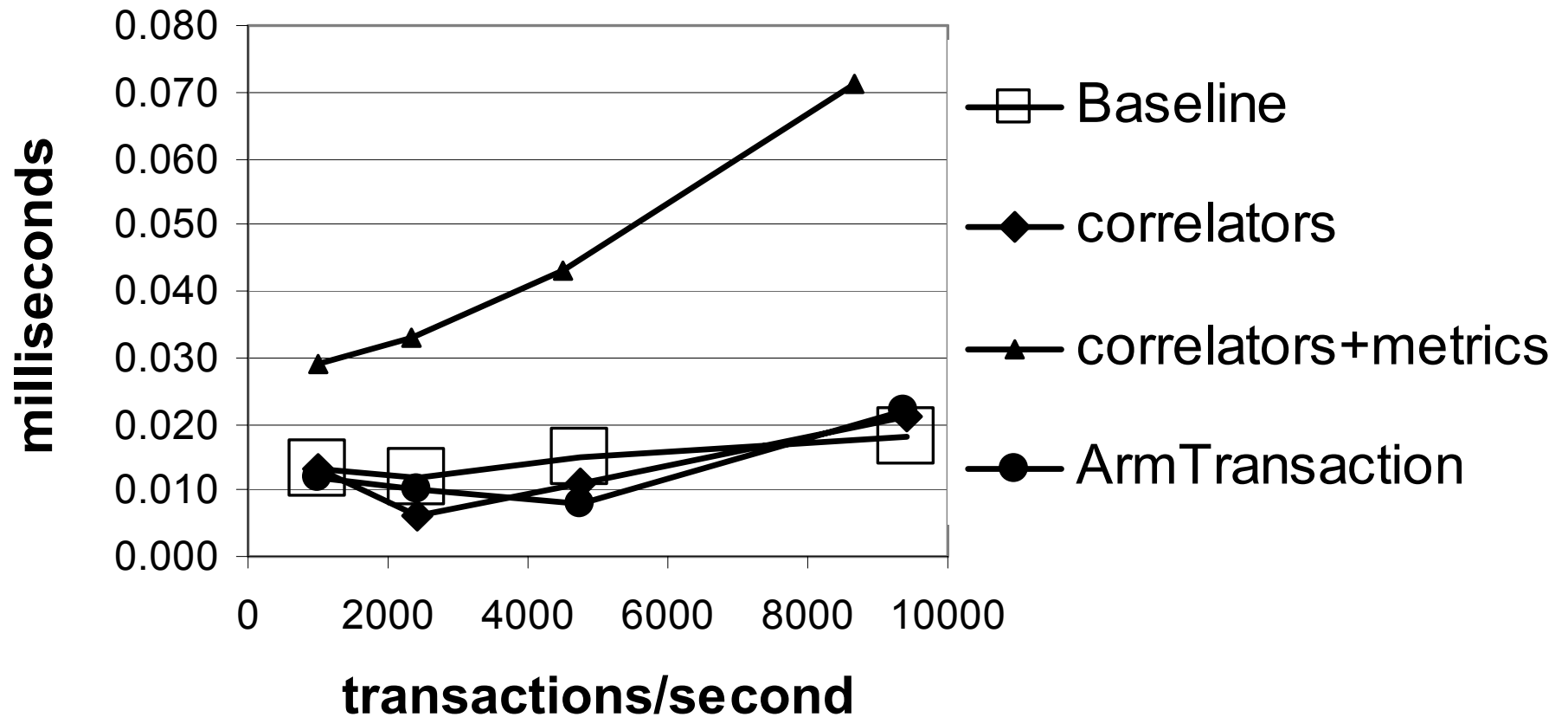
← Call ArmTransaction.stop(status)

Get ARM processing stop time & sum counters

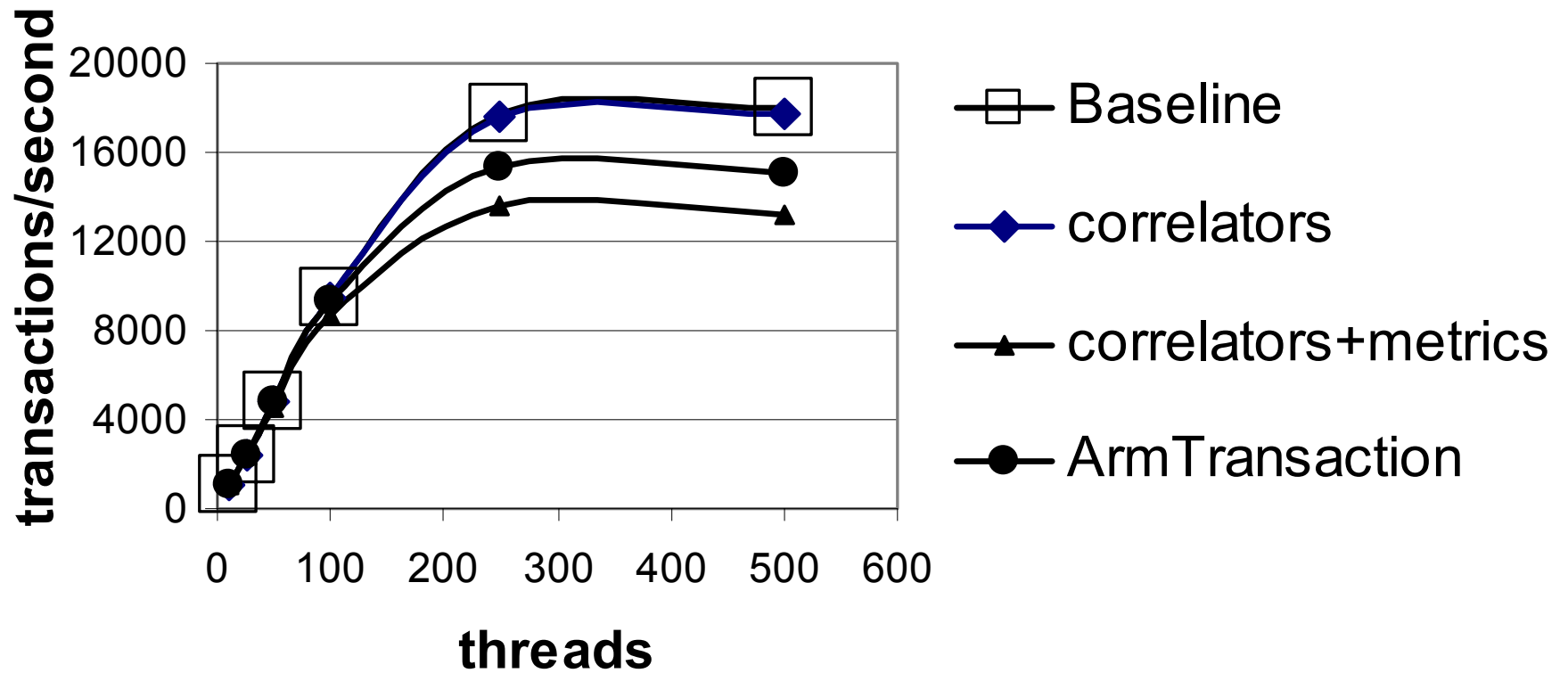
Store summaries in statistics object for report

*Experimental error*

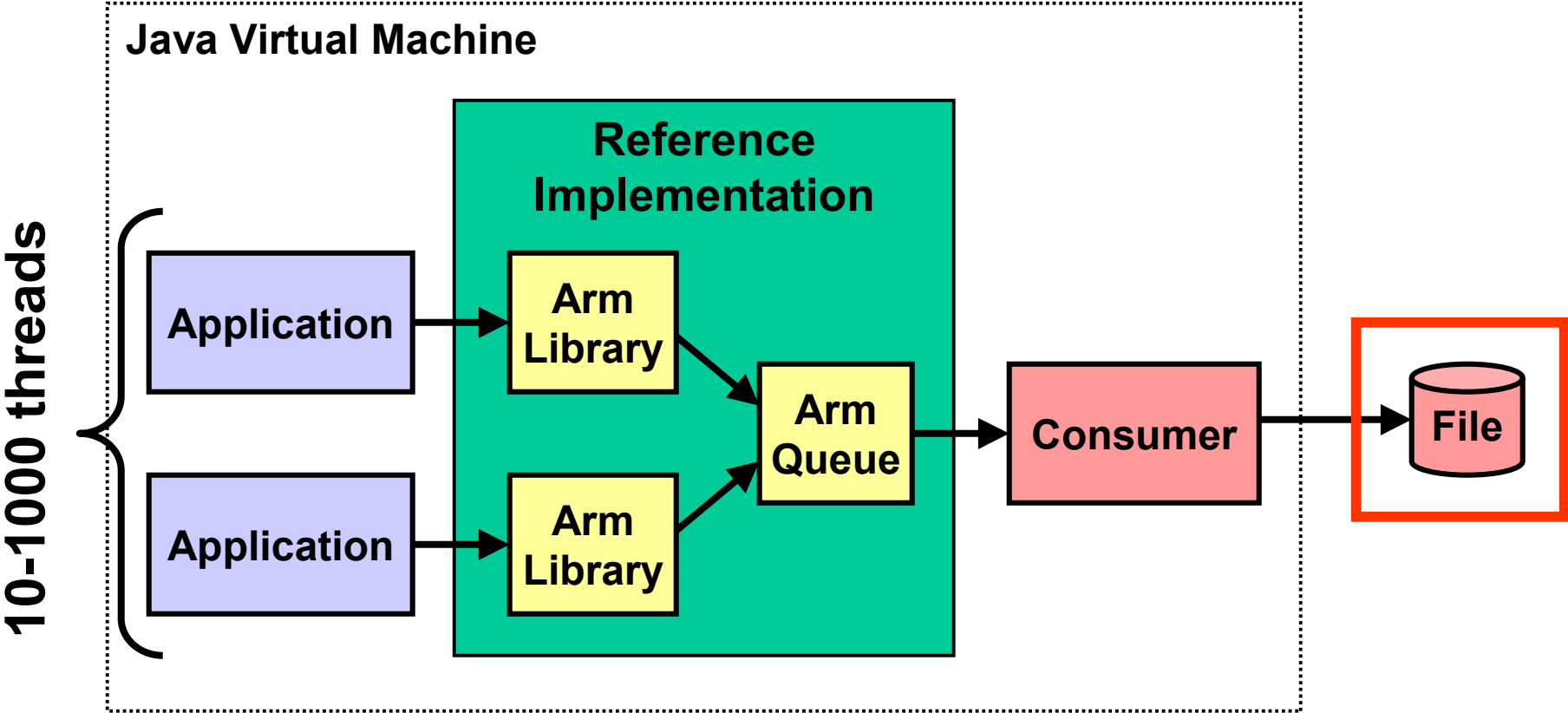
## Application latency vs. transaction rate



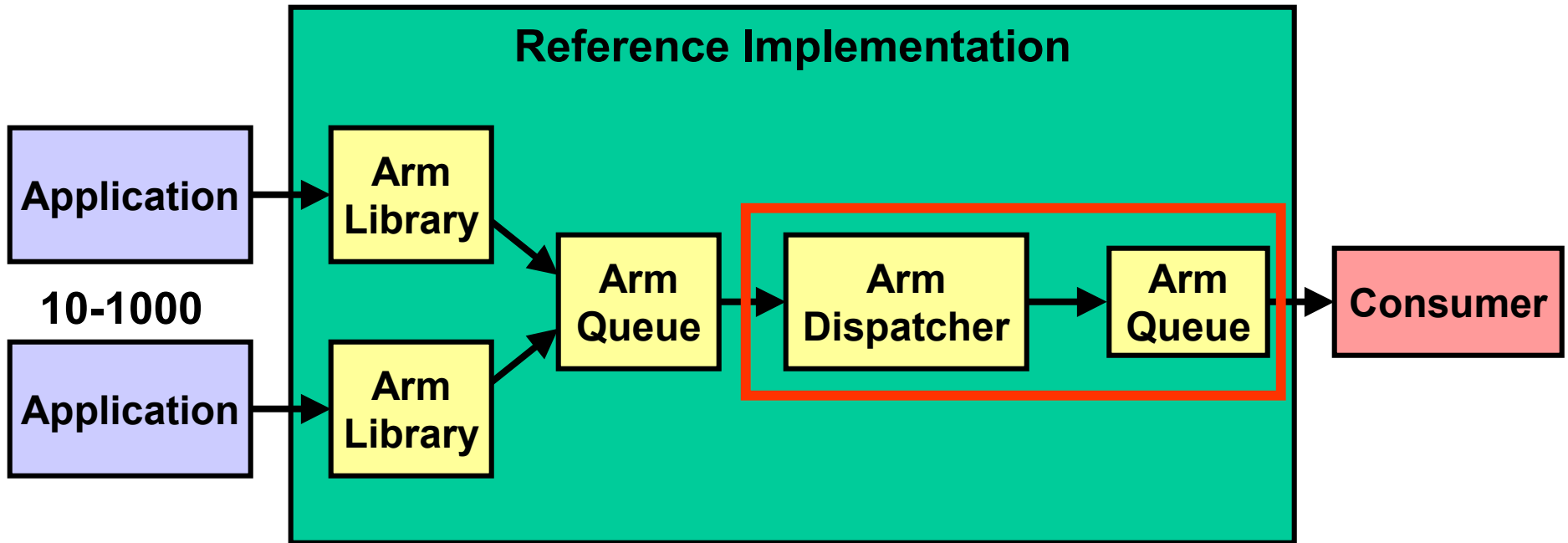
## Throughput versus workload



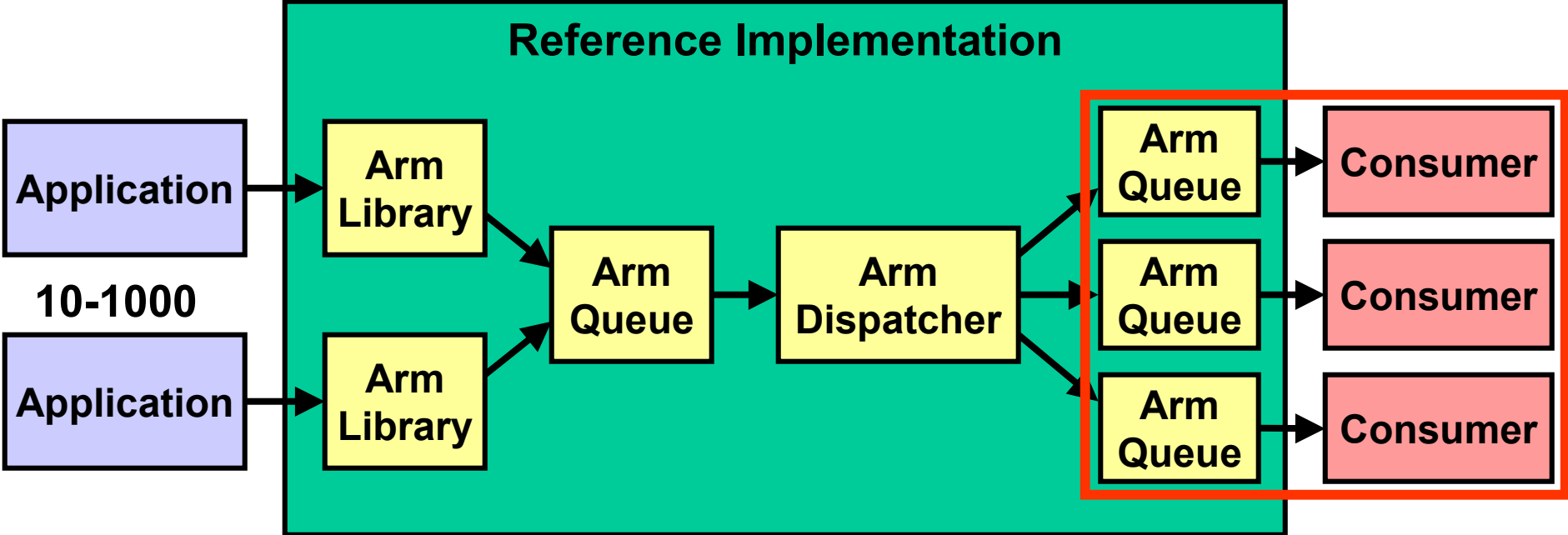
# Delta from baseline: Records to a file



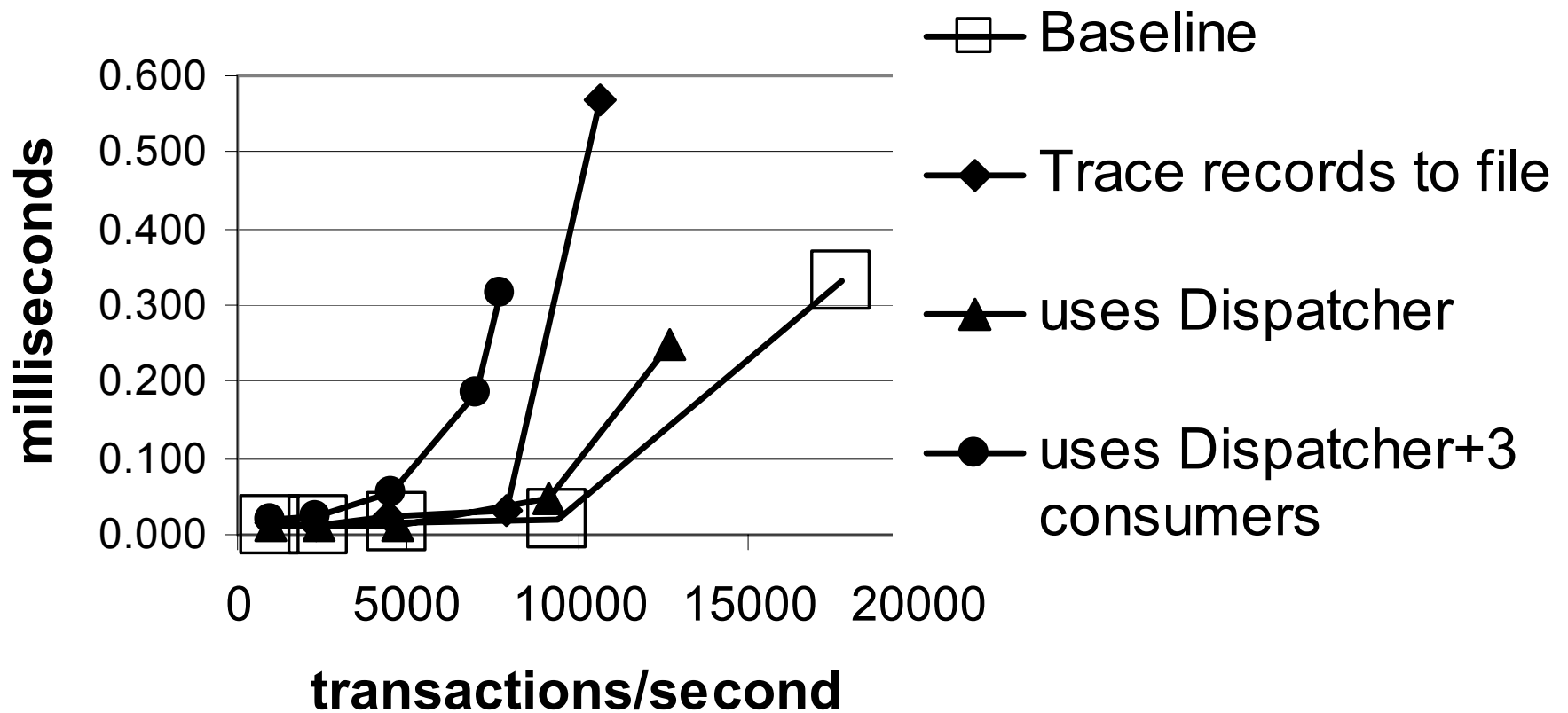
# Delta from baseline: use dispatcher



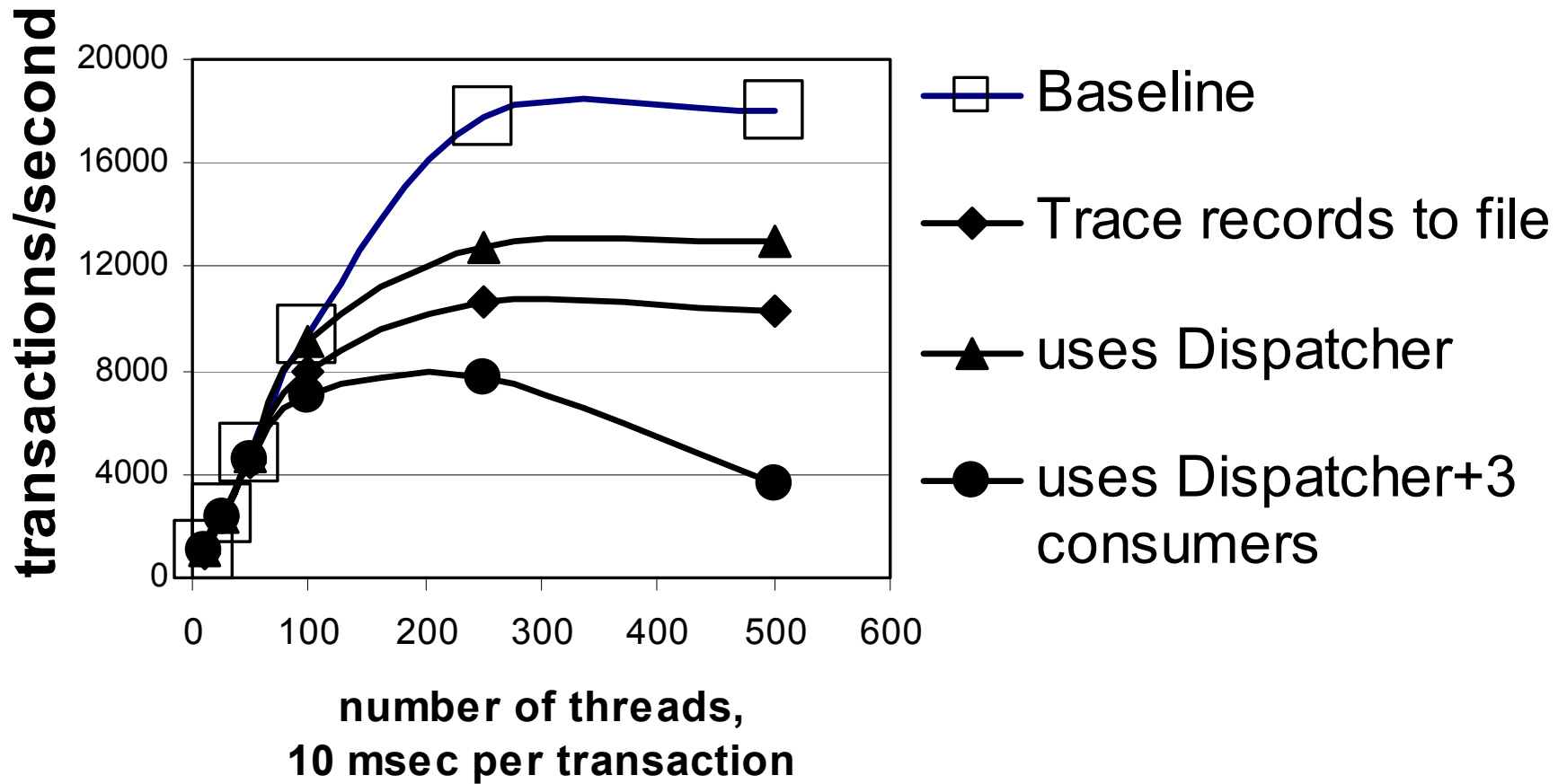
# Delta from baseline: use 3 consumers



## Application Latency in Different Configurations



## Throughput in Different Configurations



# Conclusions

# Insights for Application Developers

- ◆ **ARM 3.0 for Java is a good fit for measuring business transactions**
- ◆ **There is no significant performance penalty when using correlators**
- ◆ **Using metrics is relatively expensive compared to simple transaction measurements**
- ◆ **There isn't a significant performance difference between using ArmTransaction with start/stop calls to measure versus the application measuring the response time itself**

# Insights for System Administrators

- ◆ **Using an ARM implementation written entirely in the Java language is a reasonable option**
- ◆ **Measuring transactions in a non-trace mode requires just a few microseconds per transaction on this class of system**
- ◆ **Tracing can be expensive when carried out for many transactions**
- ◆ **Tracing on a selective basis is reasonable**
- ◆ **Using multiple consumers should only be done selectively, like tracing**

**Questions?**