

# Vertical Profiling

Understanding the Behavior  
of Object-Oriented Applications

Lei Liu

CS 699/CS-IT 803

1

## Acknowledgments

- Portions of this presentation taken from:
  - [Matthias auswirth](#), and [Michael Hind](#)  
“Vertical Profiling”

2

# Outline


- **1. Background**
- **2. Case Study**
  - Dip before GC
  - Gradual Increase
  - Sudden Increase
  - Periodic Pattern
  - Scalability of Multithread
- **3. Summary**

3

# Motivation

## **C Program**

---

 Application  
Operating System  
Hardware

## **Java Program**

---

 Application  
Framework  
Java Library  
Virtual Machine  
Native Library  
Operating System  
Hardware

4

## Motivation

- Layering introduces obstacles to understanding application performance
  - With dynamic recompilation, the same source code statement can be translated to different machine instructions at different memory locations throughout the execution of the program
  - Garbage collection can both relocate objects and reuse addresses, resulting in a dynamic mapping between objects and addresses

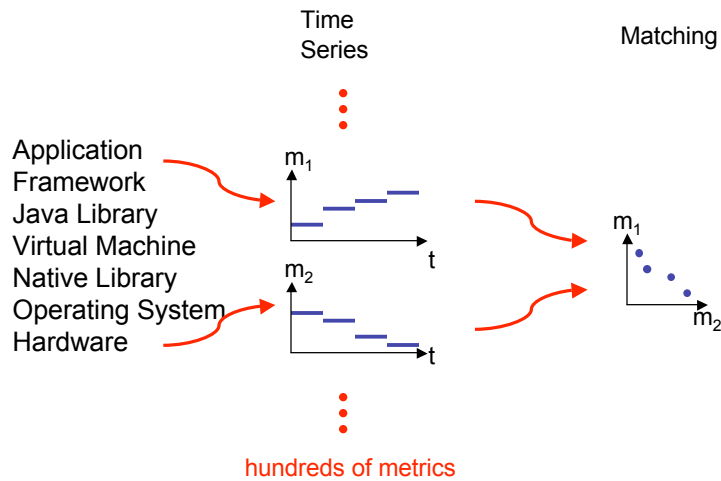
5

## Solution

- Gather more complete profiles, containing information about system behavior on various levels
- Further the understanding of system behavior through correlation of profile information from different levels-HPM, SPM

6

# Vertical Profiling



7

## HPM: Hardware Performance Monitor

- Hardware performance monitor capability that exists in JVM, implementation dependent
  - processor cycles,
  - instructions completed
  - L1 cache misses
  - Thread switch status

8

## **SPM: Software Performance Monitors**

- **Application.**
  - by adding software performance monitors to applications
- **Virtual Machine**
  - Memory Manager
  - Runtime Compilers
  - Synchronization
- **Operating System**

9

## **SPM(cont)**

- **Software performance monitors are implemented on two levels:**
  - In native code, each Pthread has its private array of software performance monitors. A pointer to this array is stored as Pthread-specific data. Instrumentations in native code can update the monitor of the currently running thread
  - On the Java level, keep a reference to a Java array of software performance monitors in the VirtualProcessor object

10

# Challenge

- Overhead
  - Fast enough to be useful?
- Perturbation
  - Vertical profiling adds instrumentation to the system, will it perturb the very behavior that it is trying to understand?

11

# Overhead

Benchmark	Production	Vertical Profiling	
compress	9.77	10.15	3.6%
db	22.42	23.85	6.4%
jack	13.38	14.41	7.8%
javac	19.14	20.57	7.5%
jess	8.23	8.64	5.0%
mpegaudio	8.52	9.69	13.8%
mtrt	7.64	7.99	4.6%
jbb	27.17	31.84	17.2%
hsql	19.19	19.39	1.1%
Average			7.4%

user might be interested in only a small subset of the monitors, and would thus be able to reduce overhead even more

12

## Perturbation Analysis

- For example, instrumentation for collecting data on a software performance monitor may change the cache behavior of the application.
- Perturbation analysis assures that the data collection is not perturbing the behavior of interest

13

## HPM perturbation analysis

- End-to-end perturbation analysis of HPMs.
  - conduct an additional run that does not collect any data at each thread switch, compare the end-to-end differences with the aggregate HPM values that are recorded at each thread switch. If these two sets of values are close, we have some confidence that our mechanism for recording HPMs does not perturb behavior at the macro level.
- Temporal impact of HPMs
  - use qualitative analysis based on our knowledge of our HPM implementation to argue that vertical profiling is not perturbing the microlevel behavior of the application

14

## SPM perturbation analysis

- Impact of SPMs on HPMs.
  - Visually compare the HPM signals collected with and without SPM tracing enabled
- Impact of SPMs on SPMs
  - Argue from knowledge of our SPM implementation that one SPM does not significantly perturb another SPM of interest to our case study

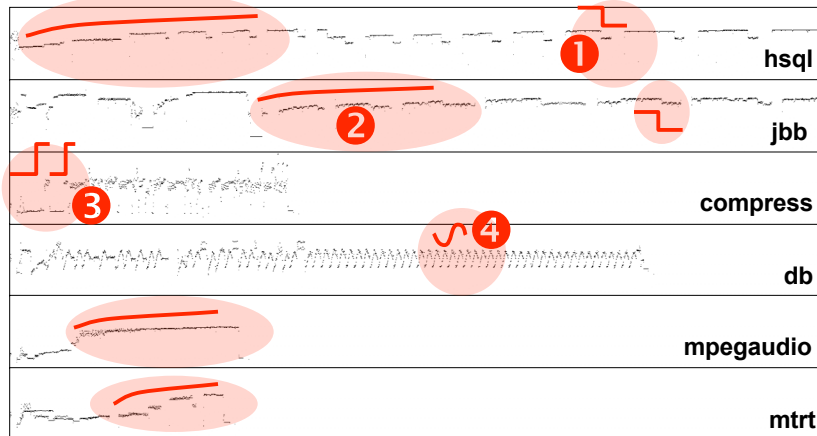
15

## Outline

- 1. Background
- **2. Case Study**
  - Dip before GC
  - Gradual Increase
  - Sudden Increase
  - Periodic Pattern
  - Scalability of Multithread
- 3. Summary

16

## Analysis of Unexpected Behavior (IPC over Time, JikesRVM on AIX on POWER4)



17

## Outline

- 1. Background
- 2. Case Study
  - Dip before GC
  - Gradual Increase
  - Sudden Increase
  - Periodic Pattern
  - Scalability of Multithread
- 3. Summary

18



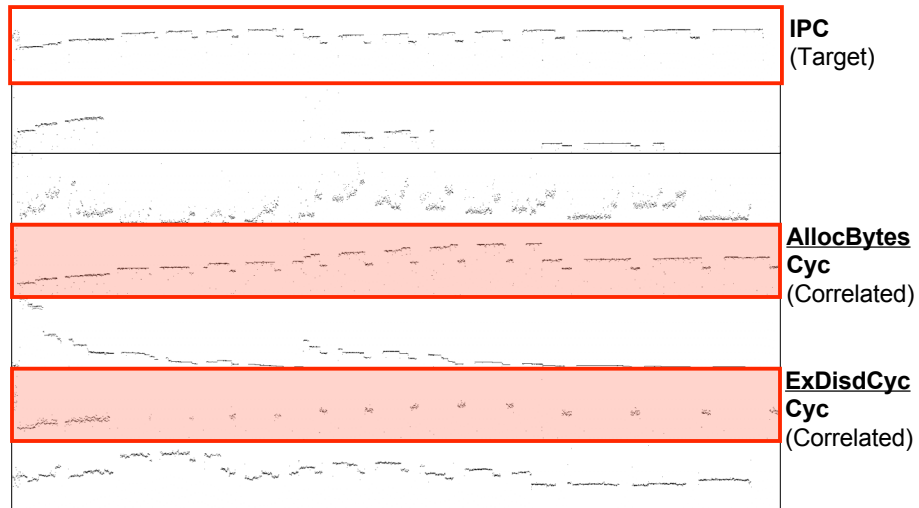
## Phenomenon ① Dip Before GC



Benchmark: **hsql**

19

## ① Dip Before GC Compare with 200+ Metrics



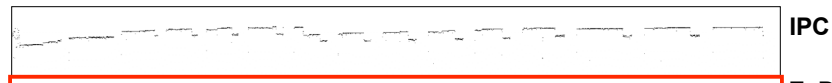
20

# 1 Dip Before GC Find Cause given Correlation



21

# 1 Dip Before GC Test Hypothesis



## Background knowledge:

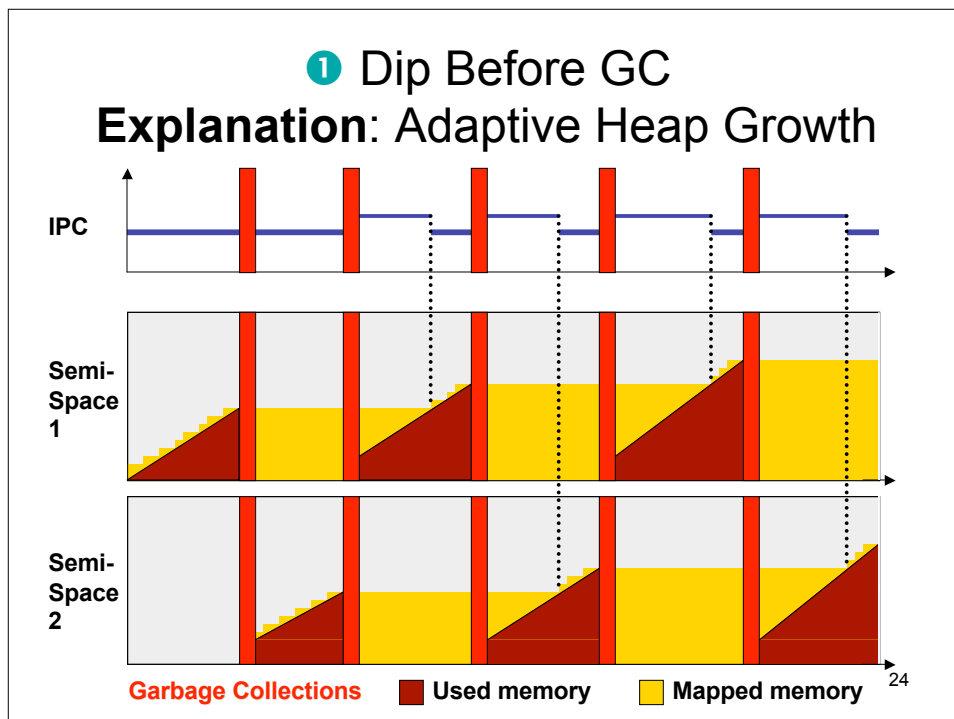
- After mmap, first access to page causes page fault
- Page faults are handled by executing interrupt handler code

22



# mmap

- System call `mmap()` will establish a mapping between a process' address space and a file, shared memory object, or typed memory object

23



## 1 Dip Before GC Results & Insights

- Cause-effect chain
  - $Mmap_{OS}$    $ExDisabled_{OS}$    $IPC_{HW}$
- Validation
  - When heap resizing is disabled, the pre-GC dip disappears
- Meta-Insight
  - Need many metrics, from various levels (vertical)
  - Need way to find correlated metrics
  - Need way to identify causality

25

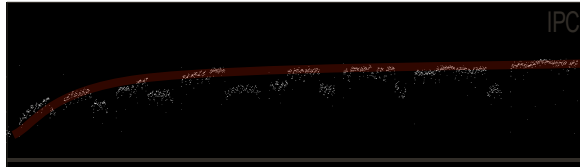
## Outline

- 1. Background
- 2. Case Study
  - Dip before GC
  - **Gradual Increase**
  - Sudden Increase
  - Periodic Pattern
  - Scalability of Multithread
- 3. Summary

26

## 2

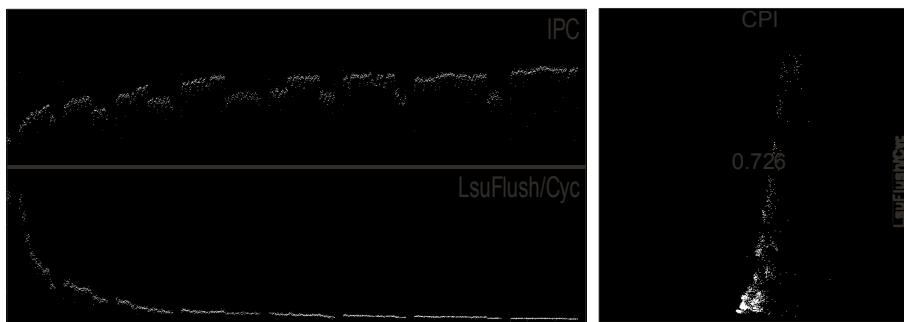
## Phenomenon 2 Gradual Increase



Benchmark: **jbb**

27

## 2 Gradual Increase Compare with HW-Layer Metric

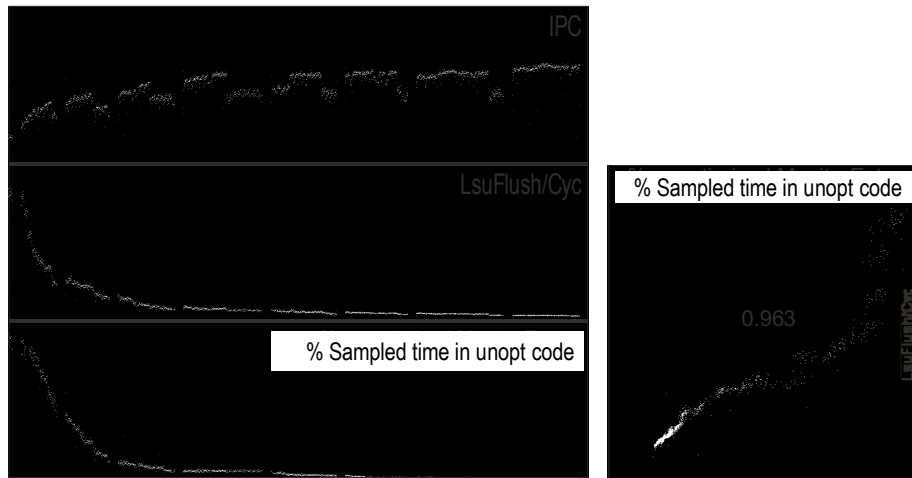


### Background knowledge:

- LSU (Load/Store Unit) flushes because of mis-speculation in out-of-order execution of loads and stores to same address.
- Unoptimized code uses operand stack.

28

## ② Gradual Increase Compare with VM-Layer Metric



29

## ② Gradual Increase Results & Insights

- Cause-effect chain
  - $\text{UnoptCode}_{\text{VM}} \rightarrow \text{LsuFlush}_{\text{HW}} \rightarrow \text{IPC}_{\text{HW}}$
- Validation
  - IPC falls *between* IPC of unoptimized and fully optimized code

30

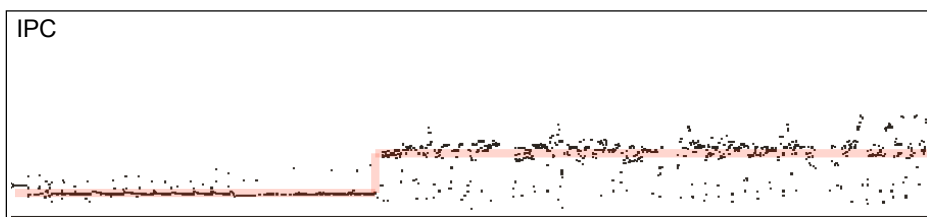
# Outline

- 1. Background
- 2. Case Study
  - Dip before GC
  - Gradual Increase
  - **Sudden Increase**
  - Periodic Pattern
  - Scalability of Multithread
- 3. Summary

31



## Phenomenon ③ Sudden Increase

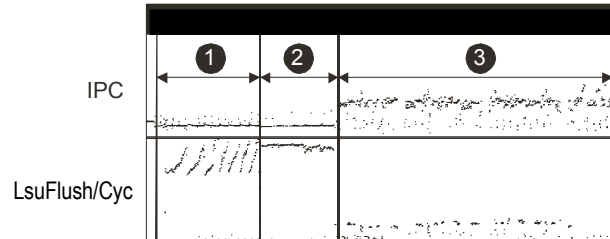


Benchmark: **compress**

32

### 3 Sudden Increase

**Explanation:** Optimization of Long-Running Method



**Insight from previous case study:**

If IPC goes up, while LsuFlush/Cyc comes down

- Non-Opt Code may come down
- Non-Opt Code may be root cause

33

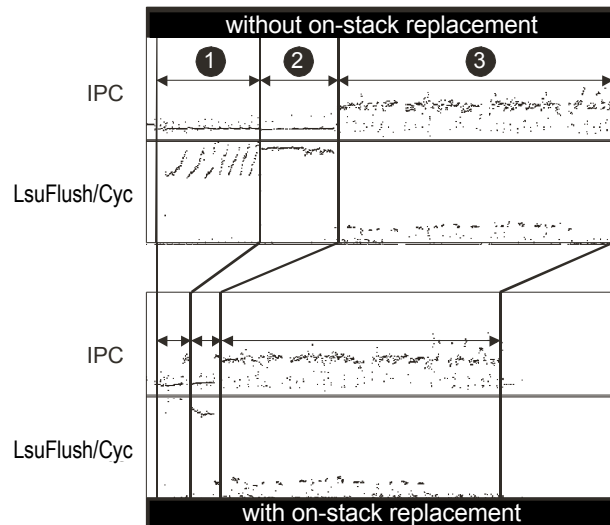
## OSR

- OSR replaces an unoptimized version of a method with an optimized version while the method is running

34

### ③ Sudden Increase

**Explanation:** Optimization of Long-Running Method



35

### ③ Sudden Increase Results & Insights

- Cause-effect chain
  - $\text{UnOpt Code}_{\text{VM}} \rightarrow \text{LsuFlush}_{\text{HW}} \rightarrow \text{IPC}_{\text{HW}}$
- Validation
  - Enabling OSR has the expected outcome
- Meta-Insight
  - Reuse insight from ② Gradual Increase

36

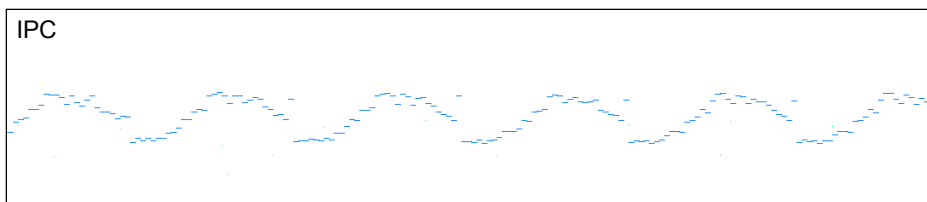
# Outline

- 1. Background
- 2. Case Study
  - Dip before GC
  - Gradual Increase
  - Sudden Increase
  - **Periodic Pattern**
  - Scalability of Multithread
- 3. Summary

37



## Phenomenon 4 Periodic Pattern

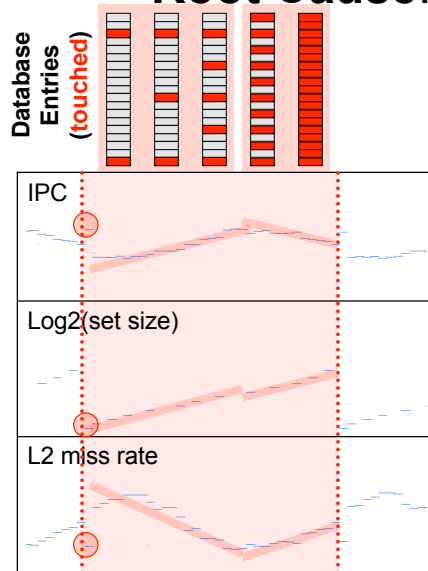


Benchmark: **db**

38

#### 4 Periodic Pattern

### Root Cause: Sort Algorithm



- Set size > 2178
  - Set too large to fit in L2
- Set size  $\leq$  2178
  - Temporal locality increases
- Smallest set sizes
  - Copying array: high IPC

39

#### 4 Periodic Pattern

### Result & Insights

- Cause-effect chain
  - $\text{SetSize}_{\text{App}} \rightarrow \text{L2Misses}_{\text{HW}} \rightarrow \text{IPC}_{\text{HW}}$
- Validation
  - Performed manual object inlining [SamGuyer] which sped up program by almost a factor of 2

40

## Perturbation Analysis

- This case study examines the Cyc, InstCmpl, and L2Misses HPMs. The end-to-end perturbation for HPMs is either within or close to the standard deviation of the executions without HPMs. End-to-end perturbation is
  - 2.77% for Cyc and 0.11% for InstCmpl, which is less than
  - the standard deviation for executions without HPMs: 2.92%
  - For Cyc, 0.23% for InstCmpl. Whereas, L2Misses HPM has only slightly more perturbation at 7.49% than its standard deviation at 5.97%.
- Tracing and periodic pattern are not correlated
- We were able to visually correlate the IPC and L2Misses patterns with and without SPMs

41

## Outline

- 1. Background
- 2. Case Study
  - Dip before GC
  - Gradual Increase
  - Sudden Increase
  - Periodic Pattern
  - **Scalability of Multithread**
- 3. Summary

42

# 5



## Phenomenon Scalability of Multithreaded

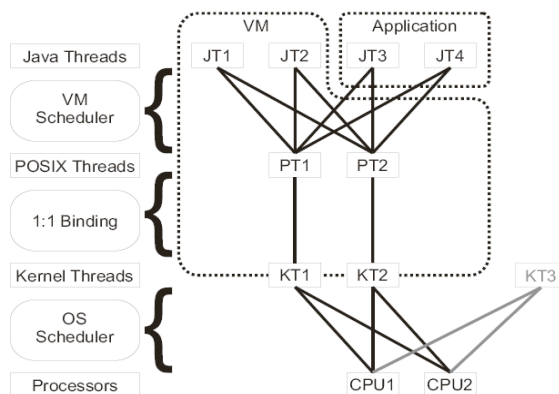
Wall Time column: the wall clock time from the point where the first worker thread starts running to the point where the last worker thread stops running

Sample D. column: the average duration (in million ticks) of the worker thread time slices

Lock Yields column: the number of yields during lock acquisitions in the worker threads

Benchmark	Scale	Wall time (M,%)	CPU time (M,%)	Samples	Sample D. (M)	Lock Yields		
mtrt	1 on 1	1,070	100%	828	100%	575	1.440	0
mtrt	2 on 2	754	70%	957	116%	694	1.379	40
mtrt	4 on 4	666	62%	1,224	148%	1,129	1.084	302
jbb	1 on 1	4,276	100%	3,227	100%	2,221	1.453	0
jbb	2 on 2	2,401	56%	3,509	109%	4,979	0.705	2,703
jbb	4 on 4	1,346	31%	3,836	119%	10,409	0.369	7,940
hsq1	1 on 1	434	100%	265	100%	192	1.380	0
hsq1	2 on 2	313	72%	277	105%	28,849	0.010	28,600
hsq1	4 on 4	367	85%	317	120%	72,256	0.004	51,363

## 5 ThreadScheduling in JVM





## Periodic Pattern Analysis

- In Java a critical section corresponds to a synchronized block or a synchronized method. On entry to a critical section, Java executes the MonitorEnter bytecode
- In Jikes RVM MonitorEnter is implemented as a call to the VM Thread.lock() method. When calling lock, a thread either immediately gets the lock, retries a few times in a tight loop, or yields if neither of the former is successful. The number of yields happening from within the lock method are an indication of lock contention, and thus a measure of parallelization overhead

45



## Periodic Pattern Analysis (cont)

- A large number of time slices in multithreaded runs is caused by lock contention. Worker threads try to acquire a lock that is already held by a different thread, and thus they yield. A thread that yields ends its time slice prematurely, and thus the length of the time slice is shorter than the scheduler's time quantum

46



## Periodic Pattern Result & Insights

- Cause-effect chain



- Validation

- need to change the amount of lock contention in the application. Left for future work

47

## Temporal **Vertical** Profiling

- Gather data about multiple levels:

- ① ② ③ ④ ⑤ Application
- ① ② ③ ④ Framework
- ① ② ③ ④ Java Library
- ① ② ③ ④ ⑤ Virtual Machine
- ① ② ③ ④ ⑤ Native Library
- ① ② ③ ④ ⑤ Operating System
- ① ② ③ ④ ⑤ Hardware

vertical



48

## Outline

- 1. Background
- 2. Case Study
  - Dip before GC
  - Gradual Increase
  - Sudden Increase
  - Periodic Pattern
  - Scalability of Multithread
- 3. Summary

49

## Performance Analysis Approaches

- *Browsing*
  - Browsing through the signals of all available performance metrics helped us find new hints. We thus depended on the set of available metrics to cover as many subsystems (and thus as many causes) as possible
- *Searching*
  - When we had a hypothesis, we knew what information we needed to test it, it may be necessary to add a new software performance monitor

50

## Performance Understanding Concrete Strategies Worth Exploring

- Reduce overhead due to heap growth
  - Pre-mmap memory using large pages
  - Be less aggressive with shrinking the heap
- Get to high IPC faster
  - Be more aggressive with dynamic recompilation
- Reduce slow-down due to long-running methods
  - Enable on-stack replacement
- Improve sorting performance
  - Use different sorting algorithm
  - Reduce memory footprint

51

## Limitation

- Too much manual work
- Hard to find general metrics for general purpose
- Some metrics might be difficult to acquire
- Correlation decided by human is not always accountable

52

## Related Work

- Data Collection
  - “Binary” Instrumentation
    - Native - [Pixie, QPT/EEL, ATOM, Vulcan, PIN]
    - Byte code - [BIT, BCEL, JikesBT]
  - “Source” Instrumentation
    - Hardware - [DCPI, PAPI, PCL, HPMTk, OProfile]
    - Software - [WMI, K42, LTT, TNF, JVMPi]
- Performance Analysis
  - Performance Metrics  
[Dufour et al., Sprunt's HPM, Fortier et al., Lilja]
  - Visualization & Analysis Tools  
[PV, Jinsight, HPCView, Paradyn, Jumpshot, Pablo, Paraver, TAU, VTune]
  - Statistical Performance Analysis  
[Eeckhout, Anh/Vetter]
  - Performance Modeling  
[PSpec, JavaPSL, PerfContracts, ParallelPerfPredicates]

53

## Future work

- [OOPSLA'05—Automating Vertical Profiling](http://www.oopsla.org/2005/ShowEvent.do?id=18)
  - <http://www.oopsla.org/2005/ShowEvent.do?id=18>

54

## Conclusion

- To understand the performance of Java programs, in general one needs
  - Profile information from all system layers
  - Interactive approach for exploring the profile information
  - Comparison of many metrics