

# **Managed Runtime Environments: Salient Characteristics and Code Optimization**

**CJ Newburn  
DPG Architecture  
Intel**

# Outline

- ◆ **What's a managed runtime workload?**
  - ❖ Usage models and workloads
  - ❖ Where have all the cycles gone?
- ◆ **Inherent characteristics**
- ◆ **Empirical findings**
  - ❖ Heavy lifting tasks
  - ❖ Scaling for logical and physical processors
  - ❖ Objects: sizes, lifetimes, classes
  - ❖ Locks
  - ❖ Branches
  - ❖ Convolution with the microarchitecture
- ◆ **Summary and code optimizations**

# Usage Models and Workloads

## ◆ Common app characteristics

- ❖ Large data sets, poor locality
- ❖ Common data types:
  - strings for supply chain
  - floats for eng/scientific/data mining
- ❖ Lots of threading
- ❖ Lots of communication, parsing, reformatting
- ❖ Lack of steady state, poor I-side locality
- ❖ Managed activity has noticeable impact, not overwhelming

## Client Usage Models and Coverage

Productivity	Interpersonal Communication	Digital Media
Education	Device Management	Entertainment & Creativity

## Server Usage Models and Coverage

Collaboration	Business Logic	Database
Web Application / Web Service	HPC	Infrastructure

# MRTE Time Breakdown Analysis

## ◆ ORP on SPECjbb

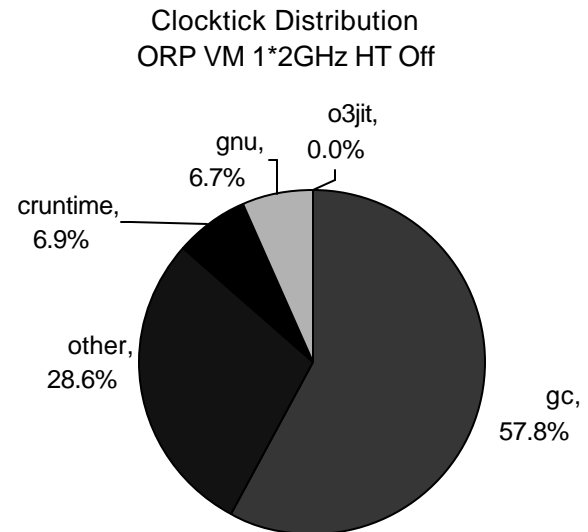
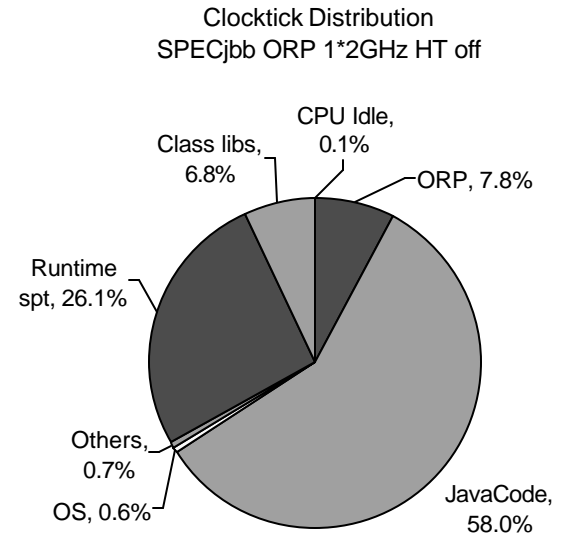
- ❖ 8% MRTE, JITted perf matters
- ❖ JIT time doesn't
- ❖ 5% GC, heap/alg matter

## ◆ Perf strong f(heap size), if small

- ❖ Must be  $\geq 600\text{MB}$  for JRockit on SPECjbb
- ❖ 40% of perf @ 200MB

## ◆ Components of interest

- ❖ 26% runtime
- ❖ 0-15% OS
- ❖ 0-6% memmove
- ❖ 0-5% sync
- ❖ 0-4% file lock



*Managed performance approaches native; path lengths similar for JRockit, ORP*

# Inherent MRTE Characteristics

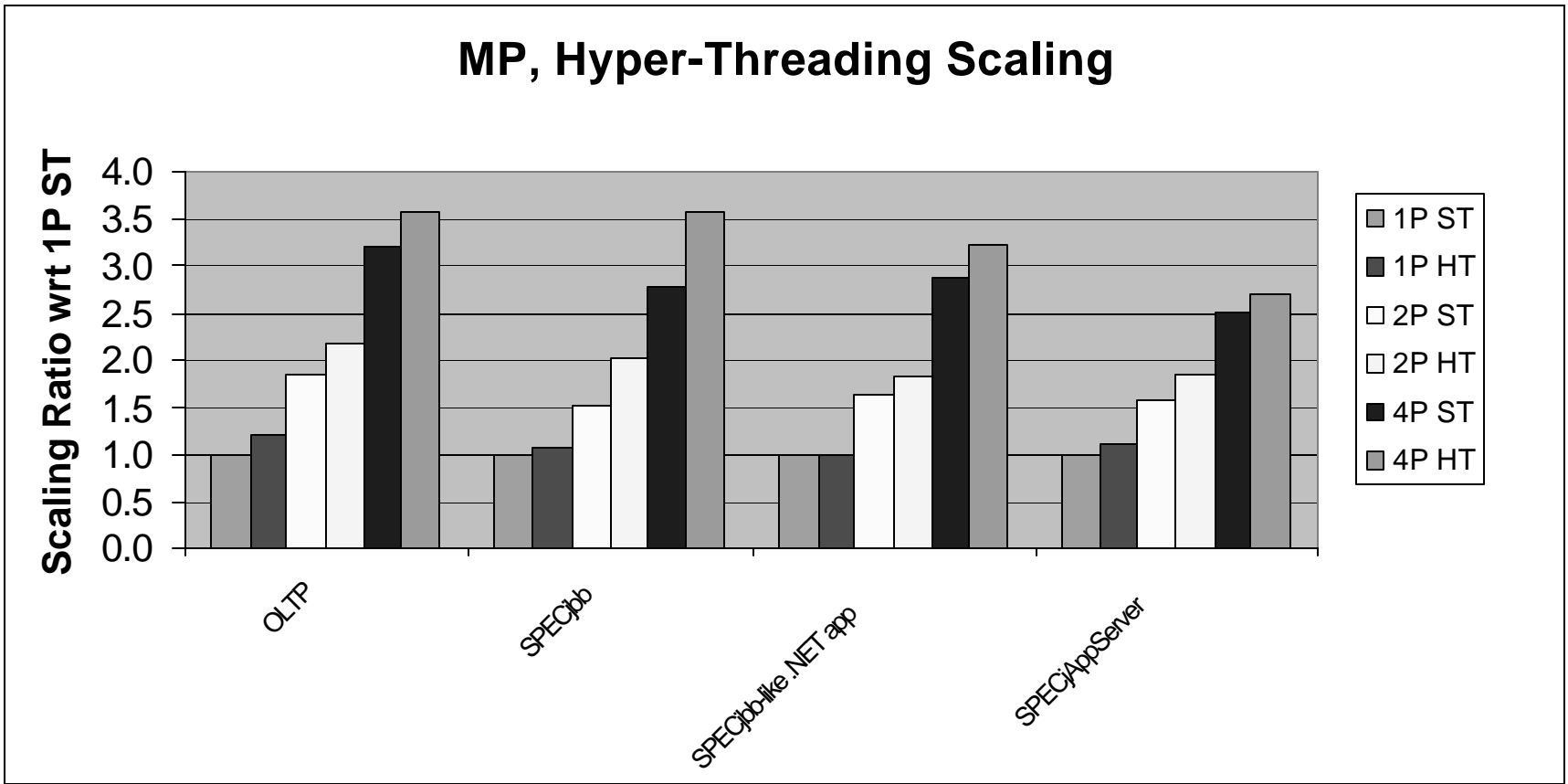
- ◆ **“Modern” language features**
  - ❖ Multithreading & synchronization
  - ❖ Garbage collection – indirect memory references
  - ❖ Object oriented
  - ❖ Exception handling at user level
  - ❖ Type safety
- ◆ **Platform neutral code distribution format**
  - ❖ Byte code with rich semantics
  - ❖ Dynamic class loading
  - ❖ Compiler is part of program execution
- ◆ **Language system enforces security & reliability**
  - ❖ Bounds check, verification, type check
  - ❖ Language based security
- ◆ **Rich standard class libs**
  - ❖ New APIs that move a lot of functionalities from OS
  - ❖ Rich component framework from J2ME to J2EE
- ◆ **Time to market and overall TCO drive decisions, not perf**

# Heavy Lifting Tasks

- ◆ Huge variance among workloads
- ◆ 7% in GC on .NET similar to SPECjbb
- ◆ 5-10% in sync on SciMark2, SPECjbb
- ◆ 13% on serialization TPC-W-like B2C ASP.NET app
- ◆ 13% on moving memory in IPDemo
- ◆ ~50% of supply-chain app in functions with string/char parameters
- ◆ 37% in XML processing on supply-chain app
- ◆ Digital signing and verification can lead to 5x increase in kernel execution time

# MP, Hyper-Threading Scaling Analysis

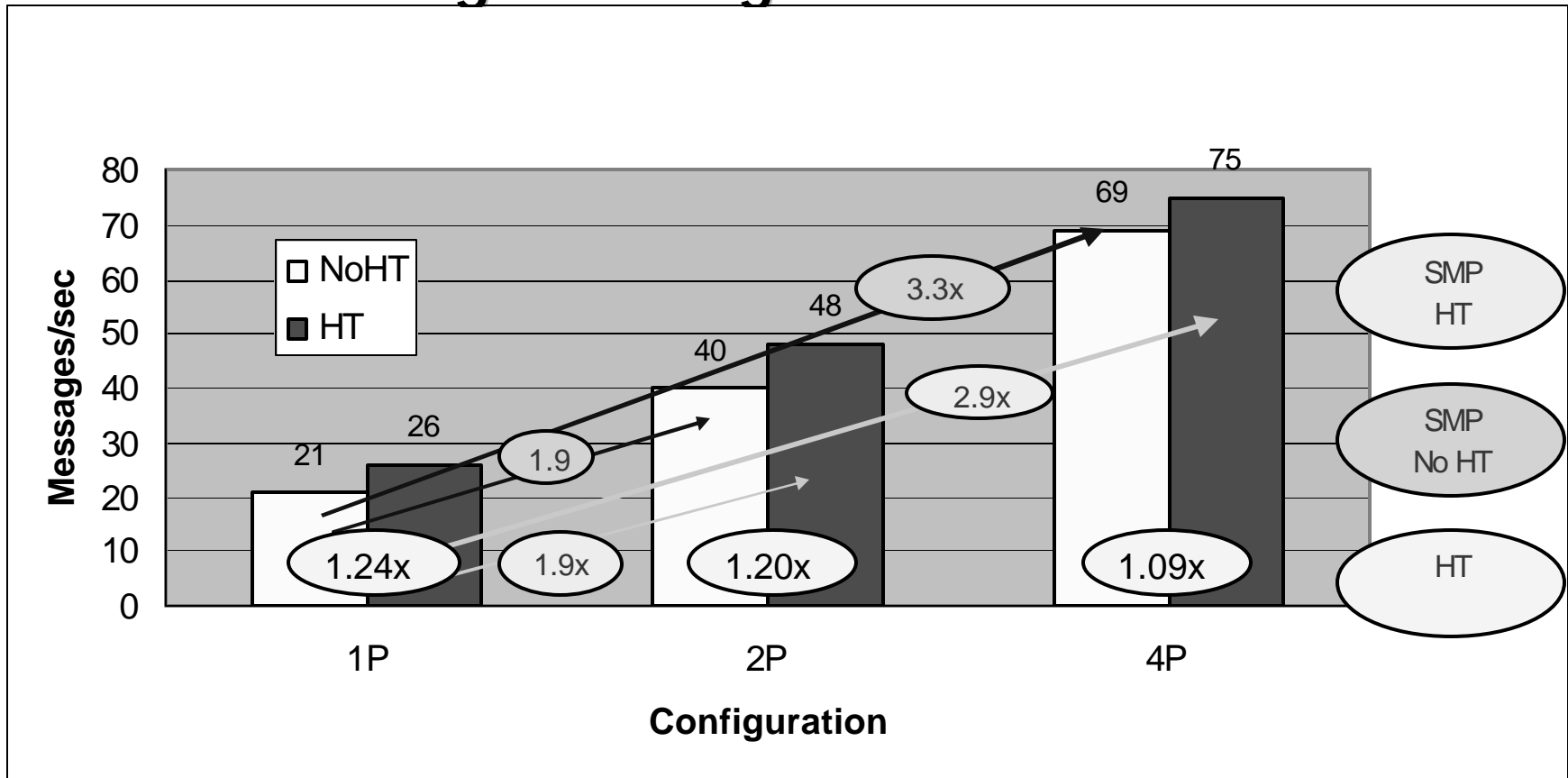
- ◆ Similar to databases
- ◆ SPECjAppServer has highest bus utilization



*Intel-internal, unpublished measurements on Intel Xeon™ MP processor with Hyper-Threading platform*

# Intel Xeon™ MP processor at 2.0GHz with Hyper-Threading

## 50000 msg/s setting for Financial Services



- ◆ Good HT, MP scaling
- ◆ Scaling tapers off as relative bus load increases

# Objects

## ◆ Size: mostly small, but a few large objects

Object Size	SPECjAppServer	SPECjbb	TPC-W
Mean.median/mode (bytes)	112/40/24	18/16/6	51/24/16
Standard Deviation (bytes)	637	60	278
90% cutoff obj size for objs	94	24	88
90% cutoff obj size for bytes	16400	128	4112
Allocations	737	111	1076

## ◆ Lifetimes, measured in method calls (MC), for objects < 128B

	0 MC	1 MC	2 MC	3 > MC
<b>SPECjbb</b>	<b>26.1%</b>	<b>23.0%</b>	<b>24%</b>	<b>26.1%</b>
<b>Volano</b>	<b>78.6%</b>	<b>5%</b>	<b>14%</b>	<b>0.4%</b>

Large # small objects (32B) with short lifetimes (<4 method calls),  
fewer large objects with long lifetimes

Poor locality beyond next \$ line

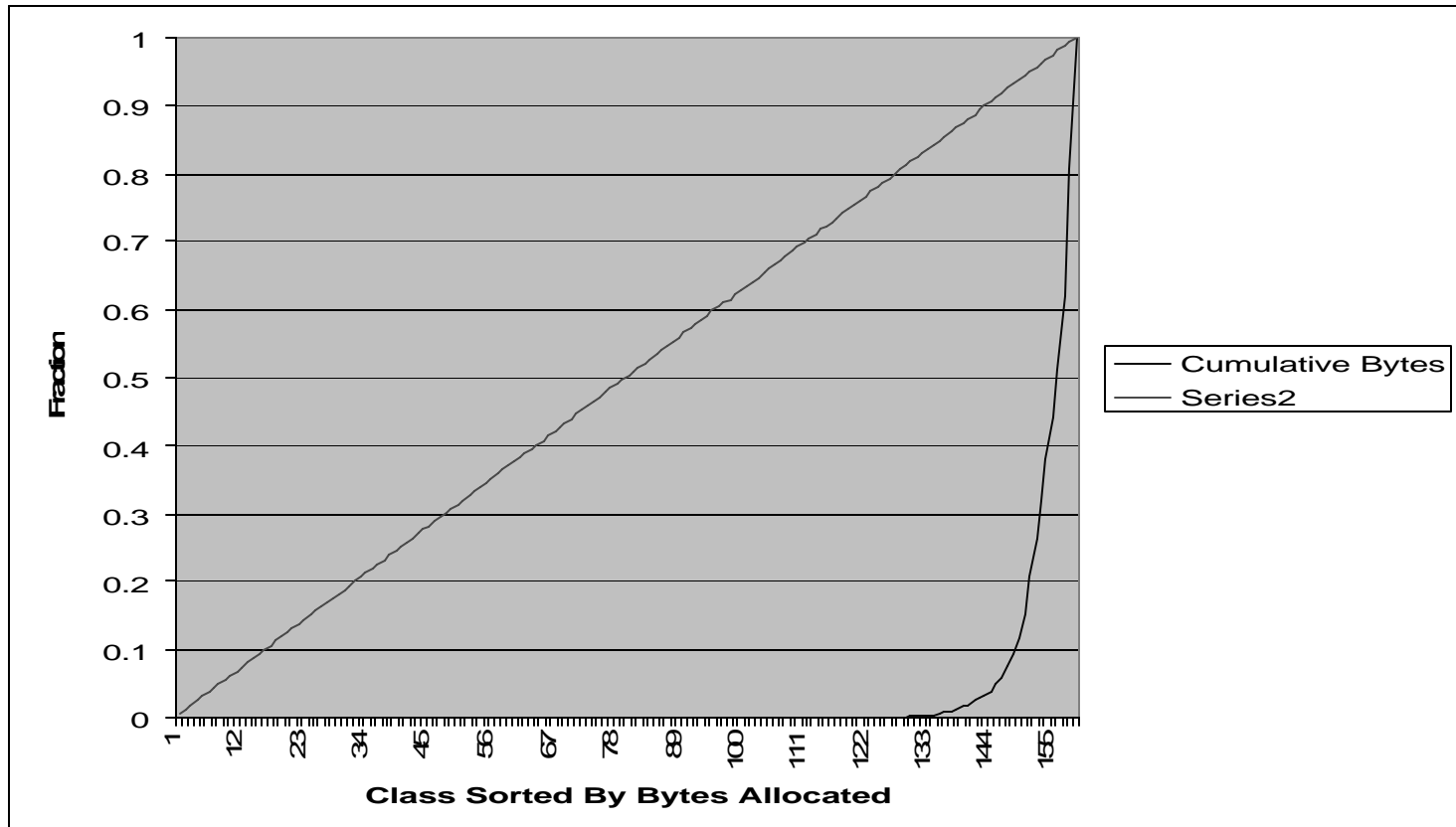
# Object Liveness

Workload	MaxLiveBytes	TotalAllocedBytes	Ratio
check	82880	1150639	0.072
db	2643	55283	0.097
jack	428466	12580724	0.034
javac	136790	3813380	0.036
jess	195337	3798314	0.051
mpegaudio	163485	1547080	0.105
mtrt	310482	8732363	0.035
specjbb	8399135	284024774	0.029

- ◆ Ratio of total allocated to max live bytes in rightmost column
- ◆ Only about 3-10% of allocated bytes are live at any given time

◆ Note that this is based on the max

# Class Distribution



- ◆ 5% of classes hold 95% of data
  - ◆ Plot shows per-class cumulative bytes for MTRT workload
  - ◆ 95% of the bytes are in 9% of the classes for this workload
- Optimization focus on a few classes could be fruitful

# Locks

## ◆ Source-level causes

- ❖ Conservative libraries
- ❖ Conservative coding styles
- ❖ Gratuitous use of synchronize keyword

## ◆ Frequency

- ❖ 3-42 locks/10k instructions in cooperative-threaded workloads
- ❖ Locks rarely even shared (order of magnitude lower), much less contended

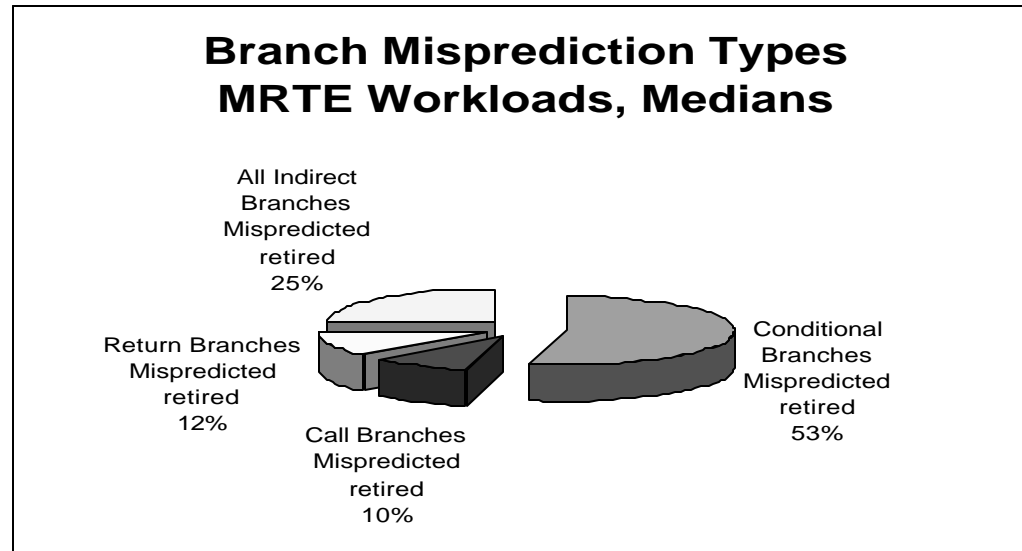
## ◆ Lock impact

- ❖ Locks are executed non-speculatively
- ❖ Traditionally inhibit instruction-level parallelism
- ❖ Traditionally lock costs increase with pipeline depth

**Eliminate locks wherever possible**

# Branches

## ◆ Breakdown of mispredicted branches by branch type



## ◆ Indirect branches

- ❖ Degree of dynamic polymorphism generally low: 1-2 targets
- ❖ # static indirect branches and calls can be in the 1000s
- ❖ Make branches more predictable: conditional to likely target + indirect

## ◆ Parsers have more mispredicted branches

# Object-Oriented Programming

## ◆ More calls and returns

- ❖ ~50 instructions per call across several Java apps
- ❖ 13% of time in call/return, push/pop
- ❖ From branch mispredictions, rare but expensive misses

## ◆ Flat execution profiles

- ❖ Max method hotspot < 5% of total time
- ❖ 1000s of methods
- ❖ Lots of inlining
- ❖ Difficult to tune: small gains for each optimization

# EMON Analysis

## ◆ Stall causes

- ❖ Locks
- ❖ Branch mispredictions
- ❖ Large working set: DTLB
- ❖ Low code locality: ITLB, trace cache
- ❖ Some opportunity for code tuning (forwarding stalls)
- ❖ No SMC, UC, split problems

## ◆ High CPI

- ❖ TPCC: 9.2
- ❖ SPECjbb: 2.7
- ❖ SPECjAppServer: 8.2

## ◆ Significant variance among workloads

## ◆ Relative to ISPEC

- ❖ Same range
  - Mispredicted branches
  - 1<sup>st</sup>, 2<sup>nd</sup>--level misses
- ❖ Higher than ISPEC:
  - ITLB page walks (3-43/10k instr vs. <1/10k instr)
  - Bus utilization (26-62% vs. <51%)
- ❖ Lower than ISPEC
  - % uops from trace cache vs. decode (26-56% vs. >65%)

# Summary

## ◆ **MRTE workloads are statistically different**

- ❖ Databases have high CPIs, parsers branch intensive, highly threaded
- ❖ Code locality is more of an issue, more OO
- ❖ More difficult to tune because of flat profile

## ◆ **Opportunity for SW management of memory**

- ❖ Careful management of many small objects
- ❖ Minimize use of virtual space (TLB)

## ◆ **Avoid locks and remove them in SW**

- ❖ Maximize thread locality
- ❖ Avoid overuse of “synchronize” keywords

## ◆ **Avoid branch mispredictions in SW**

- ❖ Peel likely target off of indirect branches & calls as conditional

**Much to be done in SW to make MRTEs perform better**