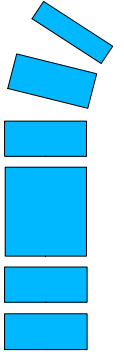


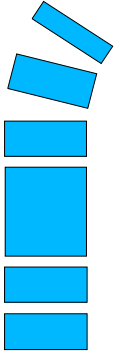
Optimistic Stack-Allocation in Java-Like Languages



Erik Corry <ecorry @ esmertec.com>

June 11 2006

Performance in Java



“Even though, as of release 1.3, allocating small objects is relatively inexpensive, allocating millions of objects needlessly can do real harm to performance”

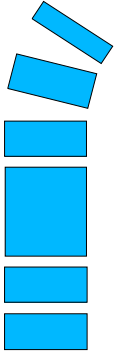
Joshua Bloch, Effective Java Programming Language Guide

“Jmol does not perform any heap memory allocation during the repaint cycle. For performance reasons, it is important that we continue to follow this guideline.”

Jmol Technical Notes

Other O-O languages (Smalltalk, BETA) do much more allocation.

Avoiding allocation considered harmful



Premature optimization is the root of all evil

Donald E. Knuth

Don't sacrifice sound architectural principles for performance

Joshua Bloch

This extends to language design and implementation!

I became convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). *Edsger W. Dijkstra*

Why is allocating objects slow?

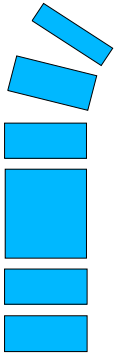
Space must be found for objects

Space must be cleared

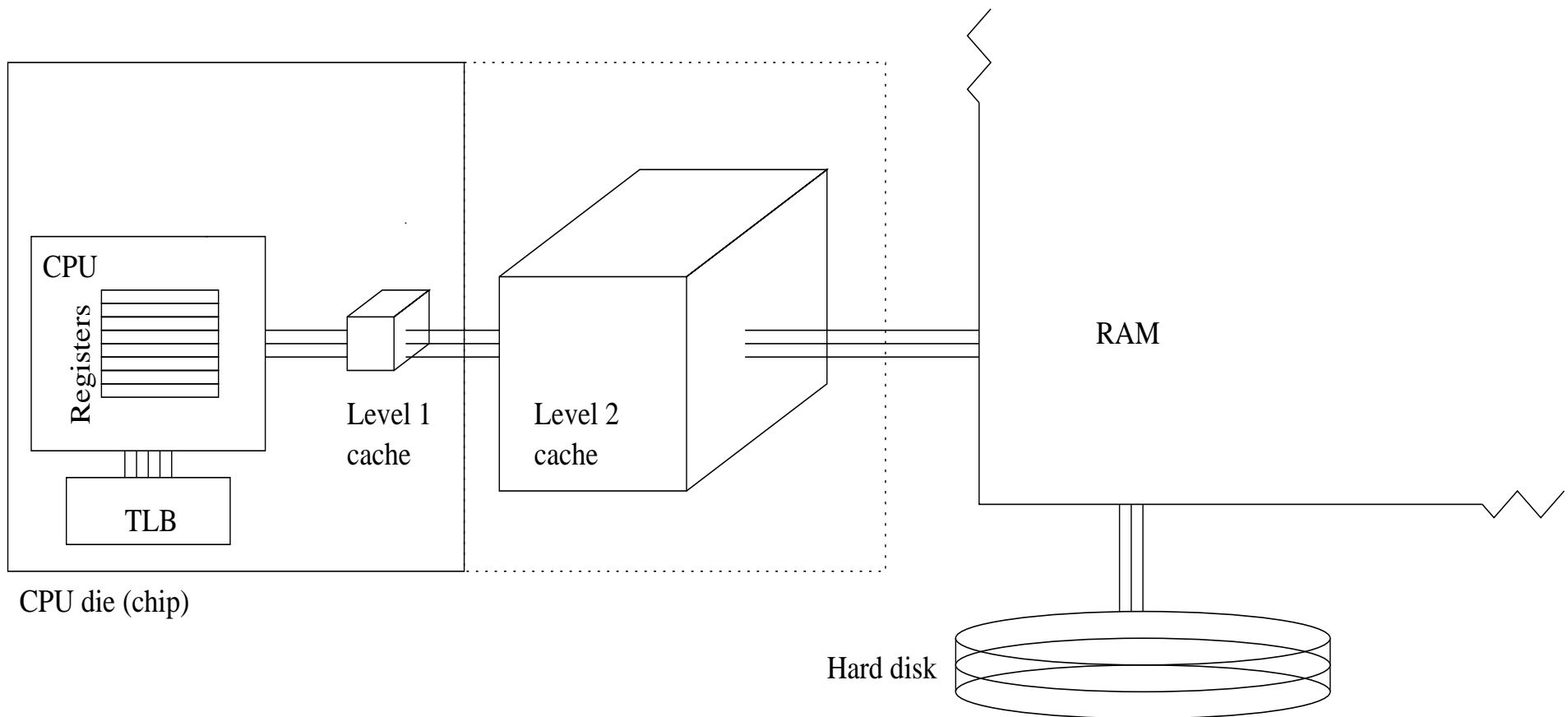
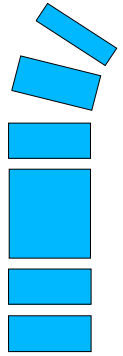
Space must be reclaimed

Modern garbage collection algorithms can do all these fast

So why is allocating objects still slow?

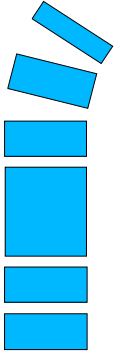


Cache hierarchies are deep and CPUs are fast



(Thesis p. 9)

“The most 'convenient' resolution to the problem would be the discovery of a cool, dense memory technology whose speed scales with that of processors” *Wulf & McKee in “Hitting the Memory Wall”*



Stack allocation makes better use of caches

Caches reward reuse of recently used memory

Stacks naturally reuse recently used memory

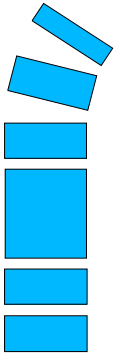
If we can allocate in last-in-first-out order then we can use a stack.

Stacks also have low allocation and deallocation overhead (just move the stack pointer up and down)

But last-in-first-out is too limiting in general

The demands of modern programming languages make stacks complicated to implement efficiently and correctly. *Andrew Appel and Zhong Shao*

How do we know which objects can be stack allocated?



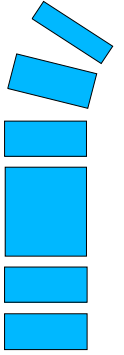
Static program analysis tells us which objects are guaranteed to be garbage at termination of *some* method. We say that these objects do not *escape* the method.

These non-escaping objects can be stack allocated in the invocation record of *that* method.

Many objects do not escape, but it can be difficult to prove.

Many objects escape the method in which they are created, but do not escape the method that called that method.

Using *deep allocation* we can allocate on the stack of a method that is buried in the stack.



Problems with static escape analyses

All are unsafe for space complexity in the presence of deep recursion

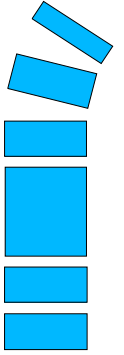
Many are unsafe for space complexity around loops, especially those with deep stack allocation.

All work best when they can assume global knowledge of the program, but we want to support dynamic-loading, run-time generated code and interactive development

Often encourage poor O-O design: Don't work well with factory methods and prefer large monolithic methods.

Language specific/Ineffective/Difficult for programmer to understand

My proposal:



Can be made 100% safe for space complexity

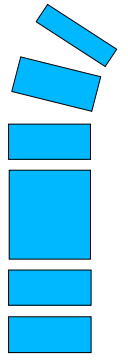
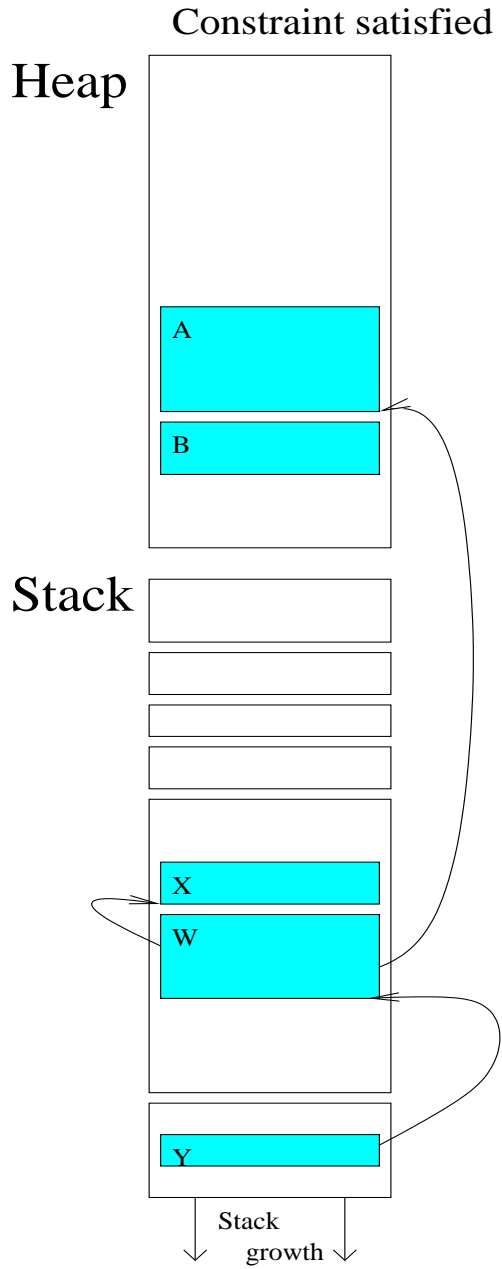
Requires no global knowledge: All analysis is simple and *intraprocedural*

Uses a simple write barrier and no read barrier

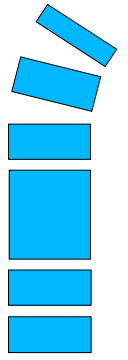
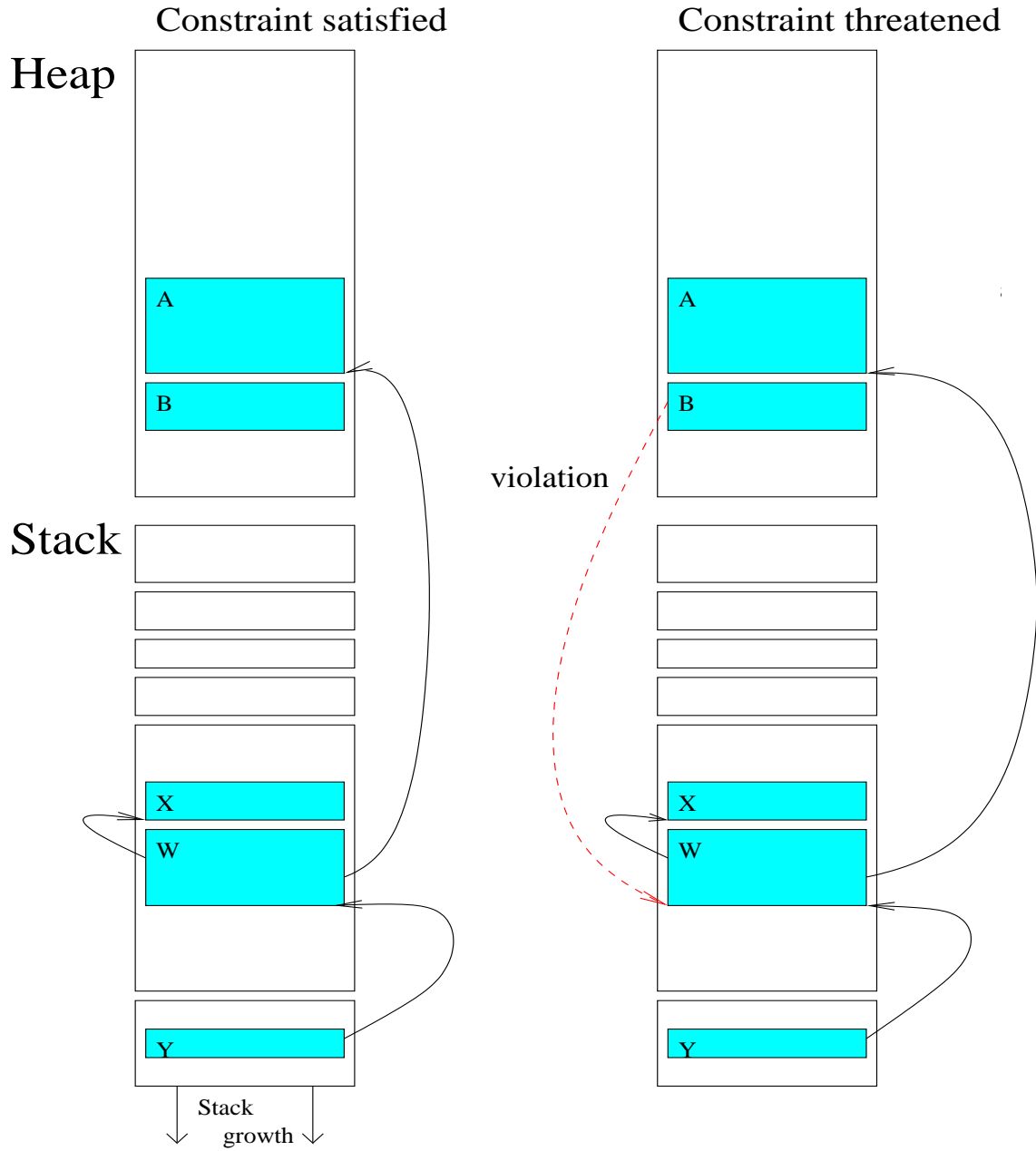
Achieves good stack allocation rates

Optimistic and heuristic-based: We do not need proofs in order to stack allocate objects.

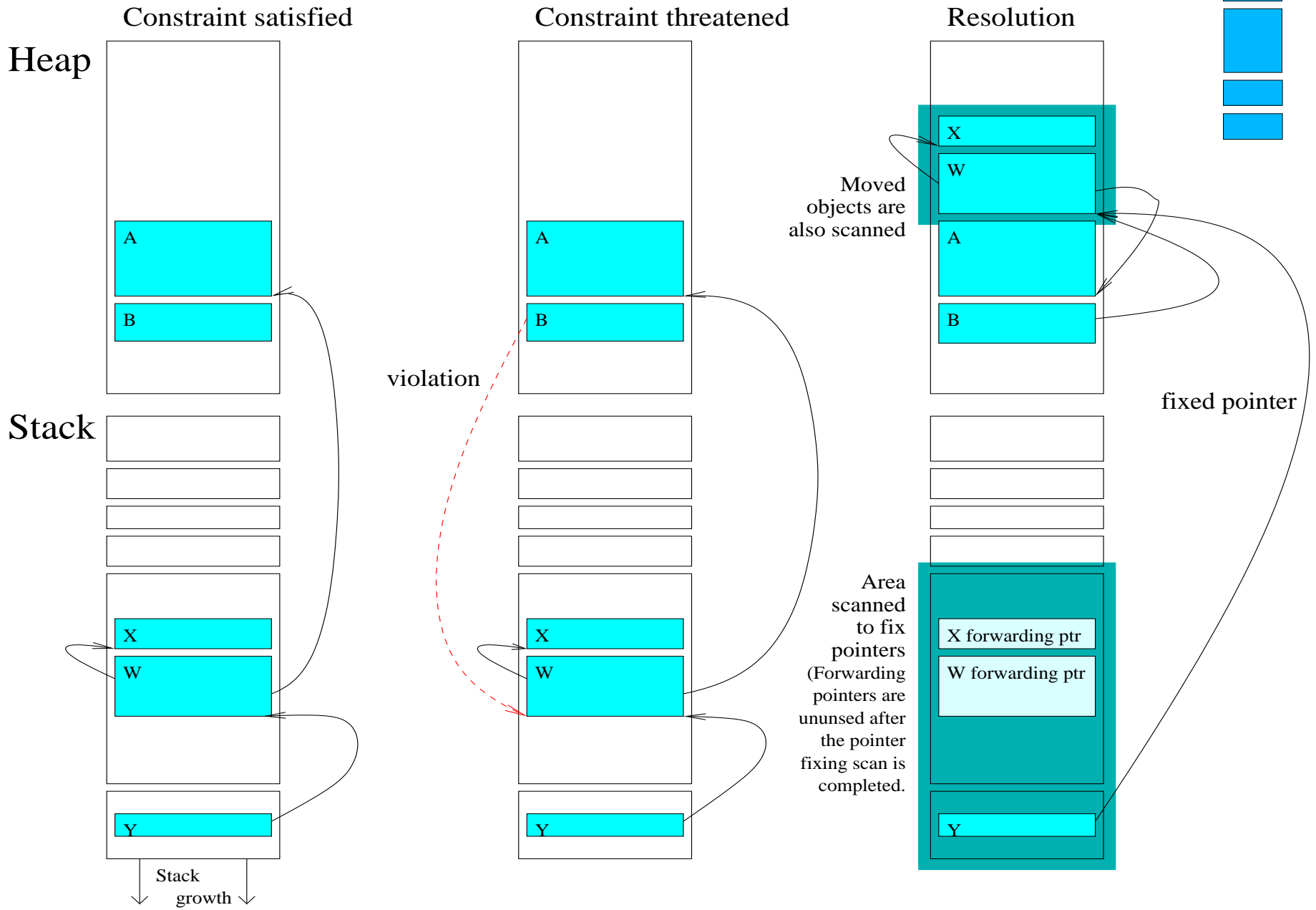
Modified lazy alloc



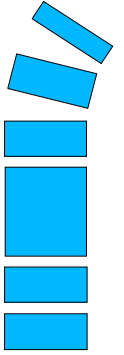
Modified lazy alloc



Modified lazy alloc



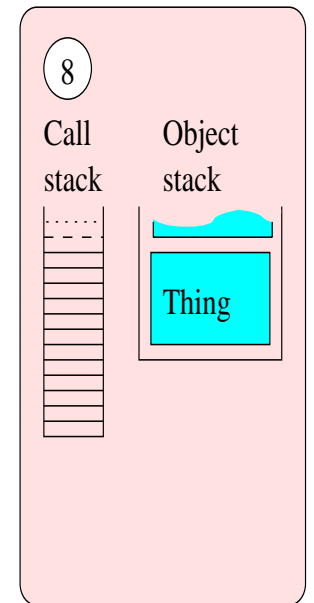
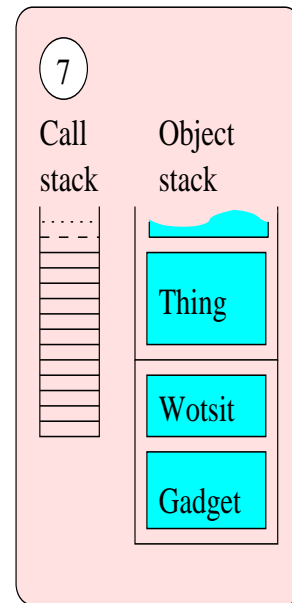
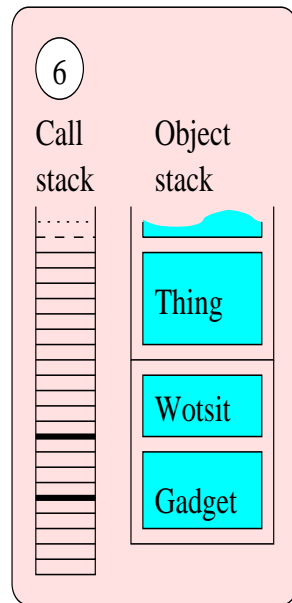
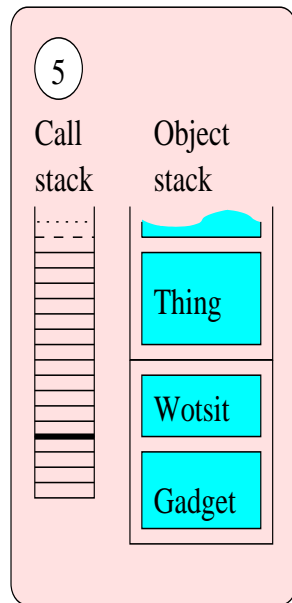
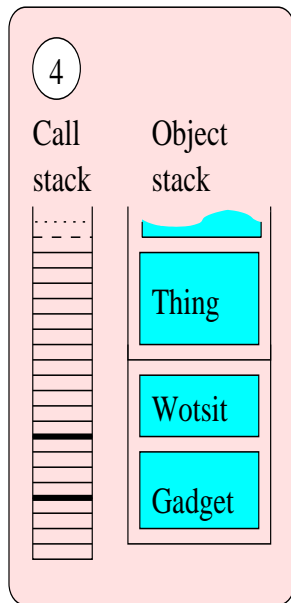
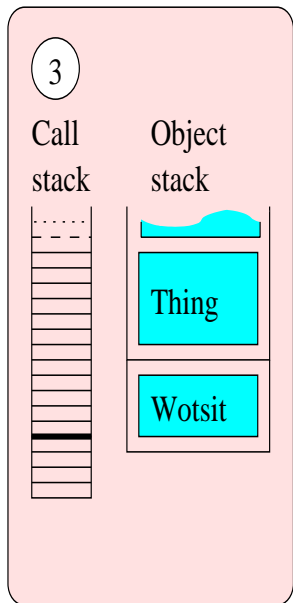
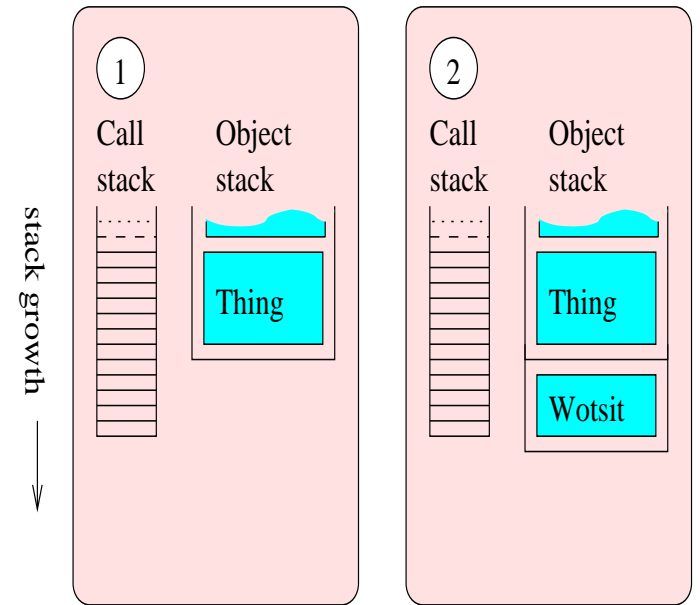
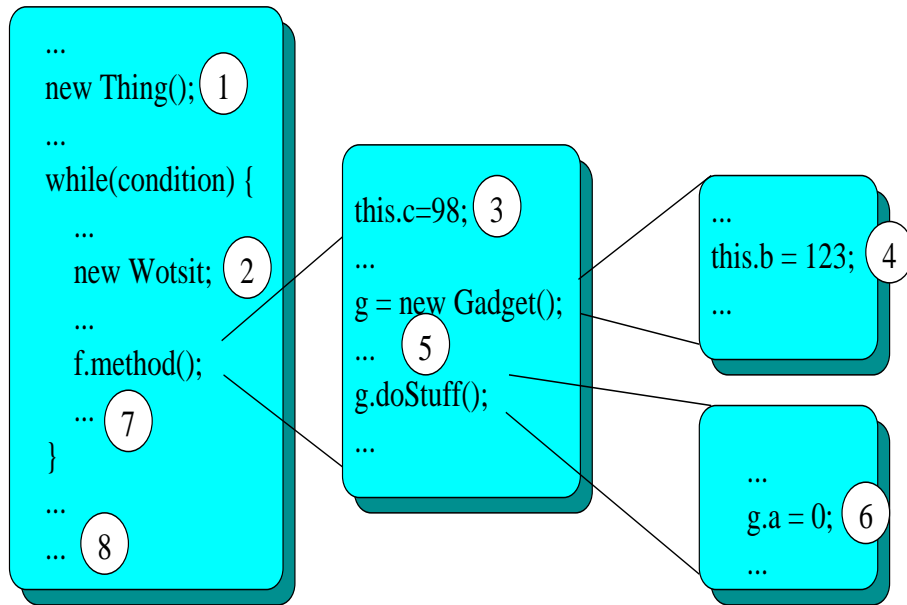
Why scan instead of read barriers?

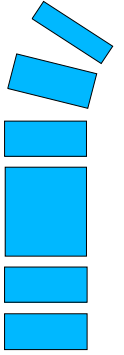


With heuristics we can reduce scans to a minimum

Read barriers cause code size blow up and slow down the program

Perhaps this is wrong!





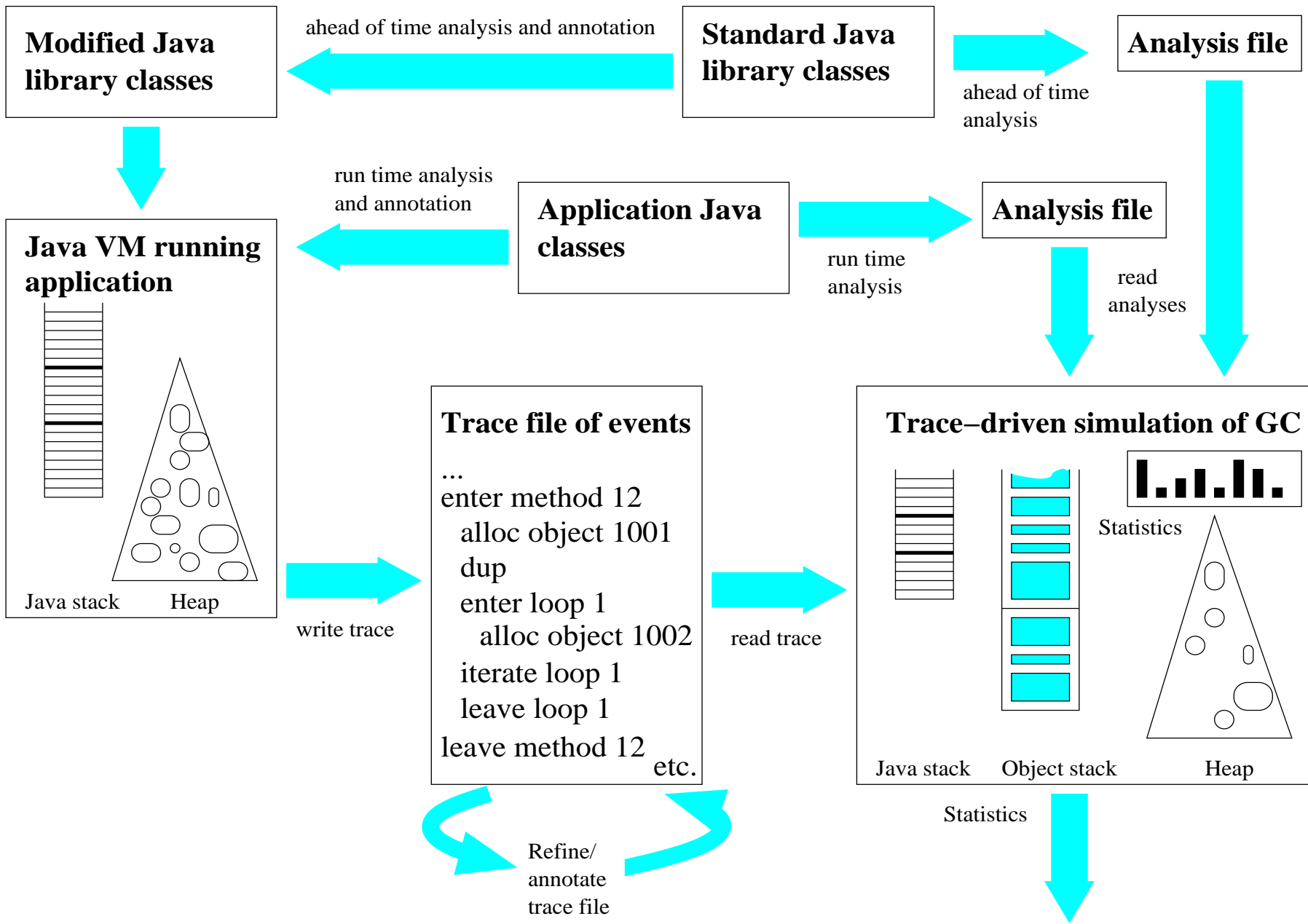
Why base allocation regions on loops rather than method invocations?

Simpler: We have to treat loops specially anyway in order to avoid space complexity issues

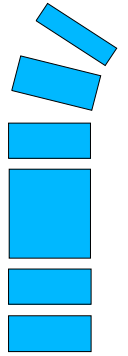
Better O-O: Method invocations are fundamental to O-O design. We don't want our optimiser to punish their use.

Less overhead: Java programs have many times more method invocations than (non-trivial) loop iterations (page 71)

More effective: We stack allocate more data than Hendren and Qian who are method invocation-based (page 72)



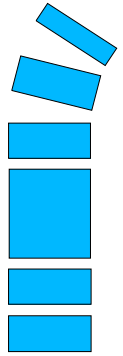
Test programs



Program	<i>Memory allocated (bytes)</i>	<i>Objects allocated</i>	<i>Read barriers</i>	<i>Write barriers</i>	<i>Method invocations</i>	<i>Stack regions destroyed</i>	<i>Non-empty stack regions destroyed</i>
202 jess	3,969,853	101,592	15,620,216	485,506	5,867,568	4,296,007	35,344
209 db	3,783,533	113,273	5,439,867	503,179	1,216,964	498,671	53,757
213 javac	7,602,866	214,734	3,566,420	517,346	2,982,147	733,094	34,447
228 jack	20,003,402	694,988	7,671,859	1,277,803	7,119,896	1,764,054	301,992

Cartesian

0	1	2	3	4	
128	129	130	131	132	
256	257	258	259	260	
384	385	386	387	388	
512	513	514	515	516	

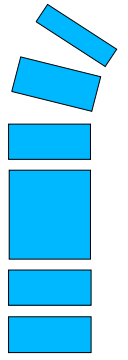


Peano

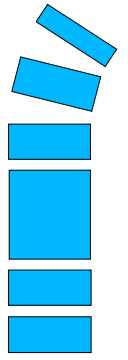
0	3	4	5	58	
1	2	7	6	57	
14	13	8	9	54	
15	12	11	10	53	
16	17	30	31	32	

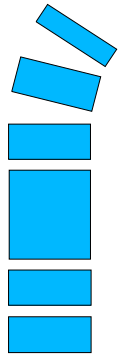
[Peano curve] cannot possibly be grasped by intuition; it can only be understood by logical analysis. *H. Hahn*

		Zero tolerance	Calling method	Zero tolerance with feedback	Calling method with feedback	Oracle
202 jess	Allocated and deallocated on stack	8.39%	12.34%	8.34%	12.16%	55.58%
	Evicted from stack	0.42%	1.09%	0.03%	0.06%	0.18%
	Data scanned to fix pointers	4.80%	6.72%	0.06%	0.22%	0.33%
	Stack scanned for semispace GC	3.61%	3.51%	3.59%	3.51%	2.11%
	Maximum region stack size	15kbyte	46kbyte	3kbyte	5kbyte	11kbyte
	Average regions scanned per eviction	1.0522	1.0483	1.0000	1.0286	1.0000
209 db	Allocated and deallocated on stack	8.48%	8.65%	8.32%	8.48%	8.55%
	Evicted from stack	0.20%	0.61%	0.00%	0.01%	0.16%
	Data scanned to fix pointers	0.92%	1.53%	0.01%	0.02%	0.07%
	Stack scanned for semispace GC	1.37%	1.37%	1.37%	1.37%	1.37%
	Maximum region stack size	16kbyte	31kbyte	3kbyte	3kbyte	9kbyte
	Average regions scanned per eviction	1.0000	1.0000	1.0000	1.0000	1.0000
213 javac	Allocated and deallocated on stack	21.09%	34.02%	21.00%	33.91%	64.73%
	Evicted from stack	0.54%	0.91%	0.18%	0.31%	0.16%
	Data scanned to fix pointers	0.85%	1.96%	0.31%	1.15%	0.72%
	Stack scanned for semispace GC	2.43%	2.16%	2.43%	2.20%	1.50%
	Maximum region stack size	27kbyte	27kbyte	14kbyte	14kbyte	22kbyte
	Average regions scanned per eviction	0.9455	0.9771	0.8973	0.9634	1.0000
228 jack	Allocated and deallocated on stack	27.05%	60.61%	27.02%	60.57%	62.99%
	Evicted from stack	0.06%	0.16%	0.00%	0.00%	0.07%
	Data scanned to fix pointers	3.33%	3.34%	0.00%	0.05%	0.11%
	Stack scanned for semispace GC	0.87%	0.50%	0.85%	0.51%	0.47%
	Maximum region stack size	66kbyte	92kbyte	51kbyte	62kbyte	68kbyte
	Average regions scanned per eviction	0.9846	1.0181	1.0000	1.0000	1.0000

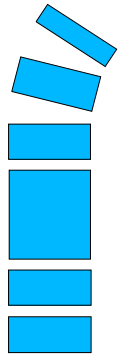


		ZT	CM	ZTwF	CMwF	Orcl
202 jess	Write barriers that were null	1.55%	1.55%	1.55%	1.55%	1.55%
	Write barriers that trigger	12.09%	12.50%	12.12%	12.49%	10.95%
	... in heap	0.02%	0.02%	0.02%	0.02%	0.02%
	... in nursery	11.43%	11.28%	11.51%	11.49%	6.37%
	... in top frame	0.02%	0.60%	0.01%	0.41%	4.04%
	... in other frames	0.00%	0.00%	0.00%	0.00%	0.00%
	... intergeneration	0.59%	0.57%	0.58%	0.57%	0.52%
	... Baker violation	0.03%	0.04%	0.00%	0.01%	0.00%
209 db	Write barriers that were null	1.27%	1.27%	1.27%	1.27%	1.27%
	Write barriers that trigger	93.66%	93.79%	93.79%	93.79%	93.80%
	... in heap	73.75%	73.76%	73.58%	73.58%	73.58%
	... in nursery	17.99%	17.96%	18.14%	18.14%	18.13%
	... in top frame	0.00%	0.17%	0.00%	0.01%	0.03%
	... in other frames	0.00%	0.00%	0.00%	0.00%	0.00%
	... intergeneration	1.91%	1.90%	2.07%	2.07%	2.07%
	... Baker violation	0.00%	0.00%	0.00%	0.00%	0.00%
213 javac	Write barriers that were null	15.33%	15.33%	15.33%	15.33%	15.33%
	Write barriers that trigger	23.44%	23.72%	23.45%	23.86%	27.42%
	... in heap	2.57%	2.58%	2.57%	2.56%	2.31%
	... in nursery	18.44%	17.97%	18.62%	18.32%	15.21%
	... in top frame	0.35%	1.07%	0.20%	0.91%	7.92%
	... in other frames	0.01%	0.01%	0.01%	0.01%	0.01%
	... intergeneration	2.03%	2.00%	2.03%	1.99%	1.93%
	... Baker violation	0.04%	0.10%	0.02%	0.08%	0.04%
228 jack	Write barriers that were null	26.86%	26.86%	26.86%	26.86%	26.86%
	Write barriers that trigger	5.04%	5.24%	5.09%	5.34%	4.88%
	... in heap	0.24%	0.10%	0.24%	0.19%	0.20%
	... in nursery	3.44%	3.43%	3.51%	3.51%	2.86%
	... in top frame	0.03%	0.40%	0.02%	0.33%	0.50%
	... in other frames	0.07%	0.07%	0.07%	0.07%	0.07%
	... intergeneration	1.25%	1.23%	1.25%	1.25%	1.24%
	... Baker violation	0.01%	0.01%	0.00%	0.00%	0.01%



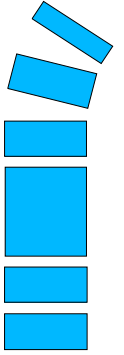


		Zero tolerance	Calling method	Zero tolerance with feedback	Calling method with feedback	Oracle
	Read barriers $\times 4 \times 10^3$	Bytes scanned $\times 10^3$ (including bytes skipped)				
202 jess	62480	191 (356)	267 (537)	2 (4)	9 (15)	13 (36)
209 db	21760	35 (71)	58 (240)	0 (1)	1 (1)	3 (15)
213 javac	14264	64 (178)	149 (445)	23 (43)	88 (131)	55 (149)
228 jack	30688	666 (1998)	668 (3150)	0 (0)	10 (41)	22 (44)



Program	Zero tolerance Section 6.2	Blanchet (safe) Section 2.8.4	Gay & Stensgaard Section 2.8.5	Hendren & Qian Section 7.1
202 jess	8.4%	21%	10.5%/5.3%	6.50%
209 db	8.5%			0.74%
213 javac	21.1%	13%		8.80%
228 jack	27.1%	20%		22.87%

Conclusions



Loop-based stack allocation is good at finding stack-allocatable objects and leads to small stack sizes

The read barrier in Baker's optimistic stack allocation can be replaced by stack scanning combined with allocation heuristics. This is likely to be more efficient.

Optimistic stack allocation sidesteps the issues around whole-program analysis and space complexity usually associated with stack allocation based on static escape analysis.