Empirical Evaluation of an Approach to Resource Constrained Test Suite Execution

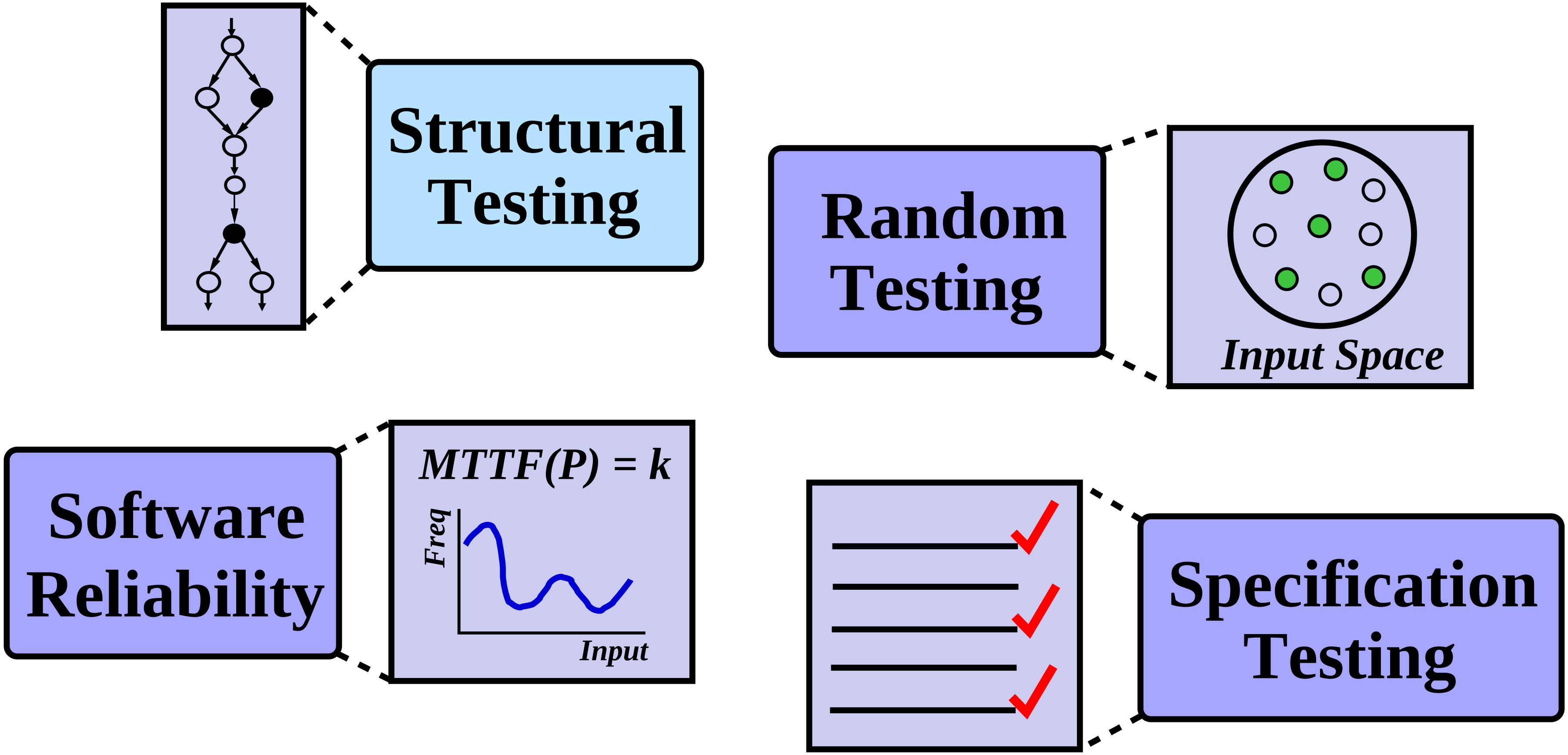
Gregory M. Kapfhammer
Department of Computer Science
Allegheny College

(in conjunction with Mary Lou Soffa and Daniel Mossé)

Looking Ahead

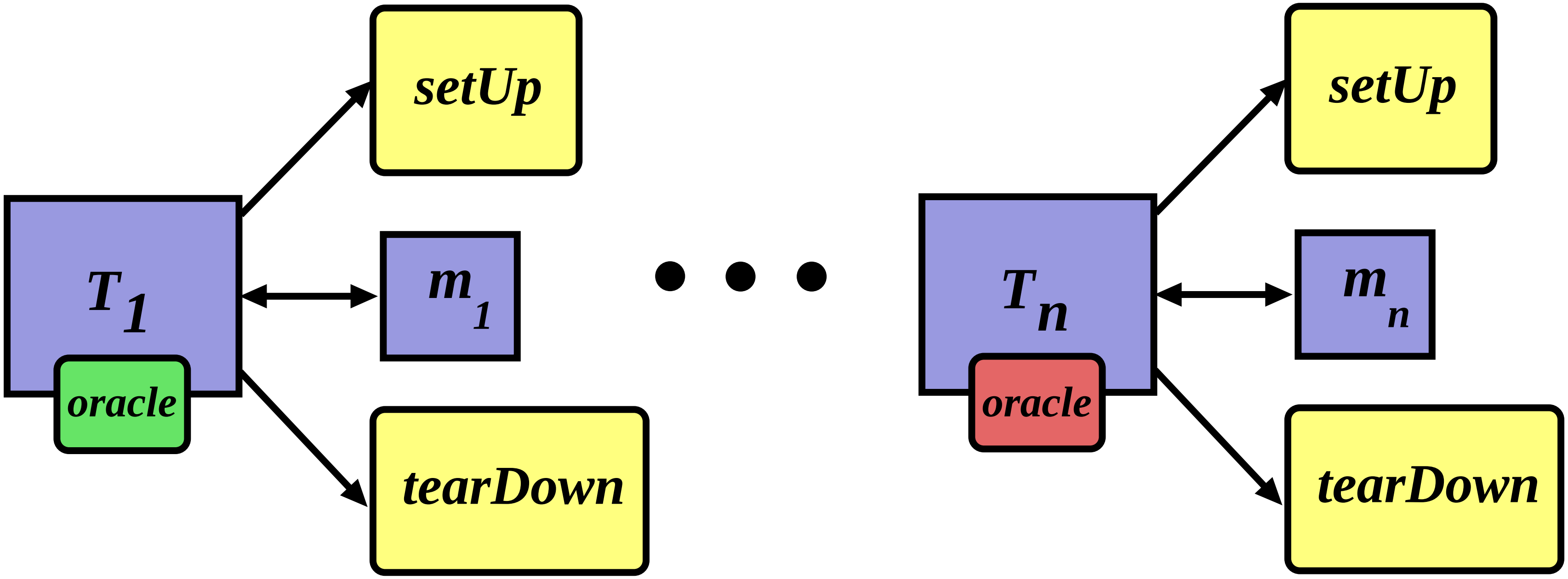
- Use of native code unloading during test suite execution in a resource constrained environment
- Identification of the testing techniques that yield the greatest reduction in execution time and native code size
- Characterization of how software applications and test suites restrict and/or support resource constrained testing
- Cost-benefit analysis for the use of sample-based and exhaustive profiles of program testing behavior
- **Executes test suites faster when memory resources are limited!**

Traditional Testing Techniques



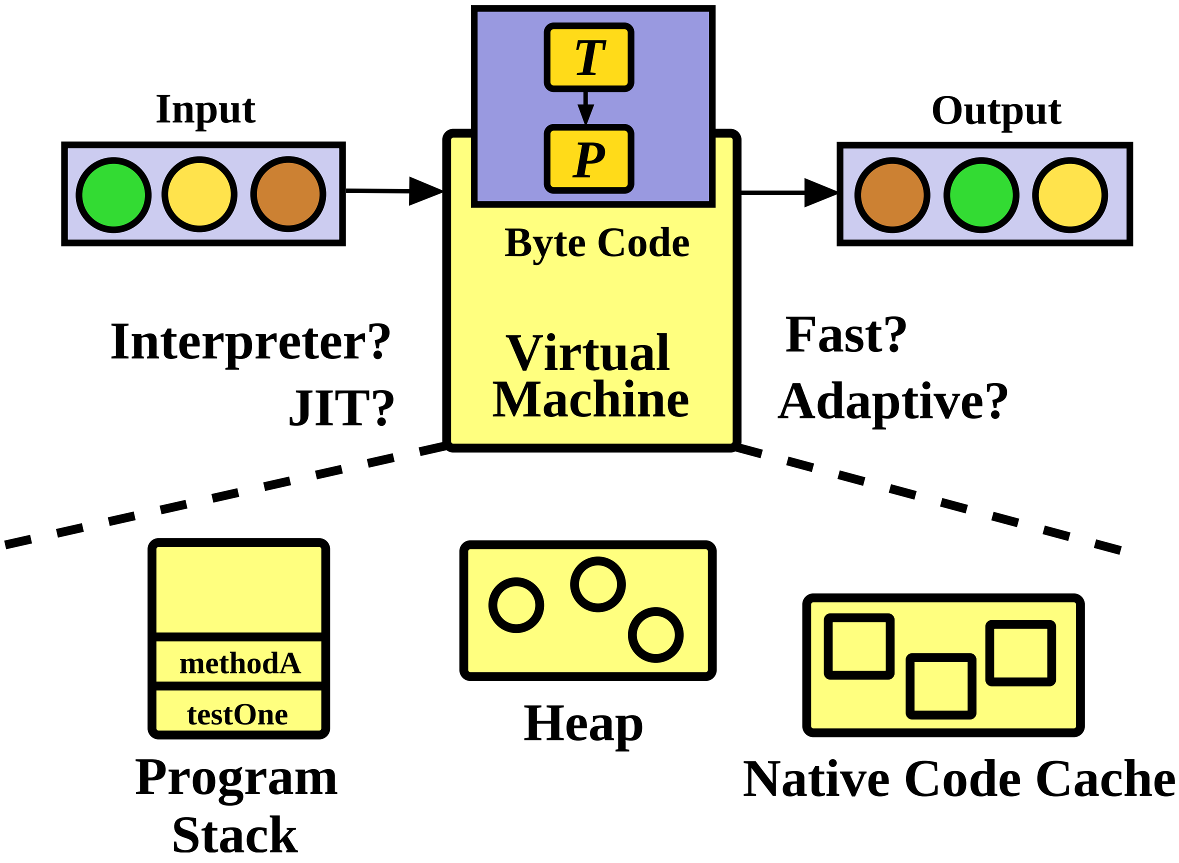
- Different approaches to establishing a confidence in the correctness of and finding faults within software

Test Suite Execution



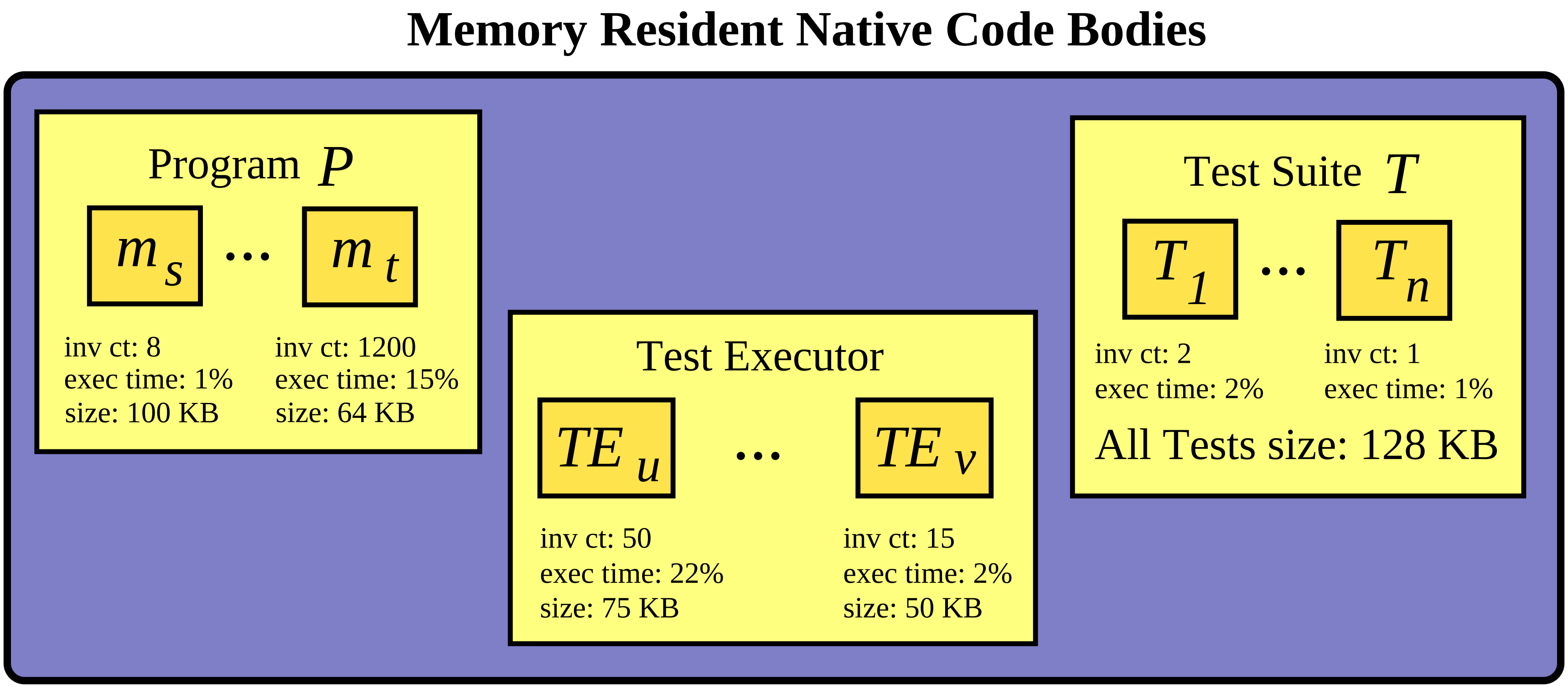
- A test executes a method and uses an oracle to determine if the method's output is correct
- Test suite execution frameworks exist for many different programming languages (e.g., JUnit for Java)

Test Suite Execution with a JVM



- During testing the JVM must manage limited resources

Resource Constrained Testing

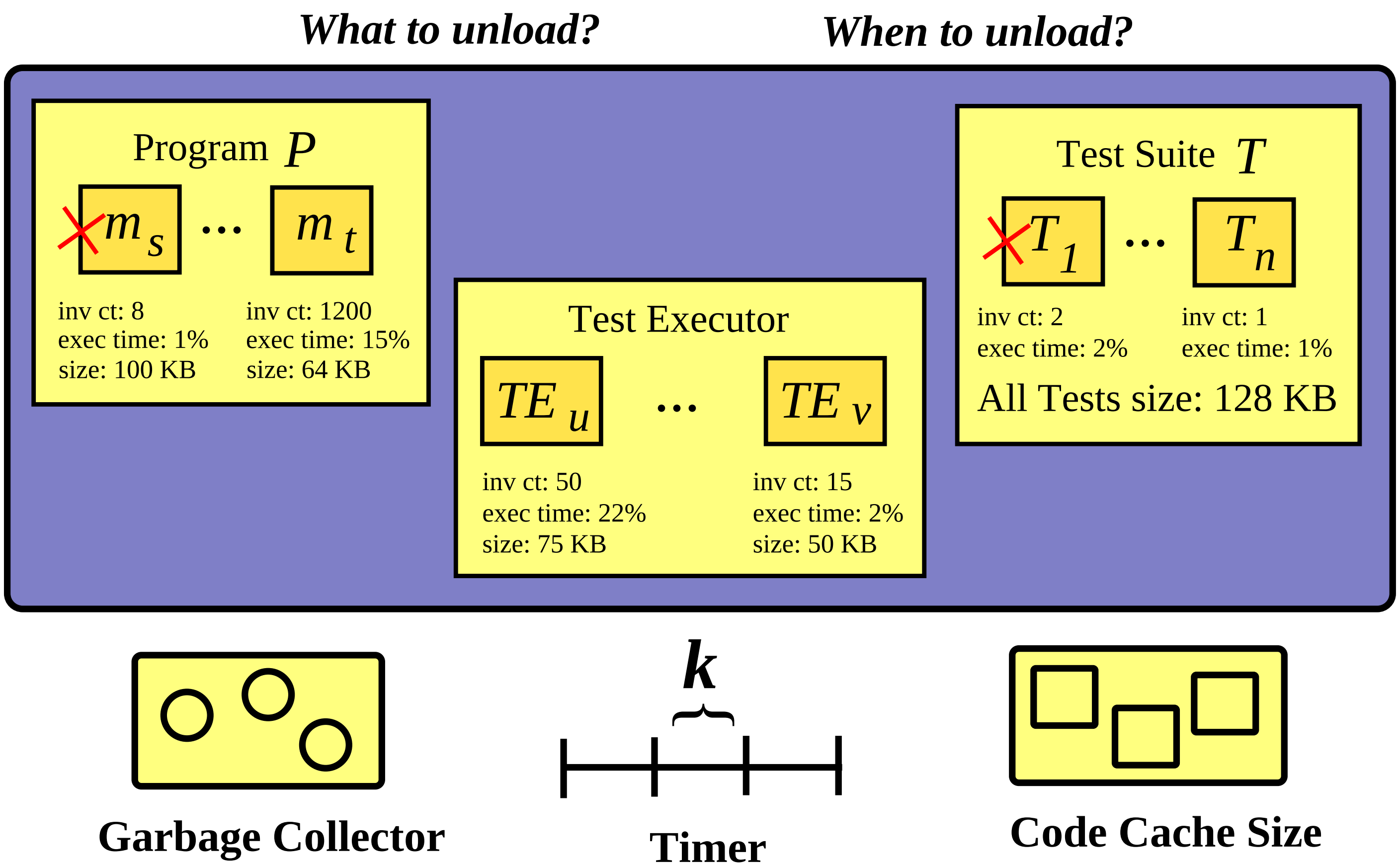


- JIT compiler produces native code that exhausts limited memory resources
- Frequent invocation of GC increases testing time

Test Suite Execution Strategies

- **Omit tests?** - Could reduce overall confi dence in the correctness of P
- **Use non-constrained environment?** - Defects related to P's interaction with environment might not be isolated
- **Execute tests individually?** - Might increase overall testing time and violate test order dependencies
- **Unload with offline profile?** - Not useful if P and T change frequently during regression testing
- **Our Approach**: Use online behavior profi les to guide the unloading of native code

Testing with Native Code Unloading



- *When* - three separate techniques and *What* - behavior profile stored in code body (Zhang and Krintz)

Unloading Configurations: *GC* and *TM*

$$\{S, X\} - \{GC, TM\} = \langle C, UC, U, H \rangle$$

Parameter	**Meaning**
C	init period (GC cycles, secs)
UC	init unload freq (GC cycles, secs)
U	non-init unload freq (GC cycles, secs)
H	heap residency threshold ($\%$)

- Use GC execution or timer expiration to trigger code unloading

Unloading Configuration: *CS*

$$\{S, X\} - CS = \langle Z_{init}, Z_{incr}, U_{CS} \rangle$$

Parameter	**Meaning**
Z_{init}	init code cache size (bytes)
Z_{incr}	code cache increment size (bytes)
U_{CS}	unload session resize trigger (count)

- Unload when native code cache is full

Experiment Goals and Design

- **Research Question:** Can an adaptive code unloading JVM reduce time and space overheads associated with memory constrained testing?
- **Experiment Metrics**: percent reduction in time, $\mathcal{T}_R^{\%}(P, T)$ and space, $\mathcal{S}_R^{\%}(P, T)$
- Jikes RVM 2.2.1, JUnit 3.8.1, GNU/Linux 2.4.18
- Sample-based (*S*) and exhaustive (*X*) program profi les
- Timer (*TM*), garbage collection (*GC*), and code cache size (*CS*) triggers the unloading technique

Case Study Applications

Name	*Min* Size (MB)	# Tests	NCSS
`UniqueBoundedStack` (UBS)	8	24	362
`Library` (L)	8	53	551
`ShoppingCart` (SC)	8	20	229
`Stack` (S)	8	58	624
`JDepend` (JD)	10	53	2124
`IDTable` (ID)	11	24	315

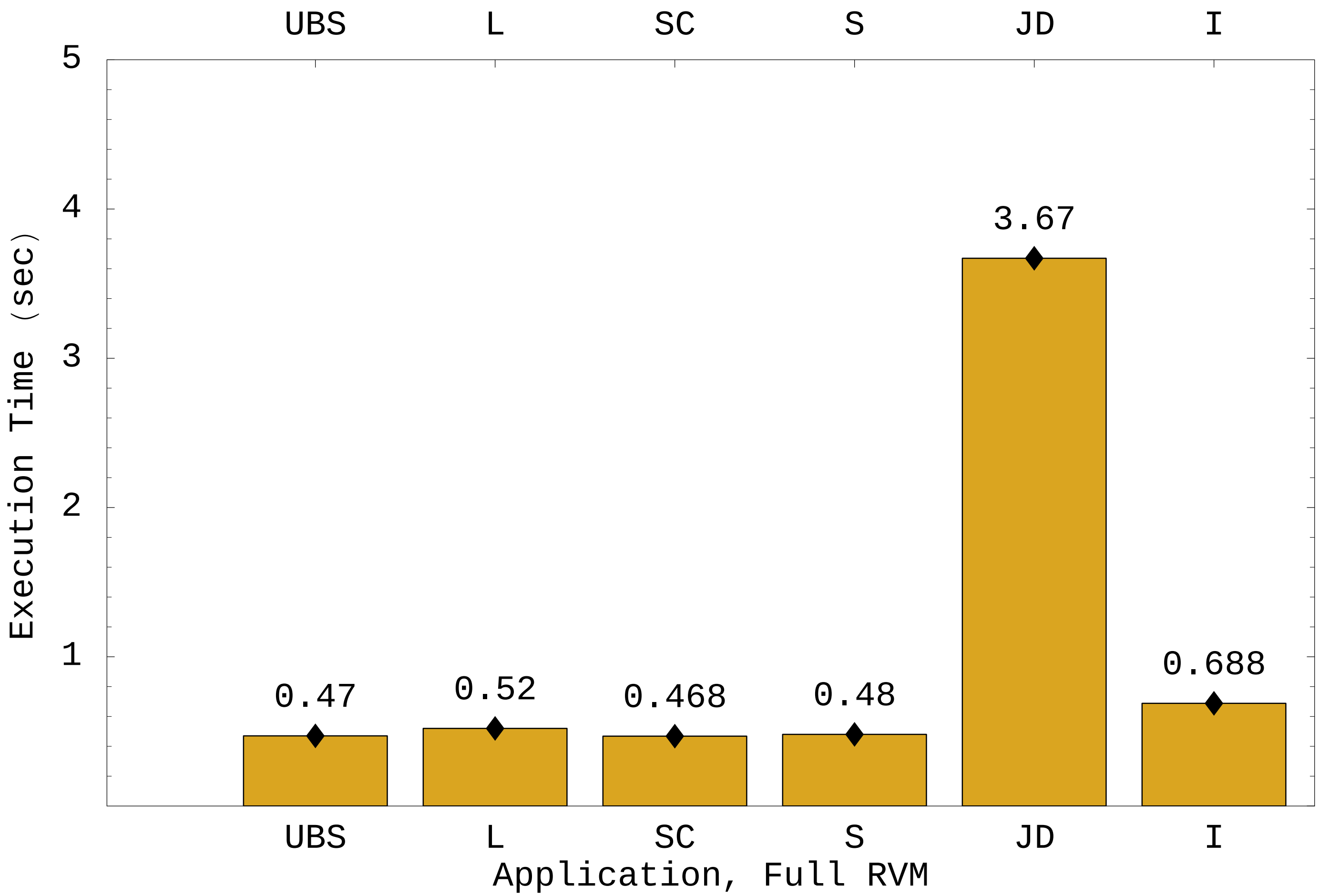
- Empirically determined the *MIN* Jikes RVM heap size

Native Code Unloading Configurations

Name	***GC***	***CS***	***TM***
UBS	$\langle 4, 1, 1, 0.0 \rangle$	$\langle 49370, 512, 5 \rangle$	$\langle 3, .5, 1, 0.0 \rangle$
L	$\langle 5, 1, 3, 0.0 \rangle$	$\langle 49370, 512, 5 \rangle$	$\langle 3, .5, 1, 0.0 \rangle$
SC	$\langle 3, 1, 1, 0.0 \rangle$	$\langle 49370, 512, 5 \rangle$	$\langle 2, .5, 1, 0.0 \rangle$
S	$\langle 4, 1, 1, 0.0 \rangle$	$\langle 49370, 512, 5 \rangle$	$\langle 3, .5, 1, 0.0 \rangle$
JD	$\langle 8, 1, 4, 0.0 \rangle$	$\langle 49370, 512, 5 \rangle$	$\langle 3, .5, 1, 0.0 \rangle$
ID	$\langle 1, 1, 3, 0.0 \rangle$	$\langle 65536, 8192, 5 \rangle$	$\langle 2, .5, 1, 0.0 \rangle$

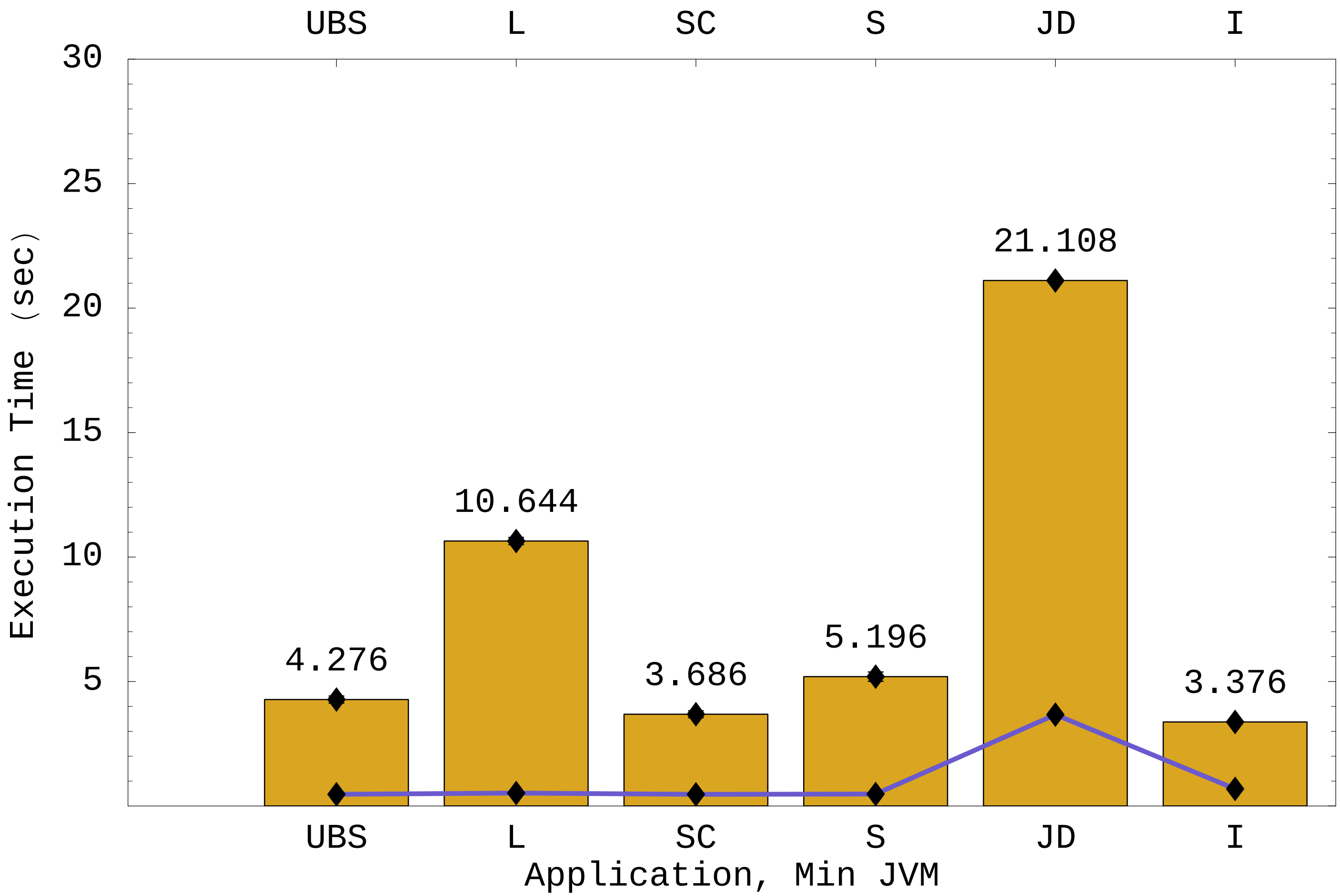
- Ten minutes (or less) of configuration time for each technique
- *S* and *X* use same configuration to a ensure fair comparison

Testing Time Overhead: *Full* RVM



- When memory is not constrained, testing time is acceptable

Testing Time Overhead: *Min* RVM



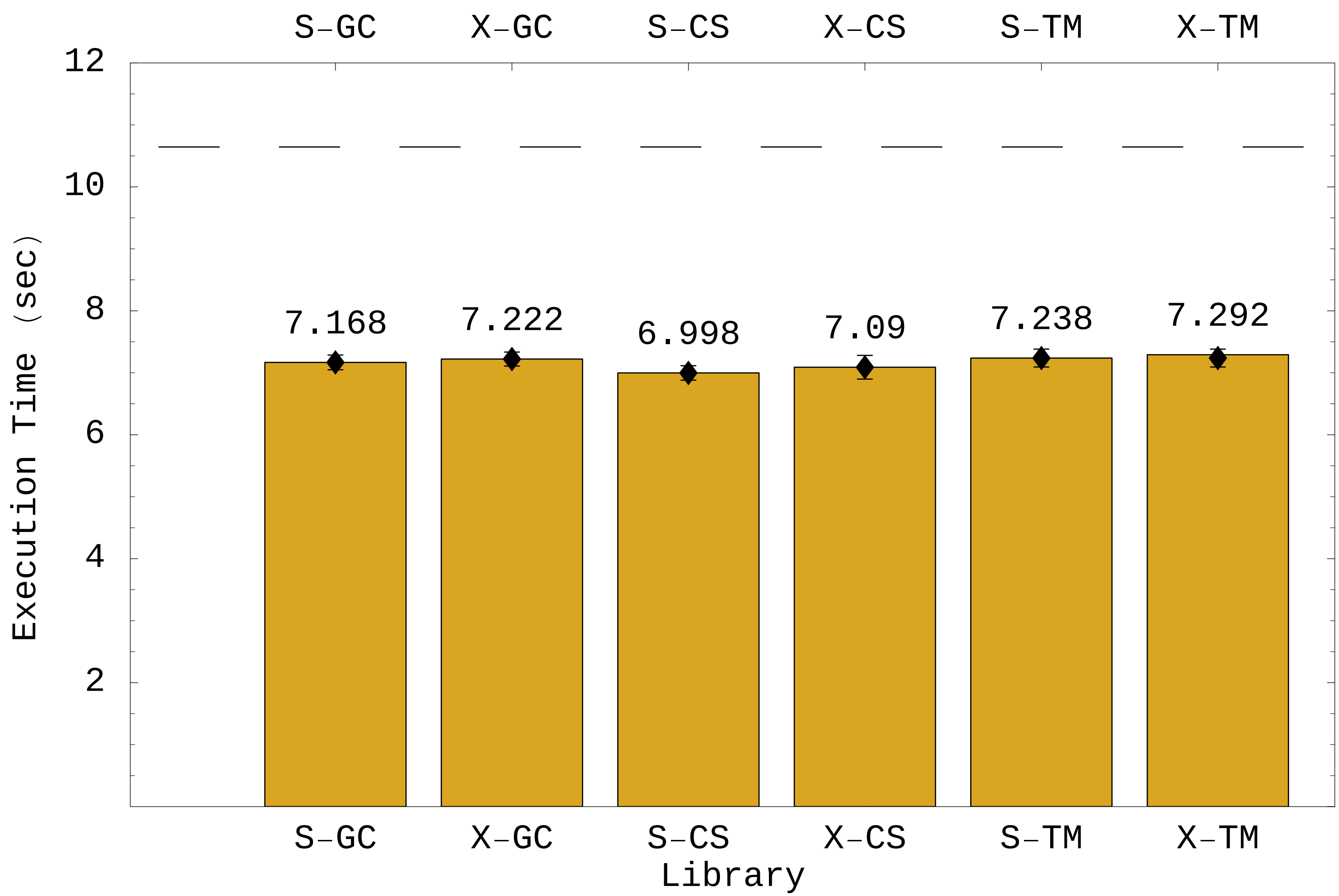
- Testing time increases signifi cantly when memory is *Min*

Summary of Reductions for `Library`

Name	$\mathcal{T}_R^{\%}(P,T)$	$\mathcal{S}_R^{\%}(P,T)$
S-GC	32.7	78.8 ✓
X-GC	32.1	65.0
S-TM	32.0	72.8
X-TM	31.5	62.3
S-CS	34.3 ✓	61.4
X-CS	33.4	59.8

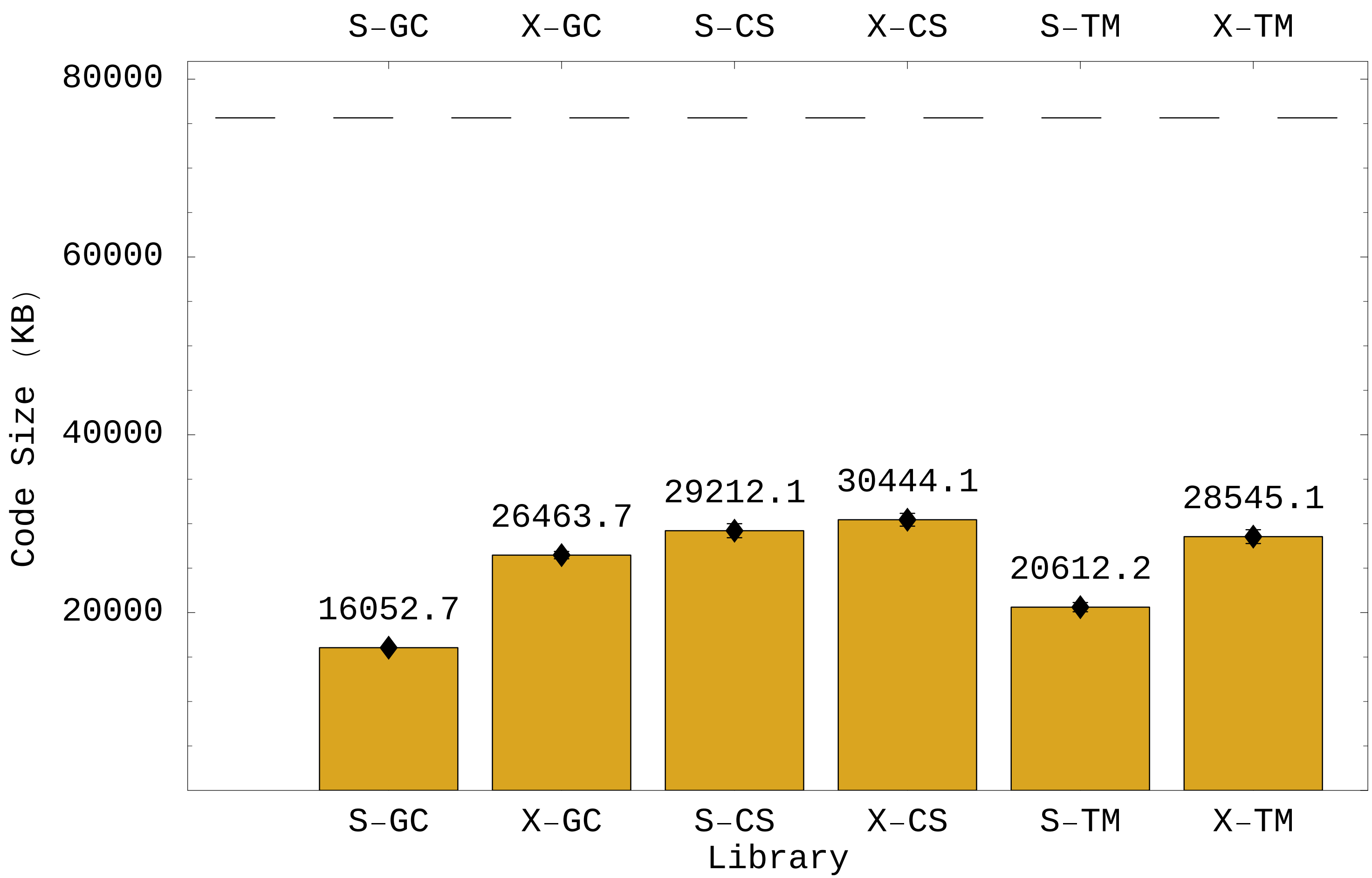
- Signifi cant reductions in time and space required for testing

Testing Time Overhead: Library



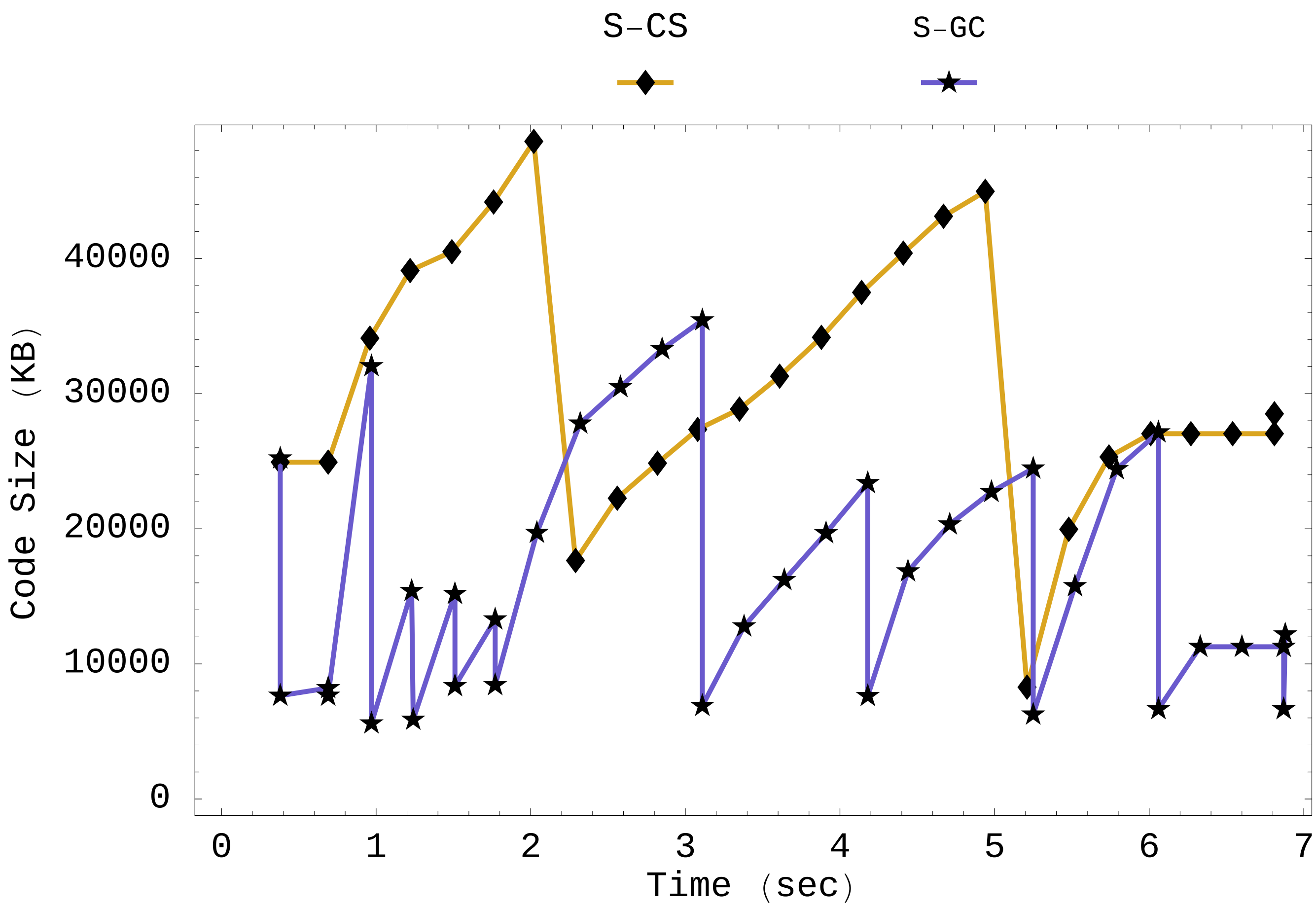
- *S* vs. *X*: Similar reductions for all code unloading techniques

Testing Space Overhead: Library



- Code size reduction does not always yield best testing time

Code Size Fluctuation: `Library`



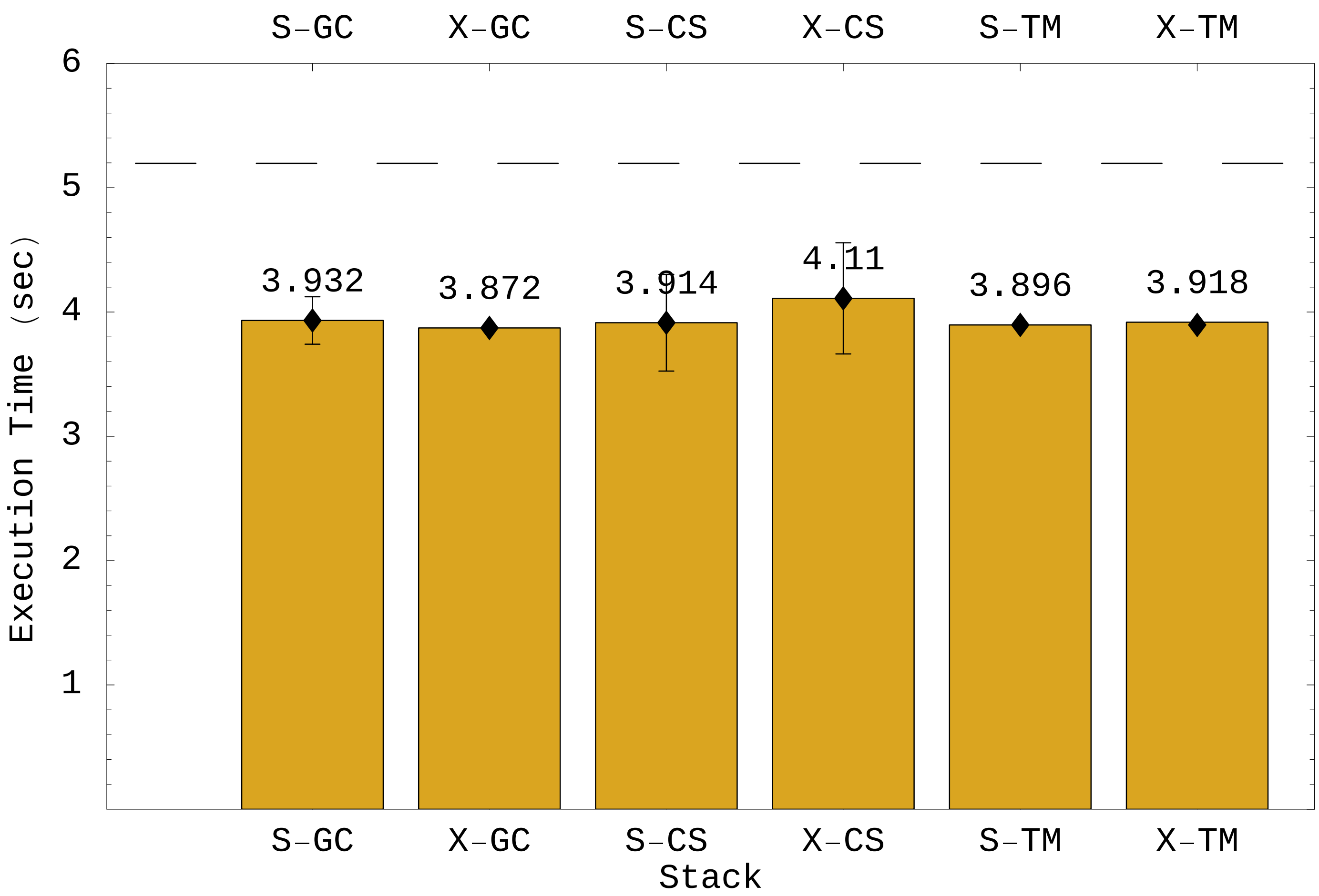
- *S-GC* causes code size fluctuation that increases testing time

Summary of Reductions for `Stack`

Name	$\mathcal{T}_R^{\%}(P,T)$	$\mathcal{S}_R^{\%}(P,T)$
S-GC	24.3	79.0 ✓
X-GC	25.4	63.4
S-TM	25.0 ✓	64.9
X-TM	24.6	47.8
S-CS	24.7	61.6
X-CS	20.9	46.9

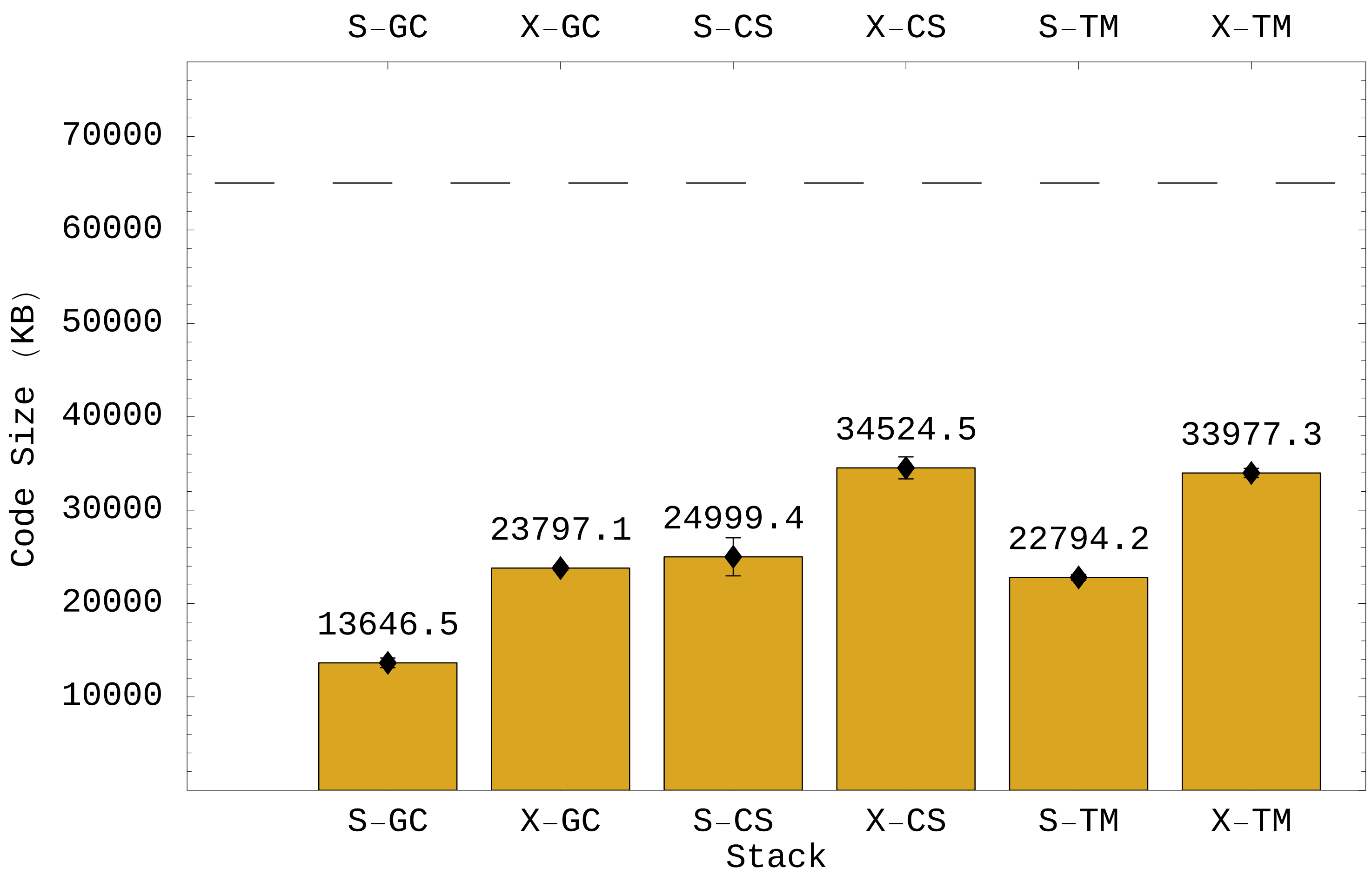
- Across all applications: *S-TM* or *S-CS* normally produce the largest reduction in testing time overhead

Testing Time Overhead: `Stack`



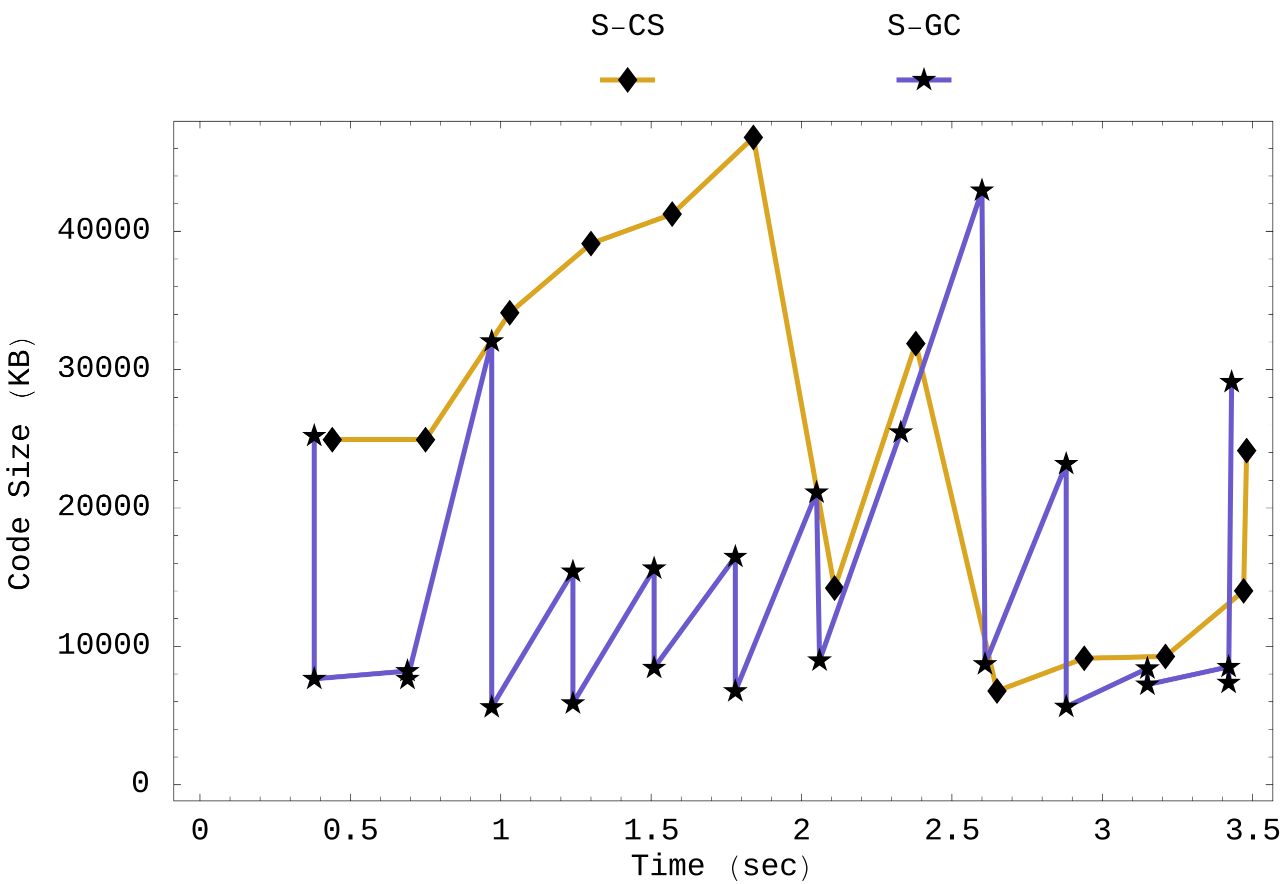
- *S* vs. *X*: Similar time reductions for another application

Testing Space Overhead: Stack



- *S-GC* also produces best code size reduction without creating greatest reduction in testing time

Code Size Fluctuation: `Stack`



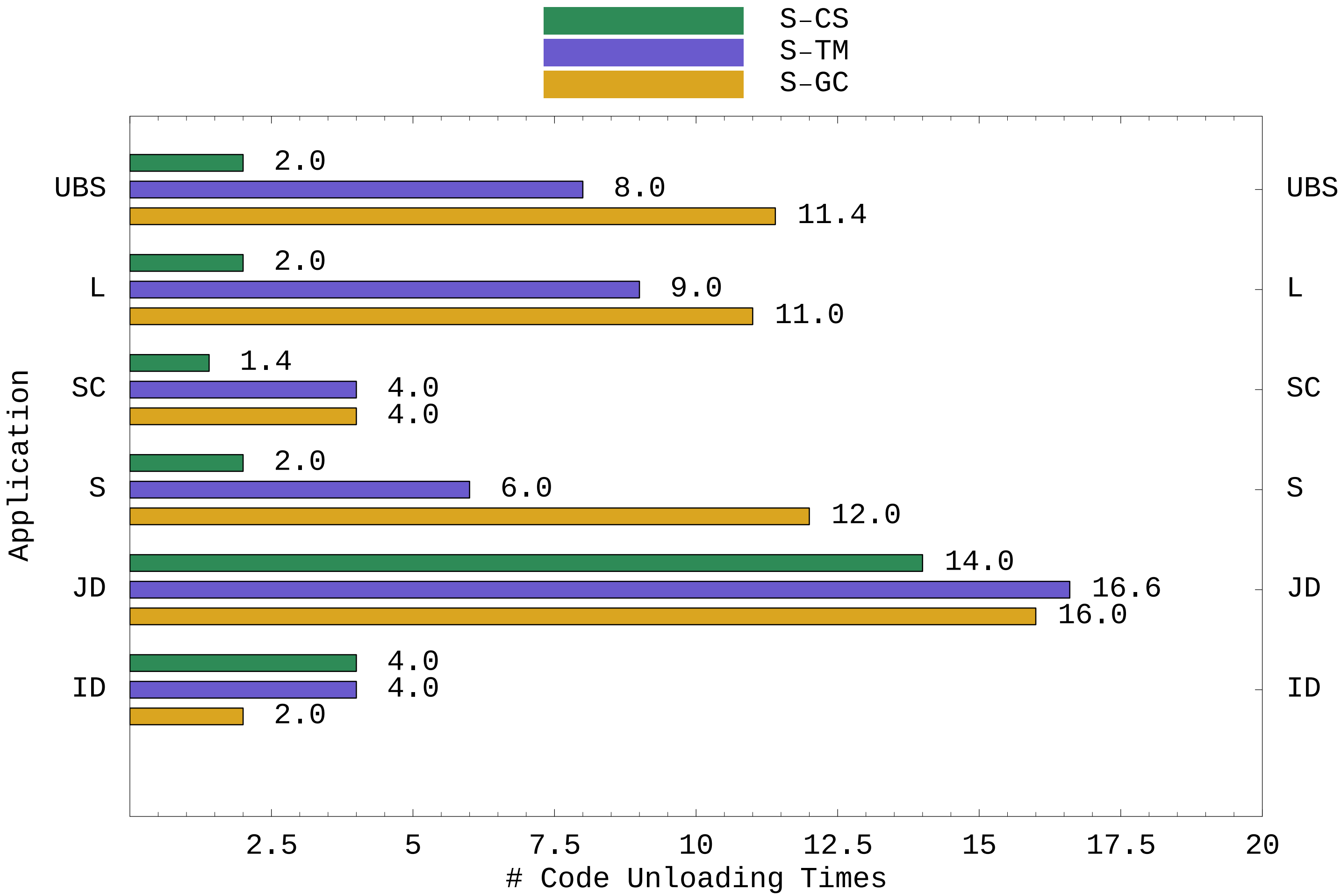
- *S-GC* also causes code size fluctuations in other applications

Summary of Reductions for `ID`

Name	$\mathcal{T}_R^{\%}(P,T)$	$\mathcal{S}_R^{\%}(P,T)$
S-GC	-1.1	42.5
X-GC	-1.1	26.7
S-TM	-1.2	44.5
X-TM	-.29 ✓	28.8
S-CS	-.77	51.4
X-CS	-1.4	61.4 ✓

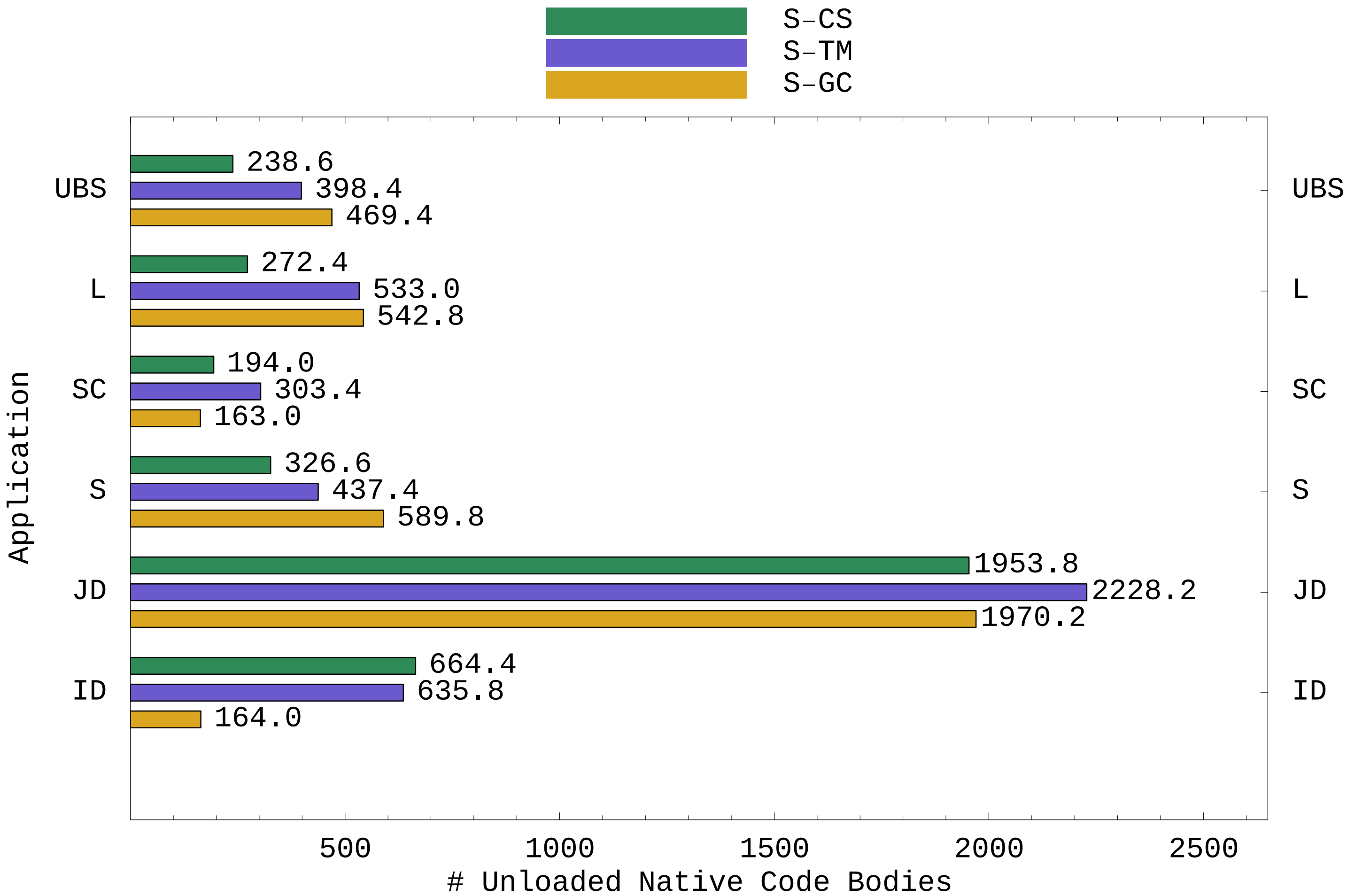
- Unloading can decrease native code size while still creating an overall increase in testing time

Number of Code Unloads



- All techniques cause `ID` to perform few unloading sessions

Number of Unloaded Code Bodies



- ID's large working set forces unloading of many code bodies

Summary of Reductions

Name	$\mathcal{T}_R^{\%}(P,T)$	$\mathcal{S}_R^{\%}(P,T)$
S-GC	16.1	68.4 ✓
X-GC	16.4	52.8
S-TM	17.1	62.6
X-TM	16.4	45.9
S-CS	17.6 ✓	58.8
X-CS	15.3	54.8

- Across all applications, adaptive code unloading techniques reduce both testing time and space overhead

Conclusions

- Dynamic compilation in JVMs can increase testing time if memory is constrained
- Adaptive unloading of native code often reduces memory overhead, avoids GC invocation, and reduces testing time
- Impact of unloading varies with respect to size of application's working set and program testing behavior
- Code unloading JVM can be rapidly configured to produce useful reductions in the time and space overheads of testing

Future Work

- Include new case study applications and test suites
- Experiments to measure the impact of garbage collection and heap compression algorithms (e.g., Jikes RVM MMTk)
- Regression test suite prioritization and reduction techniques that consider structural coverage *and* time or space overheads
- Real testing framework for emerging operating systems that support Java in ad hoc networks (e.g., MagnetOS)

Additional Resources

- Kapfhammer et al. Testing in Resource Constrained Execution Environments. In *IEEE/ACM Automated Software Engineering*. November 7 - 11, 2005.

http://cs.allegheny.edu/~gkapfham/research/juggernaut/