Speculative Multithreading in a Java Virtual Machine

Chris Pickett and Clark Verbrugge
School of Computer Science
McGill University

May 17, 2005

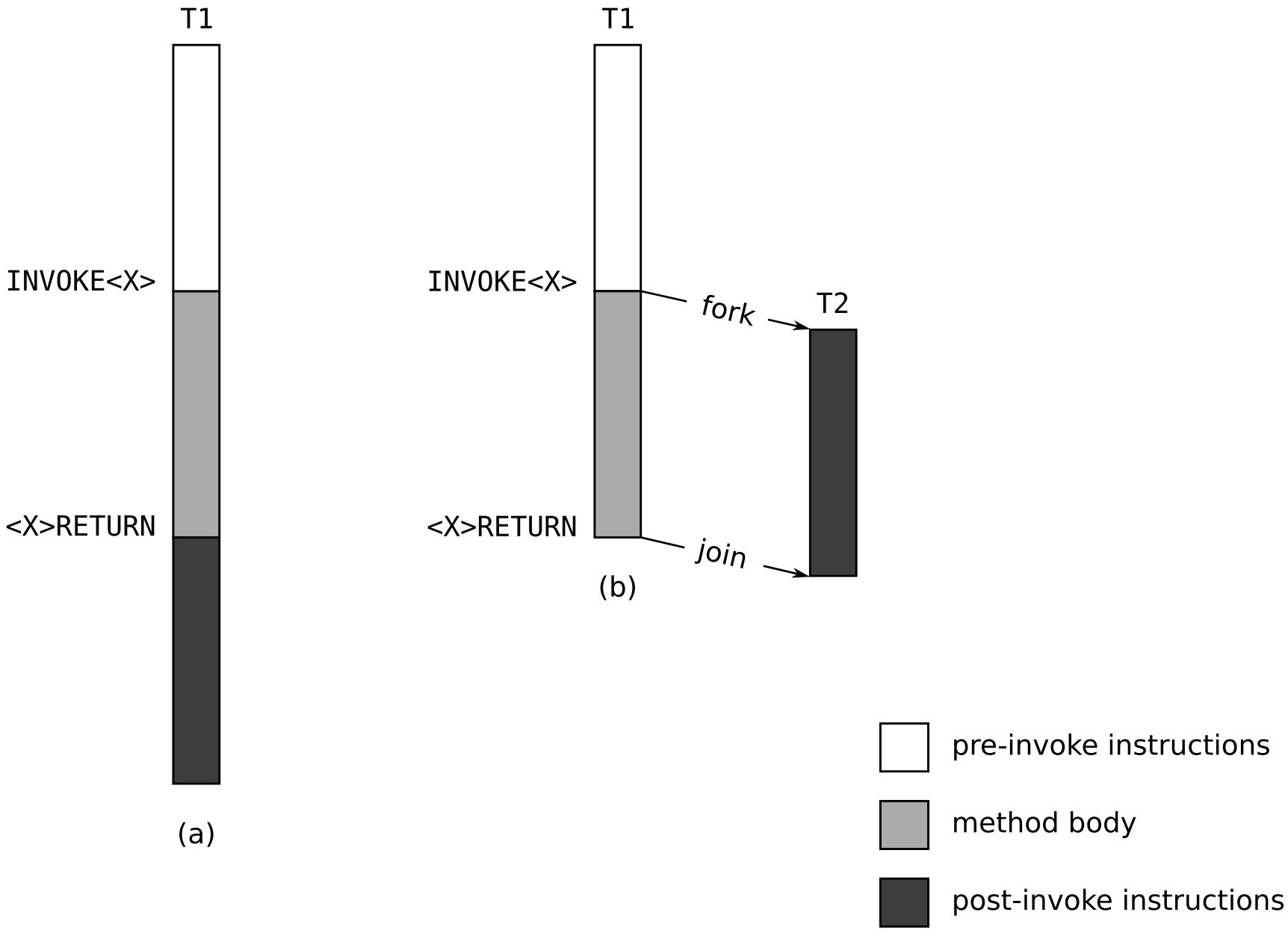
Outline

Outline

Motivation

- Speculative multithreading (SpMT) has great promise:
  - Dynamic parallelisation of irregular, non-numerical programs
  - Good potential for speedup in Java (1.5 to 5.0 over SPECjvm98 on a simulated 8-way machine).
- Simulated hardware is the primary target; software SpMT is rare.
- Not being hardware people, we wanted to try our hand at a software implementation.
- The Java Virtual Machine provides a convenient hardware abstraction layer.
- Decided to use SableVM, our lab's free/open source JVM.

Speculative Method-Level Parallelism (SMLP)

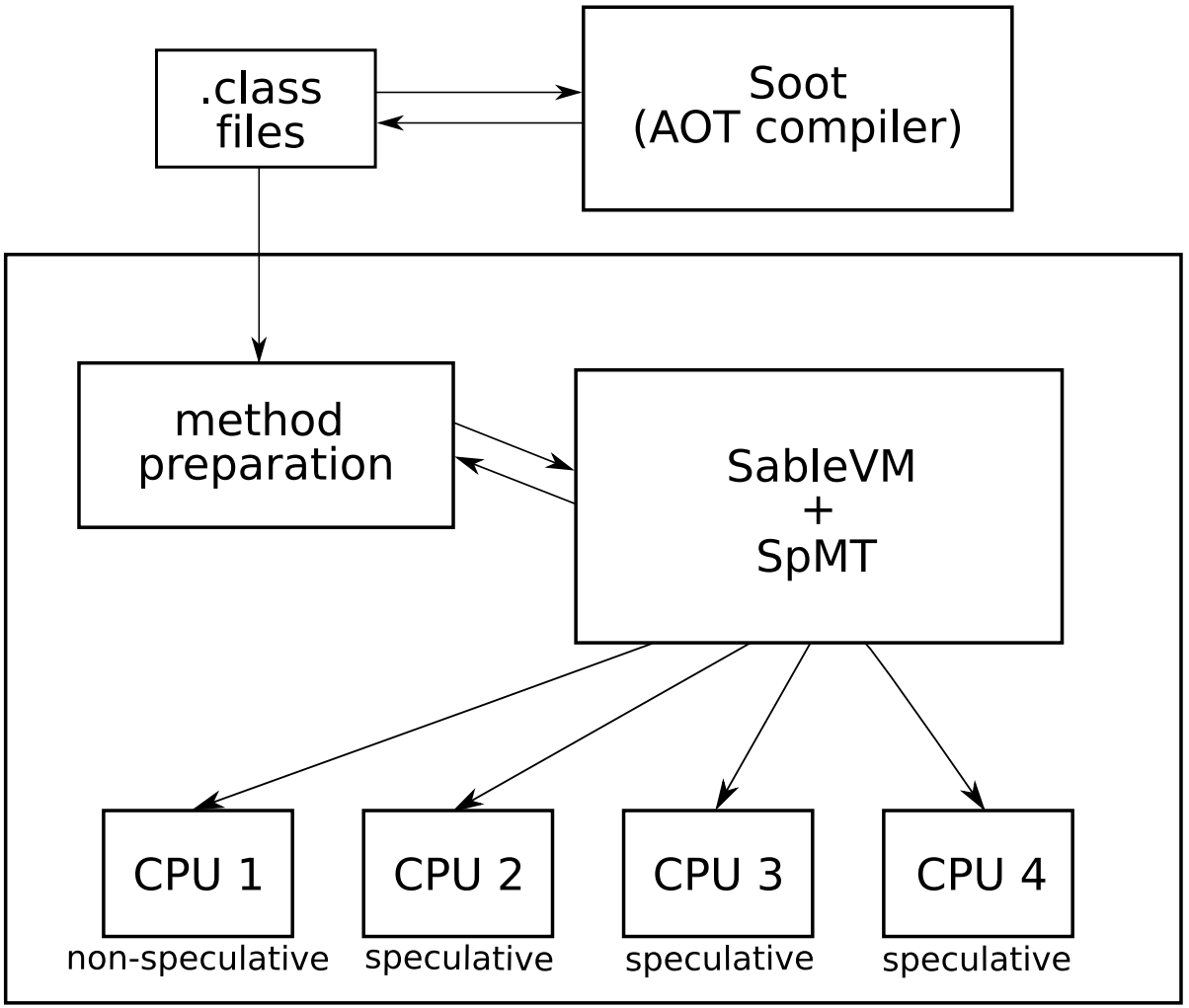


Contributions

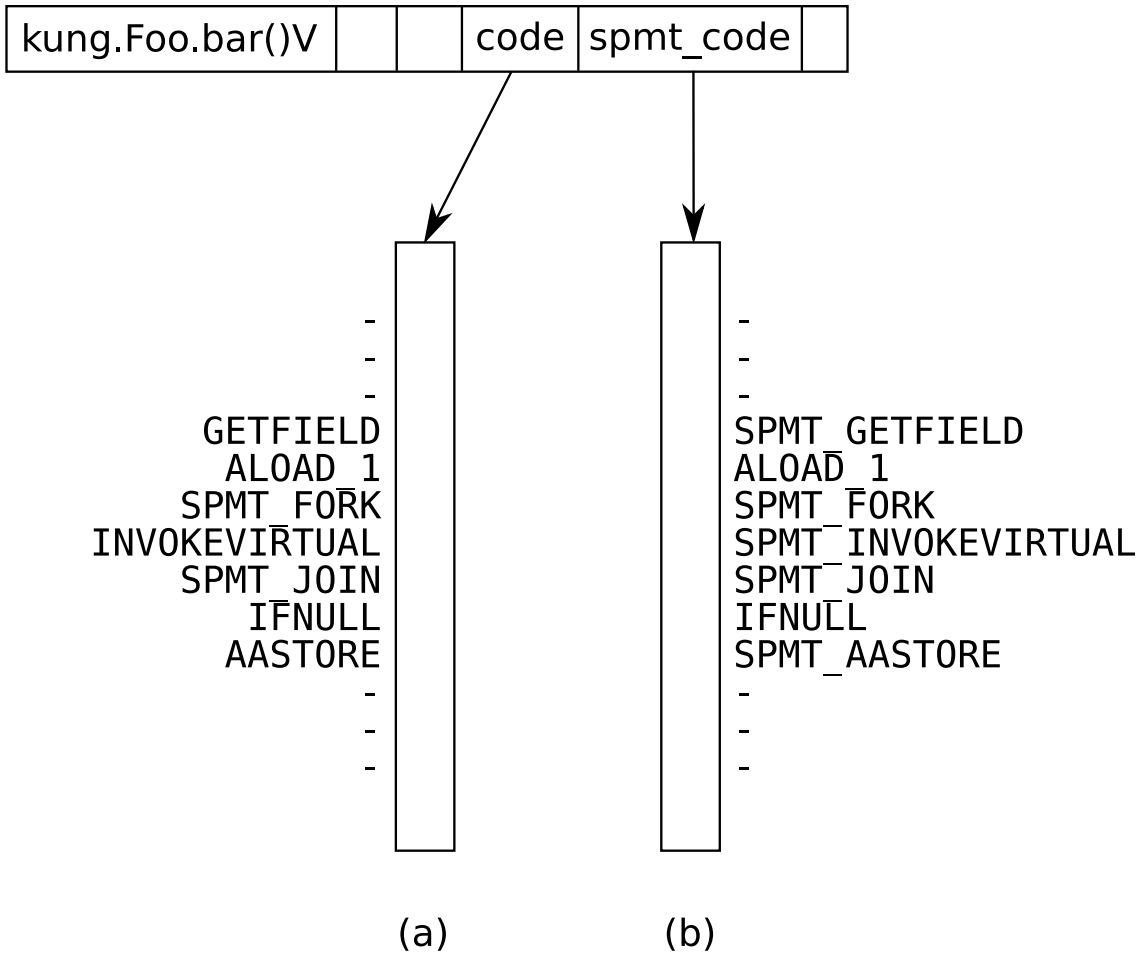
1. First complete implementation of (SMLP-based) SpMT in a (Java) virtual machine (SableVM)
2. Ability to run SPECjvm98 at size 100
3. Single-threaded simulation and true multithreaded execution modes
4. Experimental analysis of overhead costs and parallelism achieved
   - Unfortunately, no speedup :(

Outline

Execution Environment



Parallel Instruction Code Arrays



Modified Java Bytecode Instructions

instruction	reads global	writes global	locks object	unlocks object	allocates object	throws exception	enters native code	loads classes	forces stop

Modified Java Bytecode Instructions

instruction	reads global	writes global	locks object	unlocks object	allocates object	throws exception	enters native code	loads classes	forces stop
GETFIELD	always					sometimes		first time	sometimes
GETSTATIC	always							first time	first time
<X>ALOAD	always					sometimes			sometimes

Modified Java Bytecode Instructions

instruction	reads global	writes global	locks object	unlocks object	allocates object	throws exception	enters native code	loads classes	forces stop
GETFIELD	always					sometimes		first time	sometimes
GETSTATIC	always							first time	first time
<X>ALOAD	always					sometimes			sometimes
PUTFIELD		always				sometimes		first time	sometimes
PUTSTATIC		always						first time	first time
<X>ASTORE		always				sometimes			sometimes

Modified Java Bytecode Instructions

instruction	reads global	writes global	locks object	unlocks object	allocates object	throws exception	enters native code	loads classes	forces stop
GETFIELD	always					sometimes		first time	sometimes
GETSTATIC	always							first time	first time
<X>ALOAD	always					sometimes			sometimes
PUTFIELD		always				sometimes		first time	sometimes
PUTSTATIC		always						first time	first time
<X>ASTORE		always				sometimes			sometimes
(I|L)(DIV|REM)						sometimes			sometimes
ARRAYLENGTH						sometimes			sometimes
CHECKCAST						sometimes		first time	sometimes
ATHROW						always			always
INSTANCEOF								first time	sometimes

Modified Java Bytecode Instructions

instruction	reads global	writes global	locks object	unlocks object	allocates object	throws exception	enters native code	loads classes	forces stop
GETFIELD	always					sometimes		first time	sometimes
GETSTATIC	always							first time	first time
<X>ALOAD	always					sometimes			sometimes
PUTFIELD		always				sometimes		first time	sometimes
PUTSTATIC		always						first time	first time
<X>ASTORE		always				sometimes			sometimes
(I|L)(DIV|REM)						sometimes			sometimes
ARRAYLENGTH						sometimes			sometimes
CHECKCAST						sometimes		first time	sometimes
ATHROW						always			always
INSTANCEOF								first time	sometimes
RET									sometimes

Modified Java Bytecode Instructions

instruction	reads global	writes global	locks object	unlocks object	allocates object	throws exception	enters native code	loads classes	forces stop
GETFIELD	always					sometimes		first time	sometimes
GETSTATIC	always							first time	first time
<X>ALOAD	always					sometimes			sometimes
PUTFIELD		always				sometimes		first time	sometimes
PUTSTATIC		always						first time	first time
<X>ASTORE		always				sometimes			sometimes
(I|L)(DIV|REM)						sometimes			sometimes
ARRAYLENGTH						sometimes			sometimes
CHECKCAST						sometimes		first time	sometimes
ATHROW						always			always
INSTANCEOF								first time	sometimes
RET									sometimes
MONITORENTER	always	always	always			sometimes			always
MONITOREXIT	always	always		always		sometimes			always

Modified Java Bytecode Instructions

instruction	reads global	writes global	locks object	unlocks object	allocates object	throws exception	enters native code	loads classes	forces stop
GETFIELD	always					sometimes		first time	sometimes
GETSTATIC	always							first time	first time
<X>ALOAD	always					sometimes			sometimes
PUTFIELD		always				sometimes		first time	sometimes
PUTSTATIC		always						first time	first time
<X>ASTORE		always				sometimes			sometimes
(I|L)(DIV|REM)						sometimes			sometimes
ARRAYLENGTH						sometimes			sometimes
CHECKCAST						sometimes		first time	sometimes
ATHROW						always			always
INSTANCEOF								first time	sometimes
RET									sometimes
MONITORENTER	always	always	always			sometimes			always
MONITOREXIT	always	always		always		sometimes			always
INVOKE<X>	sometimes	sometimes	sometimes			sometimes	sometimes	first time	sometimes
<X>RETURN	sometimes	sometimes		sometimes		sometimes	sometimes	first time	sometimes

Modified Java Bytecode Instructions

instruction	reads global	writes global	locks object	unlocks object	allocates object	throws exception	enters native code	loads classes	forces stop
GETFIELD	always					sometimes		first time	sometimes
GETSTATIC	always							first time	first time
<X>ALOAD	always					sometimes			sometimes
PUTFIELD		always				sometimes		first time	sometimes
PUTSTATIC		always						first time	first time
<X>ASTORE		always				sometimes			sometimes
(I|L)(DIV|REM)						sometimes			sometimes
ARRAYLENGTH						sometimes			sometimes
CHECKCAST						sometimes		first time	sometimes
ATHROW						always			always
INSTANCEOF								first time	sometimes
RET									sometimes
MONITORENTER	always	always	always			sometimes			always
MONITOREXIT	always	always		always		sometimes			always
INVOKE<X>	sometimes	sometimes	sometimes			sometimes	sometimes	first time	sometimes
<X>RETURN	sometimes	sometimes		sometimes		sometimes	sometimes	first time	sometimes
NEW		always			always	sometimes		first time	sometimes
NEWARRAY		always			always	sometimes			sometimes
ANEWARRAY		always			always	sometimes		first time	sometimes
MULTIANEWARRAY		always			always	sometimes		first time	sometimes
LDC_STRING					first time				first time

Fork Decision Factors

Child threads are forked taking several factors into account.

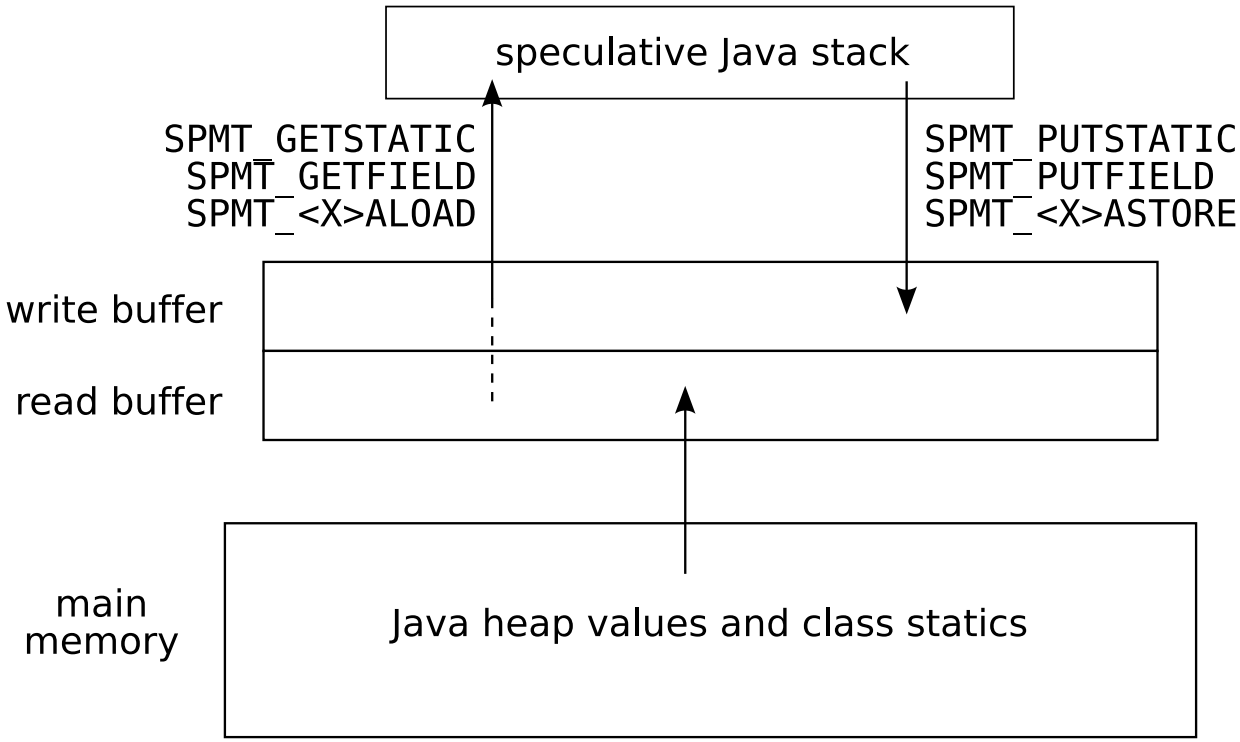
1. Static upper bound on method size
2. Dynamic min, max, and average method sizes
3. History of speculation successes and failures
4. History of sequence lengths
5. Number of zero length threads joined
6. Forced stop due to reaching another child ("elder sibling")

Forking Speculative Threads

The actual fork process consists of several steps:

1. Copy thread JNIEnv from parent to child
2. Copy parent stack to child
3. Initialize dependence buffer
4. Adjust child's operand stack height
5. Jump child `pc` over the `INVOKE<X>`
6. (optional) Predict return value for non-void methods

Dependence Buffering



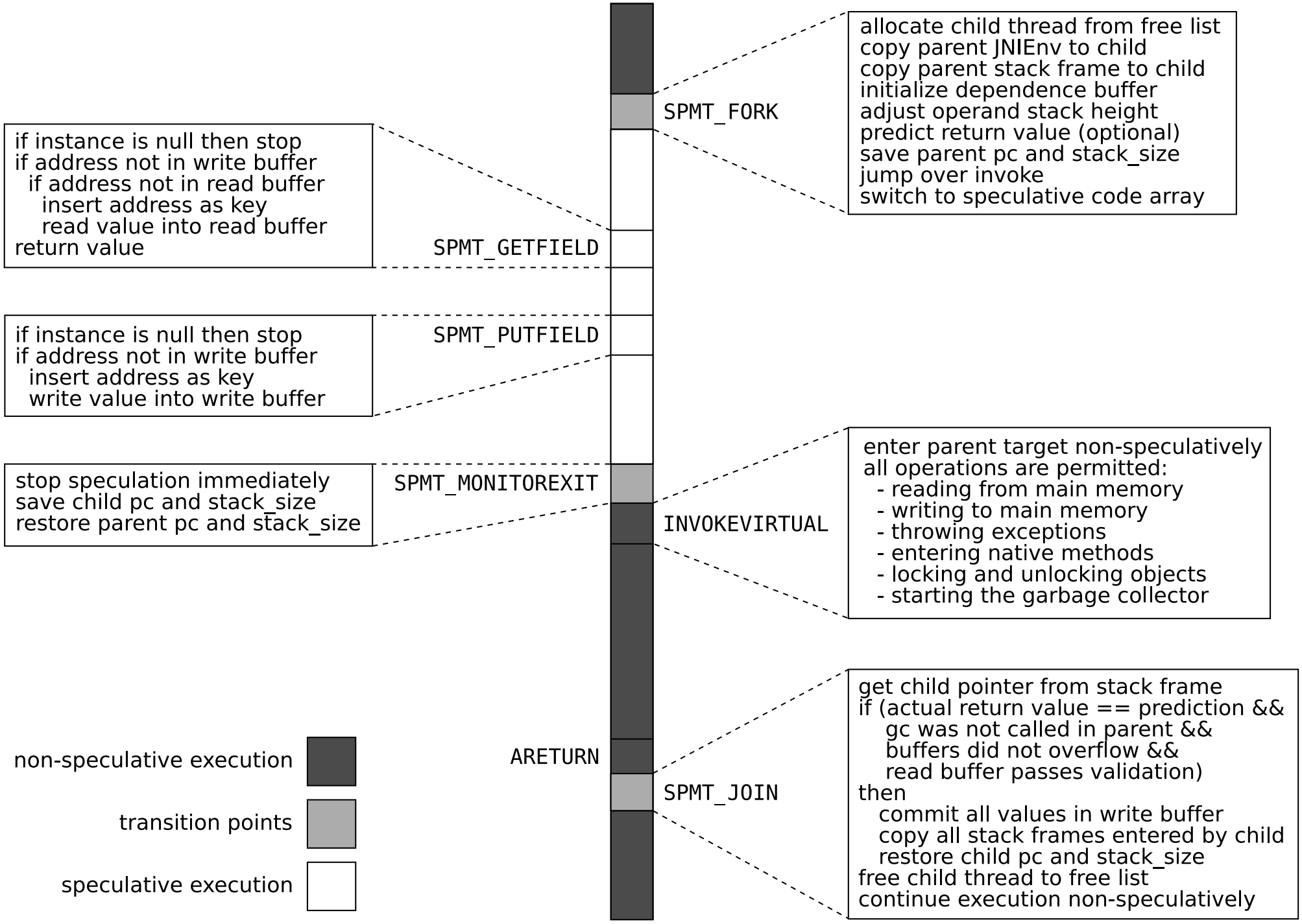
Joining Speculative Threads

- Every SpMT child eventually reaches one of four termination conditions:
  1. A pre-defined sequence length limit is reached
  2. The parent thread reaches `SPMT_JOIN` and signals the child
  3. The parent thread throws an uncaught exception, and signals the child
  4. Unsafe control flow is encountered
- Once stopped, we begin the validation process.

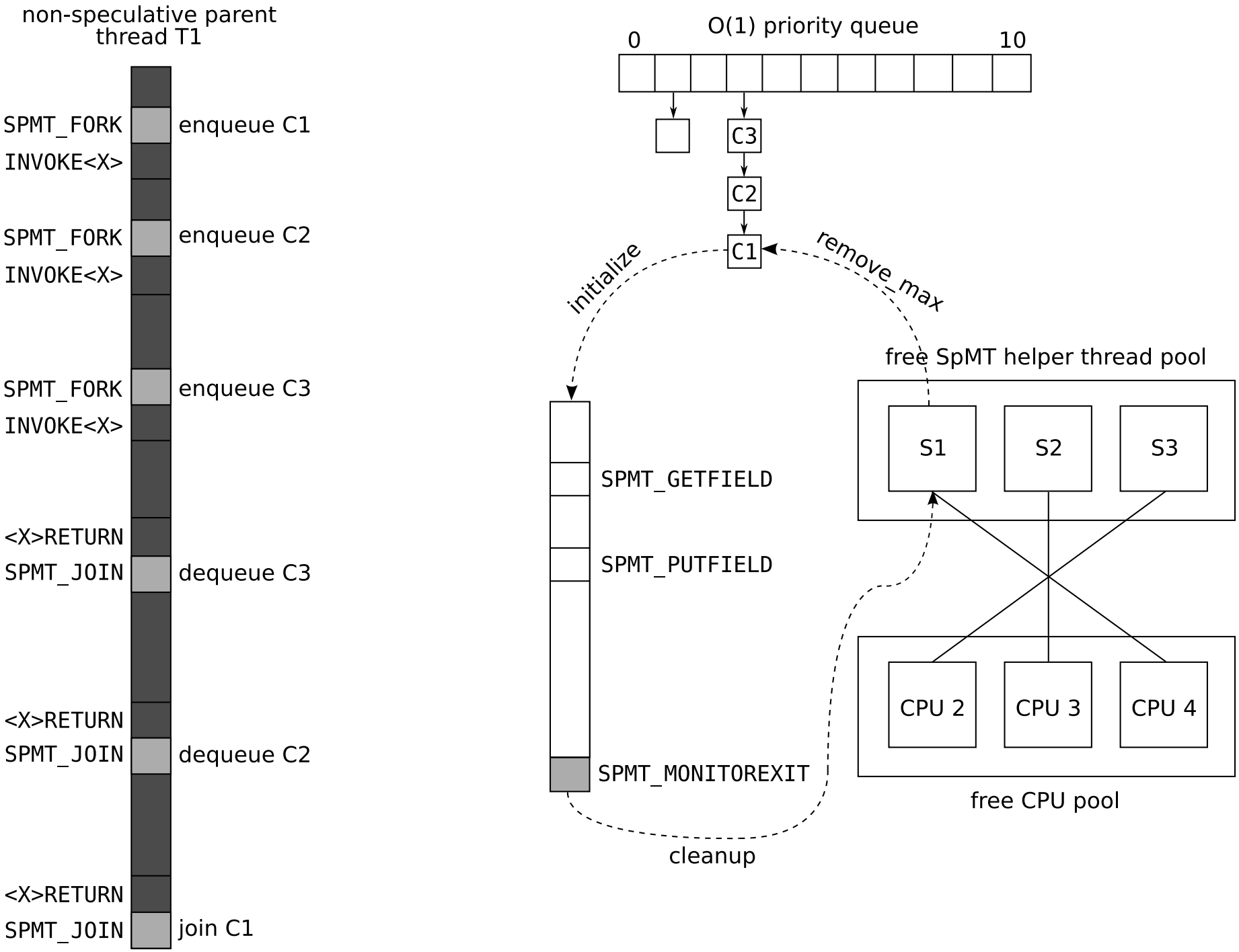
Joining Speculative Threads

- Validation consists of 4 steps
  1. Verify return value (if any)
  2. Check number of GC's in child
  3. Dependence buffers checked for corruption, overflow
  4. Values in *read* buffer compared with main memory
- If validation succeeds, then:
  - Values in *write* buffer are flushed to main memory
  - Child stack frames are copied to parent
  - Non-speculative execution resumes where the child left off
- Otherwise, the child is aborted.

Single-threaded Simulation Mode



Multithreaded Mode



Intricacies of the Java Language

- There are four Java-specific problems:
  1. Native methods
  2. Garbage collection
  3. Exceptions
  4. Synchronization

Native Methods

- Java allows for execution of non-Java, i.e. *native* code
- Native methods can be found in:
  - Class libraries
  - User code
  - VM-specific method implementations
- Native methods are needed for (amongst other things):
  - Thread management
  - Timing
  - All I/O operations
- Safe to fork children if parents encounter native methods
- Unsafe for children to enter native code

Garbage Collection

- SableVM uses a simple semi-space copying collector
- Child threads started before GC are invalidated after GC
  - Could be fixed by pinning objects, or by updating references in the dependence buffers.
- Child threads are invisible to the collector, and can continue execution during GC.
- We *are* able to allocate objects speculatively
  - Heap is protected by global mutex
  - Instead of `OutOfMemoryError`, speculation stops
  - Disadvantage is increased collector pressure from failed threads

Exceptions

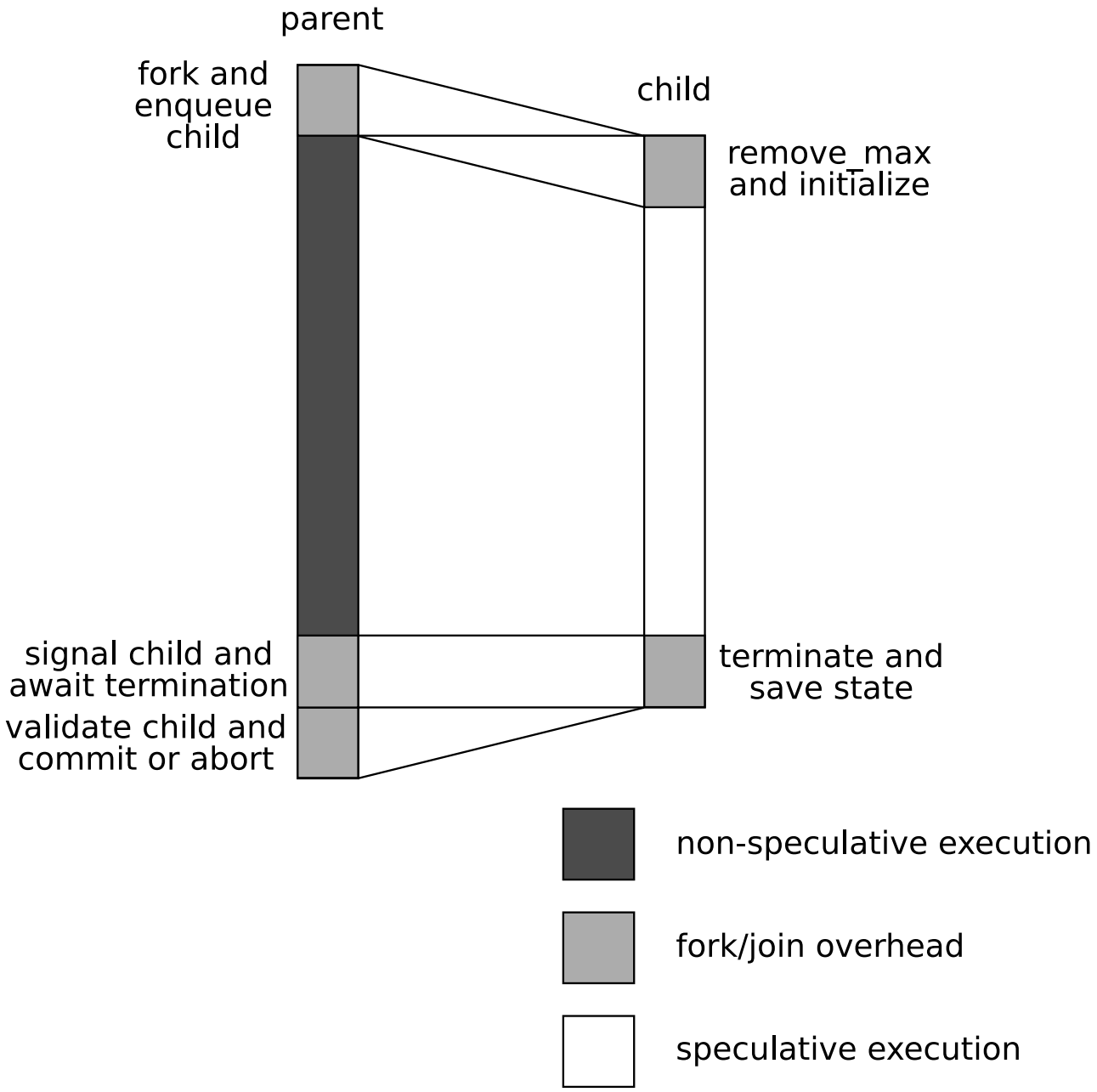
- Speculatively, exceptions force threads to stop immediately
  - Exceptions are rarely encountered
  - Writing a speculative exception handler is tricky
  - Speculative exceptions are likely to be incorrect
- Non-speculatively, exceptions can be thrown and caught in the parent
  - If uncaught, children are aborted one-by-one as stack frames are popped
- Since method calls frequently occur in exception handlers, we might expect to fork children inside them.
  - This is safe!

Synchronization

- Java allows for synchronization on a per-method and per-object basis
- Safe non-speculatively, unsafe speculatively
  - However, we *can* start child threads once *inside* a critical section; only entering and exiting is prohibited
- *Speculative Locking* allows for critical sections to be entered and exited speculatively
  - We'll look into this in the future

Outline

Speculation Overhead



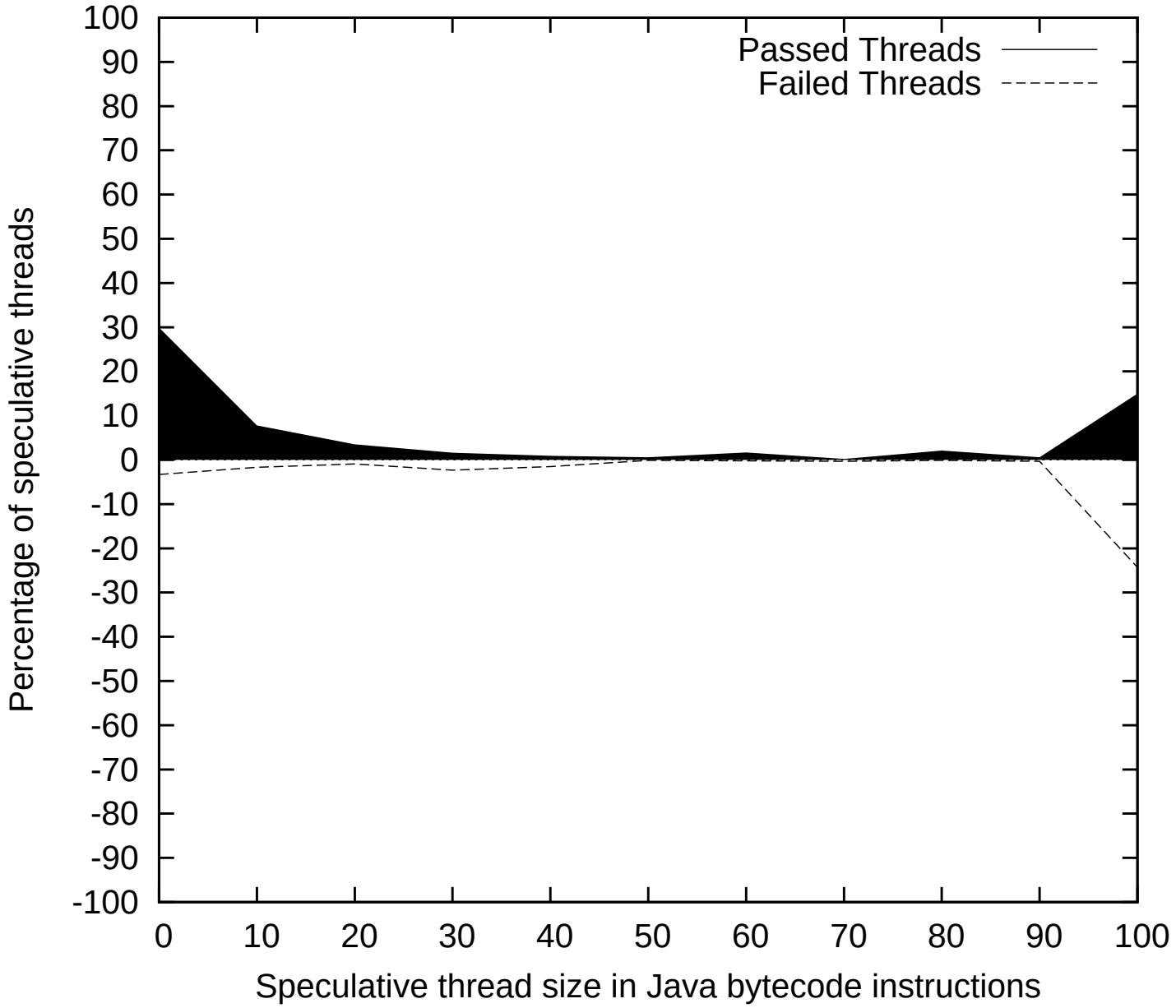
Non-speculative Thread Overhead Breakdown

execution	comp	db	jack	javac	jess	mpeg	mtrt	rt
bytecode	39%	24%	29%	30%	21%	59%	49%	58%
fork	6%	15%	13%	13%	11%	5%	3%	4%
enqueue	4%	10%	10%	9%	7%	3%	2%	2%
join	53%	59%	57%	56%	67%	34%	47%	36%
pred_update	7%	13%	12%	11%	12%	6%	7%	7%
dequeue	5%	5%	5%	4%	5%	2%	2%	2%
wait	15%	14%	11%	11%	19%	8%	26%	11%
pred_check	4%	4%	4%	5%	7%	3%	2%	3%
buffer_check	4%	6%	6%	5%	5%	3%	1%	2%
child_pass	5%	5%	7%	6%	6%	3%	2%	3%
child_fail	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%
cleanup	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%

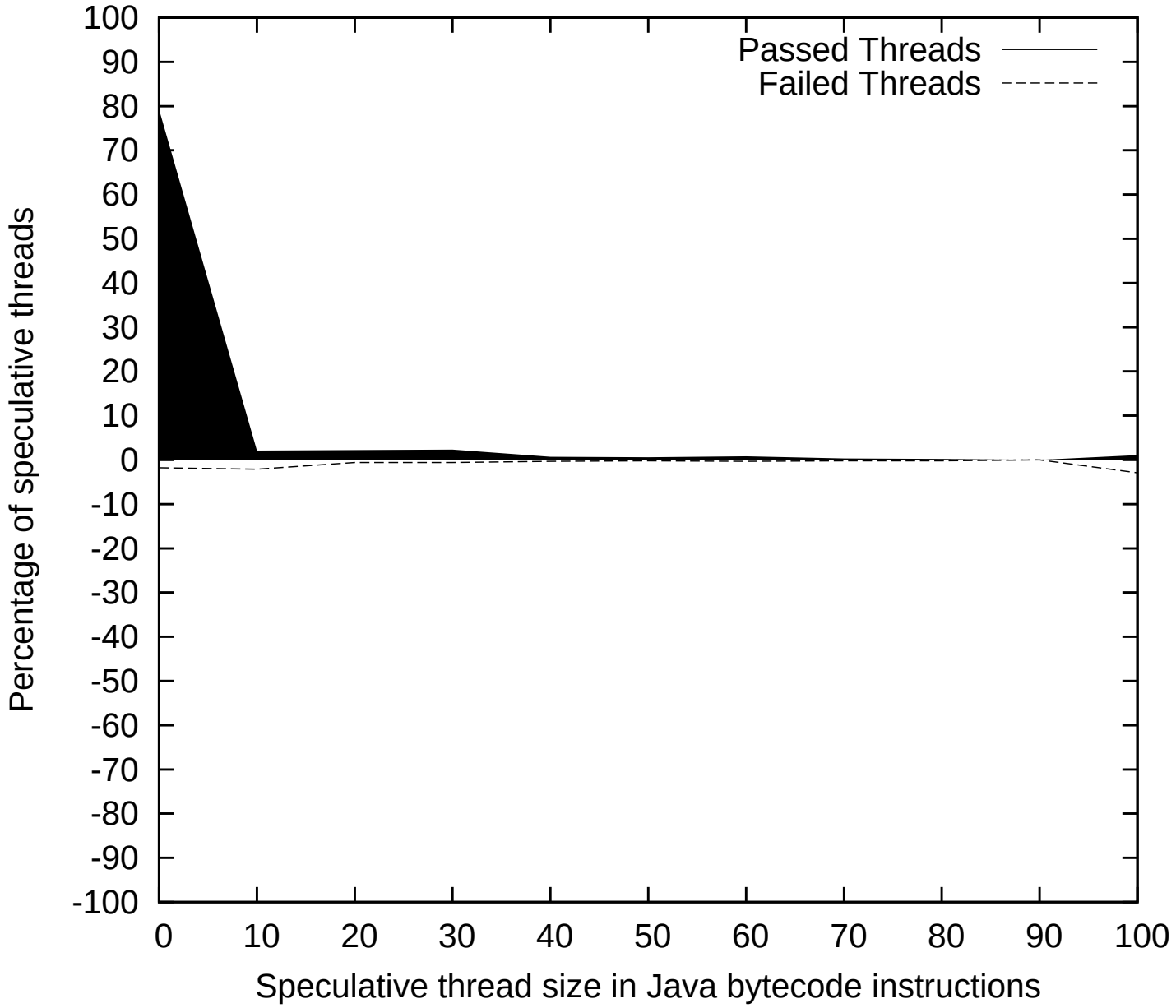
Speculative Thread Overhead Breakdown

execution	comp	db	jack	javac	jess	mpeg	mtrt	rt
child_wait	86%	82%	78%	78%	78%	55%	53%	71%
child_init	3%	4%	4%	4%	4%	2%	5%	4%
child_run	9%	12%	16%	16%	17%	41%	40%	24%
child_cleanup	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%
bytecode	58%	50%	65%	64%	57%	83%	51%	56%
fork	35%	40%	28%	29%	36%	13%	41%	36%
pred_query	33%	38%	25%	26%	33%	11%	38%	33%
join	2%	2%	2%	2%	2%	1%	2%	2%

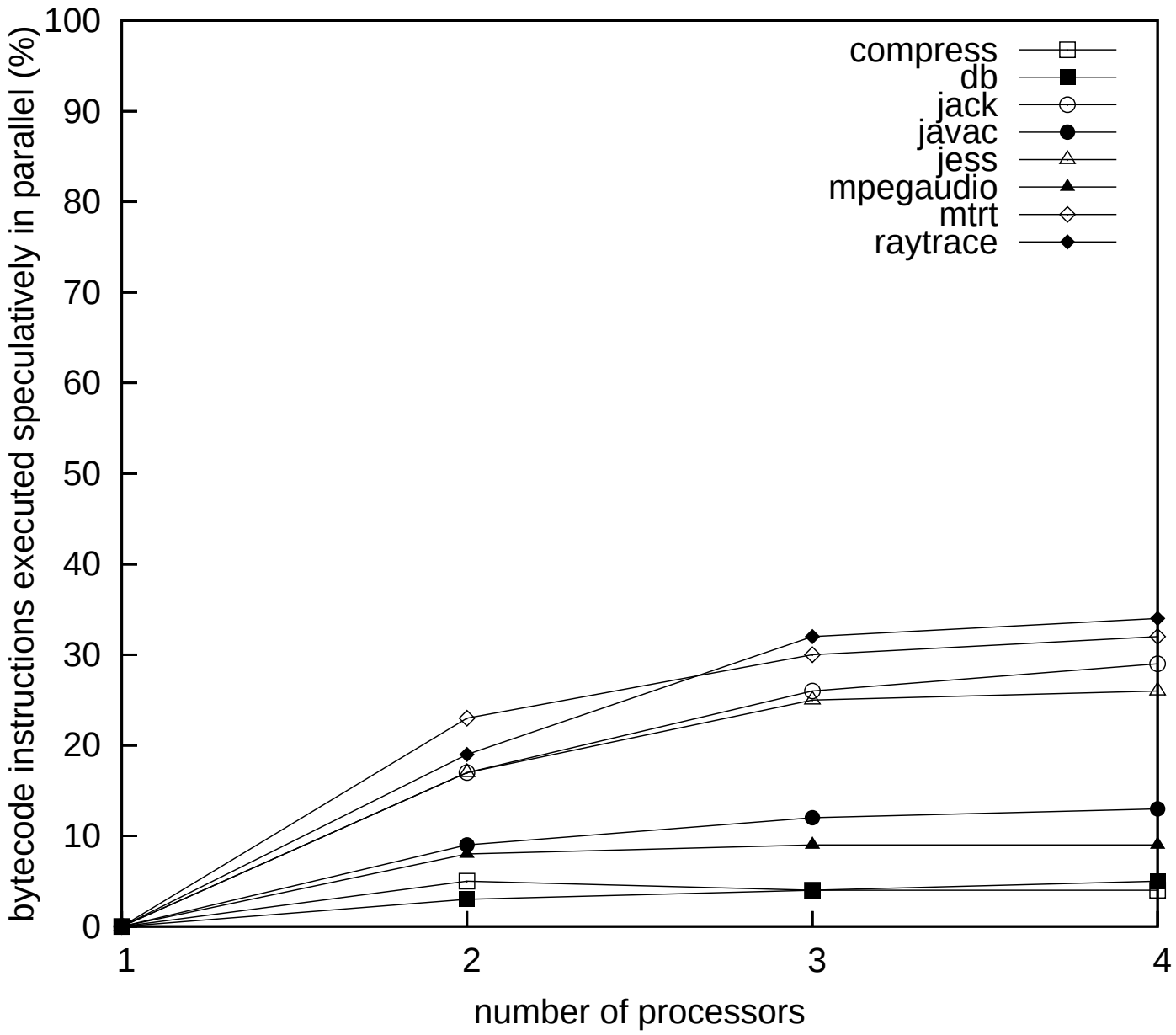
Speculative Thread Sizes (single-threaded simulation)



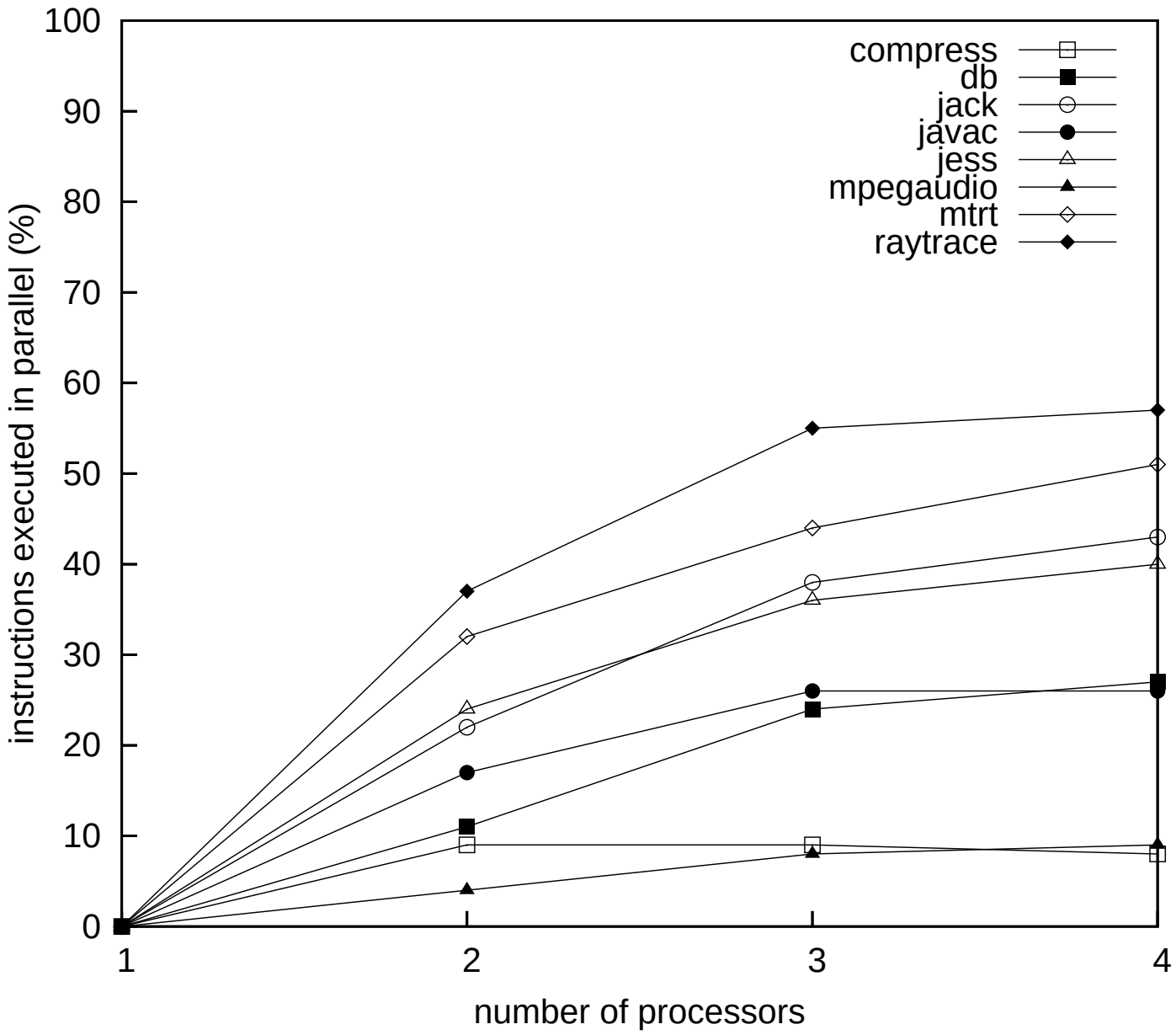
Speculative Thread Sizes (multithreaded mode)



Speculative Coverage (no RVP)



Speculative Coverage (with RVP)



Outline

Conclusions

- Automatic parellisation is a difficult goal
- We provide a complete design and working implementation
- The full Java language is handled
- Overhead costs show where to focus optimisation efforts
- We showed an increase in parallelism as:
  - Processors are added
  - Return value prediction is added
- SableSpMT is a good base for future research

Future Work in SableVM

- Eliminate overhead costs
- Implement speculative locking
- Look at processor scalability
- Allow for children to fork children
- Load value prediction
- Compiler analyses
  - Speculative dependences
  - Finding good fork points
- Clarify memory model issues
- Compare sequential algorithms running under SpMT against their hand-parallelised equivalents (start with JOlden).

Future Work in Testarossa and J9

- Implement this design in IBM's Testarossa JIT / J9 VM
- Initial target is PPC (Power4, Power5)
- Assembly versions of dependence buffer and value predictors
- Some mechanism to switch between non-speculative and speculative code (e.g. on-stack replacement)
- Goal is speedup and an (eventual) PLDI or OOPSLA paper