Software Speculative Multithreading for Java

Christopher J.F. Pickett and Clark Verbrugge
School of Computer Science, McGill University
{cpicke,clump}@sable.mcgill.ca

Allan Kielstra
IBM Toronto Lab
kielstra@ca.ibm.com

October 16th, 2006

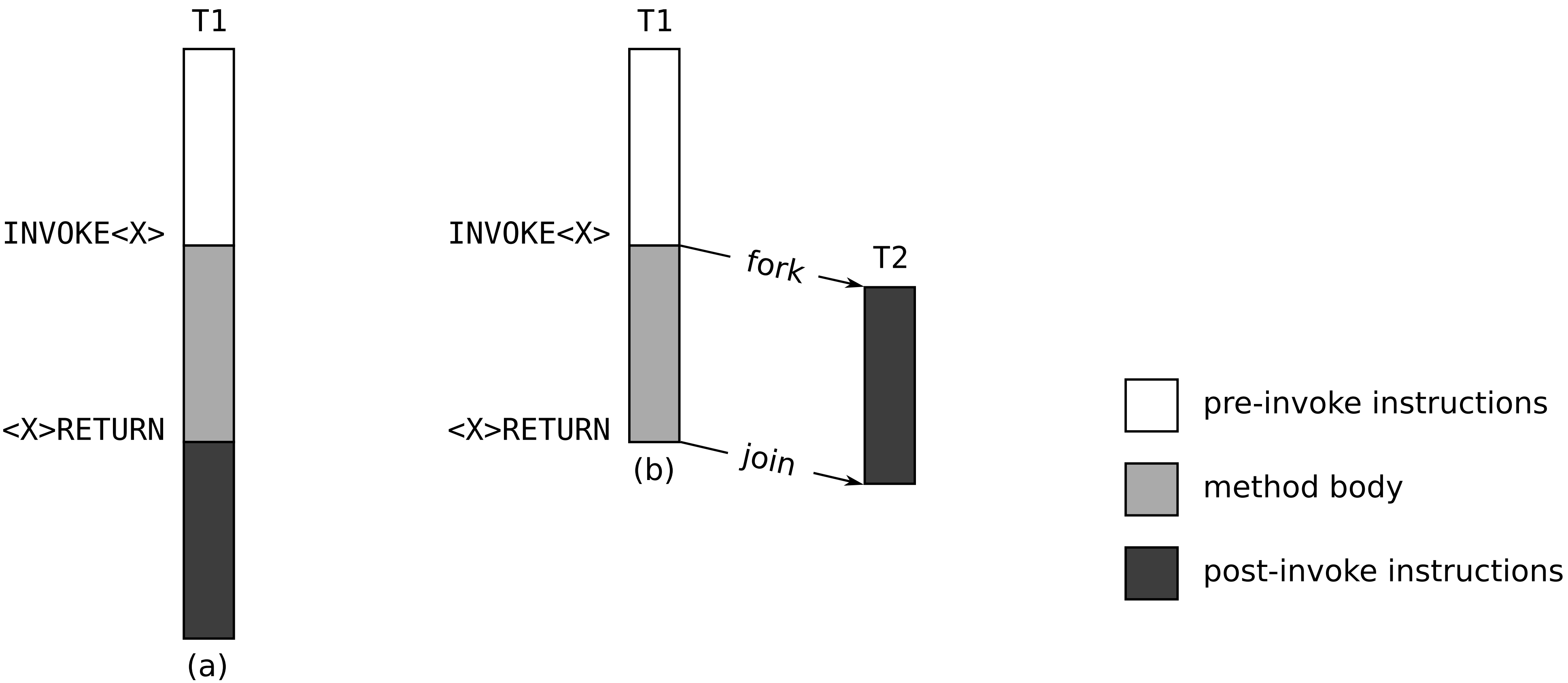
CDP'06

Outline

Motivation

- Thread level speculation (TLS) / speculative multithreading (SpMT) is a promising dynamic parallelisation technique.
- The SpMT variant *speculative method level parallelism* (SMLP) has good potential for both numeric and irregular Java programs.
- Previous work has shown 2–4x speedup on 4–8 CPU systems.
- On this basis, it seems reasonable to extend a Java virtual machine to support speculation at the bytecode level.

Speculative Method Level Parallelism (SMLP)



Problems in Speculative Multithreading

Two kinds of SpMT research, both face significant challenges.

- Problems with hardware-dependent SpMT approaches:
  1. SpMT hardware does not really exist.
  2. Hardware simulators are needed to run experiments.
  3. Accurate simulation is extremely slow.
  4. Simulated hardware implies simplifying abstractions.

- Problems with software-only SpMT approaches:
  1. Correct language semantics are not trivially ensured.
  2. Need software versions of hardware circuits, e.g. value predictors and dependence buffers.
  3. Thread overheads are a much greater barrier to speedup.
  4. Real hardware implies no simplifying abstractions.

Goals

- Our ultimate goal: speedup Java programs using a software-only SpMT-enabled JVM running on an off-the-shelf multiprocessor.
- Specific sub-goals:
  1. Determine correct semantics, implement them, characterise impact of language features and runtime support components: **LCPC'05**.
  2. Build a suitable analysis framework, characterise system performance and overhead: **PASTE'05**.
  3. Extract components into language-agnostic C library, **libspmt**.
  4. Speedup: **working on it...**

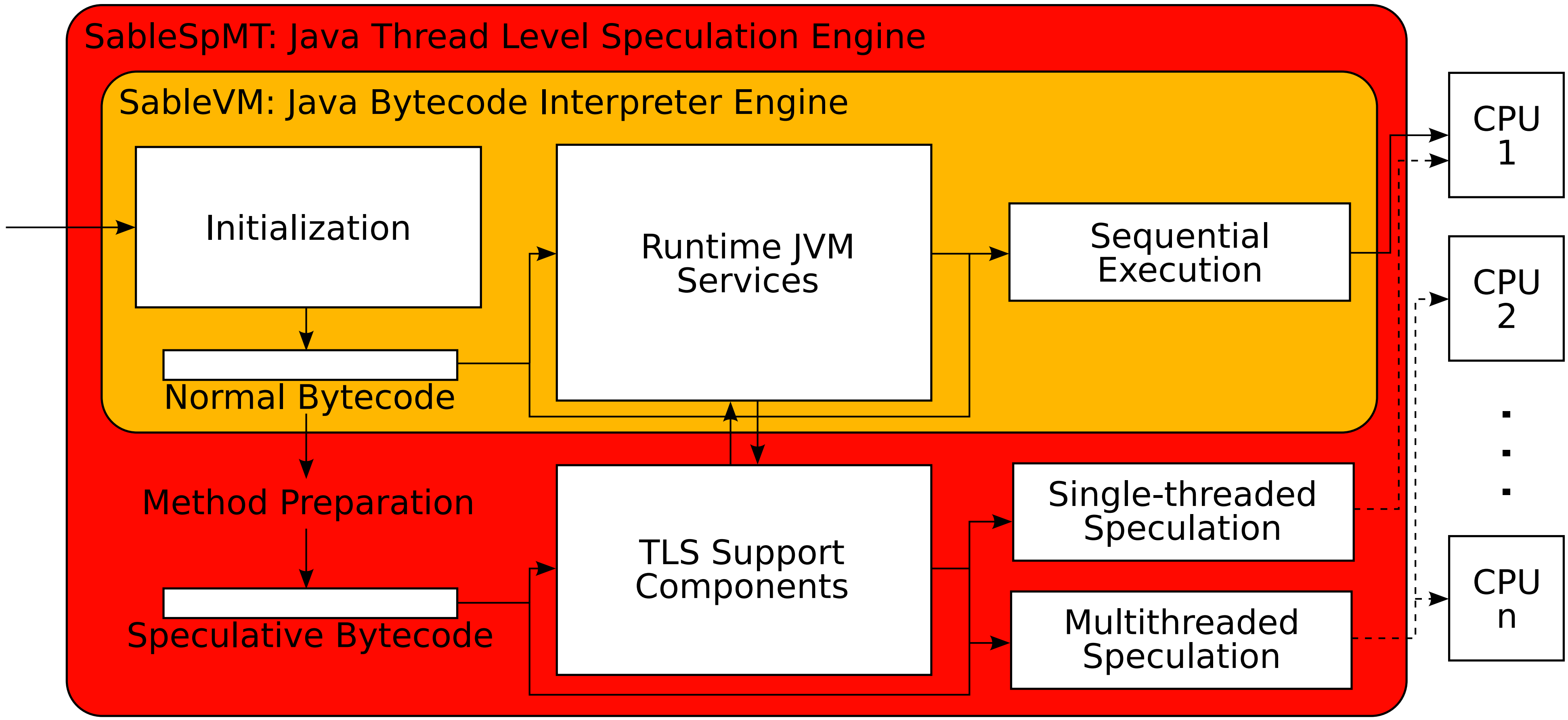
Contributions

Specific contributions:

1. SableSpMT: software SpMT implementation in SableVM
    - Runs on real multiprocessors
    - Suitable as an analysis framework
2. Semantics for high level Java language features.
3. Experimental analysis:
    - Thread overhead
    - Safety costs
    - Relative speedup

Outline

Java SpMT System Overview



Bytecode Modification

- Introduce new fork and join bytecodes.
- 25% of Java's instruction set needs non-trivial changes. Instructions might:
  - Load classes dynamically
  - Read from and write to main memory
  - Lock and unlock objects
  - Enter and exit methods
  - Allocate objects
  - Throw exceptions
  - Require a memory barrier
- Speculation terminates on unsafe operations.

Priority Queueing

- Children enqueued at fork points on $O(1)$ bounded height priority queue.
- Priority $= \min(l \times r/1000, 10)$
  - l: historical thread length at callsite in bytecodes
  - r: speculation success rate
- Queue supports `enqueue`, `dequeue`, and `delete`.
- Helper OS threads run on separate processors, and compete for TATAS spinlock on the queue.
- Helper threads only run if processors are free.

Return Value Prediction

- Return values are consumed by method continuations early on.
- Must abort children with unsafe return values on the stack.
- Accurate return value prediction benefits Java SMLP.

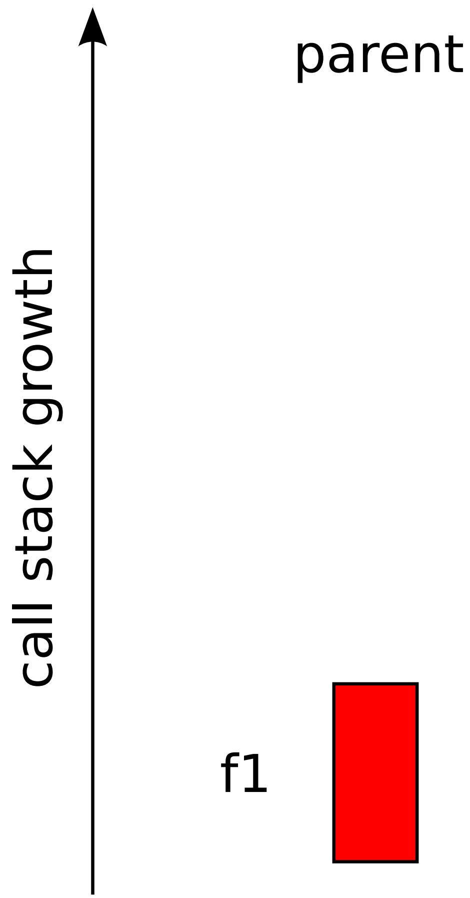
- Provide context, memoization, and hybrid predictors.
- Exploit static analyses to reduce memory and increase accuracy.
- Previously explored RVP in depth; now a system component.

Dependence Buffering

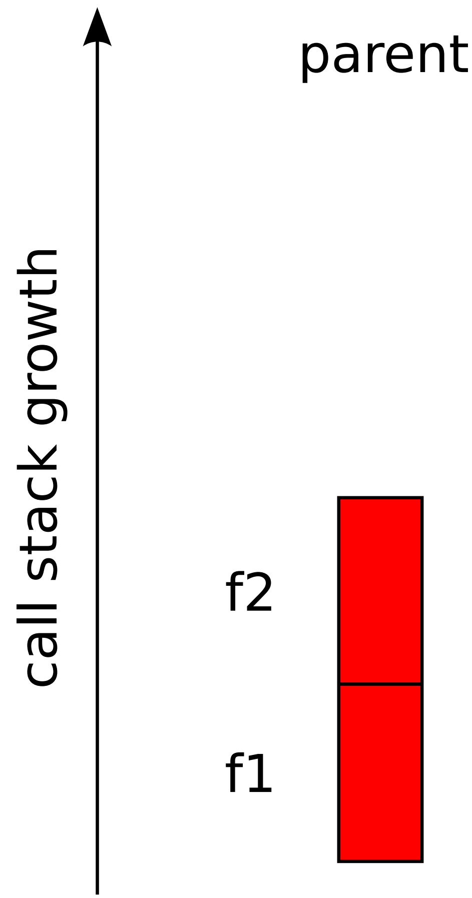
- SpMT designs usually buffer speculative memory accesses in a cache-like structure.
- Here we buffer heap/static reads/writes in a software dependence buffer, using open addressing hashtables.
- Upon joining a thread, validate all reads and then commit writes.

- Instructions touching only the stack are buffered differently.

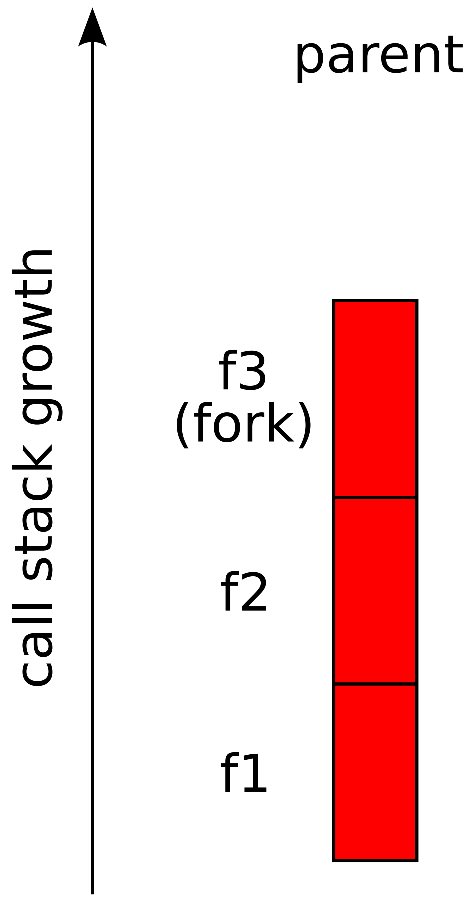
Stack Buffering



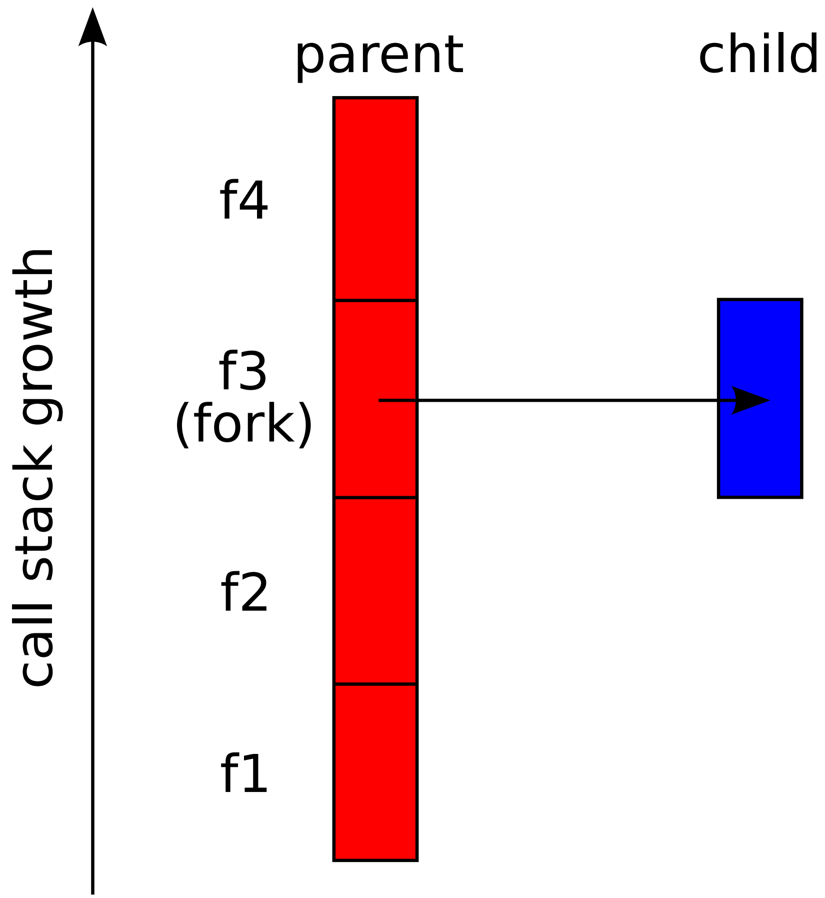
Stack Buffering



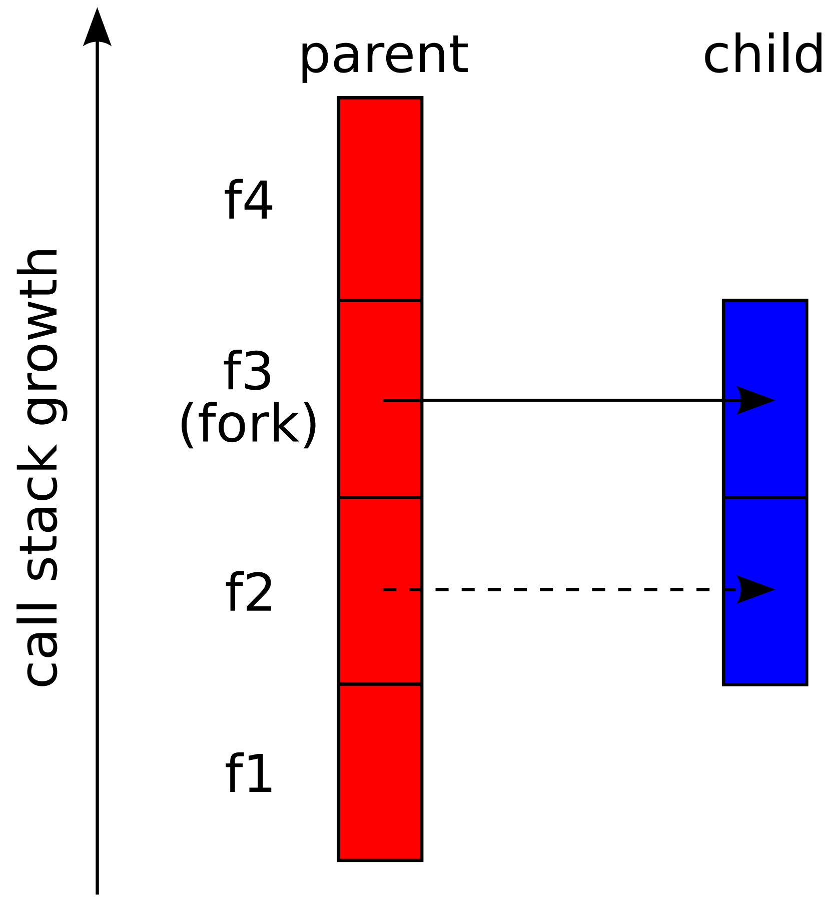
Stack Buffering



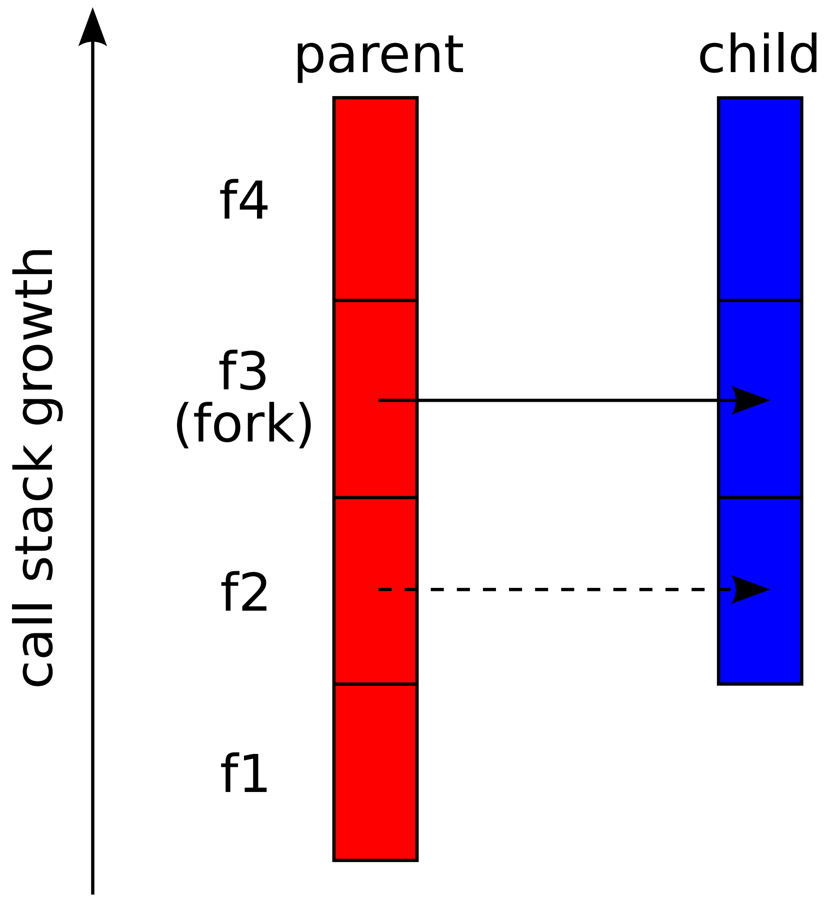
Stack Buffering



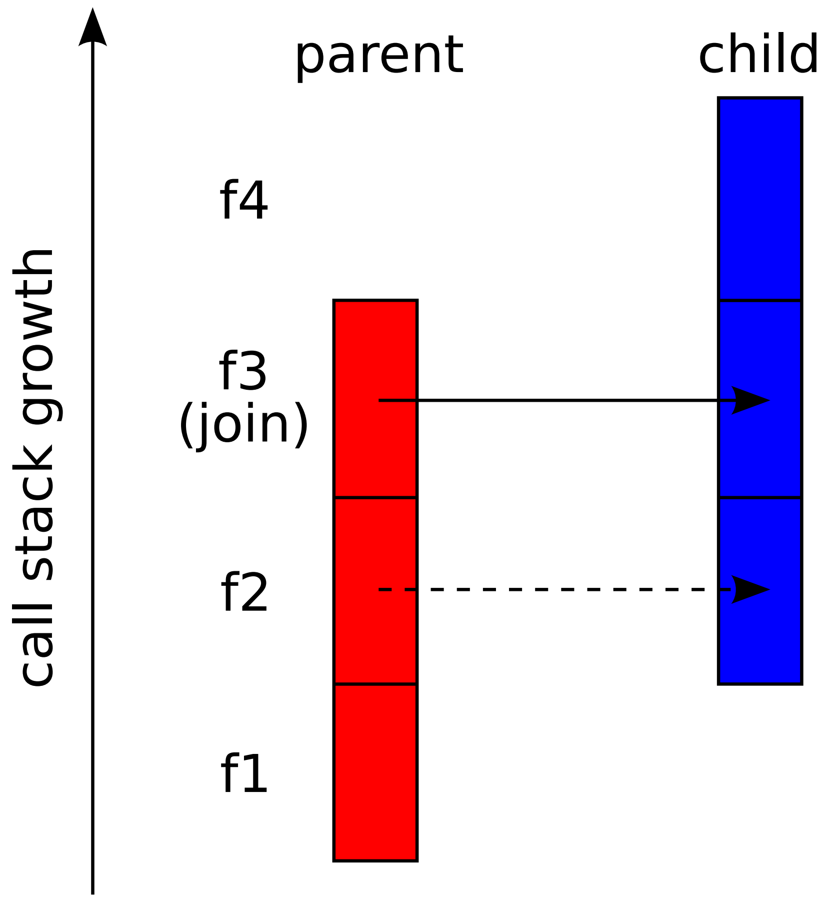
Stack Buffering



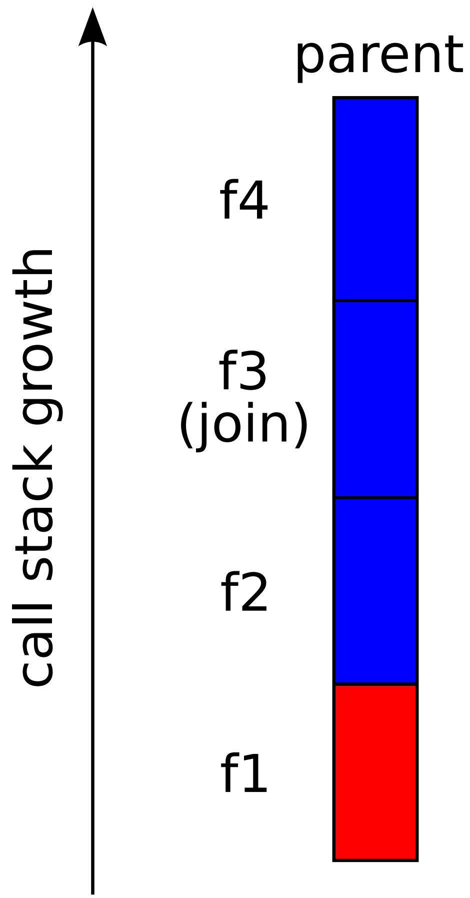
Stack Buffering



Stack Buffering



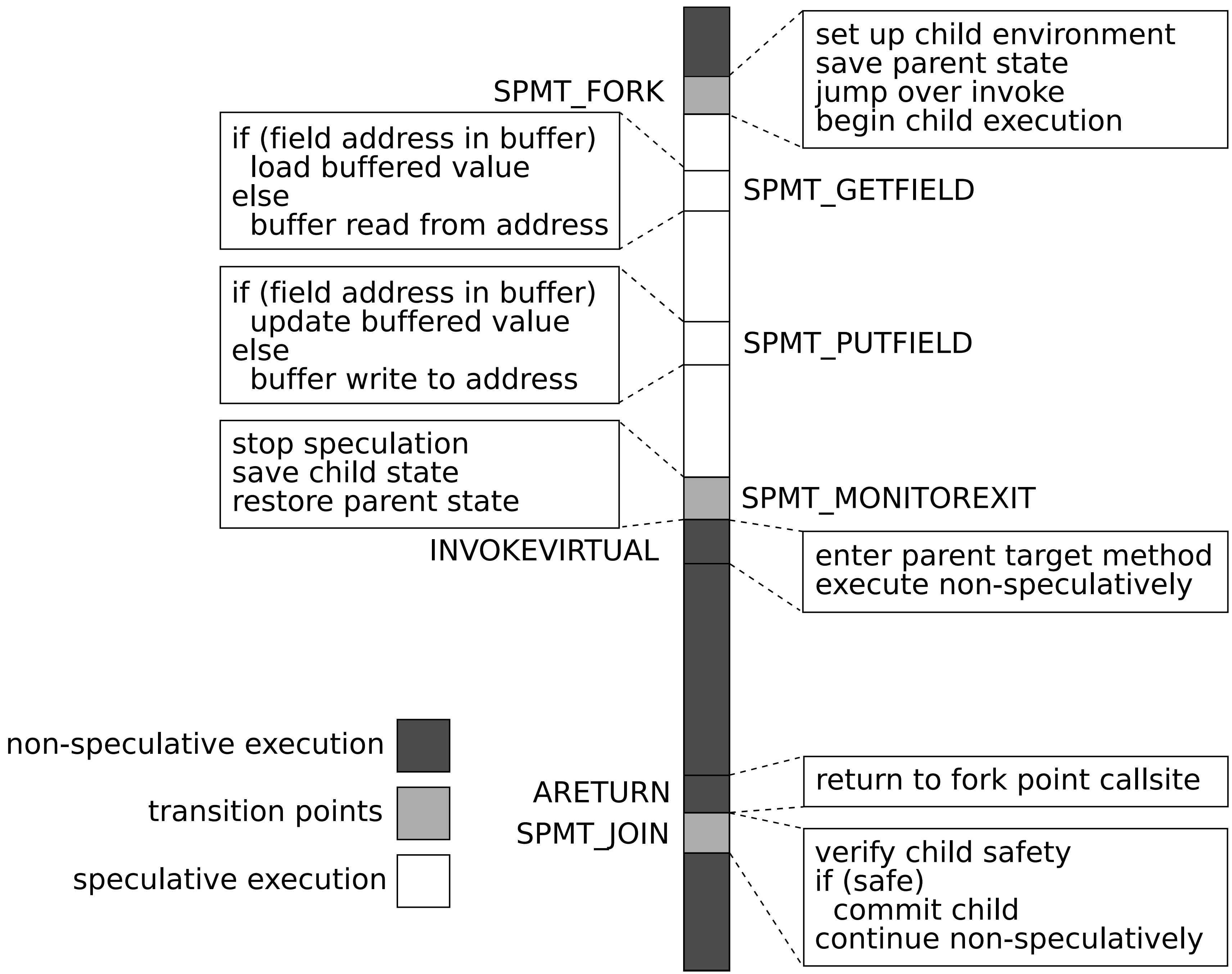
Stack Buffering



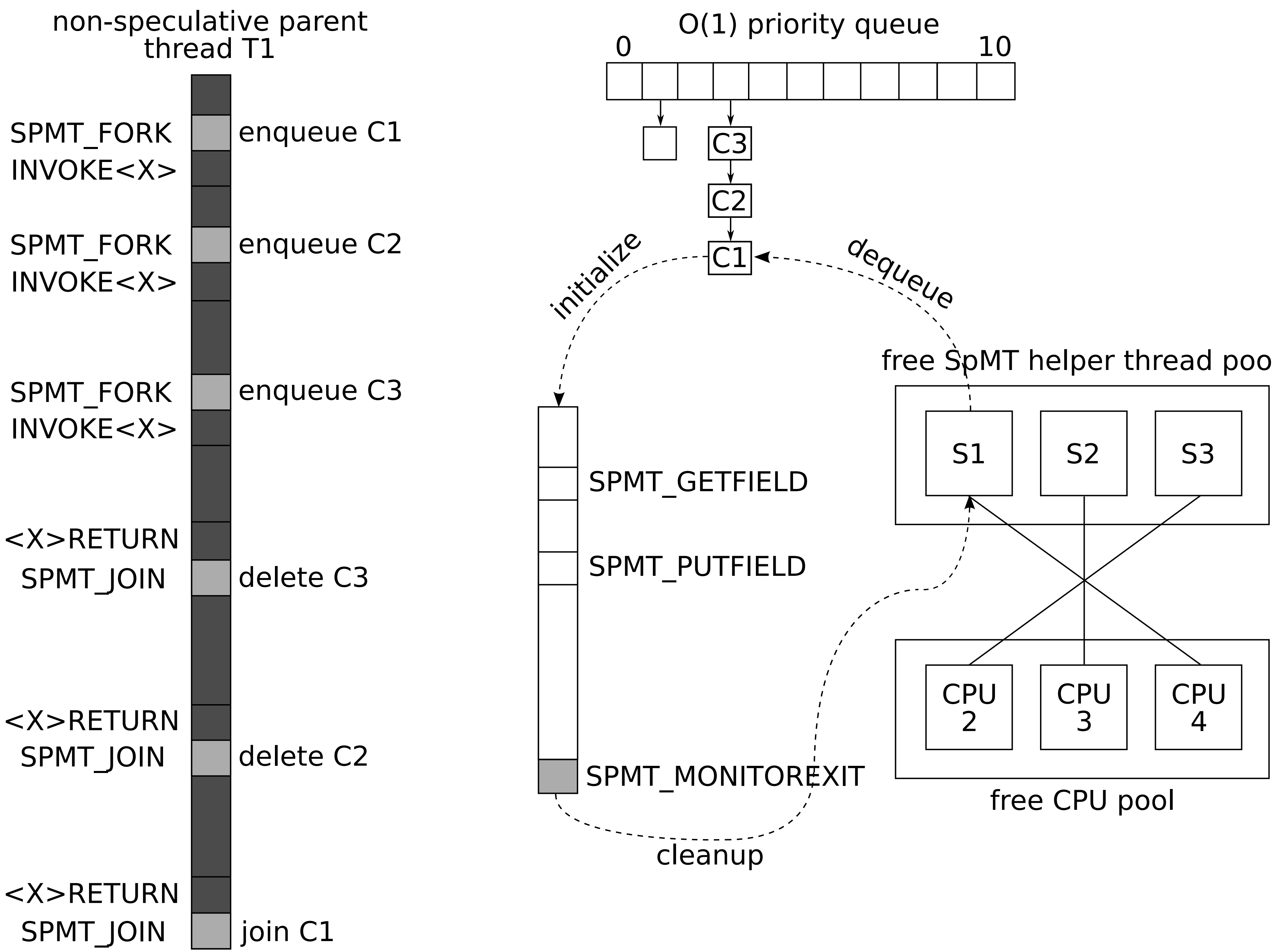
Object Allocation

- Allocate objects and arrays speculatively:
  - Compete for global or thread local heap mutexes.
  - Instead of triggering GC or an `OutOfMemoryError`, just stop.
  - No buffering needed for speculative objects.
  - Increased collector pressure, but negligible overall impact.
  - Cannot allocate objects with non-trivial finalizers.

Single-threaded Simulation Mode



Multithreaded Mode



Outline

Bytecode Verification

Speculative execution cannot depend on verification guarantees:

- Object references on the stack might be junk pointers
  - Check reference is within heap bounds.
  - Check object header is valid.
- Virtual method calls might enter the wrong target
  - Check target type is assignable to receiver type.
  - Check target stack effect matches signature.
- Subroutines might be split by speculation
  - Non-speculative `JSR`, speculative `RET`
  - Speculative `JSR`, non-speculative `RET`
  - `RET` needs to jump back to the right place.

Garbage Collection

- Simple semi-space stop-the-world copying collector
- Children are invisible to the collector, and can continue execution during GC:
  - Ignore stop-the-world requests
  - Never trigger collection
- Child threads started before GC are invalidated after GC.
  - Might consider pinning objects, or updating buffered references.

Native Methods

- Java allows for execution of non-Java, i.e. *native* code.
- Native methods can be found in:
  - Class libraries
  - Application code
  - VM-specific method implementations
- Native methods are needed for (amongst other things):
  - Thread management
  - Timing
  - All I/O operations
- Speculatively, unsafe to enter native code.
- Non-speculatively, always safe to enter native code, even for parents with speculative children.

Exceptions

- Speculatively, exceptions simply force termination because:
  1. Writing a speculative exception handler is tricky.
  2. Exceptions are rarely encountered.
  3. Speculative exceptions are likely to be incorrect.

- Non-speculatively, exceptions can be thrown and caught.
  - If uncaught, children are aborted one-by-one as stack frames are popped in the VM exception handler loop.
- Can safely fork child threads in exception handler bytecode.

Synchronization

- Java allows for per-method and per-object synchronization.
- Safe non-speculatively, unsafe speculatively
    - However, we *can* fork child threads once *inside* a critical section; only entering and exiting is prohibited.
    - In principle, this encourages coarse-grained locking.
- Rich Halpert at McGill is working on support for transactions and speculative locking.

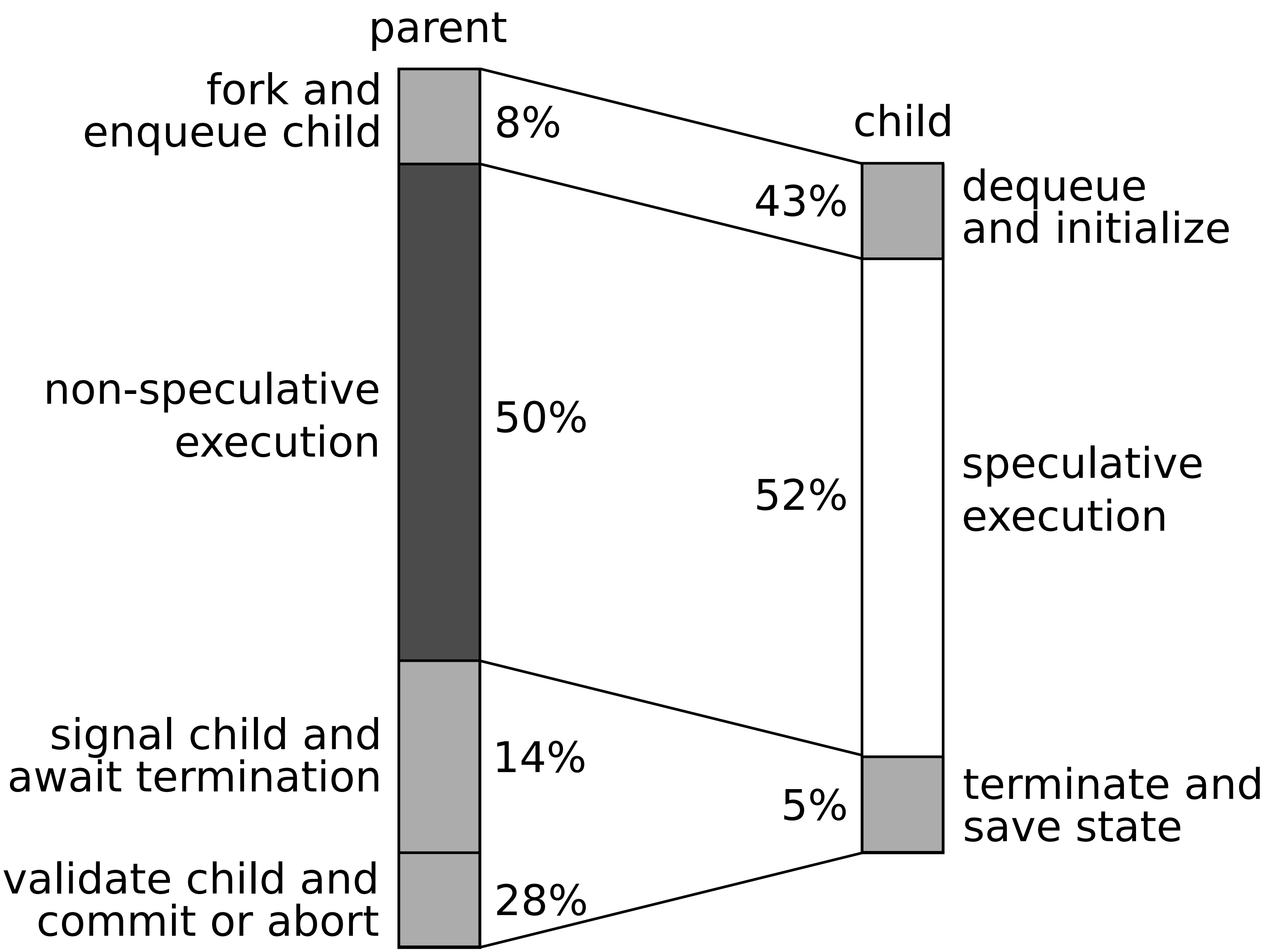
Java Memory Model

- The new Java Memory Model (JSR-133) gives specific rules about reordering, and memory barrier requirements.
- Speculation might reorder reads and writes during thread validation and committal.
- Unsafe operations we considered:
  - Locking and unlocking
  - Volatile loads and stores
  - Final stores in constructors
  - Speculation past a constructor with a non-trivial finalizer
  - `java.lang.Thread.*`
- Conservatively, terminate speculation on these conditions.
- In the future, could record barriers in dependence buffers.

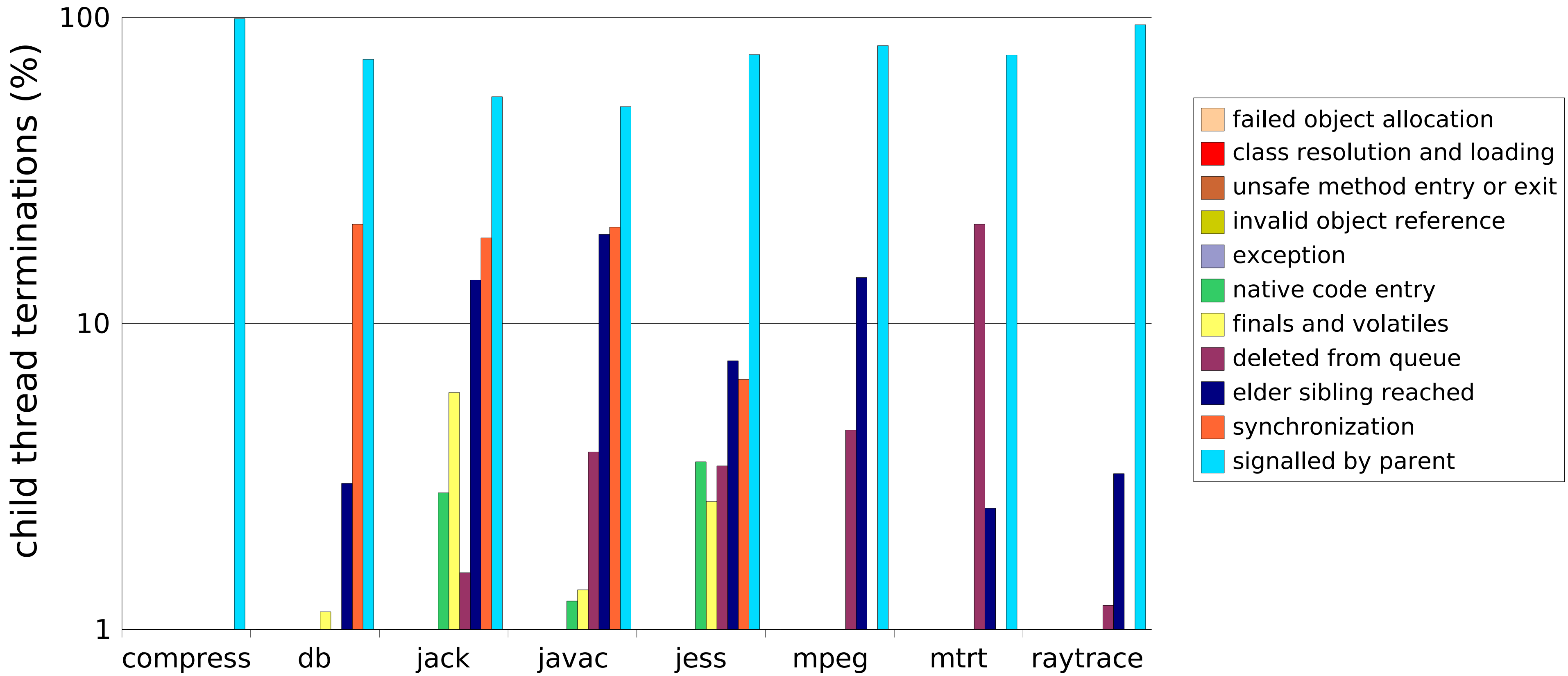
Outline

1. Introduction
2. Java SpMT Design
3. Java Language Considerations
4. Experimental Analysis
5. Conclusions and Future Work

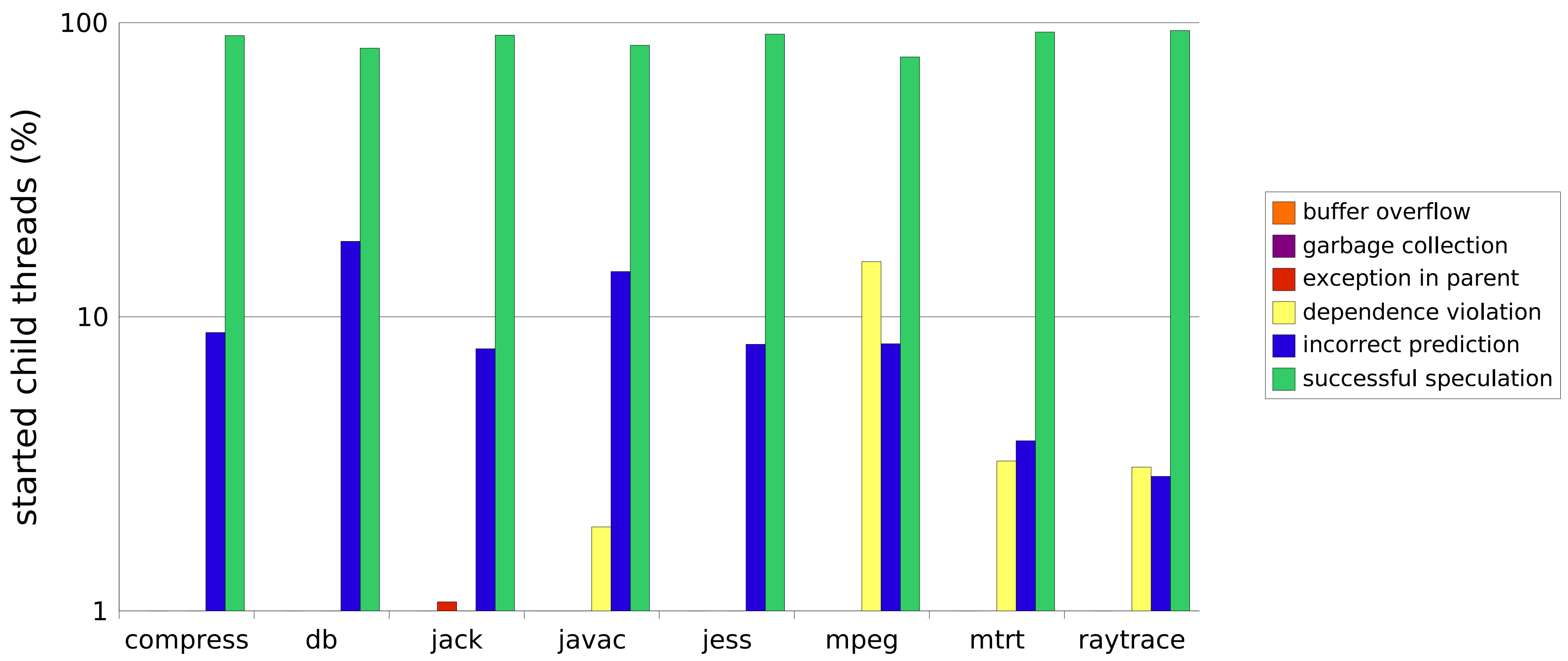
Speculation Overhead



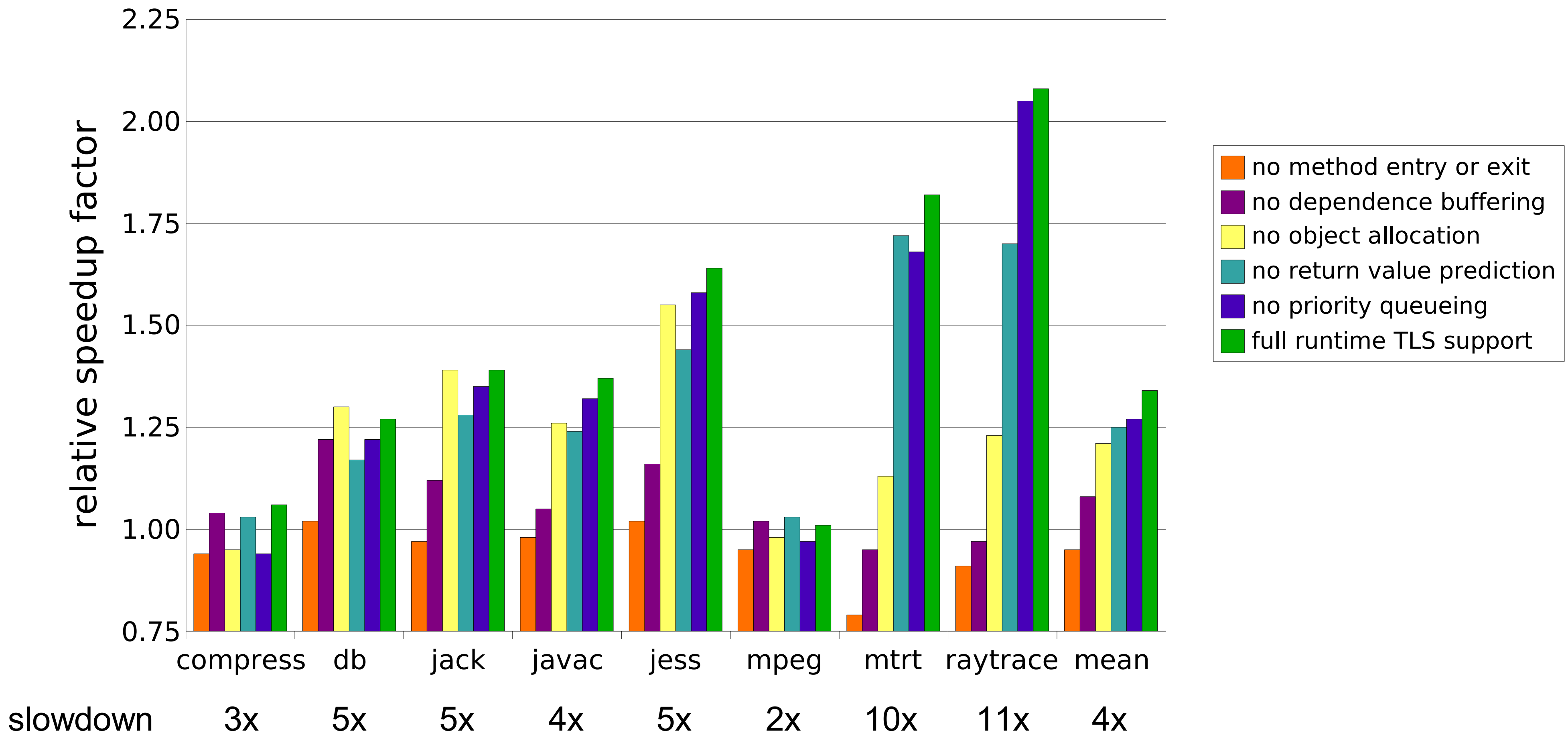
Child Termination Reasons



Child Success and Failure



Impact of Support Components on Speedup



Outline

Conclusions

- Complete design for Java SMLP
  - Handles SPECjvm98 at S100 without simplifications.
- Specific language and VM contexts affect design:
  - Non-trivial safety considerations for Java
  - Most have minimal impact on performance
    - Synchronization and JMM constraints are important
- Results show importance of runtime support components, and where to begin optimization.

Future Work

- Performance optimisations:
  - Overhead reduction
  - Forking heuristics
  - Nested speculation
  - Speculative locking
  - Load value prediction
- libspmt:
  - Migrate SableSpMT features into independent library
  - Predictors, buffer, and priority queue already implemented
  - Link to other VM's: IBM's J9/TR, OCaml
- Static analyses
  - Purity / escape analysis (Haiying Xu)
  - Lock allocation (Rich Halpert)