Software Thread Level Speculation for the Java Language and Virtual Machine Environment

Christopher J.F. Pickett and Clark Verbrugge
School of Computer Science, McGill University
Montréal, Québec, Canada H3A 2A7
{cpicke,clump}@sable.mcgill.ca

October 21st, 2005

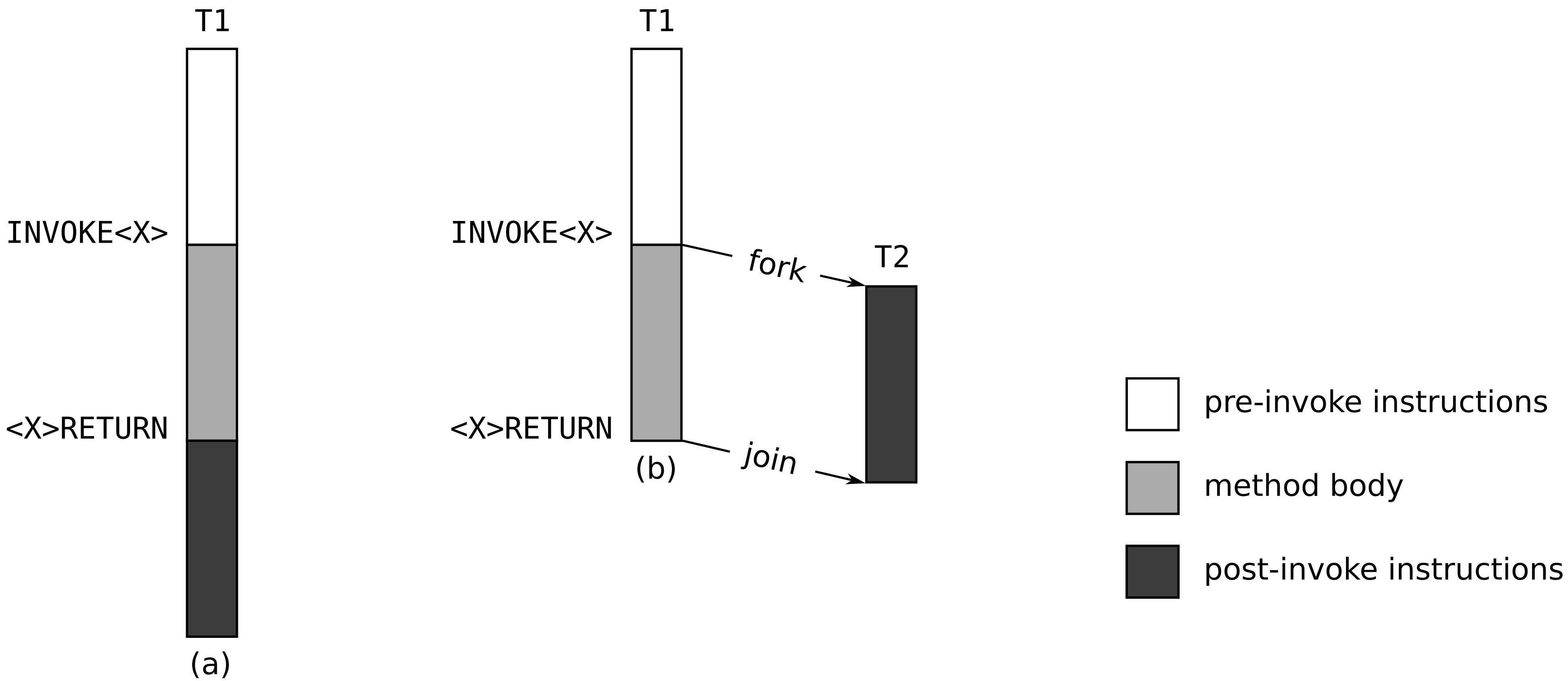
LCPC 2005

Outline

Motivation

- Thread level speculation (TLS) / speculative multithreading (SpMT) is a promising dynamic parallelisation technique.
- The TLS 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 Thread Level Speculation

Two kinds of TLS research, both face significant challenges.

- Problems with hardware-dependent TLS approaches:
  1. TLS hardware does not exist.
  2. Hardware simulators are needed to run experiments.
  3. Accurate simulation is extremely slow.
  4. All hardware studies make simplifying abstractions.

- Problems with software-only TLS approaches:
  1. Thread overheads are a much greater barrier to speedup.
  2. Correct language semantics are not trivially ensured.
  3. Generic software studies cannot make simplifying abstractions.
  4. Need software versions of hardware circuits, e.g. value predictors and dependence buffers.

Goals

- Our ultimate goal is to achieve speedup of Java programs using a software-only JVM interpreter that supports TLS running on commodity, off-the-shelf multiprocessor hardware.
- Specific sub-goals:
    1. Determine correct semantics, implement them, characterise impact of language features and runtime support components: **this paper.**
    2. Build a suitable analysis framework, characterise system performance and overhead: *SableSpMT: A Software Framework for Analysing Speculative Multithreading in Java*, **PASTE'05**.
    3. Optimise SableSpMT and achieve speedup: **future work**.

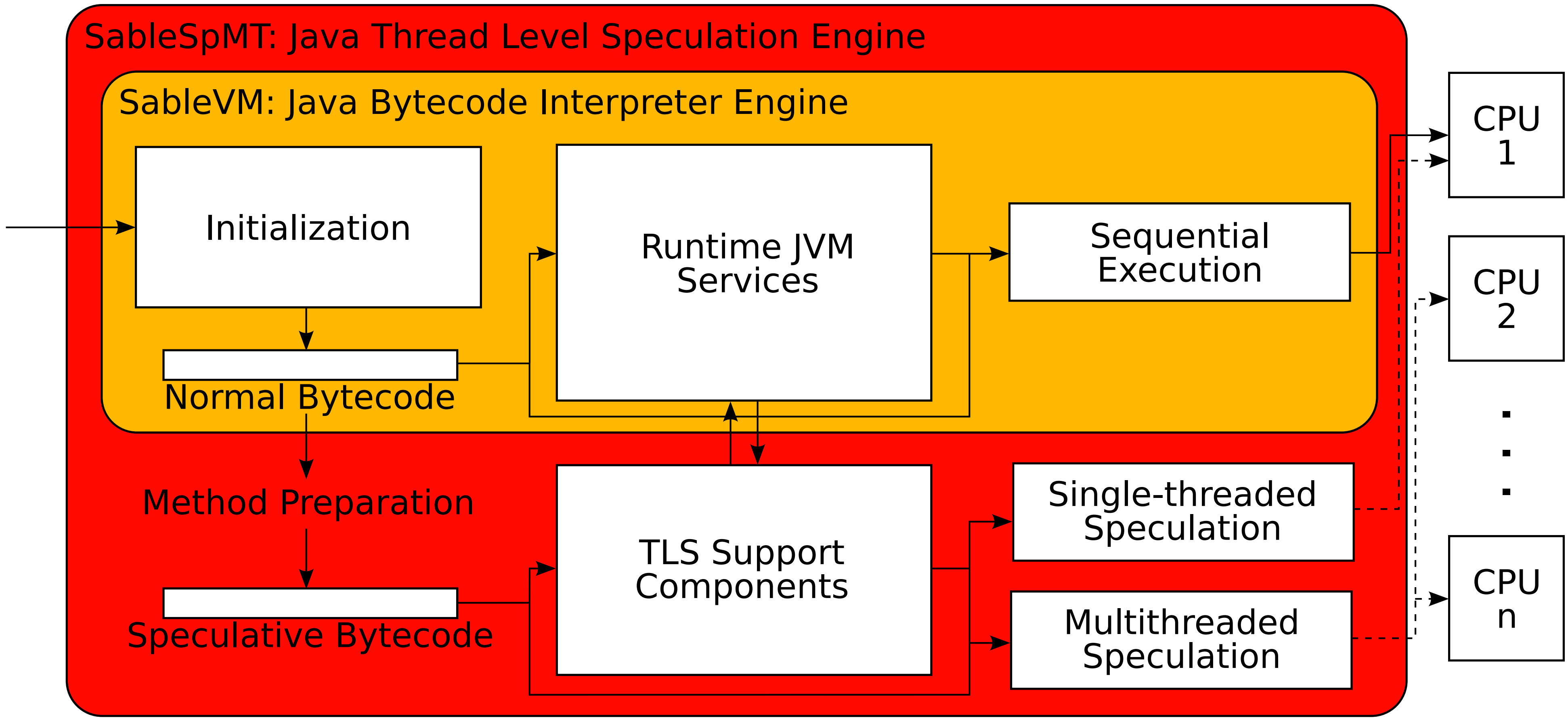
Contributions

Specific contributions:

1. Complete design for TLS at the level of Java bytecode.
2. Exposition of high level safety requirements:
   - object allocation, garbage collection, native methods, exception handling, synchronization, and the new Java Memory Model.
3. Analysis of the cost of safety considerations and benefit of runtime support components, using the SableSpMT analysis framework.

Outline

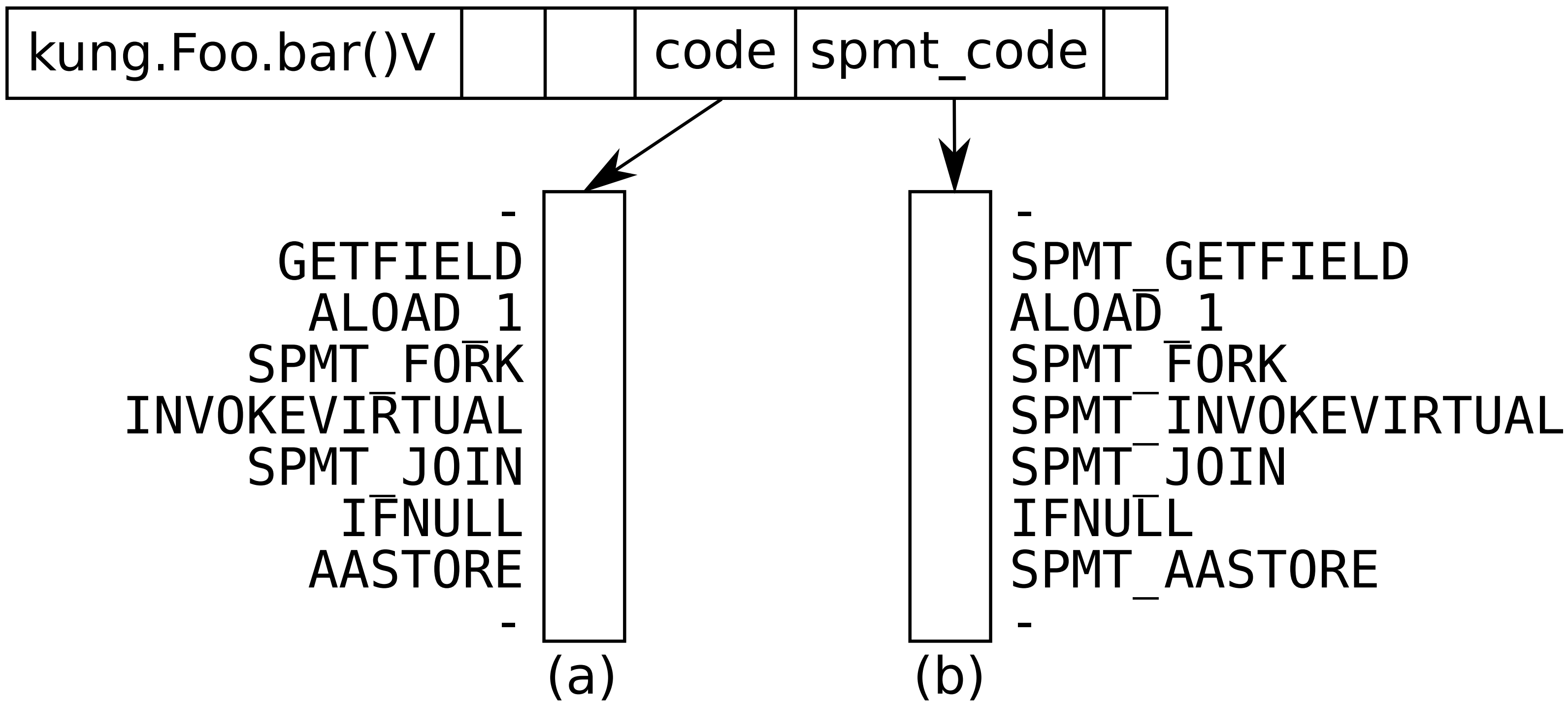
Java TLS System Overview



Method Preparation

- Need special method bodies for speculative execution.
- Insert fork and join bytecodes around every invoke.
- Duplicate normal methods, replace unsafe bytecodes with speculative versions. 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
- 25% of Java's instruction set needs non-trivial changes.
- Speculation terminates on unsafe operations.

Method Preparation



Speculative Thread Execution

- Threads are forked at every callsite.
- Out-of-order forking is permitted, but not nested speculation.
- Forking heuristics are implemented, but not currently used.
- Speculative execution depends on runtime support components.
- Threads are joined when parents return to callsites.

Priority Queueing

- Children enqueued at fork points on $O(1)$ 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.

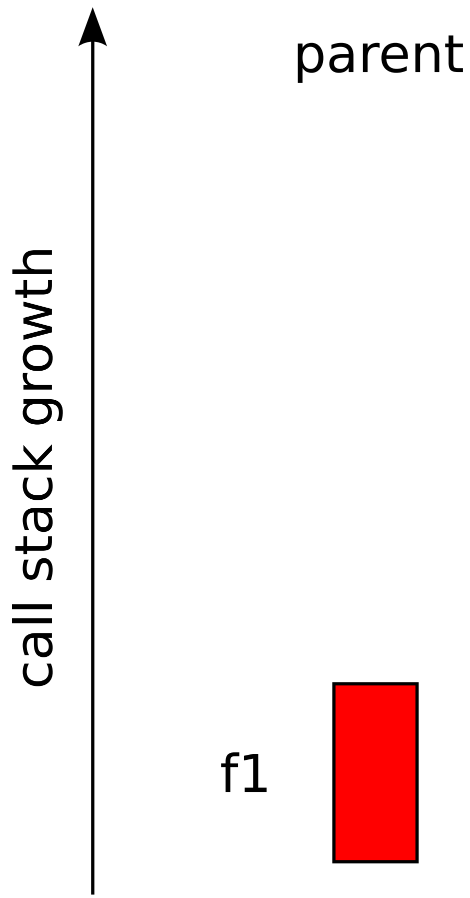
- Provide context, memoization, and hybrid predictors.
- Exploit static analyses to reduce memory and increase accuracy.
- Previously explored RVP in depth; now a system component.

Dependence Buffering

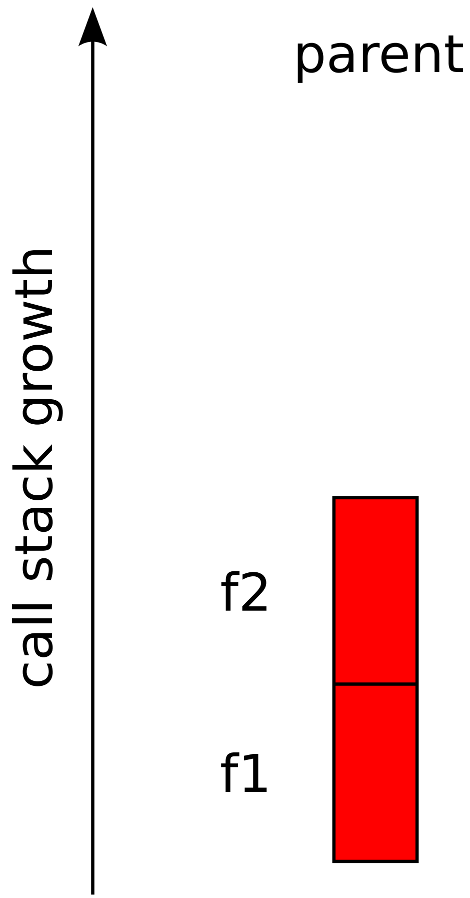
- TLS designs usually buffer speculative memory accesses in a cache-like structure.
- Here we buffer heap/static reads/writes in a software dependence buffer, using open addressing hashtables.
- Upon joining a thread, validate all reads and then commit writes.

- Instructions touching only the stack are buffered differently.

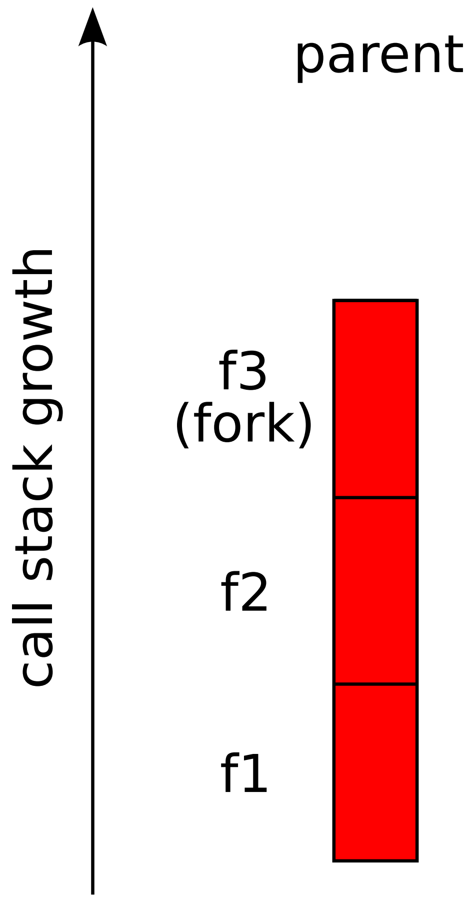
Stack Buffering



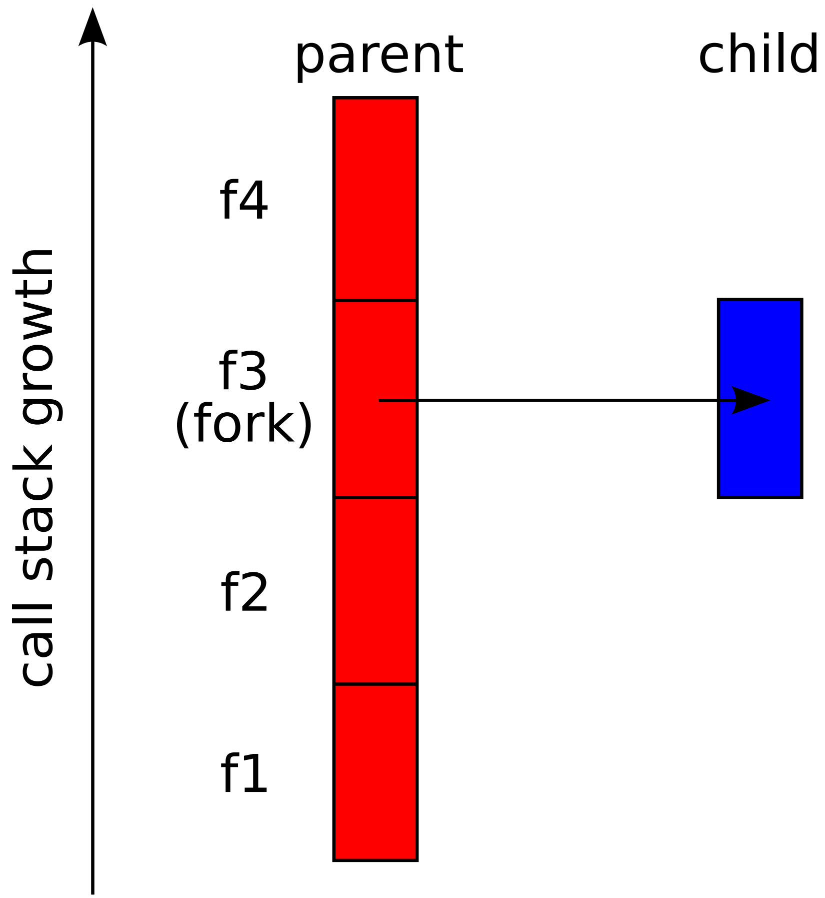
Stack Buffering



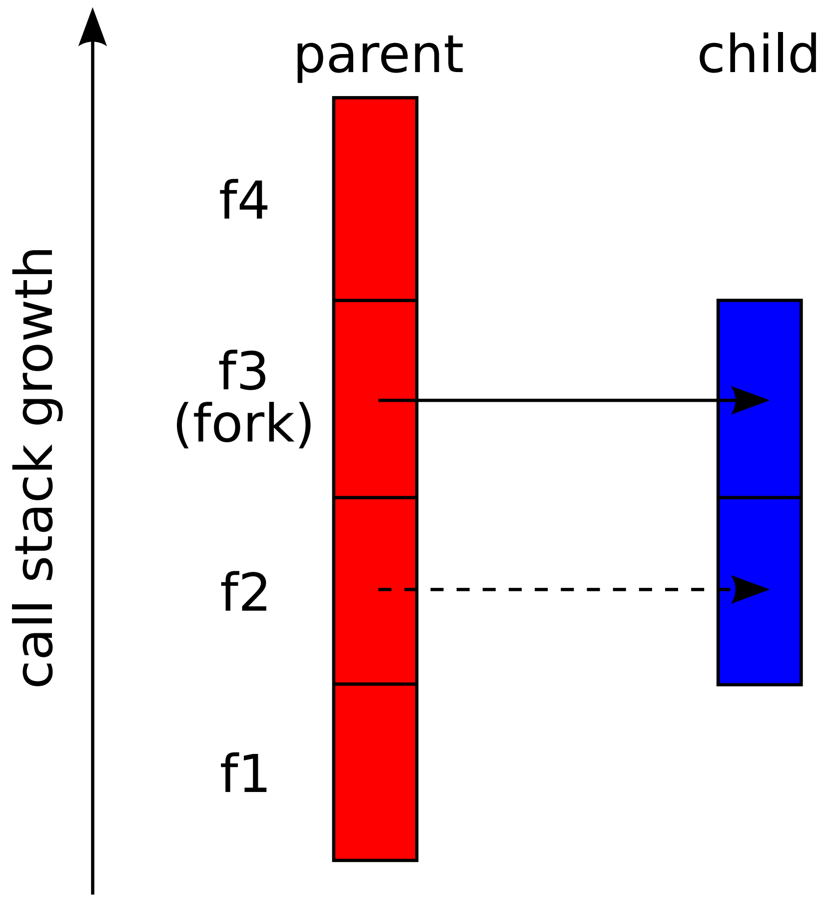
Stack Buffering



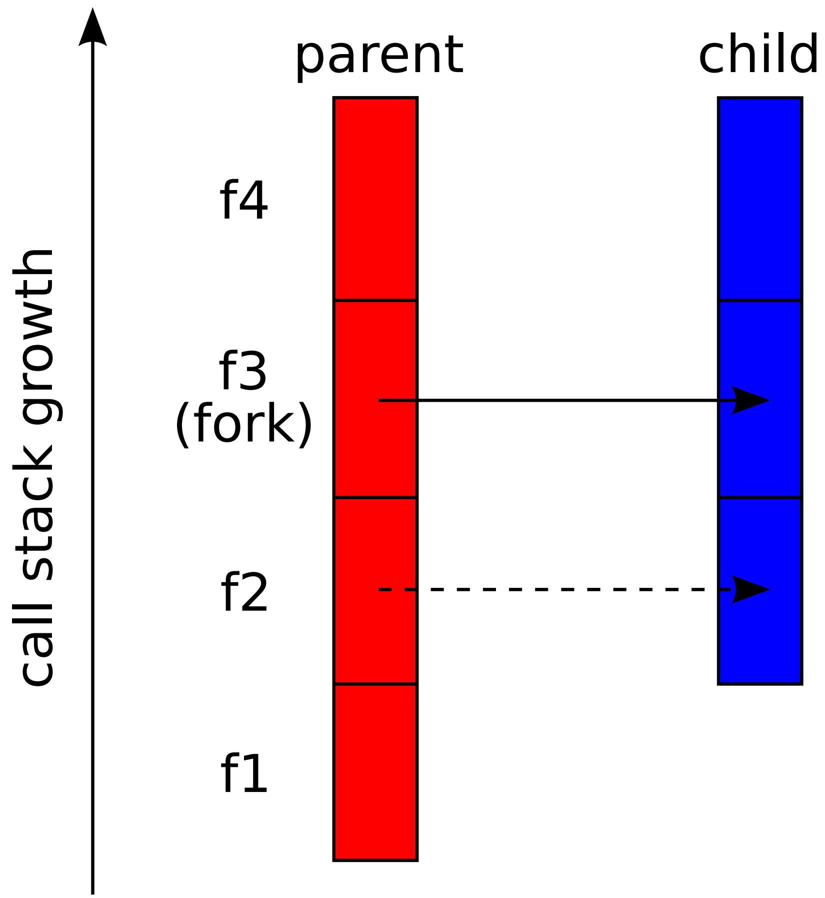
Stack Buffering



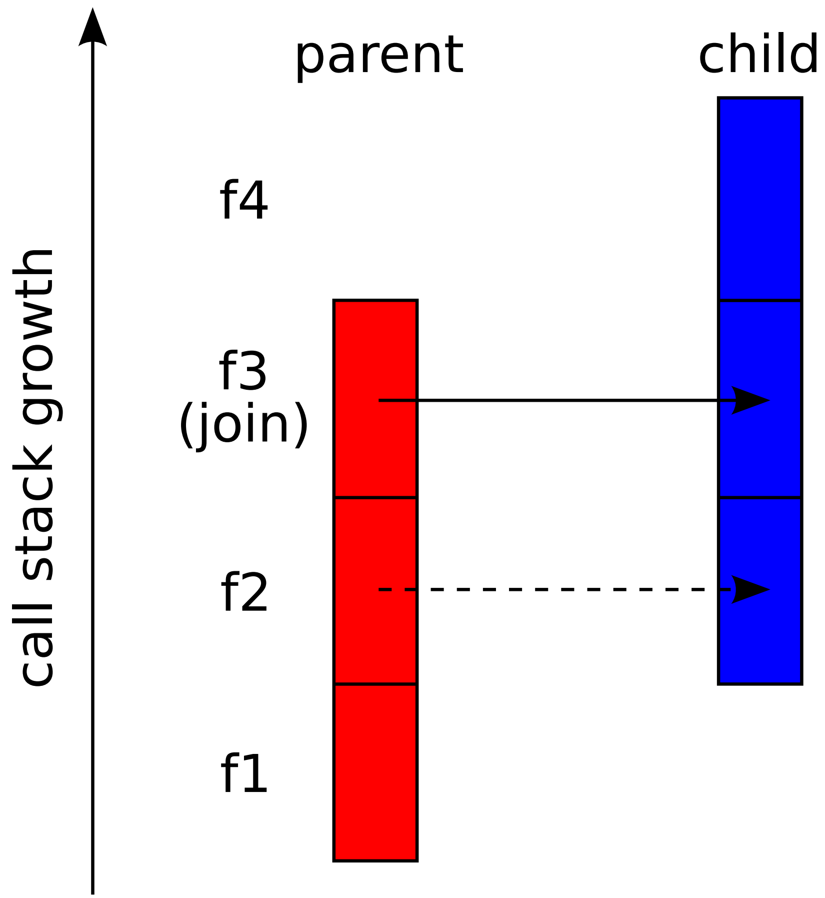
Stack Buffering



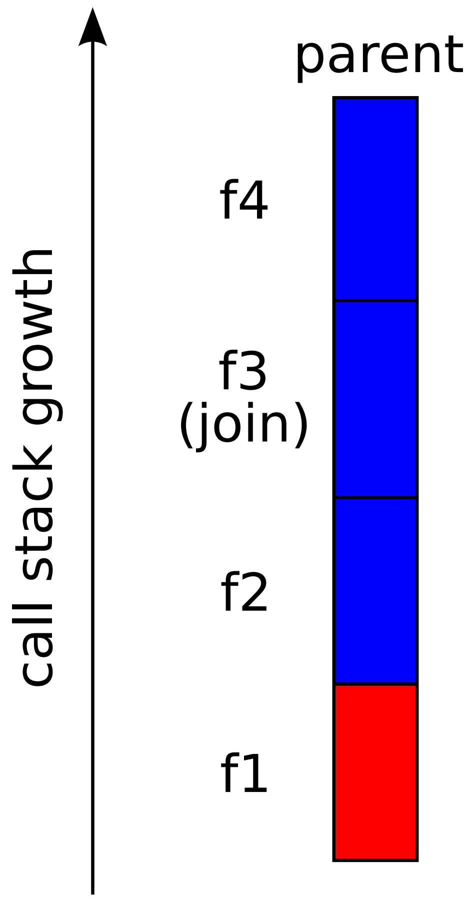
Stack Buffering



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.

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

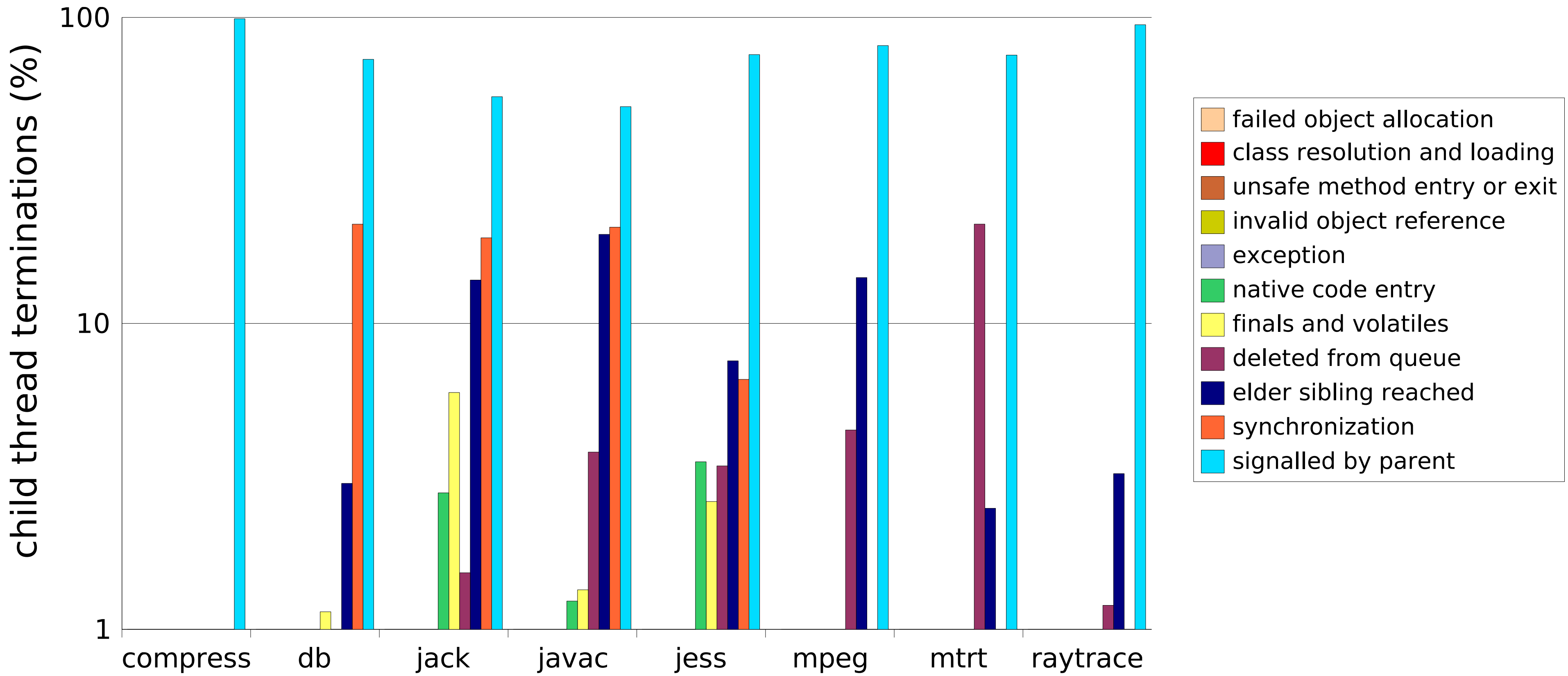
- Java allows for per-method and per-object synchronization.
- Safe non-speculatively, unsafe speculatively
    - However, we *can* fork child threads once *inside* a critical section; only entering and exiting is prohibited.
    - In principle, this encourages coarse-grained locking.
- Speculative locking is part of our future work.

Java Memory Model

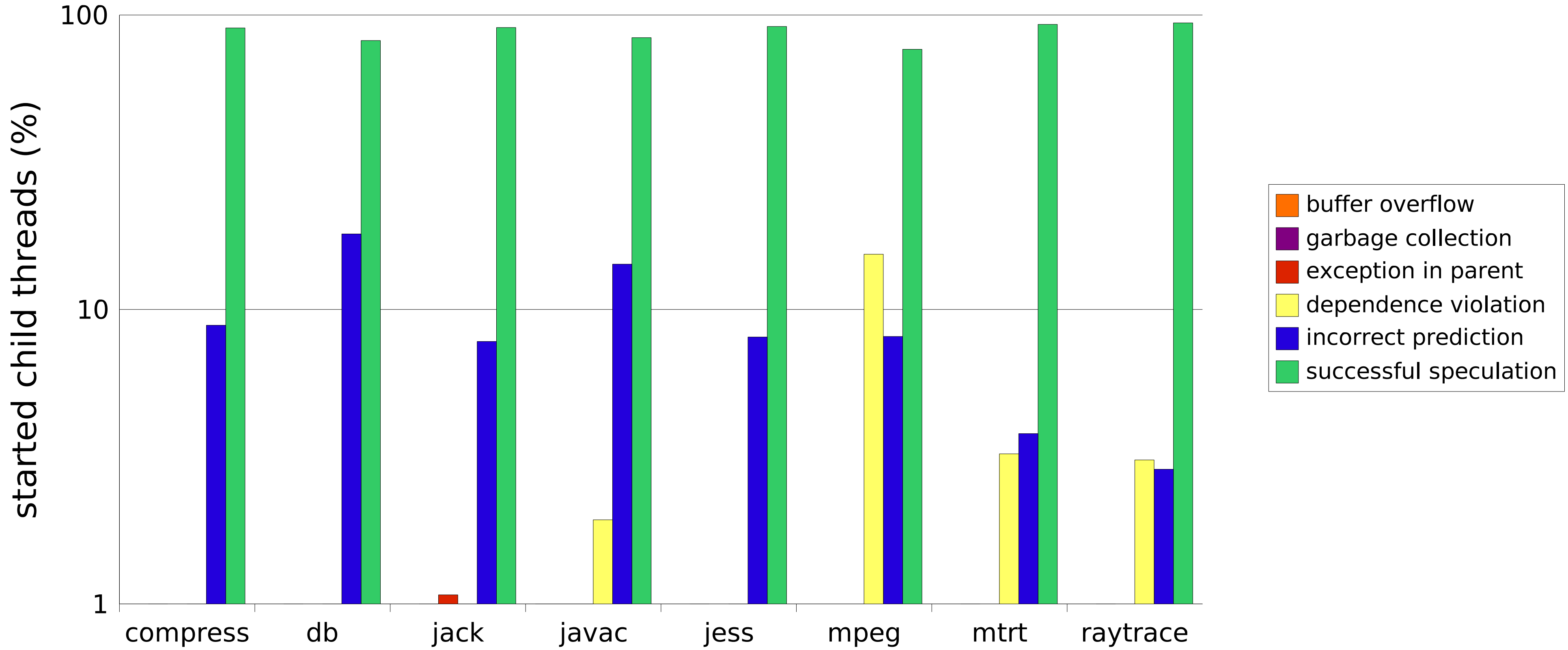
- The new Java Memory Model (JSR-133) gives specific rules about reordering, and memory barrier requirements.
- Speculation might reorder reads and writes during thread validation and committal.
- Unsafe operations we considered:
  - Locking and unlocking
  - Volatile loads and stores
  - Final stores 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

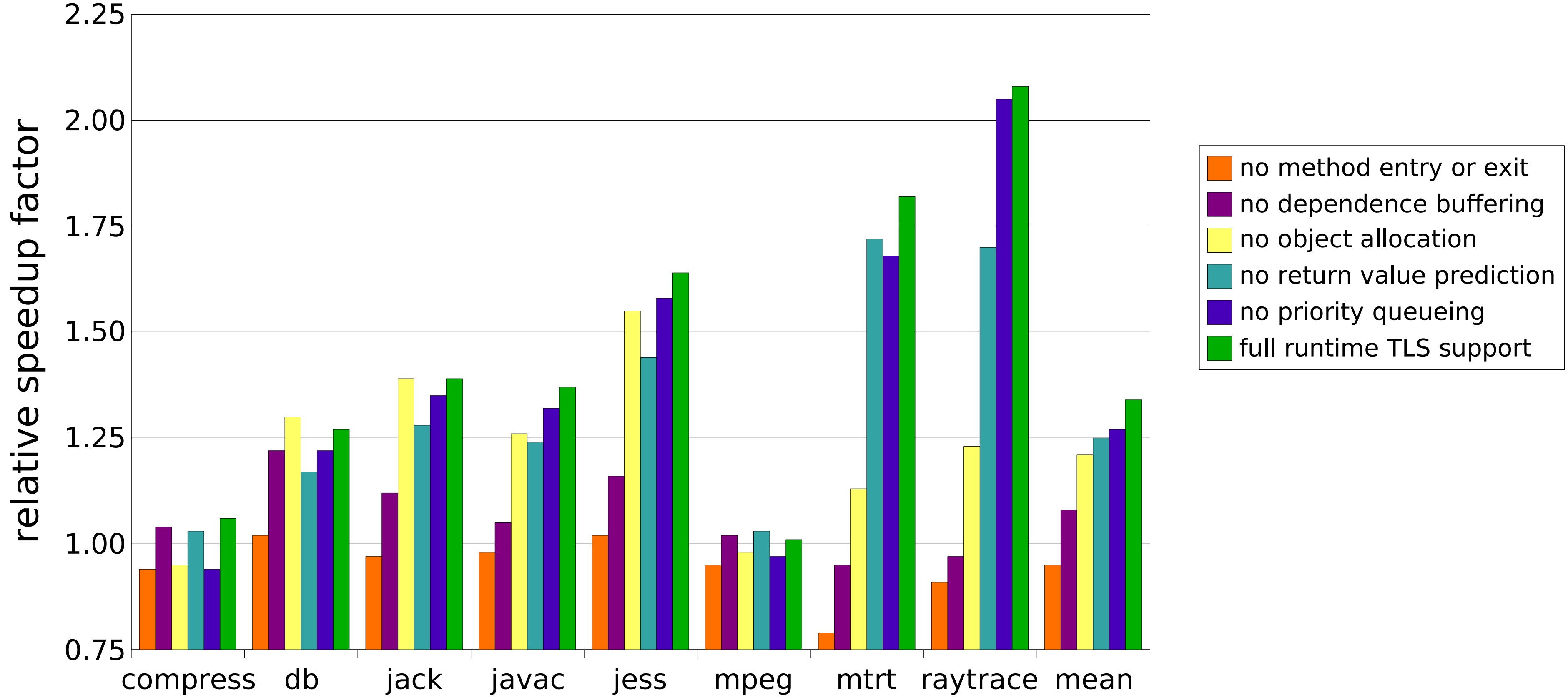
Child Termination Reasons



Child Success and Failure



Importance of TLS Support Components



Outline

Conclusions

- We provide a thorough and complete design for Java SMLP.
  - Able to handle SPECjvm98 at S100 without simplifying abstractions.
- Language and software VM contexts affect TLS designs:
  - Non-trivial safety considerations for Java
  - Most have minimal impact on performance.
    - However, synchronization can impede speculative progress significantly, as can JMM requirements.
  - Results also show an appropriate set of runtime support components is critical, and suggest relative importance.

Future Work

- Immediate performance optimisations:
  - Reduce previously characterised overhead
  - Investigate forking heuristics
  - Allow for nested speculation
  - Enable speculative locking
  - Record memory barriers in dependence buffers
  - Develop general load value prediction
- Higher level static analyses and dynamic optimisations
- Implementation in IBM's Testarossa JIT and J9 VM