libspmt: A Library for Speculative Multithreading

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

May 25th, 2007

Sable/SE Group Meeting
Outline

1 Introduction
2 Design
3 Experimental Results
4 libspmt
5 Refactoring Benefits
6 Conclusions and Future Work
Speculative Method Level Parallelism (SMLP)
Detailed SMLP Execution Model

- pre-invocation
- method body
- continuation
- post-continuation

(a) T1
(b) T1 fork
(c) T2 fork
(d) T1 fork
(e) T2 fork
(f) T1 fork

unrelated work
join & commit
join & abort
join & commit
join & abort
join & delete

time
Outline

1. Introduction
2. Design
3. Experimental Results
4. libspmt
5. Refactoring Benefits
6. Conclusions and Future Work
Java SpMT System Overview

SableSpMT: Java Thread Level Speculation Engine
SableVM: Java Bytecode Interpreter Engine

Initialization

Runtime JVM Services

Sequential Execution

Normal Bytecode

Method Preparation

TLS Support Components

Single-threaded Speculation

Multithreaded Speculation

CPU 1

CPU 2

CPU n
// execute foo non-speculatively
r = foo (a, b, c);

// execute past return point
// speculatively in parallel with foo()
Method Preparation

kung.Foo.bar()V | code | spmt_code

(a) GETFIELD
ALOAD_1
SPMT_FORK
INVOKEVIRTUAL
SPMT_JOIN
IFNULL
AASTORE

(b) SPMT_GETFIELD
ALOAD_1
SPMT_FORK
SPMT_INVokeVIRTUAL
SPMT_JOIN
IFNULL
SPMT_AASTORE
// execute foo non-speculatively
r = foo (a, b, c);

// execute past return point
// speculatively in parallel with foo()
if (r > 10)
{
    ...
}
Return Value Prediction

(a) Context Predictor

(b) Memoization Predictor
// execute foo non-speculatively
r = foo (a, b, c);

// execute past return point
// speculatively in parallel with foo()
if (r > 10)
{
    s = o1.f;  // buffer heap reads
    o2.f = r;  // buffer heap writes
}
Dependence Buffering

speculative Java stack

SPMT_GETSTATIC
SPMT_GETFIELD
SPMT_<X>ALOAD

write buffer

read buffer

main memory

Java heap values and class statics
// execute foo non-speculatively
r = foo (a, b, c);

// execute past return point
// speculatively in parallel with foo()
if (r > 10)
{
    s = o1.f; // buffer heap reads
    o2.f = r; // buffer heap writes
}

// invoke bar() speculatively
r.bar();
Stack Buffering

parent

call stack growth

f1
Stack Buffering

- parent
- call stack growth
  - f2
  - f1
Stack Buffering

parent

call stack growth

f3 (fork)
f2
f1
Stack Buffering

- Parent:
  - f1
  - f2
  - f3 (fork)
  - f4

- Child:
Stack Buffering

- Parent stack:
  - f1
  - f2
  - f3 (join)
  - f4

- Child stack:
  - (empty)

Call stack growth
Stack Buffering

- `f1`
- `f2`
- `f3 (join)`
- `f4`
// execute foo non-speculatively
r = foo (a, b, c);

// execute past return point
// speculatively in parallel with foo()
if (r > 10)
{
    s = o1.f;  // buffer heap reads
    o2.f = r;  // buffer heap writes
}

// invoke bar() speculatively
r.bar();

// stop speculation
synchronized (o4) { ... }
Single-threaded Mode

- **SPMT_FORK**
  - if (field address in buffer)
    - load buffered value
  - else
    - buffer read from address

- **SPMT_GETFIELD**
  - if (field address in buffer)
    - update buffered value
  - else
    - buffer write to address

- **SPMT_PUTFIELD**
  - stop speculation
  - save child state
  - restore parent state

- **SPMT_MONITOREXIT**
  - enter parent target method
  - execute non-speculatively

- **INVOKEVIRTUAL**
- **ARETURN**
  - return to fork point callsite

- **SPMT_JOIN**
  - verify child safety
  - if (safe)
    - commit child
    - continue non-speculatively

- non-speculative execution
- transition points
- speculative execution
Multithreaded Mode

non-speculative parent thread T1

SPMT_FORK
INVOKEX>

enqueue C1

SPMT_FORK
INVOKEX>

enqueue C2

SPMT_FORK
INVOKEX>

enqueue C3

<X>RETURN
SPMT_JOIN

delete C3

<X>RETURN
SPMT_JOIN

delete C2

<X>RETURN
SPMT_JOIN

join C1

O(1) priority queue

initialize
dequeue

SPMT_GETFIELD

SPMT_PUTFIELD

SPMT_MONITOREXIT

free SpMT helper thread pool

S1
S2
S3

free CPU pool

CPU 2
CPU 3
CPU 4

cleanup
Outline

1. Introduction
2. Design
3. Experimental Results
4. libspmt
5. Refactoring Benefits
6. Conclusions and Future Work
Overhead

- Parent:
  - Fork and enqueue child: 8%
  - Non-speculative execution: 50%
  - Signal child and await termination: 14%
  - Validate child and commit or abort: 28%

- Child:
  - Dequeue and initialize: 43%
  - Speculative execution: 52%
  - Terminate and save state: 5%
Child Termination Reasons

- failed object allocation
- class resolution and loading
- unsafe method entry or exit
- invalid object reference
- exception
- native code entry
- finals and volatiles
- synchronization
- elder sibling reached
- deleted from queue
- signalled by parent

*child thread terminations (%)*
“Speedup”

The graph shows the relative speedup for various applications under different conditions. The applications include compress, db, jack, javac, jess, mpeg, mtrt, raytrace, and mean. The conditions are indicated by different colors:
- Orange: no method entry or exit
- Purple: no dependence buffering
- Yellow: no object allocation
- Cyan: no return value prediction
- Green: full runtime TLS support

The x-axis represents the slowdown (3x, 5x, 4x, 5x, 4x, 10x, 11x, 4x) and the y-axis represents the relative speedup factor. The graph demonstrates how different optimizations affect the performance of these applications under varying slowdown conditions.
General motivation:
- SpMT projects are complex, code reuse is a nice idea.
  - Helps make projects comparable
  - Optimizations and fixes can benefit more than one project
  - Reduces individual project complexity

Specific motivation:
- We built SableSpMT, an SpMT extension to SableVM.
- Our CAS project at IBM involves adapting their Java JIT to support speculation.
- Dilemma: do we rewrite or reuse the SpMT logic?
Contributions:

- **libspmt**, a C library for speculative multithreading
  - Minimal interface
  - Modular implementation
  - Support for varying host features

- A refactored SableSpMT that uses libspmt
Development followed advice in *Refactoring*, *Test-Driven Development*, and *Code Complete 2*. General process:

- **libspmt:**
  - Start empty. Stick to modular structure and opaque types.
  - Use the Check framework for unit testing.

- **SableSpMT:**
  - `#include <spmt.h>`. Refactor, isolate module, migrate.
  - Use simple build and run scripts for functional testing.

- Avoid checking in broken builds of either libspmt or SableSpMT.
- Prefer small functions and rely extensively on the C inliner.
Forking decisions depend on dynamic fork point info:

```c
cchild = spmt_thread_request_fork (thread, fork_point);
if (child != NULL)
{
    /* save initial child state */
    ...
    spmt_thread_complete_fork (thread, &stack);
}
```
Parent returns to fork point and joins child:

```c
child = spmt_thread_request_join (thread, ret_val);
if (child != NULL)
{
    /* load final child state */
    ...
    spmt_thread_complete_join (thread);
}
```

- Child calls `spmt_child_stop_requested()` at async checkpoints
- Child may have already stopped, or never even started
- Commit is mostly handled by libspmt
Support for Host Features

Multithreading:
- Helper threads only run on unused CPU’s.
- Host calls runtime when any parent thread starts or stops
- Runtime adjusts the number of running helper threads

Garbage collection:
- Host calls runtime once for each parent when GC occurs
- Runtime invalidates and aborts all children on the stack

Exception handling:
- Host calls runtime if stack frame is popped abnormally
- Runtime invalidates and aborts the child for that frame
Core SpMT Library Modules

- Dependence buffer
- Stack mirror
- Priority queue
- Value predictors
- Thread support (parent, child, helper, runtime, thread)
Simpler “Single-threaded” Mode

(a) pre-speculation
(b) signal & wait
Before:
- Predictors were heavily tied to each other
- Hybrid predictor called each sub-predictor in order

After:
- Refactored predictors are now independently usable
- Hybrid specialization eliminates most prediction overhead
Problem 1: we want children to fork and join grandchildren

Problem 2: we want helper threads to clean up asynchronously
  - This requires `malloc()` and `free()` calls
  - We have a producer / consumer situation
  - Simple per-thread free lists do not work!
  - Frequent global synchronization is too expensive
  - SMP memory managers like Hoard are too heavyweight

Solution: manage child memory using blocks
  - Children are allocated from per-thread blocks
  - Empty and full blocks are managed globally
Outline

1. Introduction
2. Design
3. Experimental Results
4. libspmt
5. Refactoring Benefits
6. Conclusions and Future Work
Speculative multithreading is complex
We isolated the language independent parts
Refactoring exposed several new opportunities
Link IBM’s TR JIT to libspmt, generate code for it.

Explore forking heuristics

Write some papers:
  - Adaptive return value prediction
  - Nested speculation
  - libspmt paper (TR version rejected by VEE’07)
  - *Library-Centric Software Design* (LCSD’07) paper

Things I probably won’t do myself:
  - Get a non-Java host working
  - Speculative locking and software transactional memory
  - Load value prediction
  - Loop and basic block speculation