Software Speculative Multithreading for Java

Christopher J. F. Pickett

School of Computer Science, McGill University
Montréal, Québec, Canada
cpicke@sable.mcgill.ca

Abstract

We apply speculative multithreading to sequential Java programs in software to achieve speedup on existing multiprocessors. A common speculation library supports both Java bytecode interpreter and JIT compiler implementations. Initial profiling results indicate three main optimizations: adaptive return value prediction, online fork heuristics, and in-order nested method level speculation.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Process Management—Concurrency; Threads; D.2.8 [*Software Engineering*]: Metrics—Complexity measures; Performance measures; D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries; D.3.2 [*Programming Languages*]: Language Classifications—Object-oriented languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Procedures, functions, and subroutines; D.3.4 [*Programming Languages*]: Processors—Code generation; Compilers; Interpreters; Optimization; Run-time environments

General Terms Design, Experimentation, Languages, Measurement, Performance

Keywords Java, parallelism, speculative multithreading, thread level speculation, virtual machines

1. Introduction

Speculative multithreading (SpMT), also known as *thread level speculation* (TLS), is a dynamic parallelization technique that relies on out-of-order execution of sequential programs to achieve speedup > 1 on multiprocessor machines, where *speedup* is the sequential run time over the parallel run time. The key premise is that multithreaded programming is difficult and needs as much automation as possible. The advantage of SpMT is its ability to parallelize irregular applications that traditional static compilers cannot handle, at the basic block, loop, and method levels.

A large number of novel SpMT hardware designs have been evaluated along with compilers that target these architectures [6]. Most of this prior research focuses on parallelizing loops in C programs, although some work shows potential for speculation in Java programs, particularly at the method level [1, 4, 15].

OOPSLA'07 October 21–25, 2007, Montréal, Québec, Canada.
ACM 978-1-59593-786-5/07/0010.

There has been less work on software SpMT. In summary, overheads are high [11], coarser thread granularities help offset these overheads [3], manual source code changes are effective [5, 7, 14], and loop level speculation is viable [2].

This extended abstract and an accompanying poster review our work to date on software speculative multithreading for Java and outline immediate future directions. The poster is available online at `http://www.sable.mcgill.ca/publications/posters/`. We focus on Java because its object-oriented nature leads programmers to write irregular but loosely-coupled applications, a seemingly appropriate source of speculative parallelism. We focus on software SpMT because Java is a rich language with complex runtime behaviour that may not translate well to hardware SpMT designs, and because new hardware is prohibitively expensive and may not actually be required. We use *method level speculation*, which creates child continuation threads on method invocation and joins them upon method exit, because it can subsume loop level speculation and also expose additional parallelism [1]. We target Java virtual machines to maximize SpMT transparency, compatibility, and automation.

2. Current Work

Hu *et al.* show that hardware return value prediction (RVP) benefits method level speculation [4], and we start by implementing software RVP in a Java virtual machine [10]. Our baseline *hybrid* predictor is composed of previously studied predictors, and per-callsite hybrid instances select the best performing sub-predictor on each invocation. We introduce a new *memoization* predictor that hashes together method inputs from the Java operand stack, and find that it improves hybrid accuracy by 10%.

We next investigate two different compiler analyses to reduce RVP memory costs and extract more accuracy [9]. The first, a *return value use* analysis, finds unused return values as well as those compared only against constants inside boolean expressions. The runtime system uses this information to weaken predictor accuracy constraints. The second, a *parameter dependence* analysis, finds parameters that do not affect the return value. The runtime system excludes these parameters from memoization predictor inputs. The combined analyses reduce predictor memory requirements by 5% on average and improve accuracy by 5% for the `db` benchmark. The most surprising part of these two RVP studies is just how predictable Java return values are: our hybrid predictor obtains 80% accuracy over SPEC JVM98 and

95% accuracy for db. This accounts for every non-void return value in an interpreted context without inlining.

We subsequently build on this RVP implementation to create SableSpMT, a working Java bytecode interpreter SpMT prototype and analysis framework [11]. Our *out-of-order* speculation nesting model creates one child continuation for each invocation in a non-speculative parent thread. In a *multithreaded mode*, helper threads dequeue these children from a priority queue and execute them concurrently. Speculation stops either when a parent returns from a call and signals its current child or when a child encounters an illegal instruction. In a debug *single-threaded mode*, speculative children execute in the same physical operating system thread as their non-speculative parent. Both execution modes interoperate with non-speculative multithreading.

SableSpMT also supports *dependence buffering*, which buffers reads from and writes to heap and static data like strongly isolated transactional memory [8], and *stack buffering*, an optimization possible for managed code that buffers entire stack frames at once. We measure the contributions of different speculation support components to speculative parallelism, and provide an analysis of safety considerations specific to Java and their impact, including speculative bytecode execution, garbage collection, exception handling, native methods, object allocation, and synchronization [12]. The complete system slows down execution of SPEC JVM98 by 4.5x on a 4-way machine. However, we compute a *relative speedup* of 1.3x by comparing run times to a control experiment that does not allow successful speculation to commit but otherwise incurs the same overheads. High overheads are expected for an initial implementation, and profiling data indicate the optimizations in Section 3.

In the interim however, we refactor our SableSpMT implementation to create libspmt, a VM-independent library for speculative multithreading [13]. JIT compiler support is an ultimate goal of this work, and rather than rewrite the speculation support and maintain two disjoint implementations we decided to isolate the reusable logic. We establish a reasonably minimal interface and a modular C implementation that is portable and backed by unit tests. A major benefit of this approach is that SableSpMT remains as a working functional test. Initial experience with libspmt suggests that implementing new optimizations is straightforward; of course, the true design test will be JIT integration.

3. Immediate Future Work

Our JIT compiler implementation platform is the Testarossa JIT component of the IBM J9 Java Virtual Machine. Additional complexity over interpreted speculation lies in generating speculative method bodies that interact with libspmt. Otherwise, we are following the same high-level client design as used by SableSpMT.

Aggressive RVP strategies are costly. We suggest that for each speculation point there exists an optimal return value predictor and that this predictor can be discovered dynamically. We are investigating whether a hybrid predictor implementation that specializes to its ideal sub-predictor is feasible, and whether it can approximate the accuracy of an offline oracle determined by *post mortem* analysis.

Our initial system configuration forks threads on every method invocation and so child thread lengths are usually quite short. Existing dynamic speculation profiles can support online *fork heuristics* [15] that take into account child continuation lengths, parent invocation lengths, speculation success rates, and predictor confidence. JIT compiler inlining may also provide a kind of natural fork heuristic [4].

Finally, helper threads are mostly idle, which reveals a surprising *lack* of parallelism despite reasonable relative speedup. We recently extended libspmt and SableSpMT to support *in-order nesting* wherein child threads can create new child threads of their own. This populates the priority queue, reduces idle times, exposes more parallelism, and leads to improved relative speedup.

References

[1] M. K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *PACT'98*, pages 176–184, Oct. 1998.

[2] M. Cintra and D. R. Llanos. Design space exploration of a software speculative parallelization scheme. *TPDS*, 16(6):562–576, June 2005.

[3] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI'07*, pages 223–234, June 2007.

[4] S. Hu, R. Bhargava, and L. K. John. The role of return value prediction in exploiting speculative method-level parallelism. *JILP*, 5:1–21, Nov. 2003.

[5] I. H. Kazi and D. J. Lilja. JavaSpMT: A speculative thread pipelining parallelization model for Java programs. In *IPDPS'00*, pages 559–564, May 2000.

[6] A. Kejariwal and A. Nicolau. Speculative execution reading list. `http://www.ics.uci.edu/~akejariw/SpeculativeExecutionReadingList.pdf`, 2007.

[7] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI'07*, pages 211–222, June 2007.

[8] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, Dec. 2006.

[9] C. J. F. Pickett and C. Verbrugge. Compiler analyses for improved return value prediction. Technical Report SABLE-TR-2004-6, McGill University, Oct. 2004.

[10] C. J. F. Pickett and C. Verbrugge. Return value prediction in a Java virtual machine. In *VPW2*, pages 40–47, Oct. 2004.

[11] C. J. F. Pickett and C. Verbrugge. SableSpMT: A software framework for analysing speculative multithreading in Java. In *PASTE'05*, pages 59–66, Sept. 2005.

[12] C. J. F. Pickett and C. Verbrugge. Software thread level speculation for the Java language and virtual machine environment. In *LCPC'05*, volume 4339 of *LNCS*, pages 304–318, Oct. 2005.

[13] C. J. F. Pickett, C. Verbrugge, and A. Kielstra. libspmt: A library for speculative multithreading. Technical Report SABLE-TR-2007-1, McGill University, Mar. 2007.

[14] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *OOPSLA'05*, pages 439–453, Oct. 2005.

[15] J. Whaley and C. Kozyrakis. Heuristics for profile-driven method-level speculative parallelization. In *ICPP'05*, pages 147–156, June 2005.