Speculative Multithreading: An Object-Driven Approach

Simon Wilkinson and Ian Watson
University of Manchester, UK
{swilkinson, iwatson}@cs.manchester.ac.uk

Abstract

Speculative multithreading (SpMT) is a parallelizing execution model for single-threaded programs on multi-core architectures. In this paper, we introduce a new SpMT model, Object-Driven Speculative Multithreading, which exploits the structure and semantics of object-oriented programs to generate speculative parallelism. Within our technique, individual program objects take the responsibility to predict their own future behavior, and to speculatively mutate their own state and that of other objects. We present a detailed description of the Object-Driven Speculative Multithreading model, discuss our current progress and challenges, and include some preliminary results.

1 Introduction

Speculative multithreading (SpMT) has been widely investigated as a parallelizing execution model for single-threaded programs on multi-core architectures. The premise of SpMT is to execute the predicted future computation of a sequential program in 'sandboxed' parallel threads, whose side-effects are committed to memory in original program order. To maintain correctness, the control and data-flow violations that this parallelization might expose are caught at runtime by an underlying conflict detection mechanism, which rolls-back or squashes unsafe execution. Using this model, parallelization is cast from a question of correctness, to one of a cost/benefit trade-off: The fixed overheads of SpMT and the dynamic overheads of mis-speculation must be offset by the speedup brought by parallel execution.

An important component to winning this trade-off is an SpMT system's *thread spawning model*, which defines where a program is partitioned into threads and when they begin execution. The thread spawning model must balance three mutually competitive goals: to extract the greatest amount of threads as possible; to maximize the chance of any particular thread executing without violation; and to minimize the necessary complexity for the underlying mechanisms of conflict detection, side-effect buffering, rollback and commit.

In this paper, we introduce a new SpMT model, Object-Driven Speculative Multithreading (ODe-SpMT), which allows spawning to occur outside of the sequential control-flow constraints employed by existing schemes. Our aim is to abstract SpMT into an implicit object behavior for object-oriented programs, whereby individual objects take the responsibility to predict their own future behavior, and to speculatively mutate their own state and that of other objects.

In the following section we introduce the ODe-SpMT model, discuss our current progress and challenges, and present some preliminary results. In Section 3 we conclude with a summary of related work.

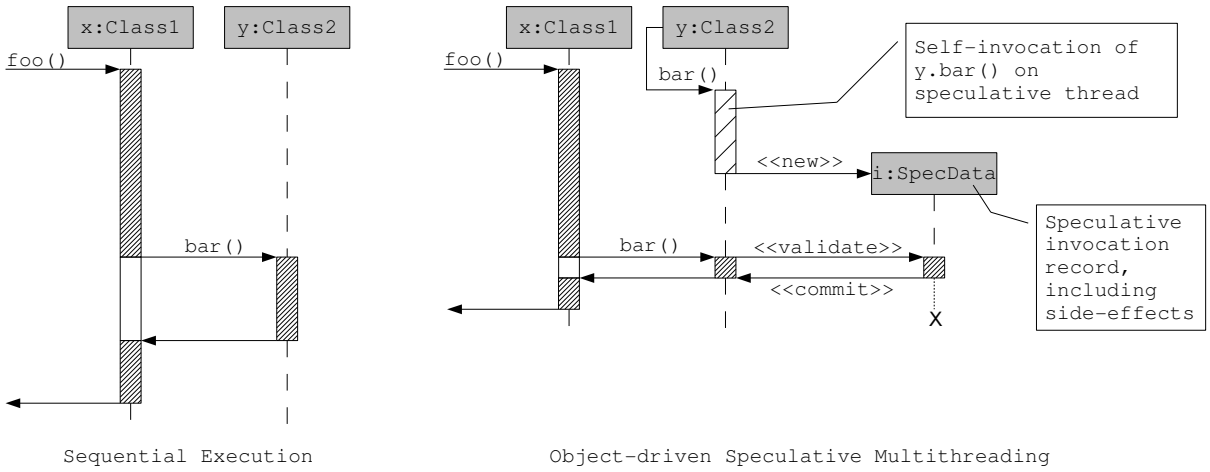

Figure 1: UML sequence diagrams of object invocation for sequential and ODe-SpMT execution.

2 Object-Driven Speculative Multithreading

The parallelization strategy of SpMT is based on the premise that in a sequential program, data is often available before the computation that consumes it. A thread spawning model must, therefore, predict both when data is likely to be available and the associated computation. Previous studies [2, 7, 8, 12] on the application of SpMT to object-oriented programs have evaluated control-flow driven threading spawning models, which essentially ignore any object semantics within the original program. For example, *method-level* spawning [2] predicts that when dynamic control-flow reaches a method call, data is available for the method's continuation. Prediction of the associated computation is implicit in the control-flow of the method's eventual return. The key observation of ODe-SpMT is that the encapsulation principle of object-orientation makes an *explicit* association between data and computation that is exploitable in making these predictions.

ODe-SpMT logically partitions a program so that a method is the unit of work for a speculative thread. The novelty of our approach is to spawn threads on a per-object basis, at a time when the object itself can predict a future incoming method invocation and the availability of the necessary data (the method's read-set). At such a time, the object *self-invokes* the method, spawning its execution on a speculative thread. The thread's side-effects are buffered, and upon method completion, are logically associated with the object, together with a record containing meta-information needed to later validate the execution. Validation and committal occur when a self-invoked method is externally-invoked by another object. Hereafter, we make a distinction between standard method invocation semantics, which we denote *external-invocation*, and object self-invocation.

Figure 1 demonstrates a simple case of object invocation within the ODe-SpMT execution model using the notation of a UML sequence diagram. In the single-threaded case, object x invokes $y.bar()$ as determined by sequential control-flow. Within ODe-SpMT, object y self-invokes $bar()$ early, on a speculative thread, in anticipation of the invocation. A record of the speculative execution is created,

which is later validated and committed when the method is externally invoked by object x.

Examining this basic model, we identify four requirements that must be fulfilled in order for ODe-SpMT to be effective:

- An object must be able to predict its future external-invocations.
- An object must be able predict when a potentially self-invoked method's read-set is available.
- Self-invoked methods must generate significant useful parallelism.
- An efficient mechanism for conflict resolution and side-effect commit must underpin the model.

The following sections describe our current work on each of these components.

2.1 Predicting Future Method Invocations

This problem is approached from a simple perspective, by predicting the *next* (in original program order) method to be externally-invoked on an object.

We have profiled invocations within single-threaded Java programs from the SPECJvm98 and Dacapo benchmark suites, and observed that approximately 92% of all objects have at most four (non-constructor) methods invoked upon them. Given this characteristic, we propose the use of per-object *finite context method* (FCM) prediction on invocation histories. FCM is a low-cost scheme for predicting a value, based on a small history of proceeding values. We note that FCM is routinely used in other speculative execution contexts, such as branch prediction and load value prediction [10].

Our FCM design uses a packed numbering scheme for a class's methods (including all those inherited), where each method is assigned a unique identifier within its class. Currently this is in the range $0 - 255$. Every object is initialized with an additional header word – the *history* word, which records the method identifier for the last four external-invocations or committed self-invocations on the object. A per-class table is kept, which records a finite number of previously seen history words, using an LRU eviction scheme. Each table entry points to a second-level table, which records the frequency of method identifiers that have followed the history word. To gain a prediction of the next method to be invoked, an object looks-up its history word in the class's prediction table, and selects the method identifier most frequently seen next. Per-class table updates occur at the return of external-invocations or the committal of self-invocations.

We have implemented per-object invocation predictors within the Jikes research virtual machine. Table 1 shows the accuracy obtained for a sample of six programs taken from the SPECjvm98 and Dacapo benchmarking suites[1]. The 'No Pred' column refers to prediction requests that could not be fulfilled

Benchmark	Invoke Qty	Correct	No Pred.
antlr	156,705,456	95.35%	1.40%
db	86,593,360	94.61%	5.13%
bloat	323,644,584	90.66%	7.10%
jess	94,440,746	83.89%	8.63%
jack	36,851,458	81.81%	10.54%
luindex	29,860,790	80.03%	8.99%

Table 1: Accuracy obtained by per-object invocation FCM predictors. SPECjvm98 benchmarks db, jack and jess were executed for 1 iteration using the s100 data set. Dacapo benchmarks antlr, bloat and luindex were executed for 1 iteration using the default data set.

[1]All single-threaded programs from these benchmarks were run, the six results shown are representative of the accuracy levels achieved. luindex proved the hardest to predict across all benchmarks.

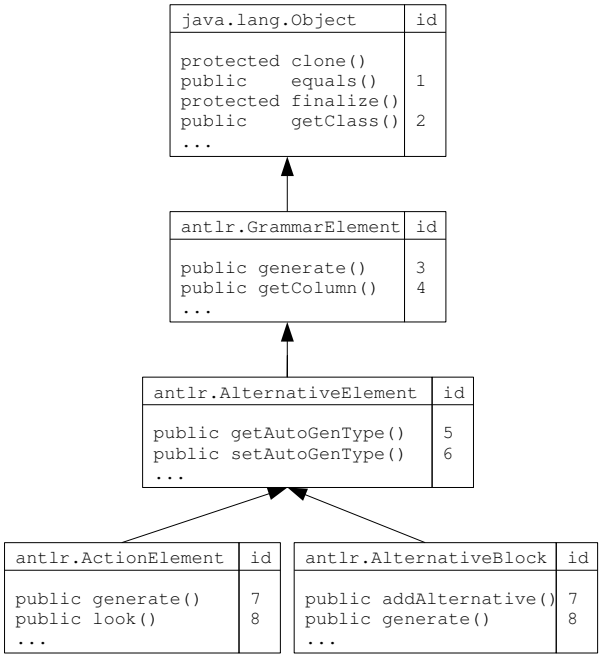

Figure 2: Example class hierachy with method identifiers for FCM prediction.

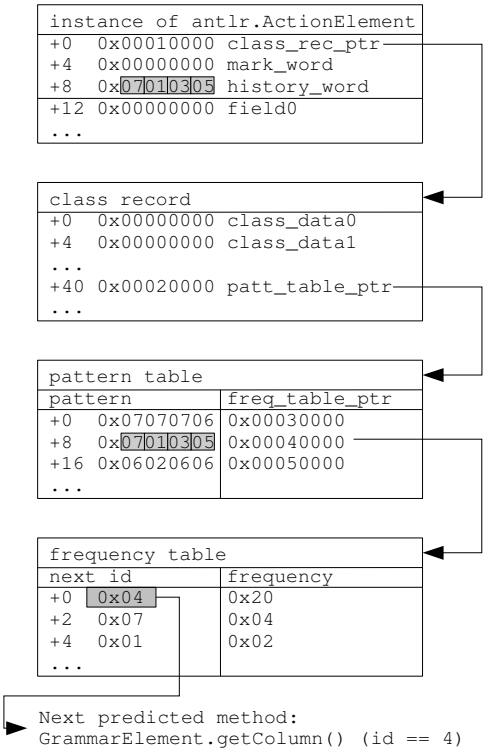

Figure 3: The application of ODe-SpMT's next-method prediction mechanism to an instance of `ActionElement`, a concrete class within the class hierarchy of Figure 2.

due to the learning time of the FCM method. We are encouraged by the relatively high accuracy, and believe the no-prediction rate can be lowered by including constructor invocations within an object's history word as a predictor warm-up.

In Figure 2, a portion of the `GrammarElement` class hierarchy from the Dacapo benchmark's antlr application is shown. Each public method is annotated with its unique prediction identifier; note that identifiers are re-used between sibling classes. Figure 3 shows the application of ODe-SpMT's next-method prediction mechanism to an instance of `ActionElement`, a `GrammarElement` subclass.

2.2 Predicting Self-invoked Method Read-set Availability

A method's read-set is composed of two subsets: the method's arguments (excluding any notion of a 'this' pointer, which is implicit to the predicting object), and the object fields read by the method. For brevity, we refer only to object 'fields'. Heap-based arrays are considered to be objects, with each array element a separate field.

Our initial approach for predicting the availability of object fields is to assume availability as soon as the previous externally-invoked method has completed. In simple terms, an object will self-invoke its next predicted method when no external-invocation is ongoing. This assumption of read-set availability is similar to that of method-continuation spawning schemes, in that we exploit the task-level parallelism inherent in the program's structural

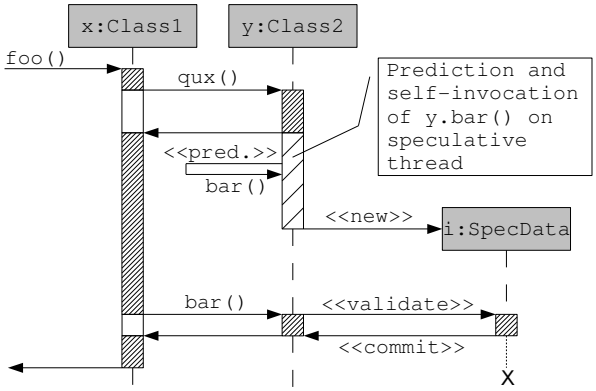

Figure 4: UML sequence diagram of object invocation for ODe-SpMT execution with next method prediction.

composition. Figure 4 shows a revised UML sequence diagram, with next-method prediction and self-invocation following external-invocation completion.

Method arguments are not available any earlier than external-invocation time, hence, they cannot be considered when predicting read-set availability. To overcome this limitation, the values themselves must be predicted. FCM prediction is attractive for this purpose as it has very low performance overheads, however, the space requirement to keep a per-object history for each method and parameter is not feasible. We are currently investigating how a time/space trade-off can be achieved around an FCM design.

2.3 Generating Parallelism

We first consider leaf methods in the context of self-invocation. A SpMT system's non-speculative thread will potentially trigger spawning on return from such a method, and commit the resulting self-invoked method's side-effects at the sequentially next invocation on the object. However, it may be the case that the speculation was incorrect – the next invocation on the object is a different method to that which has been self-invoked. In this situation we do not need to squash the speculative execution, rather, its association with the object can be maintained until resource constraints force its eviction. Therefore an object may acquire multiple speculative side-effect sets, which can be matched against incoming external-invocations and, if validated, committed out-of-order of their actual execution time.

This model is conceptually simple, but it must be expanded to generate the necessary levels of parallelism to offset its potential overheads. We therefore consider the semantics of *nesting*, which translates to the self-invocation of non-leaf methods. A self-invoked non-leaf method may *externally-invoke* methods of other objects. Therefore, we have the opportunity to trigger self-invocation within these objects, and also consume speculative side-effects from previous self-invocations. In the most unconstrained case, we need not apply any high-level ordering rules as to how this occurs, and simply let the underlying conflict resolution mechanism discard unsafe speculation from unordered nesting. This, however, may generate too much wasted execution to justify. We are currently exploring this design space experimentally in our simulator of ODe-SpMT.

2.4 Conflict Resolution and Side-effect Commit

Many existing thread spawning models use the entry and exit points of control-flow structures to drive thread creation [5]. These schemes are attractive as they create a total ordering of threads and speculative data in the system (with respect to sequential execution), which allows simple mechanisms for conflict resolution and side-effect commit. ODe-SpMT has more complex semantics, however, an underlying mechanism can enforce sequential consis-

tency, even in the case of unordered nesting.

Conflict resolution within ODe-SpMT is based on lazy validation of threads' read and write sets. Firstly, the system's non-speculative thread maintains a circular data-structure[2] recording the last n locations to which it has written. This structure is incrementally timestamped to partition it into m epochs. Speculative threads maintain two private data-structures to record their read and write locations. On thread start-up they also record the current non-speculative epoch, and any argument value predictions they will consume. Upon speculative thread completion (i.e. self-invoked method return), the read and write sets, and the buffer of speculative side-effects are associated with an invocation *record* which identifies the self-invoked method and the current non-speculative epoch. The invocation record is then maintained by the self-invoking object.

Speculative data committal occurs at the time of an external-invocation. If it is the non-speculative thread externally-invoking, then the receiver object is queried for a speculative invocation record for the method. If this exists then any argument value predictions are validated, and the record's read-set is checked for a null intersection with the non-speculative thread's write-set. The epoch timestamps are used to reduce the effective size of the write-set to those accesses which occurred at or after the speculative thread's creation. If no conflict is found, then the speculative data is committed to memory. If a conflict is found, then the invocation record is discarded and the method externally-invoked.

In the case where a *speculative* thread externally-invokes a method which has an invocation record, then we perform an equivalent operation as that for the non-speculative thread, but instead of committing results to memory, the invocation record is subsumed into that of the currently self-invoking method. If the validation is unsuccessful then the method is externally-invoked, but we need not discard the invocation record; a subsequent external-invocation by a different thread may successful commit the results.

This commit procedure guarantees correctness for the unordered nesting model discussed in Section 2.3. It also, however, clearly adds overhead to the non-speculative thread's progress. As a response, we note that ODe-SpMT operation is orthogonal to that of method-continuation speculation [2], which could be applied at method call sites to hide ODe-SpMT commit latency.

[2]For this discussion, we generalize by not making a distinction between hardware and software components.

3 Related Work

The implementation and performance of loop-iteration, loop-continuation and method-continuation thread spawning for SpMT have been widely investigated [2, 12, 5, 13, 4]. Sequential control-flow based spawning has also been considered at the level of basic-blocks [9], and optimized via graph-theoretic strategies for identifying likely control-flows [3]. We note that none of these studies consider directly applying object semantics to spawning.

Value prediction has been applied to SpMT to overcome inter-thread register and flow dependencies [6, 11, 9]. We use value prediction to speculate directly on control-flow, via per-object invocation prediction.

Program Demultiplexing (PD) [1] has similar semantics to ODe-SpMT, in that procedure bodies are the units of speculative work, and spawning occurs when a procedure's read-set is available. However, PD spawning heuristics are based on offline profiling of data-flow, and do not consider object behavior or structure.

References

[1] Saisanthosh Balakrishnan and Gurindar S. Sohi. Program demultiplexing: Data-flow based speculative parallelization of methods in sequential programs. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 302–313, 2006.

[2] M. K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 176–184, 1998.

[3] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proceedings of the SIGPLAN '04 Conference on programming language design and implementation (PLDI)*, pages 59–70, 2004.

[4] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: a TLS compiler that exploits program structure. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 158–167, 2006.

[5] Pedro Marcuello and Antonio González. Thread-spawning schemes for speculative multithreading. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, pages 55–64, 2002.

[6] Pedro Marcuello and Antonio González. Thread partitioning and value prediction for exploiting speculative thread-level parallelism. *IEEE Transactions on Computers*, 53, 2004.

[7] Jeffrey T. Oplinger, David L. Heine, and Monica S. Lam. In search of speculative thread-level parallelism. In *PACT '99: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 303–313, 1999.

[8] Christopher J. F. Pickett and Clark Verbrugge. Software thread level speculation for the Java language and virtual machine environment. In *LCPC'05: Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, pages 304–318, 2005.

[9] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 269–279, 2005.

[10] Yiannakis Sazeides and James E. Smith. The predictability of data values. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 248–258, 1997.

[11] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. Improving value communication for thread-level speculation. In *Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, pages 65–75, 2002.

[12] Fredrik Warg and Per Stenström. Limits on speculative module-level parallelism in

imperative and object-oriented programs on CMP platforms. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 221–230, 2001.

[13] Fredrik Warg and Per Stenström. Reducing misspeculation overhead for module-level speculative execution. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 289–298, 2005.