Safe Programmable Speculative Parallelism

Prakash Prabhu
Princeton University
pprabhu@cs.princeton.edu

G. Ramalingam Kapil Vaswani
Microsoft Research, India
kapilv,grama@microsoft.com

Abstract

Execution order constraints imposed by dependences can serialize computation, preventing parallelization of code and algorithms. Speculating on the value(s) carried by dependences is one way to break such critical dependences. Value speculation has been used effectively at a low level, by compilers and hardware. In this paper, we focus on the use of speculation *by programmers* as an algorithmic paradigm to parallelize seemingly sequential code.

We propose two new language constructs, *speculative composition* and *speculative iteration*. These constructs enable programmers to declaratively express speculative parallelism in programs: to indicate when and how to speculate, increasing the parallelism in the program, without concerning themselves with mundane implementation details.

We present a core language with speculation constructs and mutable state and present a formal operational semantics for the language. We use the semantics to define the notion of a correct speculative execution as one that is equivalent to a non-speculative execution. In general, speculation requires a runtime mechanism to undo the effects of speculative computation in the case of mispredictions. We describe a set of conditions under which such rollback can be avoided. We present a static analysis that checks if a given program satisfies these conditions. This allows us to implement speculation efficiently, without the overhead required for rollbacks.

We have implemented the speculation constructs as a C# library, along with the static checker for safety. We present an empirical evaluation of the efficacy of this approach to parallelization.

Categories and Subject Descriptors D.1.3 [*Programming Techniques - Concurrent Programming*]: Parallel Programming; D.3.3 [*Programming Languages - Language Constructs and Features*]: Concurrent Programming Structures

General Terms Languages

Keywords speculative parallelism, value speculation, safety, purity, rollback freedom

1. Introduction

Speculation refers to the act of taking some risks in anticipation of reward. While speculation is almost second nature of humans, it has been recognized as an important system design principle [12]. Many high performance systems such as microprocessors, file systems, and databases use speculation to improve performance. For example, software transactions and futures use speculation to increase parallelism in programs.

In this paper we focus on the use of *value speculation* to achieve parallelism. Value speculation is a mechanism for increasing parallelism by predicting values of data dependencies between tasks. Value speculation is by no means a new concept. Compiler writers and computer architects have investigated the use of value speculation for extracting instruction-level parallelism. This type of value speculation is completely transparent to the programmer and the compiler/processor decide *when* and *how* to speculate.

In this paper, however, we focus on the use of speculation *by programmers* as an algorithm design idiom to parallelize seemingly sequential code. We show, using real world examples, that value speculation can be used to extract thread level parallelism and develop speculatively parallel algorithms. This motivates our development of *language features* that enable programmers to conveniently express such speculatively parallel algorithms and to declaratively expose value speculation opportunities, without concerning themselves with mundane implementation details. We show how these constructs can be implemented efficiently, relying on static safety checkers to avoid expensive runtime mechanisms.

1.1 Speculatively Parallel Algorithms

Examples Of Inherently Sequential Algorithms. We start with some motivating examples. Lexical analysis is the problem of converting a sequence of characters into a sequence of tokens, encoded as a finite state machine (FSM). Given an FSM and a sequence of characters, lexical analysis starts in the initial state and processes characters one at a time. At every step, the analyzer transitions to the next state in the FSM based on the character just read. When the analyzer reaches a final state, it emits a token, resets its state to the initial state and continues processing the rest of the sequence.

As a second example, consider Huffman decoding. Huffman coding is a widely used data compression technique that uses a variable length binary encoding. Given a document, the encoder constructs a binary tree representing codes for all symbols in the document as well as a binary string representing the document. The decoder walks the binary tree starting at the root, matching bits in the compressed data with bits on the tree edges. When a leaf is reached, the symbol corresponding to the leaf is emitted and decoding continues from the root and the next available input bit.

A common feature of both these algorithms is the presence of a data dependence between successive iterations in the computation. In lexical analysis, the source of the dependence is the FSM state: each iteration uses the state value computed by the previous iteration. Such dependences prevent a straightforward parallelization of the algorithm. Recent studies show that such computations are not hard to find [3]. These computations are often key components of large applications such as browsers, databases, games, media

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.

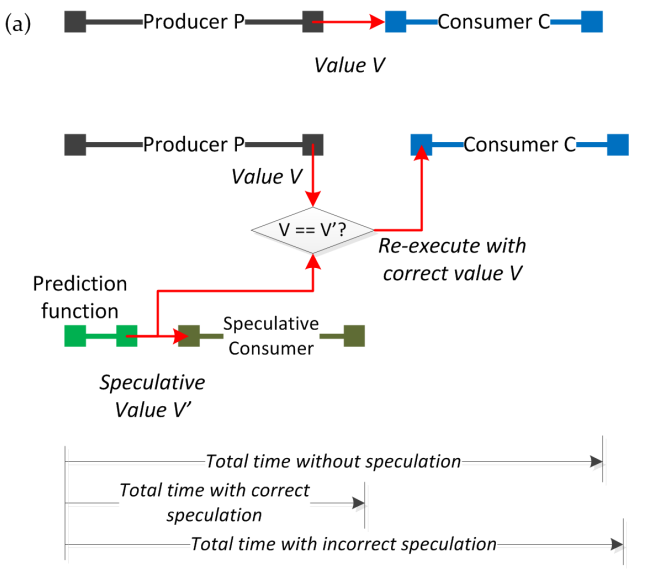

Figure 1. Value speculation

players and machine learning applications. If not parallelized, they inhibit performance of these applications as a whole.

Speculative Parallel Composition. We now illustrate how "sequential" algorithms such as lexical analysis can be parallelized [8] with the use of value prediction. The task of lexical analysis of an input sequence $c_1 \cdots c_n$ can be decomposed into two subtasks, the analysis of a segment $c_1 \cdots c_k$ and the analysis of a second segment $c_{k+1} \cdots c_n$. Unfortunately, the second subtask needs the FSM state computed by the first subtask as input and cannot be run in parallel with the first subtask.

This is an instance of a common scenario, depicted abstractly in Fig. 1. A computation P produces a value and a dependent computation C consumes the value. This data dependence between P and C serializes the execution of P and C. *Value speculation* involves *predicting* the value computed by P ahead of time and using the predicted value to execute C speculatively and concurrently with P. When P completes execution, we validate the prediction by checking the actual value computed by P and the predicted value. If the values match, we gain performance because the execution of P and C overlapped in time. However, if the values do not match, corrective action must be taken and C must be re-executed with the correct value to preserve semantics.

We can use this idea to obtain a speculatively parallel lexical analysis algorithm, as long as we have some means of predicting the FSM state at the end of the first subtask (i.e., after processing the first input segment $c_1 \cdots c_k$). Whenever the prediction is correct, the segments are processed in parallel. Whenever the prediction is incorrect, we effectively have a sequential algorithm (with some added overhead).

Speculative Parallel Iteration. It is straightforward to extend the previous idea to partition the input into k segments (e.g., where k is chosen based on the number of available processors) and process all k segments in parallel speculatively. In the worst case, it is possible for mis-predictions to produce a cascading effect, forcing corrective action and re-execution of all subsequent partitions. However, it is possible for subsequent predictions to be correct even when predictions for one or more of the previous partitions fail. Therefore, even in the presence of mispredictions, we can gain performance as long as the number of mispredictions is reasonably small.

The Prediction Function. The performance benefit of speculative parallelization depends critically on the accuracy of the value prediction mechanism. A key point to note here is that it is not necessary to use a single, generic, value prediction mechanism for exploiting speculative parallelism. We believe that, in general, better prediction accuracy can be achieved by using problem-specific prediction functions that incorporate domain knowledge and insights. We believe that speculative parallelization based on *programmer-specified prediction* functions is a very useful paradigm.

In the case of lexical analysis, the state of the lexical analyzer when it is about to process character c_{k+1} can be predicted quite accurately by lexically analyzing just a few characters preceding c_{k+1} [8]. This prediction scheme relies on the insight that in lexical analysis, some states of the FSM are frequently visited (such as the end of token states). Hence, it is likely that the prediction function, while lexically analyzing a few characters preceding c_{k+1}, reaches one of these states at the same character as the sequential analysis. If this happens, the prediction function will make the right guess. We were able to come up with accurate prediction functions for several other problems we looked at as well.

1.2 Language Support For Speculative Parallelism

Even after an algorithm designer has identified suitable opportunities for speculative parallelism and reasonably accurate prediction schemes, implementing such speculative algorithms while ensuring correctness and high performance, can be a non-trivial programming exercise. This is because in existing languages, speculation must be expressed using low level primitives such as threads and locks. Tasks such as spawning and scheduling speculative and non-speculative computations, checking for mis-predictions, and re-executing speculatively executed code if required, must all be performed manually. A more important concern is that of ensuring correctness, especially when the implementation uses mutable state. (a) What happens to the side-effects of speculatively executed code in the case of mis-prediction? (b) What happens if the producer and consumer access the same mutable state, and at least one of these accesses is a write?

In this paper, we propose new language constructs that enable programmers to declaratively express value speculation opportunities, as well as the prediction functions, exposing more parallelism in the program. The compiler and/or the runtime can take care of the tedious aspects mentioned above, ensuring performance and safety.

Specifically, we enhance a simple language that permits mutable state with two language constructs, *speculative application* and *speculative iteration* The construct "spec p g c" is used for the idea of speculative application illustrated earlier: it has three parameters, identifying a *producer*, a *consumer* that depends on the value produced by the producer and a prediction ("guess") function that predicts the value computed by the producer.

The construct "specfold f g m n" is used for the idea of speculative iteration illustrated earlier: it takes four parameters, a function f representing the loop body, a prediction function g, and the lower and upper bounds m and n for the iteration.

The semantics of these two constructs was informally described earlier. In the paper, we present a formal operational semantics for these constructs, which we call the speculative semantics. The notable aspect of this semantics is that any speculative subcomputation that used a mis-predicted value is not used, but its side-effects are not undone! We also define a *non-speculative* semantics for the language. This semantics ignores the "speculation hints" contained in the speculation constructs and, instead, executes them sequentially. The non-speculative semantics provides us with a way to interpret the program as a "specification". It serves to define the *desired behavior*.

Ideally, we would like to ensure that any speculative execution should be equivalent to some non-speculative execution. As a first step towards this goal, we formalize the notion of an equivalence between speculative and non-speculative executions. This formalization is based on the notion of a *dependence-preserving embedding* of a non-speculative execution into a speculative execution, which adapts the notion of conflict-serializability [23] to support speculation.

Several options exist for ensuring equivalence: (a) Use a pure functional language. In the absence of destructive updates, the speculative optimizations are inherently safe. (b) Use a static checker to determine whether the use of speculation in a given program context is safe. (c) Use a runtime mechanism for detecting conflicts and/or taking corrective action to ensure correctness. A rollback mechanism to undo the side-effects of a computation (as used in STMs) is an example of such a runtime solution. We can also use, e.g., a combination of static checking and runtime mechanisms.

In this paper, we explore the second option of using a static checker to ensure the safety of speculation in a language that supports mutable state. We define a safety condition called *rollback freedom* for a program. We show that in a program that satisfies rollback freedom, any speculative execution is equivalent to a non-speculative execution, even without any runtime mechanism for logging, conflict detection and rollback. In a rollback-free program, it suffices to simply re-execute speculatively executed consumers when mis-predictions occurs. Rollback freedom is weaker than conditions such as side-effect freedom and purity [19] that are traditionally used to ensure safety, as it allows certain forms of side-effects that are quite useful in practice.

We finally describe an inter-procedural, flow sensitive static analysis that checks for rollback freedom. Our analysis uses a flow-sensitive heap abstraction and an interval abstraction to precisely characterize parts of the heap that may be accessed by a given computation. This analysis ensures safe speculative parallelism at no runtime complexity and overheads.

We have implemented the speculation constructs as a C# library, along with the static checker for safety. We present an empirical evaluation of the efficacy of this approach to parallelization.

1.3 Contributions

In summary, our paper makes the following contributions:

- We propose new language constructs to enable programmers to express speculative parallelism in programs.
- We present a formal operational semantics for a language that supports mutable state as well as the proposed speculation constructs. The semantics formalizes an *unsafe but efficient* implementation that does not rollback side-effects of mis-predicted speculative computation. We also present a non-speculative semantics for the language, which we use to define the notion of a *correct speculative execution* as one that is equivalent to a non-speculative execution.
- We present a set of safety conditions (rollback freedom) for a program that guarantee that every speculative execution of the program is correct. We also present a static analysis to check that a program satisfies these safety conditions.
- We have implemented the speculation constructs as a C# library. We present experimental results showing how speculative versions of several real-world applications implemented using our library are able to exploit parallelism. We have also implemented an initial version of our static safety analysis.

2. A Language for Speculative Parallelism

In this section, we present the syntax and semantics of Speculate, a language with explicit support for speculative parallelism. The core language consists of call-by-value lambda calculus with dynamically allocatable mutable heap cells.

Syntax and Informal Semantics. Figure 2(a) presents the abstract syntax of Speculate. The language includes constants (integer constants as well as arithmetic operators), standard λ-calculus expression forms such as variables, functions, function application, sequential composition and conditionals. We support shared memory with constructs for memory allocation (new e), assignment ($e_1 := e_2$) and memory dereferences ($!e$).

The language includes fold expressions, similar to those found in functional languages, which model a simple form of iteration where each iteration depends on the value computed in the previous iteration. Informally, fold $f\ s\ l\ u$ represents the value $f\ u\ (\cdots f\ (l+1)\ (f\ l\ s)\cdots)$ computed by the loop:

```
result := s;
for i = l to u do
   result := f (i, result);
```

The first new construct in the language is speculative application (spec $p\ g\ c$), which we also refer to as speculative composition. This construct takes three parameters, a producer p and a predictor g, and a consumer g. Informally, speculative application computes the value of $c(p)$, but does this concurrently with the computation of p by predicting that the value of p will be g and computing $c(g)$ and taking corrective action if the prediction fails.

The second new construct is speculative fold (specfold $f\ g\ l\ u$), which we also refer to as speculative iteration). This computes a fold expression using speculative parallelism. This construct has a signature slightly different from that of fold: the second parameter g is a prediction function that takes an integer (in the range $l..u$) as a parameter. The value $g(l)$ is taken to be the initial value s used in a fold expression. For any $i > l$, $g(i)$ is the value predicted to be the value of `result` at the beginning of the i-th iteration of the loop described above (that evaluates a fold expression). Operationally, speculative fold executes all the iterations of the loop in parallel, using the predicted values, and takes corrective action when the prediction fails.

The language does not provide any other parallelism construct. However, the speculation constructs are expressive enough to encode common parallel programming patterns. E.g., the parallel evaluation construct "$e_1 \parallel e_2$" is a special case of speculative composition with no dependence between the producer and consumer. Similarly, a *do all* parallel loop is a special case of speculative iteration with no loop-carried dependence. (These special cases can be encoded using the unit value $()$ as the predicted value.)

Semantic Domains. The execution state of a program is represented by a configuration H, T consisting of a heap H and a set of threads T. A heap is a partial map from locations l to values v. A thread $t[e]$ consists of a thread-id t and an expression e that is being evaluated by the thread. We use the construct $t_1[e_1] \parallel \cdots \parallel t_k[e_k]$ to represent the set of threads $\{t_1[e_1], \cdots, t_k[e_k]\}$ mnemonically, but note that the ordering of these threads is immaterial.

The language of runtime expressions is richer than the language of expressions allowed by our language, as it includes some auxiliary constructs we use to define the semantics of the language. The new constructs include wait t and cancel t, where t is a thread-id. wait and cancel are used to synchronize between threads and preempt threads in speculative computations.

Operational Semantics. Fig. 2 presents *two* different operational semantics for Speculate. Rules C $\cup$ S define the actual semantics,

(a) Syntax and Semantic Domains

$x \in \text{Var}$ $\quad c \in \text{Const}$ $\quad t \in \text{Tid}$ $\quad l \in \text{Loc}$ $\quad v \in \text{Val}$ $\quad e \in \text{Exp}$ $\quad \mathsf{H} \in \text{Heap} = \text{Loc} \hookrightarrow \text{Val}$

$$
\begin{array}{lcl}
e & ::= & c \mid x \mid \lambda x.e \mid e_1\ e_2 \mid e_1; e_2 \mid \mathsf{if}\ e_1\ e_2\ e_3 \mid \mathsf{new}\ e \mid e_1 := e_2 \mid\ !e \mid \mathsf{fold}\ e_f\ e_i\ e_l\ e_u \mid \mathsf{spec}\ e_p\ e_g\ e_c \mid \mathsf{specfold}\ e_f\ e_g\ e_l\ e_u \mid r \\
r & ::= & \mathsf{wait}\ t \mid \mathsf{cancel}\ t \mid \mathsf{check}\ t_p\ t_g\ t_c\ e_c \mid \mathsf{auxfold}\ e_f\ e_g\ e_l\ e_u t_p \\
v & ::= & c \mid x \mid \lambda x.e \mid l \mid t \mid \mathsf{unit}
\end{array}
$$

(b) Evaluation Context

$$E ::= [\cdot] \mid E\ e \mid v\ E \mid E;\ e \mid \mathsf{if}\ E\ e_2\ e_3 \mid \mathsf{new}\ E \mid\ !E \mid E\ :=\ e \mid l\ :=\ E \mid \mathsf{spec}\ e_p\ e_g\ E \mid op_k\ v_1 \cdots v_{i-1}\ E\ e_{i+1} \cdots e_k$$
$$(\mathsf{fold}, \mathsf{specfold} \in op_4, \mathsf{check}, \mathsf{auxfold} \in op_5)$$

(c) Common Evaluation Rules (C)

[THREAD]
$$\frac{\mathsf{H}, e \to \mathsf{H}', e'}{\mathsf{H}, t[e] \parallel \mathsf{T} \Rightarrow \mathsf{H}', t[e'] \parallel \mathsf{T}}$$

[CONTEXT-1]
$$\frac{\mathsf{H}, e \to \mathsf{H}', e'}{\mathsf{H}, E[e] \to \mathsf{H}', E[e']}$$

[CONTEXT-2]
$$\frac{\mathsf{H}, t[e] \parallel \mathsf{T} \Rightarrow \mathsf{H}', t[e'] \parallel \mathsf{T}'}{\mathsf{H}, t[E[e]] \parallel \mathsf{T} \Rightarrow \mathsf{H}', t[E[e']] \parallel \mathsf{T}'}$$

[APPLY]
$$\frac{}{\mathsf{H}, (\lambda x.e)\ v \to \mathsf{H}, e[v/x]}$$

[SEQ]
$$\frac{}{\mathsf{H}, v; e \to \mathsf{H}, e}$$

[IF-ZERO]
$$\frac{}{\mathsf{H}, \mathsf{if}\ 0\ e_2\ e_3 \to \mathsf{H}, e_3}$$

[IF-NON-ZERO]
$$\frac{c \neq 0}{\mathsf{H}, \mathsf{if}\ c\ e_2\ e_3 \to \mathsf{H}, e_2}$$

[ALLOC]
$$\frac{l \notin Dom(\mathsf{H})}{\mathsf{H}, \mathsf{new}\ v \to \mathsf{H}[l \mapsto v], l}$$

[SET]
$$\frac{}{\mathsf{H}, l := v \to \mathsf{H}[l \mapsto v], v}$$

[GET]
$$\frac{}{\mathsf{H}, !l \to \mathsf{H}, \mathsf{H}(l)}$$

[FOLD-1]
$$\frac{v_l > v_u}{\mathsf{H}, \mathsf{fold}\ v_f\ v_{init}\ v_l\ v_u \to \mathsf{H}, v_{init}}$$

[FOLD-2]
$$\frac{v_l \leq v_u}{\mathsf{H}, \mathsf{fold}\ v_f\ v_{init}\ v_l\ v_u \to \mathsf{H}, \mathsf{fold}\ v_f\ (v_f\ v_l\ v_{init})\ (v_l + 1)\ v_u}$$

(d) Speculative Evaluation Rules (S)

[WAIT]
$$\frac{}{\mathsf{H}, t'[\mathsf{wait}\ t] \parallel t[v] \parallel \mathsf{T} \Rightarrow \mathsf{H}, t'[v] \parallel t[v] \parallel \mathsf{T}}$$

[CANCEL]
$$\frac{}{\mathsf{H}, t'[\mathsf{cancel}\ t] \parallel t[e] \parallel \mathsf{T} \Rightarrow \mathsf{H}, t'[()] \parallel \mathsf{T}}$$

[SPEC-APPLY]
$$\frac{t_p, t_g, t_c \text{ fresh in } \mathsf{T}}{\mathsf{H}, t[\mathsf{spec}\ e_p\ e_g\ v_c] \parallel \mathsf{T} \Rightarrow \mathsf{H}, t_p[e_p] \parallel t_g[e_g] \parallel t_c[v_c\ (\mathsf{wait}\ t_g)] \parallel t[\mathsf{check}\ t_p\ t_g\ t_c\ v_c] \parallel \mathsf{T}}$$

[CHECK]
$$\frac{x_p, x_g \text{ not free in } v_c}{\mathsf{H}, \mathsf{check}\ t_p\ t_g\ t_c\ v_c \Rightarrow \mathsf{H}, (\lambda x_p, x_g.\mathsf{if}(x_p =_{int} x_g)\ (\mathsf{wait}\ t_c)\ (\mathsf{cancel}\ t_c; v_c\ x_p))(\mathsf{wait}\ t_p)(\mathsf{wait}\ t_g)}$$

[SPEC-ITERATE-1]
$$\frac{v_l \leq v_u \text{ and } t_g, t_b \text{ fresh in } \mathsf{T}}{\mathsf{H}, t[\mathsf{specfold}\ v_f\ v_g\ v_l\ v_u] \parallel \mathsf{T} \Rightarrow \mathsf{H}, t[\mathsf{auxfold}\ v_f\ v_g\ (v_l + 1)\ v_u t_b] \parallel t_g[v_g\ v_l] \parallel t_b[v_f\ (\mathsf{wait}\ t_g) v_l] \parallel \mathsf{T}}$$

[SPEC-ITERATE-2]
$$\frac{v_l \leq v_u \text{ and } t_g, t_b, t_c \text{ fresh in } \mathsf{T}}{\mathsf{H}, t[\mathsf{auxfold}\ v_f\ v_g\ v_l\ v_u t_p] \parallel \mathsf{T} \Rightarrow \mathsf{H}, t[\mathsf{auxfold}\ v_f\ v_g\ (v_l + 1)\ v_u t_c] \parallel t_g[v_g\ v_l] \parallel t_b[v_f\ v_l\ (\mathsf{wait}\ t_g)] \parallel t_c[\mathsf{check}\ t_p\ t_g\ t_b\ (v_f v_l)] \parallel \mathsf{T}}$$

[SPEC-ITERATE-3]
$$\frac{v_l > v_u}{\mathsf{H}, t[\mathsf{auxfold}\ v_f\ v_g\ v_l\ v_u t_p] \parallel \mathsf{T} \Rightarrow \mathsf{H}, t[\mathsf{wait}\ t_p] \parallel \mathsf{T}}$$

(e) Non-Speculative Evaluation Rules For Speculative Constructs (N)

[NONSPEC-APPLY]
$$\frac{}{\mathsf{H}, \mathsf{spec}\ e_p\ e_g\ e_c \to \mathsf{H}, e_c\ e_p}$$

[NONSPEC-ITERATE]
$$\frac{v_l \leq v_u}{\mathsf{H}, \mathsf{specfold}\ v_f\ v_g\ v_l\ v_u \to \mathsf{fold}\ v_f\ (v_g\ v_l)\ v_l\ v_u}$$

Figure 2. Syntax and two operational semantics for Speculate. The expressions r are used by the runtime; they are not available in the source language. Rules C ∪ S define the *speculative semantics*. Rules C ∪ N define the *non-speculative semantics*.

.

which we call the *speculative semantics*. Rules C ∪ N define a semantics, the *non-speculative semantics*, that ignores the speculation hints and is used subsequently in our discussion of safety. (Semantics of primitive arithmetic operations such as + are assumed.)

Reductions that do not involve multiple threads are described using rules of the simpler form $\mathsf{H}, e \rightarrow \mathsf{H}', e'$, which indicates that an expression e, with an initial heap H, reduces to e', transforming the heap to H' in the process. Reductions involving multiple threads are described using rules of the form $\mathsf{H}, \mathsf{T} \Rightarrow \mathsf{H}', \mathsf{T}'$, which indicates that state $\mathsf{H}; \mathsf{T}$ reduces to state $\mathsf{H}'; \mathsf{T}'$. The rule THREAD relates the two forms of reduction rules. Furthermore, the rule also indicates that the scheduling is non-deterministic: any thread ready to perform a reduction may be chosen to execute at any point during execution. The evaluation of a program e starts in the initial state $\circ; main[e]$ consisting of an empty heap and a main thread evaluating e.

As usual, we use evaluation contexts (see Fig. 2(b)) to define the order of evaluation within an expression. An evaluation context is an expression with a hole, denoted $[\cdot]$, that identifies the next reduction to be performed. Given an evaluation context E, let $E[e']$ denote the expression obtained by replacing the hole in E by e'. For any expression e, there is at most one evaluation context E and a reducible sub-expression e' such that $e = E[e']$, and this identifies the sub-expression e' that is reduced next. E.g., the definition of evaluation context vE indicates that in an application $e_1 e_2$, the sub-expression e_2 is evaluated only after e_1 has been reduced to a value v. The CONTEXT rules show how the reduction is performed at the appropriate sub-expression chosen by the evaluation context.

Common expression forms such as function application, sequential composition, conditional expressions and memory allocation have standard semantics.

The WAIT rule defines semantics of wait, a synchronization primitive that allows threads in Speculate to communicate and co-ordinate. The evaluation of wait t in a thread t' *blocks* until t completes evaluation and the expression evaluates to the value computed by t. Observe that the rule permits multiple threads to successfully wait and retrieve the value computed by any thread t.

A thread t' may abort another existing thread t using the cancel t primitive, as indicated by the CANCEL rule. Note that the evaluation of a cancel blocks until the cancellation is successful. The non-deterministic scheduling may permit t to execute any number of reductions before the cancellation completes successfully. (Preemptive cancellation is hard to implement in practice. We will discuss some of the implications of this aspect later on.)

We now consider the semantics of spec. Rule NONSPEC-APPLY presents a straightforward non-speculative semantics for this construct that ignores the speculation hint. The rule SPEC-APPLY describes the speculative semantics of spec. The evaluation of spec e_p e_g v_c proceeds as follows: a producer thread t_p starts evaluating e_p and a predictor thread t_g starts evaluating e_g. A speculative consumer thread t_c waits for the predicted value and applies the consumer function v_c to the predicted value. The original thread t uses the auxiliary function check to coordinate the speculative computation, as defined by rule CHECK. The check function waits for both the producer and predictor to complete execution, and compares the value produced by these two threads. If they are the same, then the check function waits for the value computed by the speculative consumer. Otherwise, the speculative consumer is aborted, and the consumer function is applied to the correct input value.

Some key observations about the semantics of spec:

- We restrict the predicted values to be of type integer, and the equality operator used is the integer equality operator.
- We use three new threads solely to simplify the description. In an implementation, one new thread (for the predictor and speculative consumer) is sufficient, while the original thread can evaluate the producer and do the final check.
- The evaluation of spec blocks until the producer and the consumer complete evaluation.
- The speculative executions may behave differently from non-speculative executions for two reasons. (i) In the speculative semantics, side-effects of speculative consumers are not rolled back in the case of mispredictions. The consumers are simply re-evaluated. (ii) Producers and consumers may also race if they access the same shared state. As explained later, we rely on a static analysis to ensure "correct speculation".
- In the presence of speculation, scheduling policies (e.g., thread prioritization) and thread-cancellation policies (e.g., preemptive vs. non-preemptive) have implications for both termination and performance. A program that terminates under the non-speculative semantics may fail to terminate in the speculative semantics if a (mispredicted) speculative consumer is non-terminating and the scheduling policy is unfavorable. We discuss these issues in more detail in section 3.3.

We now consider the semantics of specfold. Rule NONSPEC-ITERATE presents the non-speculative semantics for this construct. The speculative semantics of specfold are a natural generalization of speculative application to loops with loop-carried dependences. As the rules SPEC-ITERATE-1 and SPEC-ITERATE-2 indicate, threads are created to execute all the iterations in parallel, using predicted values instead of the loop-carried dependence. The predictor v_g is a function that takes the iteration index i as a parameter and predicts the incoming value for the i-th iteration. The first iteration is non-speculative and is described by rules SPEC-ITERATE-1. Rule SPEC-ITERATE-2 describes the remaining iterations, which are speculative and utilize a checker thread t_c to validate the prediction and re-execute the loop body in case of mispredictions. Rule SPEC-ITERATE-3 indicates the final termination of the computation, which happens once the final loop iteration successfully completes.

Note that our semantics describes one specific validation scheme. It is possible to increase the speculative parallelism in the program with other validation schemes. E.g., assume that the speculative execution of the i-th iteration completes before the $i-1$-th iteration. The speculative output of the i-th iteration, if different from the value predicted by the $i+1$-th iteration, can be used to initiate further speculative executions of the $i+1$-the iteration. We discuss such a variation in Section 4.

3. Safe Speculation Without Rollback

One of the potential advantages of language support for speculation is that speculation constructs can be seen as hints used solely to improve performance, without affecting correctness. In general, a program's behavior under the speculative semantics may not be the same as its behavior under the non-speculative semantics due to side-effects. We would like to treat the non-speculative semantics as the intended behavior and statically check a speculative program to see if we can guarantee that its speculative behavior will be equivalent to its non-speculative behavior. We take the following three step approach to this goal:

1. We define a notion of *equivalence* between speculative and non-speculative executions. We define a speculative execution to be *correct* if it is equivalent to a non-speculative execution.
2. We define a safety criterion, *rollback freedom*, for the use of speculation constructs. We show that all speculative executions of a program that satisfies rollback freedom are correct.

3. We then present (in Section 5) a static analysis to check if a given speculative program satisfies rollback freedom.

Some of these notions become complicated if the predicted value consists of a (mutable) heap-allocated data structure. We focus on the essence of speculation by restricting attention to the case where the predicted value is of primitive type (e.g., integer).

3.1 Correctness Criterion

We now define two, related, notions of equivalence between speculative and non-speculative executions: a weaker notion that requires the final-states produced by the two executions to be equivalent and a stronger notion that also requires a correspondence between the (interesting) *transitions* of the two executions. The stronger notion is useful because it covers operational aspects and is also relevant for language extensions such as I/O with externally visible effects. (The second notion is a natural adaptation of the notion of conflict equivalence and conflict serializability [23] to permit speculation.) The following definitions are asymmetric because the speculative execution is allowed to have extraneous steps (transitions) as long as they don't affect the rest of the computation or the final state, and the heap is allowed to have extraneous heap cells (created by the extraneous steps) that are garbage (unreachable).

Preliminary Definitions. A *transition* $c \stackrel{a}{\Rightarrow} c'$ represents a single execution step, with a label a explained below. An *execution* is a sequence of transitions $c_0 \stackrel{a_1}{\Rightarrow} c_1 \cdots \stackrel{a_n}{\Rightarrow} c_n$, where c_0 is the initial configuration. Let $label(\tau)$ denote the label on a transition τ. We define the labels of transitions generated by the ALLOC, SET, and GET rules to be "ALLOC (ℓ, v)", "SET (ℓ, v)", and "GET (ℓ, v)", respectively, where ℓ and v represent the location and the value read/written respectively. We refer to these transitions as *interesting* transitions. We define the labels of all other transitions to be ϵ.

We say that a transition τ_1 is *data-dependent* on a transition τ_2 if τ_2 writes to a location ℓ that τ_1 reads and no transition in between τ_2 and τ_1 writes to ℓ. We say that a location ℓ in the final heap produced by an execution is data-dependent on a transition τ if τ writes to location ℓ and no transition after τ writes to ℓ.

Since different executions may use different location addresses, we compare different executions modulo a correspondence between the heap locations of the two executions. For this reason, we define a *correspondence mapping* from one execution π_1 to another execution π_2 to be a function μ that maps every interesting transition τ of π_1 to a distinct interesting transition $\mu(\tau)$ of π_2, and every heap location ℓ in π_1 to a distinct heap location $\mu(\ell)$ in π_2. We extend such a given mapping μ to map action labels and values: $\mu(a)$, where a is an action label or value, is obtained by replacing every occurrence of a location ℓ in a by $\mu(\ell)$.

Final-State Equivalence. The final configuration of a complete execution is of the form $\mathsf{H}, main[v] \parallel \mathsf{T}$, indicating that the main thread $main$ has completed evaluating the original program. We say that the *final state* of this execution is (H, v). Let π_n and π_s be a complete non-speculative and speculative execution with final states (H_n, v_n) and (H_s, v_s) respectively. We say that π_s is *final-state equivalent* to π_n, modulo a correspondence mapping μ, if

1. $v_s = \mu(v_n)$, and
2. for every location ℓ in H_n, $\mathsf{H}_s(\mu(\ell)) = \mu(\mathsf{H}_n(\ell))$.

Dependence Equivalence. We say that a correspondence mapping μ from an execution π_n to an execution π_s is a *dependence-preserving embedding* if

1. The mapping preserves transition labels: for every interesting transition τ in π_n, we have $label(\mu(\tau)) = \mu(label(\tau))$
2. The mapping preserves data-dependences between transitions: for every transition τ_1 and τ_2 in π_n, τ_1 is data-dependent on τ_2 iff $\mu(\tau_1)$ is data-dependent on $\mu(\tau_2)$.
3. The mapping preserves data-dependences of the final heap: for every ℓ in the final heap produced by π_n and transition τ in π_n, ℓ is data-dependent on τ iff $\mu(\ell)$ is data-dependent on $\mu(\tau)$.

We say that a speculative execution π_s is *dependence equivalent* to a non-speculative execution π_n if there exists a dependence-preserving embedding from π_n into π_s. Note that dependence equivalence is a stronger guarantee than final-state equivalence. We say that a speculative execution is *correct* if it is dependence equivalent to some non-speculative execution.

Note. The notion of dependence equivalence can be further strengthened by including other primitive reductions as interesting transitions and enriching transition labels to include the id of the thread performing the execution step, the redex being performed, and the evaluation-context to establish a much tighter correspondence between the executions. As this would significantly complicate the notation, we restrict ourselves to the simpler equivalence notion in this paper.

3.2 Rollback Freedom: A Safety Criterion

We now consider the problems that can prevent a speculative execution of $\mathsf{spec}\ e_p\ e_g\ e_c$ from satisfying the correctness criterion.

The first problem is the common one of data races: if the producer and the speculative consumer both access the same location, and at least one of these accesses is a write, the resulting execution may not be equivalent to the non-speculative execution (where the producer and consumer execute sequentially one after another). If no such conflicting accesses are possible between the producer and speculative consumer, then this problem does not arise.

The second problem, however, is specific to speculative execution without rollback. In the case of misprediction, heap updates performed by the speculative consumer can affect correctness. The key idea we exploit in our safety check for this problem is the following. If *the locations read by the consumer re-execution are disjoint from the locations written by the speculative consumer*, then the invalid speculative consumer execution does not affect the correctness of the subsequent consumer re-execution. Furthermore, if *the consumer re-execution is guaranteed to overwrite all locations written by the speculative consumer*, then the invalid speculative consumer execution will not affect the correctness of any subsequent computation (or the correctness of the final state).

E.g., consider an iterative loop in which the i-th iteration computes a value v_i and stores it in a (pre-existing) location ℓ_i. If the speculative execution of the i-th iteration, because of a misprediction, stores a wrong value v_i' in ℓ_i, the execution will still be correct as long as the re-execution of the i-th iteration stores the right value v_i in ℓ_i (and it does not read the pre-existing value of ℓ_i).

We now state the safety condition formally. For any expression e and heap H, we define $\mathcal{R}(e, \mathsf{H})$ to be the set of locations *in the initial heap* H that are read before they are written by the *non-speculative* evaluation of e with initial heap H. We define $\mathcal{W}(e, \mathsf{H})$ to be the set of locations *in the initial heap* H that are written during the *non-speculative* evaluation of e with initial heap H. (These definitions can be extended to accomodate the case where the evaluation of e in initial heap H is non-terminating.)

We say that $(\mathsf{spec}\ e_p\ e_g\ e_c, \mathsf{H})$ is *safe* if

$$
\begin{array}{l}
(a)\ \mathcal{W}(e_p, \mathsf{H}) \cap \mathcal{R}(e_c e_g, \mathsf{H}) = \emptyset, \\
(b)\ \mathcal{R}(e_p, \mathsf{H}) \cap \mathcal{W}(e_c e_g, \mathsf{H}) = \emptyset, \\
(c)\ \mathcal{W}(e_p, \mathsf{H}) \cap \mathcal{W}(e_c e_g, \mathsf{H}) = \emptyset, \\
(d)\ \mathcal{R}(e_c e_p, \mathsf{H}) \cap \mathcal{W}(e_c e_g, \mathsf{H}) = \emptyset, \\
(e)\ \mathcal{W}(e_c e_p, \mathsf{H}) \supseteq \mathcal{W}(e_c e_g, \mathsf{H}).
\end{array}
$$

Note that condition (b) is subsumed by condition (d), but we include it for expository reasons. We obtain an analogous safety condition for the construct specfold $v_f\ v_g\ v_l\ v_u$ by treating, for every $v_l \leq i < v_u$, the i-th iteration as the producer and the $i+1$-the iteration as the consumer.

We say that the pair (H, e_2) is reachable from e_1 if, in the non-speculative semantics, $\circ, e_1 \rightarrow^* \mathsf{H}, E[e_2]$. We say that a program e is *safe* if every $(\mathsf{H}, \text{spec}\ e_p\ e_g\ e_c)$ and $(\mathsf{H}, \text{specfold}\ v_f\ v_g\ v_l\ v_u)$ reachable from e is safe. We also say that a program satisfies *rollback freedom* if it is safe.

Note that rollback freedom is expressed in terms of the non-speculative semantics (which is a sequential semantics). This simplifies the analysis required to check for rollback freedom because the analysis does not have deal with interleaved (concurrent) executions produced by speculation.

Theorem 1. *If e satisfies rollback freedom, then every complete speculative execution of e is correct.*

Proof. We present a brief proof sketch here. We first prove a weaker result that a given speculative execution of a construct spec $e_p\ e_g\ e_c$ that satisfies the safety conditions is correct. Conditions (a) to (c) imply that there is no conflicting access between the execution steps of the producer and the speculative consumer. Hence, the execution steps of the producer and the speculative consumer commute with each other. Thus, the speculative execution is equivalent to the complete execution of the producer followed by the speculative consumer. If the prediction is correct, then this particular execution is correct. If the prediction is incorrect, then consider the speculative consumer and the re-execution of the consumer. Since the locations read by the consumer re-execution are disjoint from the locations written by the speculative consumer (condition (d)), the speculative consumer execution does not affect the correctness of the consumer re-execution. Since the consumer re-execution *overwrites* all locations written by speculative consumer (condition (e)), the writes by the speculative consumer do not affect the correctness of any subsequent computation (or the correctness of the final state). It follows that the speculative execution is correct.

However, our safety criterion checks for conditions (a)-(e) only using non-speculative executions. This may seem unintuitive, but is sufficient. We can prove, by induction, that speculative executions must also satisfy the same conditions and be equivalent to non-speculative executions. Details omitted. □

3.3 Termination Guarantees

Ideally, the speculation scheme should provide a termination guarantee: if a non-speculative execution terminates, the speculative execution should also terminate. With minor modifications, the speculative semantics does provide this guarantee.

First, note that the speculation-validation step waits for both producer and predictor to complete execution. If the predictor is non-terminating, this is a problem as the non-speculative execution does not invoke the predictor. This is easy to fix: if the producer completes execution before the predictor, there is no point in continuing with the speculation; we can abort the predictor and speculative consumer and execute the consumer with the correct input.

Second, when the validation step detects mis-prediction, it attempts to cancel the speculative consumer. If the speculative consumer is non-terminating (e.g., because it is processing the wrong input), then we need to ensure that the cancellation step eventually takes place to guarantee termination. Either a fair scheduler or a prioritized scheduler that prioritizes non-speculative threads higher than speculative threads suffices to guarantee termination.

```
1 public class Speculation {
2   public static void Apply<T>(
3     Func(T) producer,
4     Func(T) predictor,
5     void Action(T) consumer)
6
7   public enum ValidationMode { Seq, Par };
8
9   public static void Iterate<T>(
10    int low, int high,
11    Func(int, T, T) loopBody ,
12    Func(int, T) predictor,
13    ValidationType val /* optional */)
14
15  public static void Iterate<T, U>(
16    int low, int high,
17    Func(U) initializer,
18    Func(int, U, T) loopBody,
19    Func(int, T) predictor,
20    Action(int, U) finalizer,
21    ValidationMode val)
22 }
```

Figure 3. An API for speculative parallelism. Func is a C# generic type for delegates that take zero or more arguments and return a value. Action is a C# type for delegates that do not return a value.

4. A Speculation Library

In this section we describe how we realized the speculation constructs presented earlier as a C# library.

Programming model. Our API (see Fig. 3) consists of a method Speculation.Apply that implements speculative application, and a set of methods Speculation.Iterate that support speculative iteration. The classes and methods in the API are generic with types representing the type of the value(s) being speculated. The API requires that all computations be specified as C# delegates.

Apply provides the same interface and semantics as spec. Our API supports several variations of specfold. There are several methods that support speculative iteration. In its simplest form, the Speculation.Iterate method requires four arguments, the *low* and *high* loop indices, and delegates representing the loop body and prediction function. Both delegates are parameterized with the loop index; this permits the prediction delegate to make iteration specific predictions.

The call to Speculation.Iterate takes an optional validation mode parameter. This parameter can be used to specify the validation mode an implementation should use to validate speculative iterations. We currently support two validation modes. In the *sequential* validation mode, speculatively executed iterations are validated (and re-executed) *in sequence*, starting from the low to the high loop index. Sequential validation matches the semantics described in section 2. The *parallel* mode does validation speculatively. In this mode, a speculatively executed iteration i is validated speculatively as soon as the previous iteration $i-1$ completes, even if iteration $i-1$ itself has not been validated.

We find that a common usage scenario for speculative iteration is one where each iteration allocates local objects, computes results in the objects and then publishes the results to global state. We provide a variant of speculative iteration that explicitly supports this scenario. This variant requires the user to provide a local initialization and a local finalization delegate. The local initializer is a function delegate that returns the initial state of the local data for each iteration and the local finalizer is a delegate that performs a final action on the local state of each iteration.

```
1 public class LexicalAnalysis {
2  private TokenCollection[] tc =
3   new TokenCollection[NUM_TASKS];
4
5  public Token[] SpeculativeLex(char[] input) {
6
7   int fragmentSize =
8    input.Length() / NUM_TASKS;
9
10  Speculation.Iterate<State>(0, NUM_TASKS,
11   (i, state) => /*loop body */ {
12    State finalState;
13    tc[i] = SequentialLex(input,
14     i * fragmentSize,
15     (i + 1) * fragmentSize,
16     state, out finalState);
17    return finalState;
18   },
19   (i) =>  /* prediction function */ {
20    if (i == 0)
21     return START_STATE;
22    else {
23     State predictedState;
24     SequentialLex(input,
25      i * fragmentSize - 10 /* overlap */,
26      i * fragmentSize,
27      START_STATE, out predictState);
28     return predictedState;
29    }
30   });
31
32   /* assimilate tokens in tc */
33   ...
34 }
35
36 private TokenCollection SequentialLex(
37  char[] input,
38  int from, int to,
39  State inputState,
40  out State finalState)
41 {
42  LatexLexer lexer =
43   new LatexLexer(input, from, to);
44  return lexer.Lex(inputState, out finalState);
45 }
46 }
```

Figure 4. An implementation of speculative lexical analysis using the Speculation API.

Figure 4 shows an implementation of speculative lexical analysis using our API. The implementation lifts an existing sequential lexcial analysis routine (SequentialLex) to a speculatively parallel version. The usage closely resembles the usage of Parallel.For (a C# API for parallel loops), with the only difference being the presence of a prediction function.

Implementation. We have implemented this API using the .NET Task Parallel Library [1]. Some aspects of our implementation not covered by our earlier theoretical description include the following, and extending the theory to handle these aspects is future work.

- The implementation ensures that only validated consumers/iterations throw exceptions. The library hides all exceptions from code that was speculatively executed with the wrong values. We also try and provide *sequential* exception semantics. Under these semantics, a call to Speculation.Iterate throws the exception corresponding the first valid iteration, irrespective of the order in which the runtime executes iterations.
- Our implementation is based on the .NET Task Parallel Library, which supports a co-operative cancellation model (as opposed to preemptive cancellations). Consequently, our implementation does not preserve termination. We rely on the user to inject cancellation checks to ensure that speculatively executed codes terminates even with incorrect values. These checks can be automatically injected in user code, as is done in STMs.
- We use the abstract `Equals` method of the generic type T to check for equality. This allows a user of the library to supply their own abstract equality function. This may be useful in scenarios where strict equality is not required and a prediction can be considered correct as long as it satisfies a more relaxed equality check. If the programmer specifies her own equality function, it is her responsibility to ensure that the equality function does not affect correctness. Our safety correctness results currently apply only when strict equality is used.

5. Static Analysis For Safety

In this section, we describe a static analysis that checks whether a C# program that uses our API satisfies rollback freedom conditions described in Section 3.2.

The core analysis is combined pointer and escape analysis (similar to the purity and side-effect analysis described in [19]). The goal of the analysis is to compute precise over-approximations of the $\mathcal{R}$ and $\mathcal{W}$ sets defined in Section 3.2. The analysis is an interprocedural flow and field sensitive analysis that computes an abstract model of the heap at every program point. The model represents the heap accessed by the given method when the program point is reached. The heap is modeled as a graph, with nodes representing abstract heap locations, and edges representing heap references. All heap locations that share the same allocation site are represented by the same abstract location.

This analysis is specifically designed to distinguish parts of the heap allocated for internal use by a method during its execution from parts of the heap exist before the call to the method such as parameters, objects loaded through parameters and static objects. This is a critical distinction since speculatively executed code may create new objects in the heap, but we do not have to worry about writes to such objects as long as they do not escape to the caller. (The definitions of $\mathcal{R}$ and $\mathcal{W}$ in Section 3.2 reflect this.)

For example, consider the speculative lexical analysis implementation in Figure 4. Figure 5 shows a simplified heap graph obtained after analyzing SequentialLex. The method requires three parameters, an input character array input, an object initialState representing the state in which lexical analysis should start, and finalState, which is a reference to the final state of the FSM. Edges are annotated with names of fields (e.g. yy_input, yy_initial, tokens and yy_final are fields in the class LatexLexer; these fields are accessed inside the constructor and the method Lex). Edges from array objects are annotated with intervals representing a set of array indices (e.g. [from, to]). As in [19], dashed nodes represent pre-existing objects (such as parameter objects) and dashed edges represent reads from escaping objects. This method creates an object of type LatexLexer for processing the input string. Since this object does not escape to the caller, any writes to fields of this object do not affect safety.

The analysis is modular analysis that analyzes each method independent of its calling contexts. The analysis assumes that formal parameters do not alias and computes a summary that describes the set of abstract locations in the pre-existing heap that are read/written during the method execution. The inter-procedural component of the pointer analysis utilizes aliasing information available at the call site to merge the summary of the callee into the caller.

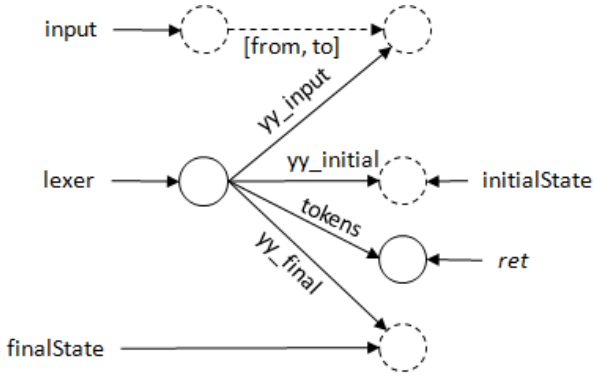

Figure 5. Model representing the heap accessed by SequentialLex

Modeling array accesses. We extend the core analysis to model array accesses and accesses to static objects more precisely. This analysis identifies locations in an array that may be read/written and describes them using symbolic intervals, whose lower bounds and upper bounds are linear expressions over program variables (in particular, the procedure parameters). The inter-procedural component of the range analysis re-interprets intervals in terms of parameters of the caller. This is an important extension because some of the common scenarios where speculative parallelism is likely to be used (including the benchmark programs in this paper) involve arrays and other indexed collections. For example, in SequentialLex, the interval [from, to] describes the set of indices of the array input read by the method. Similarly, the internal [i, i] describes the set of indices of the array tc written to by the loop body delegate.

Computing must information. The analysis defined above computes an over approximation (*may* information) of $\mathcal{R}$ and $\mathcal{W}$ sets. Using over-approximate read and write sets is sound in all cases except the LHS of condition (e). In condition (e), ensuring soundness requires that we use an under-approximation of the write set $\mathcal{W}(e_c e_g)$. We achieve this using the following two extensions to the core pointer and range analysis. We extend the pointer analysis to maintain a single bit with every abstract heap node, which represents whether the node models a single (concrete) object or summary objects. We say that a reference is must-written in a method if the object containing the reference is a single object *and* if the reference has been written to on all paths in the method. We also extend the range analysis to compute an under-approximate interval of values for every variable. The must write set at any program point consists of the writes to all single objects in the heap graph.

6. Experiments

In this section, we present an empirical evaluation of the proposed approach, using real-world applications we implemented using the library described in Section 4.

Benchmark programs. We implemented three representative benchmark programs using the speculation library. These include lexical analyzers for C, HTML, Java and Latex, Huffman decoding algorithm, and a dynamic programming algorithm for finding the maximal weighted independent set (MWIS) of a given path graph. We implemented prediction functions for all the algorithms. The prediction functions use a specified number of elements preceding a given segment (referred to as the *overlap*) to predict the required value at the beginning of the segment (similar to the lexical analysis prediction function described in Section 1).

Our implementation of MWIS has two speculatively parallel phases. In the first phase, we identify the elements that will be a part of the MWIS using dynamic programming. This phase operates on segments of the path graph in parallel. The prediction function predicts whether the pair of nodes immediately preceding the current segment will be a part of MWIS. The second phase walks backwards along the path graph and emits the MWIS.

We note that the speculation library itself took about 4 man months to design, develop and validate. Subsequently, each speculative algorithm took only about a day's effort to implement starting from a pre-existing sequential implementation.

Setup. We conducted all our experiments on a 2.66 Ghz, Intel Core 2 Duo server with 4 GB RAM, running Windows Server 2008. We measured time using the `DateTime` class in C#. All the times we report are averages over 10 executions measured using the ptime utility. We used the .NET PeakVirtualMemorySize64 API to measure peak memory consumption.

We also identified input datasets of different flavors for each benchmark to study the effect of the input dataset on the observations. We generated lexical analyzers for 4 different languages (C, HTML, Java and Latex). We created three representative data sets for Huffman decoding, *media* (mp3 files), *rawdata* (trace data from an Intel profiling tool) and *text* (a collection of books). For MWIS, we generated path graphs with uniformly distributed numbers between intervals 0 to 50 and 0 to 5000.

Speedup and Scalability. First, we measured the speedup of the parallel implementations for each dataset, while varying the number of threads and the amount of overlap. Figure 6 shows the results of this experiment. For each data set, we plot two sets of speedup, a *max* speedup (on the left) obtained with an overlap that is large enough to eliminate mis-predictions (this represents our best case), and a *min* speedup (on the right) obtained with a very small overlap. We make several observations from this data.

- Several of the benchmark/dataset combinations scale almost linearly with the number of threads when the overlap is large enough to make accurate predictions. The actual speedups depend on the datasets. For example, the latex lexical analysis implementation achieves a speedup of 4 with 4 threads.
- The speedups obtained with smaller overlap vary from no speedup (Huffman decoding/media dataset) to near linear speedup (Java lexical analysis). In the case of the Java lexical analyzer, we find that predictions are often accurate even with a small overlap.
- In the lexical analysis implementations, we observe a correlation between the speedup and the size of the finite state machine. The lexical analyzer for C has the largest FSM whereas the one for Latex has the smallest FSM. This suggests that the speedup may be influenced by the memory subsystem, which is stressed more when finite state machines are larger.
- The MWIS benchmark does not scale as well as the other benchmarks even with a large overlap . We find that this benchmark is memory bound and performance is limited by the memory subsystem.
- There are a small number of cases where speedup is marginally less than 1. This shows that the runtime overheads introduced by our library are negligible.

Prediction Accuracy. The scalability of a speculative algorithm largely depends on the accuracy of the prediction function. If a prediction function rarely mis-predicts, we expect a speculative implementation to behave like a parallel implementation. On the other hand, if a prediction function mis-predicts often, the implementation reduces to a sequential implementation, plus the library's own overheads.

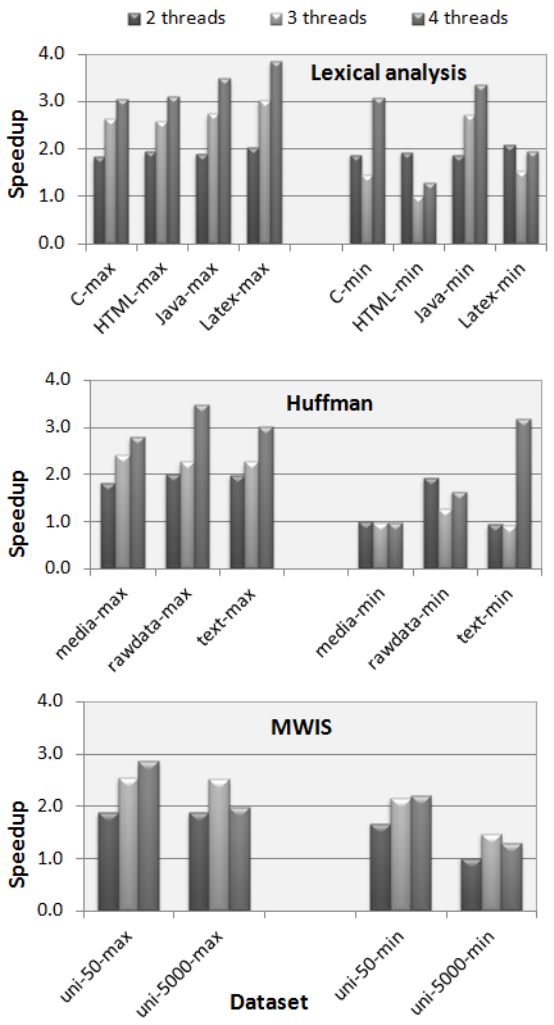

Figure 6. Variation in scalability of the three benchmark programs with number of threads, data sets and prediction quality.

We measured the accuracy of prediction using different overlap lengths for all our datasets. In each case, we made 32 predictions using the prediction function at equally separated points in the input data, and measured the prediction accuracy. As shown in Figure 7, we find that the prediction accuracy increases as the overlap increases. Except for the HTML Lexer and MWIS (uni-5000), a 100% prediction accuracy was achievable with reasonable overlap sizes. We also repeated this experiment increasingly larger number of predictions (upto 500,000 predictions) and found that the prediction accuracy remains more or less the same. These experiments highlights the potential for *domain specific* prediction functions to be very accurate and the value of *programmer specified* prediction functions.

Validation modes. As described in Section 4, we support two validation modes in the Speculation.Iterate API, a sequential validation mode (*seq*) and a parallel/optimistic validation mode (*par*). Figure 8 shows the variation in speedup with the choice of the validation mode and the accuracy of prediction for one dataset in each of the benchmark programs. We present two sets of speedups, a *min* speedup obtained with a small overlap, and a *max* speedup obtained with a large overlap that eliminates mis-predictions.

Benchmark	*Overlap*	*Prediction Accuracy %*		
		HTML	*Java*	*Latex*
Lexical analysis	16	28%	90%	62%
	64	41%	100%	100%
	256	50%	100%	100%
		media	*rawdata*	*text*
Huffman	2	38%	72%	66%
	4	47%	81%	75%
	8	72%	100%	91%
	16	91%	100%	100%
	64	100%	100%	100%
		uni-50	*uni-5000*	
MWIS	8	81%	38%	
	16	97%	38%	
	32	100%	38%	

Figure 7. Variations in prediction accuracy for various data-sets.

Both validation modes performed equally well in many cases, but sequential validation seems to do better when the number of threads is increased to 4 and we use a good predictor. This is slightly counter intuitive since we would expect parallel validation to perform better in exactly such scenarios. Our investigation suggests that the overheads of creating a larger number of validation tasks outweighs the benefits of parallel validation as the number of speculative threads increase. We believe this is an artifact of the task library we are using and we are working on optimizing this aspect of our implementation. We propose to further investigate the relative performance of the two options, e.g., in machines with more than 4 cores.

Dataset size. We also conducted experiments to study the effect of the data set size on scalability. For example, in the case of Huffman decoding, we varied the size of input data from 10 MB to 50 MB. Our experiments suggest that the speedups do not vary significantly within the data size intervals we chose. On an average, we see a small drop in speedup with the increase in data set size. We studied this behavior closely using performance counters and we find that the memory subsystem plays an increasingly important role as data sets sizes increase, which limits speedup. We omit the details of this experiment due to space constraints.

Static analysis. We have built a prototype implementation of the analysis for checking rollback freedom (described in Section 5) and used it to verify correctness of our benchmark programs. Currently, our implementation does not analyze methods in the .NET base class library. We manually provided summaries for BCL methods called from within our programs. Table 9 shows some of the characteristics of the benchmarks programs and the time taken by our analysis to verify correctness. All of our benchmarks are moderately sized with a reasonably small number of methods. In all three case, the time taken to verify rollback freedom was under 30 seconds (unfortunately, with relatively high peak memory usage). We believe that the scalability of our implementation can be improved further. We leave a more detailed analysis of the precision-performance trade-off for future work.

7. Related Work

Speculatively Parallel Algorithms. Several speculatively parallel algorithms have been designed in recent years, for use in specific domains. Jones et al. [8] devised an overlap-based speculative lexical analysis for parallelizing a web browser's front-end. Luchaup et al. [14] speculatively identify attacks within an intrusion detection system using *hot* state prediction within a pattern matching FSM. Klein et al. [10] designed a parallel JPEG decoder by speculative identification of synchronization points within Huffman codes.

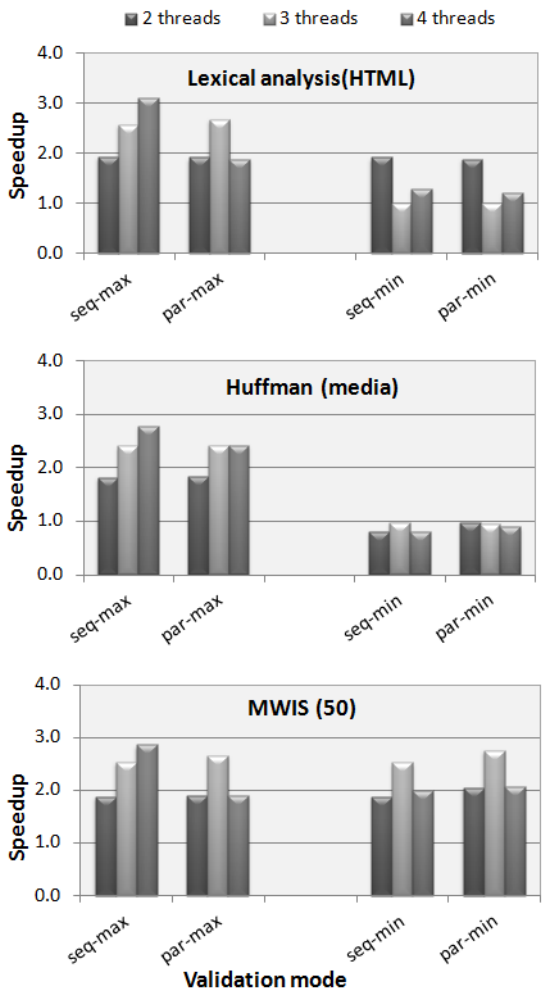

Figure 8. Variation in scalability of benchmarks with the type of speculation validation sequential or parallel.

Benchmark	*LOC*	*# methods*	*Time (sec.)*	*Memory Usage (MB)*
Lexical Analysis (Java)	493	76	23.62	50
Huffman Decoding	578	83	21.25	66
MWIS	412	44	29.89	64

Figure 9. Characteristics of benchmark programs and time & memory consumed to verify rollback freedom.

Speculative parsing [9] and speculative simulated annealing [25] are other examples of speculative algorithms. All these algorithms can be easily expressed using our speculation constructs.

Language Constructs and Implementation Mechanisms For Safe Parallelism. Futures is a well-known language construct for initiating the concurrent computation of a value for later use. Welc *et al.* [24] formalized the concept of safe futures for Java as one that guaranteed behavior equivalent to a sequential implementation. They utilize a runtime mechanism to ensure sequential semantics in the presence of memory conflicts. Software transactions [4, 7, 2, 22] are a well-known construct for safe parallelism, with a well-studied formal semantics [15, 2]. Implementations of software transactions typically rely on optimistic concurrency and runtime techniques to detect conflicts and use rollback for corrective action. The Galois system provides set iterators for specifying optimistic parallelism and uses high level commutativity checks for runtime validation [11]. The constructs we propose meet a need not addressed by the above constructs: programmable value speculation. Further, we use static analysis techniques to ensure safety and avoid the need for runtime techniques. The use of runtime techniques to detect conflicts and/or take corrective action in an implementation of our speculation constructs (e.g., when static verification is not possible) is an interesting problem worth exploring.

Software BOP [6] is a system that speculatively executes code regions annotated as "possibly parallel" by the programmer. Value checking is used as an optimization for speculation validation in certain restricted cases, while address-based checking is used in the general case. Speculatively parallel algorithms which can be expressed using our constructs cannot be expressed in Software BOP as there is no support for specifying *ordered* "possibly parallel" regions. This ordering is important in deciding the regions to re-execute on mis-speculation detection. Software BOP supports rollback at runtime by leveraging page-based protection mechanisms, while we rely on static analysis to prove rollback freedom.

Compiler and hardware driven value speculation. Computer architects and compiler writers have proposed the use of value speculation for extracting parallelism at both the instruction level (ILP) and thread-level (TLP). For example, value speculation has been used to to drive ILP optimizations such as register value re-use [21] and load/store reordering [13].

Hardware-assisted TLS systems use value prediction for masking communication latency between threads [20] and predicting silent stores [5] and return values [16]. Mitosis is a system that uses slice based value prediction for initializing speculative thread state [17]. SPICE [18] uses selective value prediction to convert loops into data parallel form.

In this paper, we focus on user-programmable speculation for TLP. Our constructs are complementary to low-level value speculation used in compilers and the hardware. Our constructs enable programmers to specify custom prediction functions that exploit domain knowledge, which is often critical for prediction accuracy.

8. Future Work

We believe that speculative parallelism is an important algorithm design idiom for improving performance of algorithms that are commonly perceived as sequential. The language extensions we propose simplify writing speculatively parallel programs. However, these are initial designs and there are several interesting avenues for future work.

We defined the semantics of our speculative constructs in a simple language. Formally defining the semantics of these constructs in a richer language that supports features such as exceptions, synchronization primitives is an interesting problem. Ideally, we would like to guarantee non-speculative exception semantics for the speculative constructs. However, providing these guarantees statically or even with low runtime overheads appears challenging. For example, consider the scenario when a producer or a speculative iteration throws an exception. In such a scenario, the consumer or any subsequent speculative iterations should not appear to have been executed at all.

We would also like to weaken the conditions we use check for rollback freedom to permit common programming idioms such as caching. By weakening these conditions and statically verifying larger fragments of code as rollback free, we may be able to significantly reduce the burden of runtime mechanisms that ensure safety such as transactional memory.

Finally, we believe that expressing speculative algorithms using constructs may permit more efficient implementations of these algorithms. For example, it may be possible to exploit knowledge about whether or not a task is speculative (and the *degree* of speculation) to improve task scheduling.

Acknowledgments

We would like to acknowledge Tim Harris for his inputs during the initial stages of this work. We also thank Ashish Agarwal for his contributions to this paper.

References

[1] What's new in beta 2 for the task parallel library. In *blogs.msdn.com*, 2009.

[2] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *Proc. of POPL*, pages 63–74, 2008.

[3] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Communications of ACM*, 2009.

[4] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The atomos transactional programming language. *SIGPLAN Not.*, 41(6):1–13, 2006.

[5] Marcelo Cintra and Josep Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *Proc. of HPCA*, page 43, Washington, DC, USA, 2002. IEEE Computer Society.

[6] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. Software behavior oriented parallelization. In *Proc. of PLDI*, pages 223–234, 2007.

[7] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proc. of PPoPP*, pages 48–60, New York, NY, USA, 2005. ACM.

[8] Christopher Grant Jones, Rose Liu, Leo Meyerovich, Krste Asanovic, and Rastislav Bodik. Parallelizing the web browser. In *Proc. of HOTPAR*, 2009.

[9] Blake Kaplan. Speculative parsing patch. In *bugzilla.mozilla.org*, 2009.

[10] Shmuel Tomi Klein and Yair Wiseman. Parallel huffman decoding with applications to jpeg files. *Journal of Computing*, 46(5):487–497, 2003.

[11] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI*, pages 211–222, 2007.

[12] Butler W. Lampson. Lazy and speculative execution in computer systems. In *Proc. of ICFP*, pages 1–2, New York, NY, USA, 2008. ACM.

[13] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *Proc. of ASPLOS*, pages 138–147, 1996.

[14] Daniel Luchaup, Randy Smith, Cristian Estan, and Somesh Jha. Multi-byte regular expression matching with speculation. In *Proc. of RAID*, pages 284–303, 2009.

[15] Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. In *Proc. of POPL*, pages 51–62, 2008.

[16] Jeffrey T. Oplinger, David L. Heine, and Monica S. Lam. In search of speculative thread-level parallelism. In *Proc. of PACT*, page 303, Washington, DC, USA, 1999. IEEE Computer Society.

[17] Carlos García Qui nones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. *SIGPLAN Not.*, 40(6):269–279, 2005.

[18] Easwaran Raman, Neil Vachharajani, Ram Rangan, and David I. August. Spice: speculative parallel iteration chunk execution. In *Proc. of CGO*, pages 175–184, 2008.

[19] Alexandru Salcianu and Martin C. Rinard. Purity and side effect analysis for java programs. In *Proc. of VMCAI*, pages 199–215, 2005.

[20] J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. Improving value communication for thread-level speculation. In *Proc. of HPCA*, page 65, Washington, DC, USA, 2002. IEEE Computer Society.

[21] Dean M. Tullsen and John S. Seng. Storageless value prediction using prior register values. In *Proc. of ISCA*, pages 270–279, 1999.

[22] Christoph von Praun, Luis Ceze, and Calin Cascaval. Implicit parallelism with ordered transactions. In *Proc. of PPoPP*, pages 79–89, 2007.

[23] G. Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[24] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for java. In *Proc. of OOPSLA*, pages 439–453, New York, NY, USA, 2005. ACM.

[25] E. E. Witte, R. D. Chamberlain, and M. A. Franklin. Parallel simulated annealing using speculative computation. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):483–494, 1991.