Access Annotation for Safe Speculative Parallelization: Semantics and Support

Chen Ding
The University of Rochester
Computer Science Department
Rochester, NY 14627

Technical Report TR-966
March 2011

Abstract

The safety of speculative parallelization depends on monitoring all program access to shared data. Automatic solutions use either program instrumentation, which can be costly, or hardware-based triggering, which incurs false sharing. In addition, not all access requires monitoring. It is worth considering a manual approach in which programmers insert access annotations to reduce the cost and increase the precision of program monitoring.

This report first presents an execution model and its interface for access annotation. The semantics of an annotated program is defined by the output of a (sequential) canonical execution. The report then describes a quadratic-time checker that can verify the safety of annotations, that is, whether a speculative execution always produces the canonical output. The report demonstrates the usability of the annotation interface by safely parallelizing a number of code fragments that have uncertain parallelism.

The research is supported by the National Science Foundation (Contract No. CCF-0963759, CNS-0834566, CNS-0720796) and IBM CAS Faculty Fellowships. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding organizations.
1 Introduction

Speculative parallelization divides a sequential program into possibly parallel tasks—for
example as safe futures (Welc et al., 2005), ordered transactions (von Praun et al., 2007) or
PPRs (Ding et al., 2007)—and uses a run-time system to monitor for conflicts and revert to
sequential execution if needed. Speculation is useful in addressing the problems of uncertain
parallelism due to either implementation or program input.

The main benefit of speculative parallelization is safety—the speculative execution al-
ways produces the same output as the sequential execution. The safety guarantee, however,
depends on adequate monitoring of data access in both speculative and non-speculative
tasks. In a software implementation, monitoring is typically done in two ways. The first
is program instrumentation. It is often too costly to instrument every data reference. A
compiler may remove much unnecessary instrumentation but only for applications amenable
to compiler analysis (Rauchwerger and Padua, 1995; Tian et al., 2010, 2008). The other
solution is hardware support to trigger a call to the monitoring code at a data access, either
at the granularity of a page (Ding et al., 2007; Berger et al., 2009) or a cache block (Shrir-
man et al., 2007, 2008). False sharing becomes a problem especially at page granularity. In
these solutions, program monitoring is a function of the implementation, hidden from the
programmer.

The problem of cost and precision may be better solved by allowing programmer control.
In this report, we define an interface for access annotation and integrate it into BOP, a
suggestion interface for parallelizing sequential code (Ding et al., 2007; Ding, 2011). We
call the combined interface annotated BOP or A-BOP in short. The base BOP hints divide
d a sequential execution into a serial part, called the skeleton, and a series of possibly parallel
(PPR) tasks. A-BOP adds a set of primitives for access annotation for use by the skeleton
and PPR tasks to define their visible behavior. Only the annotated accesses in a speculation
task are monitored, and only these changes are copied out and merged into the rest of the
execution.

The annotation interface gives a programmer direct control over the two major
factors in performance: the cost and precision of access monitoring. As access monitoring
becomes programmable, however, it becomes part of program semantics. Different A-BOP
annotations effectively create different programs. The goal of this work is to give well-
deﬁned semantics to the annotation interface. We describe two techniques:

- A-BOP canonical execution deﬁnes the semantics of an annotated program using a
  sequential execution of PPR tasks.

- A-BOP race checking veriﬁes that all parallel executions are guaranteed to produce
  the canonical semantics. Race checking takes time quadratic to the number of PPR
tasks.

Parallelization using A-BOP takes the following steps. A programmer starts from a
sequential program and first inserts parallelization hints and access annotations. The pro-
grammer then veriﬁes the correctness of access annotation through testing. For a test input,
she runs and debugs the (sequential) canonical execution to obtain the correct output. Then she runs the race checker to ensure that all parallel executions produce the correct output. As new annotations are added for later tests, they preserve the correctness of previous annotations inserted for earlier tests. Systematic testing can be used to gain a high degree of certainty that the parallelized program has the right semantics and is safe.

We expect the A-BOP model to have a number of software engineering benefits. The sequential semantics of the canonical execution means that there is no need for parallel debugging. It makes the parallelized code easier to write, understand, and compose with other sequential code or automatically parallelized code. A program may be fully annotated so it no longer needs speculation. Finally, an A-BOP program can run on a cluster of machines without shared memory.

2 The Execution Model

The execution model specifies what tasks are parallel, how they share data, and when access annotation is needed. Since not all tasks are run in parallel, not all shared-data access requires access annotation.

2.1 Parallel Programming by Hints

BOP provides the following three hints for a user to suggest and manage parallelism in a sequential program. The hints may be inserted incorrectly but will never change the program output.

- **bop.ppr** brackets a code block as a possibly parallel region to mean that its execution is likely parallel with the execution of the subsequent code.

- **bop.ordered** brackets a block of code inside a PPR region as an ordered region to suggest that PPR tasks should execute the ordered region sequentially one task at a time and in the order of the sequential program.

- **abort.spec**, if executed by a speculative PPR task, causes the task to abort; otherwise, it has no effect.

We first explain the effect of these hints in more detail and then show example uses in the next section.

**Parallelization hint** At the start of a PPR block, a speculative task is forked to jump to the end of the PPR block and execute the subsequent code. We call the forked tasks PPR tasks. Compared to conventional task fork or spawn, PPR has three distinct qualities. First, a PPR is possibly but not definitely parallel. This has two implications in safe

---

1Currently, A-BOP supports only iterative parallelism with no nesting. Nesting can be supported (see Section 4.3). In the current design, a **bop.ppr** hint is ignored if encountered inside a **PPR** task.
implementation. First, a PPR hint must support unsafe execution after the PPR block as well as within the PPR block. Furthermore, a PPR is fork-without-join. Fork-join primitives, including spawn/sync in Cilk (Frigo et al., 1998), future/get in Java, and async/finish in X10 (Charles et al., 2005), come in pairs: one for the fork and one for the join. PPR has fork but no explicit join. It uses speculation to ensure safe implicit join (Ding et al., 2007). In fact, not relying on a user is a requirement for safety.

**Sequential skeleton**  The execution between two PPR tasks is an inter-PPR link. We call all inter-PPR links collectively as the skeleton. The skeleton links are always run sequentially, while the PPR tasks may run in parallel. The sequential skeleton is often used to prepare for the parallel work in PPRs. Both the skeleton and the PPR code may be speculated.

**Ordering hint**  PPR tasks may synchronize with each other using the ordering hint. Ordered blocks, when executed by multiple PPR tasks, can only be executed by one task at a time and in the sequential program order. An ordering hint lets PPR tasks speculatively share results during parallel execution by communicating data changes and synchronizing between tasks. Ordering hints are implemented using dependence hints (Jacobs et al., 2009), which may also suggest task joins and implement write-value prediction. Like the parallelization hint, dependence hints do not change the program output. It affects only parallelism, not correctness.

**Speculation barrier**  A user can force speculation to end by calling abort_spec anywhere in a program. For the call to succeed, all preceding computation must finish and finish correctly, so only the sole non-speculative task can execute past the line. It erects a barrier—code before and after abort_spec cannot overlap. A speculation barrier has two common uses. The first is for a program to wait for a parallel execution to finish and to use its results. Second, since the code immediately after the barrier is executed once and only once, it can be used to implement unrecoverable operations. In A-BOP, system calls such as mmap are made to first call the speculation barrier.

### 2.2 Data Sharing

A program execution is divided into a series of skeleton links and a set of PPR tasks. We refer to them collectively as tasks. A-BOP ensures task isolation and sequential semantics using the following two mechanisms.

- *Copy-on-write data replication*, which lets a task hide its data writes from being seen by others and also prevents the task from seeing the data writes made by others.

- *Sequential commit of data writes*, which ensures that the writes from parallel tasks are merged in a sequential order when the tasks finish.

In addition to task isolation, copy-on-write has three other benefits (Ding et al., 2007). It removes all false dependences between tasks, i.e. write-read and write-write conflicts.
Parallel tasks are run without having to synchronize with each other (except when there are ordering hints). Finally, recovery is straightforward. A speculation task is simply killed if it is incorrect or too slow. Tian et al. characterized these properties in their copy-or-discard (CorD) model (Tian et al., 2008). Copy-on-write and ordered commit are also used by multi-threaded code to ensure deterministic parallelism (Section 6).

To understand data sharing between tasks, consider the four examples in Figure 1, representing all four ordering relations between inter-PPR and PPR task pairs. Listing 1 shows two skeleton links, where data is shared without copying. Listing 2 shows data assignment first in a skeleton link and then in the succeeding PPR. Copy-on-write in the PPR task replicates the datum before writing. The skeleton-skeleton and skeleton-PPR relations are always sequential. In these two examples, data sharing happens between sequentially executed code.

<table>
<thead>
<tr>
<th>Listing 1: skeleton-skeleton, sequential sharing, no copying</th>
</tr>
</thead>
</table>
| \begin{verbatim}
  i = 1
  bop_ppr {
    ...
  }
  i = i + 1
\end{verbatim} |

<table>
<thead>
<tr>
<th>Listing 2: skeleton-ppr, sequential sharing, copy-on-write in ppr</th>
</tr>
</thead>
</table>
| \begin{verbatim}
  i = i + 1
  # try setting g[x] in parallel
  bop_ppr {
    g[x] = i
  }
  ...
\end{verbatim} |

<table>
<thead>
<tr>
<th>Listing 3: ppr-ppr, possibly parallel, copy-on-write in both tasks</th>
</tr>
</thead>
</table>
| \begin{verbatim}
  # try setting g[x] in parallel
  bop_ppr {
    g[x] = 1
  }
  # try setting g[y] in parallel
  bop_ppr {
    g[y] = 2
  }
\end{verbatim} |

<table>
<thead>
<tr>
<th>Listing 4: ppr-skeleton, possibly parallel, copy-on-write in ppr</th>
</tr>
</thead>
</table>
| \begin{verbatim}
  # try setting g[x] in parallel
  bop_ppr {
    g[x] = i
  }
  i = i + 1
  # try setting g[y] in parallel
  bop_ppr {
    g[y] = 1
  }
  i = i + 1
  g[y] = i
\end{verbatim} |

Figure 1: Skeleton and PPR task ordering, data sharing, and copy-on-write data replication

Two PPR tasks are possibly parallel, shown by the third example in Listing 3. They create their own version of the data. The two versions conflict if they are for the same memory location. The conflict is detected when the speculation system tries to merge their results. At the conflict, the speculation system rolls back the second task and re-executes it correctly, now that the first task has finished.

The third example shows the two purposes of access monitoring: to compare and merge data changes in speculation tasks. The implementation of access monitoring affects both correctness and cost. The cost includes both that of error checking and data copying.
In the last example in Listing 4, a PPR task is possibly parallel with the next skeleton link. Because of copy-on-write, the two data writes after the PPR have no affect on (the memory space of) the PPR. Still, the $g[y]$ access in the inter-PPR code may depend on the PPR. The PPR hint is expressing possible parallelism with the skeleton code.

**Safe data sharing**  Skeleton links share data without copying. Other task pairs—skeleton-PPR, PPR-PPR, and PPR-skeleton—create multiple versions of data when assigning new values. Safe data sharing must record different data versions and supply the right version at each data reference.

We decompose the requirement of safe data sharing into two problems. The first is the monitoring problem, which is to identify shared-data access in parallel tasks. The second is the enforcement problem, which is to use the right data version. In A-BOP, the speculation system solves the enforcement problem automatically. To solve the monitoring problem, it uses access annotation.

### 2.3 Access Annotation

There are two types of access annotations as follows:

- **bop.promise**($data_1, data_2, ...$) is called by a PPR task to enumerate a list of data items that have been or will be modified. The PPR task promises a new value for the data.

- **bop.use**($data_1, data_2, ...$) is called by a PPR task or a skeleton link to enumerate a list of data that have been or will be read and need the most recent value. It is a request for data update.

A PPR task may write to a datum many times. Only the last value is promised. Hence a promise may be made anywhere and any number of times in a task and has the same effect—the promised value is the one at the end of the task execution. The placement of a use is similarly flexible. Wherever a use is called, it requires the correct value at the start of the task.

Access annotation has two other qualities. Like a declaration, a promise or a use is needed only once for the same data, regardless of how many times the data is accessed. On the other hand, an annotation is like a variable access and can be inserted inside arbitrary control flows. We may call access annotation a kind of floating executable declaration.

It is instructive to compare the two annotations with check-in and check-out used in version control. PPR tasks are analogous to separate individuals reading and modifying files in a shared repository. A promise is a deferred check-in, postponed to the end of the task. A use is a retroactive check-out, guaranteed to have the correct value at the start of the task.

Not all shared-data accesses in A-BOP require check-ins and check-outs. Neither do all of them need annotation. Next we show another set of four examples. The first two need access annotation for explicit data sharing. The next two are for implicit data sharing, where some or all annotations can be safely omitted.
Explicit data sharing  Safe parallelization of PPR-PPR or PPR-skeleton tasks require that their data sharing be made explicit. In Listing 5, the first function writes \( g[x] \) and the second function reads \( g[y] \). The PPR hints tell the A-BOP system to speculate them in parallel. The annotations tell A-BOP to check whether the two tasks are indeed parallel. In Listing 6, the first task assigns \( g[x] \) to 0, and the following code computes the reciprocal of \( g[y] \) if it is not 0. The hints and annotations tell A-BOP to speculate but monitor. The two examples differ in parallelism. The first is data speculation in the PPR code, and the second is control speculation in the inter-PPR code. In both cases, data sharing is made explicit by a pair of promise and use.

<table>
<thead>
<tr>
<th>Listing 5: explicit ppr-ppr sharing for data speculation</th>
</tr>
</thead>
<tbody>
<tr>
<td># try foo in parallel</td>
</tr>
<tr>
<td>bop_ppr {</td>
</tr>
<tr>
<td>bop_promise( g[x] )</td>
</tr>
<tr>
<td>g[x] = foo( x )</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td># try bar in parallel</td>
</tr>
<tr>
<td>bop_ppr {</td>
</tr>
<tr>
<td>bop_use( g[y] )</td>
</tr>
<tr>
<td>bar( g[y] )</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 6: explicit ppr-skeleton sharing for control speculation</th>
</tr>
</thead>
<tbody>
<tr>
<td># try setting g[x] in parallel</td>
</tr>
<tr>
<td>bop_ppr {</td>
</tr>
<tr>
<td>bop_promise( g[x] )</td>
</tr>
<tr>
<td>g[x] = 0</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>bop_use( g[y] )</td>
</tr>
<tr>
<td>if g[y] \neq 0</td>
</tr>
<tr>
<td>g[y] = 1/g[y]</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 7: implicit updating in delayed reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = head</td>
</tr>
<tr>
<td>i = 0</td>
</tr>
<tr>
<td>while n exists</td>
</tr>
<tr>
<td># try foo in parallel</td>
</tr>
<tr>
<td>bop_ppr {</td>
</tr>
<tr>
<td>bop_promise( s[i] )</td>
</tr>
<tr>
<td>s[i] = foo( n )</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>n = n.nxt</td>
</tr>
<tr>
<td>i = i + 1</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>abort_spec # stop speculation</td>
</tr>
<tr>
<td>return sum( s[1...n] )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 8: implicit updating in in-place reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = head</td>
</tr>
<tr>
<td>s = 0</td>
</tr>
<tr>
<td>while n exists</td>
</tr>
<tr>
<td># try foo in parallel</td>
</tr>
<tr>
<td>bop_ppr {</td>
</tr>
<tr>
<td>tmp = foo( n )</td>
</tr>
<tr>
<td>bop_ordered {</td>
</tr>
<tr>
<td>bop_promise( s )</td>
</tr>
<tr>
<td>s += tmp</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>n = n.nxt</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>bop_ordered { return s }</td>
</tr>
</tbody>
</table>

Figure 2: The upper two examples show explicit sharing of \( g[x], g[y] \). They show data and control speculation over unknown dependences. The lower two examples show implicit updating of \( s \) and implicit sharing of \( n, i \). They use synchronization to preserve known dependences.
Implicit data updating  bop_use annotations can be omitted at places where data has been updated implicitly. There are two common places of implicit data updating. The first is at a speculation barrier, where all previous promises are merged before continuing. The second is at an bop_ordered block, where promises in all previous ordered blocks are merged before continuing. The two cases are demonstrated in Listing 7 and 8, where the reduction variable s needs only the promise annotaiton but no use annotation.

Implicit data sharing  Data sharing between two sequential tasks, i.e. skeleton-skeleton and skeleton-PPR, do not need access annotation at all. For example, in the loop in Listing 7, the variable n is assigned before and used in each PPR task, but needs no promise nor use annotation.

3 The Semantics

We first define several terms. The speculation barrier divides an A-BOP execution into non-overlapping phases. In each phase, we number the inter-PPR links and the PPR tasks as link[i] and ppr[i], with ppr[i] immediately following link[i], i = 1, . . . , n. If a phase ends in the skeleton, for symmetry we assume there is an empty ppr[n] after link[n]. We divide the data accessed by each task into four sets, read, write, ano_read, ano_write. The first two are the sets of data read and modified, and the next two are the sets of data listed in bop_use and bop_promise annotations. The prefix ano is short for annotated.

The annotated promises, i.e. ano_write, take effect in following ways:

- Patch-based release. The promised data changes are released together as a patch at the end of a PPR task. If a PPR task has an ordering block, the promises in the order block are released as a (speculative) patch at the end of the ordered block.
- Non-deterministic release time. Since a PPR task is run asynchronously, the timing of its patch release is unpredictable in general.
- In-order patching. Correctness checking is done sequentially in A-BOP (as in BOP). A PPR task is verified correct only after all previous PPR tasks have completed and verified correct. As a result, PPR patches become visible in the sequential order.  

3.1 Canonical Execution

The semantics of an annotated program is defined by the canonical execution. It runs the program sequentially. At the end of each ppr[i], only the promised data changes, ppr[i].ano_write, are preserved. Other changes, i.e. ppr[i].write – ppr[i].ano_write, are erased, i.e. reverted to the pre-PPR value. The implementation is by patching. At the start of a PPR task, it forks a child process to execute the task. The ano_write set is copied back to the parent, which then continues the canonical execution.

2An exception is the speculative release by ordered blocks.
The canonical execution is sequential. It is equivalent to that of the original code plus a data resetting routine at the end of each PPR task. There is no concurrency nor non-determinism. Like sequential code, it produces the same output for the same input. Also like sequential code, a user can analyze and correct an incorrect output using a standard debugger.

### 3.2 Annotation Error

A program has an annotation error if a parallel execution may produce a different result than that of the canonical execution. Listing 9 shows an example error. The code has three PPRs. It starts with $g = 1$, assigns $g = 2, 3$ in the first two PPRs, and reads $g$ in the third PPR. Data sharing is not fully annotated because the last PPR reads $g$ without requesting an update (through the use annotation). In a parallel execution, depending on when the first two PPRs finish, the third PPR may read either 1, 2, or 3 from $g$. This is an error because $g$ should be 3 in the third PPR as stipulated by the canonical execution.

### 3.3 PPR Verification

We verify each phase of an A-BOP execution in the sequential order. For each phase, we use an inductive process, which checks $\text{link}[i]$ and $\text{ppr}[i]$ after verifying all previous skeleton links and PPRs. The cost is $O(n^2)$, where $n$ is the number of PPR tasks in the phase.

Like the canonical execution, the verifier runs the program sequentially and uses patching to incorporate PPR results. The difference is when patches are applied. If there is no
dependence between PPR tasks, the verifier runs PPR \( i \) exactly \( i \) times. First, it runs \( i \) with no prior patches. Then for \( k = 1, \ldots, i - 1 \), it includes the patches of PPR 1 through \( k \), that is, the patch combo \( \cup_{j=1}^{k} \text{ppr}[j].\text{ano}_\text{write} \).

Two PPRs may have a true dependence. PPR \( i \) depends on PPR \( p \) if and only if:

\[
p < i, \text{ppr}[p].\text{ano}_\text{read} \cap \text{ppr}[i].\text{ano}_\text{write} \neq \emptyset
\]

The verifier handles dependences as follows. For each PPR \( k \), let PPR \( i \) be the latest PPR that \( k \) depends on. For each PPR \( j \) from \( i \) to \( k - 1 \), the verifier tests PPR \( k \) with patches of PPR 1 to PPR \( j \). Consider a comparison in Table 1. The two rows shows three PPRs first without any dependence and then with one dependence. Different groups of tests are needed to verify that PPR 3 produce identical result as the canonical execution (the last test in each group).

<table>
<thead>
<tr>
<th>PPRs</th>
<th>dependences</th>
<th>tests for PPR 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3</td>
<td>none</td>
<td>(3), (1,3), (1,2,3)</td>
</tr>
<tr>
<td>1, 2, 3</td>
<td>1 \rightarrow 3</td>
<td>(1,3), (1,2,3)</td>
</tr>
</tbody>
</table>

Table 1: Verification runs needed to verify PPR 3 with and without a dependence.

Since the verifier executes PPR \( i \) at most \( i - 1 \) times, the total cost is \( O(n^2) \) for a program with \( n \) PPRs. This seems counter intuitive since \( n \) task may finish execution in \( n! \) different orders. The reason for the quadratic cost is the sequential patching in the A-BOP model. The tasks may finish out of order but their changes are applied in order. There are a quadratic number of ways patches can be serially applied, hence the cost of the verifier.

We show the effect of sequential patching using the program in Listing 10. It has three PPRs. The first two each provide an argument for the third PPR. In sequential patching, the third PPR may see the first PPR patch without the second PPR patch, but it may not see the second PPR patch without the first.

The quadratic cost is reduced if the granularity of PPRs is increased. In fact, dividing a program execution into isolated tasks is extremely beneficial for this type of verification. The cost of \( O(n^2) \) is very different when \( n \) is the number of PPRs as opposed to the number of run-time instructions.

### 3.4 Skeleton Verification

A skeleton link may conflict with an earlier PPR task. A conflict happens when the skeleton link read or write a value promised by an earlier PPR but not yet patched. If the skeleton access is a read, we call it a PPR-skeleton value flow. If the skeleton access is a write, we call it a PPR-skeleton memory (write) reuse. The example in Listing 11 shows both types of conflicts. The skeleton code can either use an update annotation to monitor the conflicts or insert a barrier to remove the conflicts.

The verifier checks whether the inter-PPR code has sufficient \texttt{bop\_use} annotations to monitor all conflicts. It first computes the intersection of the data access of the skeleton
Listing 10: Sequential patching

```plaintext
... ...
  x = y = z = 0
  bop_ppr {
    x = 1
    bop_promise( x )
  }
  ...
  bop_ppr {
    y = 2
    bop_promise( y )
  }
  ...
  bop_ppr {
    if x == 0 and y == 2
      z = 3  # unreachable code due to sequential patching
    end
  }
```

Listing 11: The need for skeleton verification

```plaintext
  bop_ppr begin
    x = 1
    bop_promise( x )
  end
  y = x  # ppr-skeleton value flow
  x = 2  # ppr-skeleton write reuse
  # are not allowed w/o a bop_use(x) or a barrier
```

and the promises of previous PPR tasks. Then it checks if all variables in the intersection have an update annotation. Formally, it checks

\[
\forall i, (\text{link}[i].\text{read} \cup \text{link}[i].\text{write}) \cap \left( \bigcup_{k=1}^{i-1} \text{ppr}[k].\text{ano}_{\text{write}} \right) \subseteq \text{link}[i].\text{ano}_{\text{read}}
\]

To implement, we can instrument all memory references in the skeleton code (but not the PPR code) to find the intersection. If the intersection is empty, there is no PPR-skeleton conflict, and the skeleton execution would always be correct, regardless of the PPR execution.

Skeleton verification uses dependence checking, not exhaustive testing (as used by PPR verification in Section 3.3). If we use exhaustive testing for both the skeleton and PPR code, we would have to consider all interleaving of the link sequence and the PPR sequence in which link \( i \) always appears before PPR \( i \). The interleaving problem is similar to arranging
parentheses correctly in a sequence, and the number of choices to check is \( \frac{2n!}{n!(n+1)!} \), known as the Catalan number. The cost is exponential. By using dependence checking, we ensure canonical semantics in quadratic time.

4 Discussion and Comparison

4.1 Composability

Composability of annotations If we fix the program, the hints, and the bop_promise annotations but test more of its inputs to find all needed bop_use annotations, we have the monotonicity property in that the refinement over a new execution will not invalidate executions that were correct before the refinement. New updates do not break the correctness of previously passed tests.

Complex software often has hidden errors. Access annotation is a semantic construct like program control flow and data reference. Because of the monotone property, the use of access annotation can be validated by systematic testing as we do for conventional software errors. Concurrency errors in parallel software are usually harder to test. The A-BOP race checker brings down the cost and complexity of concurrency error testing to a level closer to sequential error testing.

Composability of code Since A-BOP has sequential semantics, multiple A-BOP tasks can be grouped to form a single task the same way sequential tasks are stringed together. In addition, A-BOP tasks may run with auto-parallelized code, since both have sequential semantics. When new parallelism is introduced, e.g. by adding a task or removing a barrier, old access annotations need to be checked for new races.

Semi-automatic annotation Automatic techniques may be used to identify shared data accesses and annotate them using the A-BOP interface. Such analysis includes type inference as in Jade (Rinard and Lam, 1998), compiler analysis as in CorD (Tian et al., 2010, 2008), and virtual memory support as in BOP (Ding et al., 2007). A user may use automatic analysis in most of the program and then select a few critical loops or functions for manual annotation. The hybrid solution lets a programmer to obtain most of the benefit of manual annotation yet let a tool perform most of the annotation work.

4.2 Usability

Many parallel languages have primitives for annotating data or data access. There are two basic design choices. The first is whether to annotate shared data, private data, or both. If shared data is annotated, the second choice is whether the annotation should be a declaration, one for each datum; or instrumentation, one for each access. The next table shows the four basic designs and an example for each design. For example, OpenMP provides primitives to declare whether local and global variables are shared and assumes that all heap data is shared (OpenMP). DSTM annotates all accesses to shared data (Herlihy
et al., 2003). In comparison, A-BOP annotates for some shared data and for at most one of their read and write accesses.

<table>
<thead>
<tr>
<th>annotation</th>
<th>declaration</th>
<th>instrumentation</th>
<th>executable declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared data</td>
<td>Jade, OpenMP</td>
<td>DSTM</td>
<td>A-BOP</td>
</tr>
<tr>
<td>private data</td>
<td>OpenMP</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

**Precision, cost, and correctness** A declaration-based method has two difficulties annotating dynamically allocated data. First, heap data often has no static names. Second, the access is often conditional, and the location of the access is often parameterized. As a type of instrumentation, access annotation is more precise than declaration. On the other hand, access annotation is more efficient than instrumenting all data access. The cost saving comes from fewer run-time calls but more importantly from less synchronization between tasks. In fact, A-BOP does not synchronize at an annotation call. It inserts the annotated locations into a task-local record.

The downside is missing annotation, which leads to insufficient monitoring. In general, a user would rather lose parallelism than correctness. An slow program is less embarrassing than an buggy one. The A-BOP race checker provides a safe remedy, so a programmer does not have to sacrifice correctness for the benefits of precision and cost.

**Down-sizing shared data and meta-data** Most costs of speculation come from monitoring, checking, and copying shared data. A user can minimize this cost by specifying only the data that has to be shared. Furthermore, a user may annotate shared data by regions rather than by elements, reducing both the number of annotation calls and the size of meta-data that the speculation system has to track and process. For these purposes, an interface for marking shared data is more intuitive to use than the one for marking private data.

### 4.3 Extensions

**Hint widening** An access annotation specifies shared data by the address and size. A user may mistakenly record just a fraction of a memory object. We can re-interpret the user’s intent to mean the whole object. Such widening may improve program correctness but may also lose parallelism (if the augmented memory space contains private data).

**Parallelization improvements** The canonical execution can also be used to measure parallelism, granularity, and the cost of data commit and communication and to provide suggestions for the programmer to understand and improve the parallel performance.

**Nested parallelism** The execution model can be extended to spawn inner tasks at nested PPRs. The structure of the skeleton changes from a list, where there is no nesting, to a tree. The correctness checking is the same as no-nesting case once the task tree is linearized using well-established numbering techniques (Mellor-Crummey, 1991; Bender et al., 2004).
5 Demonstrations

We demonstrate A-BOP and especially the power and convenience of access annotation on three examples. The first has uncertain do-all parallelism, and the other two have do-across parallelism.

String scanning and pattern conversion  Consider the problem of sub-string substitution, which scans an array of bytes and rewrites all sub-strings that match a given pattern. The program in Listing 12 processes the input in \( m \)-letter blocks. The code uses the range syntax. For example, \( str[lo...hi] \) refers to the series of letters starting from \( str[lo] \) and ending at (and including) \( str[hi] \).

The program has uncertain parallelism. For example, we use it to scan and replace all occurrences of the 3-letter pattern “aba” with “bab”. The process may be data parallel, e.g. when the input string has no “aba”. Or it may be completely sequential, e.g. if the string is “abaa..a” and becomes “bb..bab” after conversion. For safe parallelization, we mark the inner loop a possibly parallel task. To monitor conflicts, we annotate the read of each block boundary, the write at each substitution, and the read before turning the result.

The annotation interface is used to ensure correctness and minimize cost. The annotations are precise down to the exact letter and the exact execution branch. Ranges are used when adjacent letters are accessed together, reducing the size of meta-data. In addition, only the changed letters are copied out of the PPR task. If the string is not changed, there is no copying. The precise annotations, made possible by the annotation interface, minimize the work of monitoring and data copying.
Listing 13: Limited-size buffer processing

```plaintext
inbuf[1...b], outbuf[1...b] # two buffers of bounded size

no = 1

while ( reader.has_more )
    bop_ppr {
        # try processing next input in parallel
        bop_ordered(reader) {
            bop_promise(inbuf[no]) # monitor for premature buffer reuse
            inbuf[no] = reader.next_line
        }

        bop_use(inbuf[no]) # monitor for premature buffer reuse
        outbuf[no] = lots_work(inbuf[no]) # no redundant monitoring

        bop_ordered(printer) {
            print(outbuf[no])
        }
    }

    no = no==b? 1 : no+1
end
```

Resource-constrained pipelining  The loop in Listing 13 uses a fixed-size buffer to copy lines from a reader to a printer. There are three shared resources: the reader, the printer, and the buffer. To serialize the use of the reader and the printer, we use two ordered regions (with different labels). The buffer has \( b \) entries. It is monitored to allow parallel use of different entries but not the same entry. It is sufficient to monitor just one buffer. The manual interface allows a programmer to avoid unnecessary monitoring of the second buffer. The example was used by Per Brinch Hansen to show the problem of resource-constrained parallelism. The limited buffer size makes the problem harder than 3-stage pipelining.

Time skewing  Iterative solvers are widely used to compute fixed-point or equilibrium solutions. Listing 14 shows the structure of a typical solver as a two-nested loop. The outer level is a convergence loop. Each iteration is a time step. In a time step, there is first a parallel inner loop that computes on domain data and then a convergence test. The convergence test cannot be done until all computation finishes, and the next time step cannot start until the convergence test is done. However, in all but the last time step, the computation will not converge. Speculation can allow two time steps to overlap, which is known as time skewing (Wonnacott, 2002). Previous literature shows large performance benefits for both sequential (Wonnacott, 2002; Song and Li, 1999) and parallel (Liu and Li, 2010) executions.

Time skewing can be expressed using hints, assuming complete data monitoring (Ding, 2011). Here we give a version using explicit access annotation. In Listing 14, we use a PPR
while not converged
    bop_use( converged )  # expose dependence on converged

for i in 1...n
    bop_ppr {  # try computing in parallel
        r = compute( data[i] )
        bop_ordered {
            s = s.add_result( r )
            bop_promise( s )
        }
    }
end

bop_ppr {  # try next time step in parallel
    bop_ordered {
        bop_use( s )
        if good_enough?( s )
            converged = true
            bop_promise( converged )
    }
}
end

block to parallelize the inner loop and a second PPR block to make the convergence check asynchronous. The speculative execution continues into the next time step assuming that the computation does not converge. The speculation is wrong only in the last iteration. Then the run-time system cancels the remaining speculation and returns the correct result.

Listing 14 shows that a total of four annotations are sufficient to replace automatic data monitoring. The first annotation in the loop benefits from flexible placement (Section 2.3): the annotation inserted after the convergence check ensures correct value before the check.

6 Related Work

Software speculative parallelization Software speculative parallelization was pioneered by Rauchwerger and Padua (Rauchwerger and Padua, 1995). While most techniques automatically parallelized do-all loops (with limited potential on full applications (Kejariwal et al., 2006)), several techniques provided a safe interface for expressing possible parallelism (Welc et al., 2005; von Praun et al., 2007) and likely dependence (Zhai et al., 2008; Raman et al., 2010). BOP used Unix processes to implement parallelism and dependence...
hints for sequential C/C++ programs (Ding et al., 2007; Jiang and Shen, 2008; Zhang et al., 2009; Jacobs et al., 2009). Process-based systems use the virtual memory protection mechanism, which transparently provides strong isolation, on-demand data replication, and complete abortability. A process-based system can run in a distributed environment (Jacobs et al., 2009; Kim et al., 2010; Aviram et al., 2010). Similar benefits can be realized for threads using the Copy-or-Discard model with compiler and run-time support (Tian et al., 2008). Raman et al. presented a software system called SMTX that supports pipelined execution of sequential loops (Raman et al., 2010). Recently, Tian et al. generalized the model as SpiceC (scalable parallelism via implicit copying and explicit commit) to support either speculative parallelization by default or other types of commits defined by the user (Feng et al., 2011).

The original BOP divides program data into three categories — shared, checked, and likely private — for complete monitoring (Ding et al., 2007). Shared data is monitored at page granularity. Value-based checking is precise and has no false sharing. SMTX uses value-based checking for all shared data to eliminate the false sharing (at the cost of per-access monitoring and logging) (Raman et al., 2010). Instead of automatic monitoring, this report describes an interface for a user to control access monitoring. It shows when annotations are necessary and how to ensure their correctness. A manual interface may leverage user knowledge and enable more efficient and precise monitoring than what is possible with automatic methods alone.

**Data annotations** HPF provides array templates for a user to specify data partition and alignment (Allen and Kennedy, 2001). An HPF compiler generates parallelized code for either shared memory or distributed memory execution (Adve and Mellor-Crummey, 1998; Wang et al., 2004). The Jade language lets a user specify shared data, and the Jade system derives the dependence for automatic run-time parallelization (Rinard and Lam, 1998). The Jade annotation is declarative using the *shared* keyword. It may be operational, for example, by traversing a shared linked list. In HPF and Jade, data sharing must be specified statically or at least before the task starts. In programs with uncertain control flow and data indirection, such specification must be conservative and therefore not always perfectly precise. It cannot specify data to be allocated and shared by parallel tasks. A-BOP annotations can precisely specify dynamic data sharing but require testing and race checking to ensure their completeness.

Hill et al. used check-in and check-out hints for a user to make data sharing explicit to facilitate shared-memory cache coherence (Hill et al., 1993). They were performance primitives and did not change program semantics. As discussed in Section 2.3, A-BOP promises and uses are postponed check-ins and retroactive check-outs. Not all shared-data accesses in A-BOP tasks require check-ins and check-outs. Neither do all of them need annotation. Finally, access annotations are semantics constructs and require correctness checking.

**Race detection in fork-join parallelism** On-the-fly race detection can be done efficiently for perfectly nested fork-join parallelism (Mellor-Crummey, 1991; Bender et al., 2004). Callahan and others showed that at the program level, the problem of post-wait
race checking is co-NP hard and gave an approximate solution based on dataflow analysis (Callahan and Sublok, 1988; Callahan et al., 1990). They used the term canonical execution to mean the sequential execution of fork-join parallel constructs. The post-wait race checking can be done at run time in $O(np)$ time, where $n$ is the number of synchronization operations and $p$ is the number of tasks (Netzer and Ghosh, 1992). These and other results are summarized in (Helmbold and McDowell, 1994). The execution model of A-BOP is similar except that the primitives of forks, posts, and waits are hints and do not affect program semantics. Instead of race-free execution for synchronization operations, A-BOP guarantees sequential semantics for all memory accesses, annotated or not.

**Deterministic parallelism** Deterministic Java uses a type system to ensure deterministic semantics (Bocchino Jr. et al., 2009). A type system guarantees safety but is conservative and may rule out programs that have deterministic semantics. A-BOP takes a run-time approach and permits more types of parallelism. However, access annotations have to be tested for correctness.

Scott and Lu gave five definitions of determinism and showed their containment relationships (Scott and Lu, 2011). A language-level definition is ExternalEvents, which requires that the observable events in two executions be the same. An implementation level definition is Dataflow, which requires that two executions follow the same “reads-see-writes” relationship. A-BOP lets a programmer define external events and relies on speculation to preserve data flow. The combination enables user control over both the semantics and the cost of its enforcement.

The framework of task isolation and ordered commit has been used to ensure deterministic semantics and eliminate concurrency errors in multi-threaded code. Grace used processes to implement threads, eliminating deadlocks and data races (Berger et al., 2009). Burckhardt et al. defined isolation and revision types in C# to buffer and merge concurrent data changes in r-fork/r-join threads (Burckhardt et al., 2010). Determinator was developed as a new operating system that buffers processes and threads in private workspaces and terminates an execution if concurrent data writes are detected (Aviram et al., 2010). CoreDet ensured determinism in threaded execution using versioned memory and a deterministic commit protocol (Bergan et al., 2010). The access annotation of A-BOP may help to make program monitoring more precise and generally applicable in these systems. The concept of executable declaration is applicable, so is the use of error recovery and speculative synchronization.

**Comparison with OpenMP** OpenMP is based on loop parallelism (OpenMP). The number of iterations is known before a parallel loop starts, which allows efficient parallelization. There is an implicit barrier at the end of a parallel loop. Data is shared unless declared as private. OpenMP loops may have non-deterministic results and intermittent errors. The implicit barrier prevents parallelism between a loop and the subsequent code. A-BOP is based on task parallelism and guarantees safe, sequential semantics. OpenMP allows out-of-order access to shared data, but A-BOP can allow only in-order access in order to for it to safeguard sequential semantics (Ding, 2011).
7 Summary

We have presented a new interface for a programmer or a tool to annotate shared-data access and two supporting techniques, canonical execution and race checking, to ensure that annotations are correct and complete. The interface is precise and flexible, enough to minimize the cost of access monitoring and data commit. It is economic, only needed in parallel tasks and can be omitted through implicit updating and implicit sharing. It is composable, allowing the mixed use of manual and automatic parallelization.

Speculative systems till recently are burdened with the mostly high and often unpredictable cost of monitoring and correctness checking. Much of the cost may be saved by leveraging user knowledge. By making program monitoring programmable, the A-BOP hints and annotations enable a user to control both the sequential meaning and its enforcement. As a result, it has expanded the reach of speculative systems from safe parallelization to safe parallel programming.

Acknowledgments

The relation between the Catalan number and the cost of exhaustive testing (Section 3.4) is due to Daniel Stefankovic. The author also wishes to thank Michael Scott, Tongxin Bai, Xipeng Shen, Bin Bao, Xiaoming Gu, Xiaoya Xiang and Peng Wu for their comments on the ideas and the presentation.

References


Ding, Chen. 2011. Two examples of parallel programming without concurrency constructs (PP-CC). In PPOPP, pages 299–300.


Zhang, Chao, Chen Ding, Kirk Kelsey, Tongxin Bai, Xiaoming Gu, and Xiaobin Feng. 2009. A language of suggestions for program parallelization. Technical Report URCS #948, Department of Computer Science, University of Rochester.