Software Behavior Oriented Parallelization

Chen Ding[†⋆], Xipeng Shen[‡], Kirk Kelsey[†], Chris Tice[†], Ruke Huang[⋆], and Chengliang Zhang[†]

[†]Computer Science Dept., University of Rochester
[‡]Computer Science Dept., College of William and Mary
[⋆] Microsoft Corporation

Abstract

Many sequential applications are difficult to parallelize because of unpredictable control flow, indirect data access, and input-dependent parallelism. These difficulties led us to build a software system for behavior oriented parallelization (BOP), which allows a program to be parallelized based on partial information about program behavior, for example, a user reading just part of the source code, or a profiling tool examining merely one or few executions.

The basis of BOP is programmable software speculation, where a user or an analysis tool marks possibly parallel regions in the code, and the run-time system executes these regions speculatively. It is imperative to protect the entire address space during speculation. The main goal of the paper is to demonstrate that the general protection can be made cost effective by three novel techniques: programmable speculation, critical-path minimization, and value-based correctness checking. On a recently acquired multi-core, multi-processor PC, the BOP system reduced the end-to-end execution time by integer factors for a Lisp interpreter, a data compressor, a language parser, and a scientific library, with no change to the underlying hardware or operating system.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Concurrent Programming—parallel programming; D.3.4 [*Programming Languages*]: Processors—optimization, compilers

General Terms Languages, Performance

Keywords speculative parallelization, program behavior

Dedication *This paper is dedicated to the memory of Ken Kennedy, who passed away on February 7, for his lasting leadership in parallel computing research.*

1. Introduction

Many existing programs have dynamic high-level parallelism, such as, a compression tool that processes data buffer by buffer, an English parser parsing sentence by sentence, and an interpreter interpreting expression by expression. They are complex and may make extensive use of bit-level operations, unrestricted pointers, exception handling, custom memory management, and third-party libraries. The unknown data access and control flow make such applications difficult if not impossible to parallelize in a fully automatic fashion. On the other hand, manual parallelization is a daunting task for complex programs, pre-existing ones in particular. Moreover, many programs have input-dependent behavior where both the degree and the granularity of parallelism are not guaranteed or even predictable. The complexity and the uncertain performance gain make it difficult to justify the investment of time and the risk of error.

```
while (1) {
 get_work();
 ...
 BeginPPR(1);
 step1();
 step2();
 EndPPR(1);
 ...
}
```

Figure 1. possible loop parallelism

```
...
BeginPPR(1);
work(x);
EndPPR(1);
...
BeginPPR(2);
work(y);
EndPPR(2);
...
```

Figure 2. possible function parallelism

We address these problems with behavior oriented parallelization (*BOP*). Here the behavior is defined as the set of all possible executions of a program. The goal of *BOP* is to improve the (part of) executions that contain coarse-grain, possibly input-dependent parallelism, while guaranteeing correctness and basic efficiency for other cases. Unlike traditional code-based approaches that exploit the invariance in all behavior, a behavior based system utilizes partial information to incrementally parallelize a program or to streamline it for common uses.

The new system lets a user or a profiling tool to suggest *possibly parallel regions* (*PPR*) in a program by marking the start and the end of the region with matching markers: *BeginPPR(p)* and *EndPPR(p)*. Figures 1 and 2 show the marking of possible (pipelined) loop parallelism and possible function parallelism respectively. *PPR* is region-based (which is different from communication-based `do-across` [1, 10]), the parallelism is likely but not definite (which is different from future and parallel section constructs [17, 27]), and a region may have unpredictable entries or exits (which is different from transactions [19]).

To support possible parallelism, *BOP* protects the entire address space by dividing it into possibly shared and privatizable subsets and monitoring and replicating them accordingly. The virtual memory (VM) protection mechanism in modern operating systems can be readily used for this purpose. For *BOP*, the VM protection effects on-demand data replication and supports complete rollback.

The process-based protection has a high overhead. However, much of it is inherently unavoidable for a software scheme to

PLDI'07 June 11–13, 2007, San Diego, California, USA.

support unpredictable computations. A major goal of this paper is to show that general protection can be made cost effective by mainly three techniques.

First, most overheads—starting, checking, and committing—are placed on the speculative path, while the system uses unmonitored, non-speculative execution to ensure basic progress if all speculation should fail. We will show how the system holds a tournament between speculative and non-speculative executions, so that it uses the speculative result only when the parallel execution finishes not only correctly but also faster than the sequential execution. The parallelized program in the worst case is as fast as its unmodified sequential version.

Second, *BOP* uses value-based checking, which is more general than dependence-based checking (also known as Bernstein conditions [1, 3]). It permits parallel execution in the presence of true dependences and it is one of the main differences between process-based *BOP* and existing thread-based systems (as discussed in Section 2.5). In the paper we give a proof of its correctness and show how *BOP* bounds and hides its run-time costs.

In implementation, *BOP* differs from prior techniques of speculative execution and current techniques of transactional memory in that the overhead of *BOP* depends on the size of accessed data rather than the length of the parallel execution. Consequently, the cost is negligible if the size of the parallel task is sufficiently large. This leads to its third feature: *BOP* is programmable. We will describe its profiling technique and user interface for finding appropriate parallel regions.

BOP inherits two inherent limitations of speculation: the execution may be faster but not more efficient (because of extra CPU, memory, and energy usage), and the speculative region cannot invoke general forms of I/O and other operations with unrecoverable side effects. However, the main advantage of *BOP* is ease of programming. Parallelizing a complex application will perhaps always require non-trivial code changes, but the process is greatly facilitated by the system because the programmer no longer needs to understand the entire application, spend time in parallel programming or debugging, or worry about the negative performance impact of parallelization.

Compared to thread-based techniques, *BOP* has the overhead of general protection and the problem of false sharing. However, the overhead of the system is automatically hidden by the granularity of tasks regardless of their parallelism. Furthermore, *BOP* can be more efficient because it uses unmodified, fully optimized sequential code, while explicit threading and its compiler support are often restrained due to concerns over the weak memory consistency on modern processors. As we will see in the evaluation section, the generality and basic efficiency make it possible for large, existing software to benefit from parallel execution, that is, to be parallelized without being paralyzed.

2. Behavior-oriented Parallelization

2.1 Possibly Parallel Regions

The *PPR* markers are written as *BeginPPR(p)* and *EndPPR(p)*, where p is a unique identifier. At a start marker, *BOP* forks a process that jumps to the matching end marker and speculatively executes from there. While multiple *BeginPPR(p)* may exist in the code, *EndPPR(p)* must be unique for the same p. The matching markers can only be inserted into the same function. The exact code sequence in C language is as follows.

- *BeginPPR(p)*: if (BeginPPR(p)==1) goto EndPPR_p;
- *EndPPR(p)*: EndPPR(p); EndPPR_p:;

At the presence of unpredictable control flows, there is no guarantee that a start marker is followed by its end marker, or the matching markers are executed the same number of times. For example, a `longjmp` in the middle of a parallel region may cause the execution to back out and re-enter.

The *BOP* system constructs a sequence of non-overlapping *PPR* instances using a *dynamic* scope. Based on the order the *PPR* markers are executed at run time, *BOP* dynamically delineates the boundary of *PPR* instances. While many options are possible, in the current design we use the following scheme: At any point t, the next *PPR* instance starts from the first start marker operation *BeginPPR(p)* after t and then ends at the first end marker operation *EndPPR(p)* after the *BeginPPR(p)*. For example, assume the program has two *PPR* regions P and Q marked by m_P^b, m_P^e, m_Q^b, and m_Q^e. If the program, from the start t_0, executes the markers six times from t_1 to t_6 as follows:

t_0	t_1	t_2	t_3	t_4	t_5	t_6
	m_P^b	m_P^b	m_P^e	m_Q^b	m_P^e	m_Q^e

Two dynamic *PPR* instances are from t_1 to t_3 and from t_4 to t_6, which will be run in parallel. The other fragments of the execution will be run sequentially, although the part from t_3 to t_4 is also speculative. Note that the above scheme is only one example of the many choices the run-time system can take. In principle, *BOP* may dynamically explore different schemes in the same execution.

Compared to the static and hierarchical scopes used by most parallel constructs, the dynamic scope lacks the structured parallelism to model complex task graphs and data flows. While it is not a good fit for static parallelism, it is a useful solution for the extreme case of dynamic parallelism in unfamiliar code.

A coarse-grain task often executes thousands of lines of code, communicates through dynamic data structures, and has non-local control flows (exceptions). Functions may be called through indirect pointers, so parallel regions may be interleaved instead of being disjoint. Some of the non-local error handling may be frequent, for example, an interpreter encountering syntax errors. In addition, exceptions are often used idiomatically, for example, resizing a vector upon an out-of-bound access. Some non-local jumps are rare. For example, the commonly used *gzip* program has error checking and abnormal exit in the compression code. Although in our experience no error has ever happened, if one cannot prove the absence of error in *gzip* (or other sizeable software), dynamic scopes such as *PPR* can be used to parallelize the common cases while guarding against unpredictable or unknown entries and exits.

Since the *PPR* markers can be inserted anywhere in a program and executed in any order at run-time, the system tolerates incorrect marking of parallelism, which can easily happen when the region is marked by a profiling tool based on a few inputs or given by a user unfamiliar with the code. The markers are programmable hints, so are other parts of the interface, where the quality of hints affects the parallelism but not the correctness nor the worst-case performance.

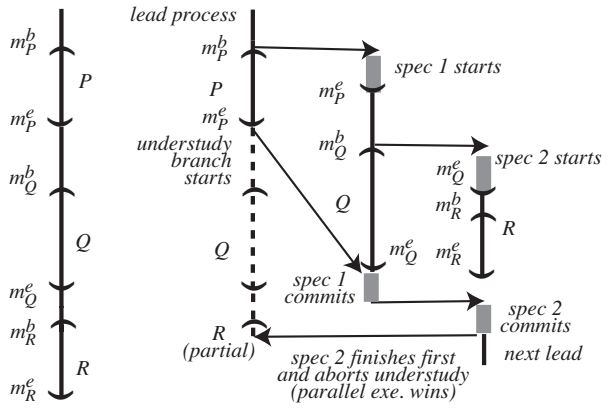
2.2 The Parallel Ensemble

The *BOP* system uses concurrent executions to hide the speculation overhead off the *critical path*, which determines the worst-case performance where all speculation fails and the program runs sequentially.

2.2.1 Lead and Spec Processes

The execution starts as the *lead* process, which continues to execute the program non-speculatively until the program exits. At a (pre-specified) speculation depth k, up to k processes are used to execute the next k *PPR* instances. For a machine with p available processors, the speculation depth is set to $p-1$ to make the full use of the CPU resource (assuming CPU is the bottleneck).

Figure 3 illustrates the run-time setup by an example. Part (a) shows the sequential execution of three PPR instances, P, Q, and R. Part (b) shows the speculative execution. When the lead process



(a) Sequential execution of PPR instances *P, Q,* and *R* and their start and end markers.

(b) A successful parallel execution, with *lead* on the left, *spec* 1 and 2 on the right. Speculation starts by jumping from the start to the end marker. It commits when reaching another end marker.

Figure 3. An illustration of the sequential and the speculative execution of three PPR instances

reaches the start marker of P, m_P^b, it forks the first speculation process, spec 1, and continues to execute the first *PPR* instance. Spec 1 jumps to the end marker of P and executes from there. When spec 1 reaches the start of Q, m_Q^b, it forks the second speculation process, spec 2, which jumps ahead to execute from the end of Q.

At the end of P, the lead process starts the *understudy* process, which re-executes the following code non-speculatively. Then it waits for spec 1 to finish, checks for conflicts, and if no conflict is detected, commits its changes to spec 1, which assumes the role of the lead process so later speculation processes are handled recursively in a similar manner. The kth spec is checked and combined after the first $k-1$ spec processes commit. When multiple spec processes are used, the data copying is delayed until the last commit. The changed data is copied only once instead of multiple times in a rolling commit. Our implementation of these steps is shown later in Section 3.1.

The speculation runs slower than the normal execution because of the startup, checking, and commit costs. The costs may be much higher in process-based systems than in thread-based systems. In the example in Figure 3(b), the startup and commit costs, shown as gray bars, are so high that the parallel execution of spec 1 finishes slower than the sequential understudy. However, by that time spec 2 has finished and is ready to commit. The second commit finishes before the understudy finishes, so spec 2 aborts the understudy and becomes the next lead process.

BOP executes *PPR* instances in a pipeline and shares the basic property of pipelining: if there is an infinite number of *PPR*s, the average finish time is determined by the starting time not the length of each speculation. In other words, the parallel speed is limited only by the speed of the startup and the size of the sequential region outside *PPR*. The delays during and after speculation do not affect the steady-state performance. This may be counter intuitive at first because the commit time does not matter even though it is sequentially done. In the example in Figure 3(b), spec 2 has similar high startup and commit costs but they overlap with the costs of spec 1. In experiments with real programs, if the improvement jumps after a small increase in the speculation depth, it usually indicates a high speculation overhead.

Table 1. *BOP* actions for unexpected behavior

behavior	prog. exit or error	unexpected *PPR* markers
lead	exit	continue
understudy	exit	continue
spec(s)	abort speculation	continue

2.2.2 Understudy: Non-speculative Re-execution

BOP assumes that the probability, the size, and the overhead of parallelism are all unpredictable. The understudy provides a safety net not only for correctness (when speculation fails) but also for performance (when speculation is too slow). For performance, *BOP* holds a two-team race between the non-speculative understudy and the speculative processes.

The non-speculative team represents the worst-case performance or the critical path. If all speculation fails, it sequentially executes the program. As we will explain in the next part, the overhead for the lead process only consists of the page-based write monitoring for the first *PPR* instance. The understudy runs as the original code without any monitoring. As a result, if the granularity of *PPR* instance is large or when the speculation depth is high, the worst-case running time should be almost identical to that of the unmodified sequential execution. On the other hand, whenever the speculation finishes faster than the understudy, it means a performance improvement over the would-be sequential execution.

The performance benefit of *understudy* comes at the cost of potentially redundant computation. However, the cost is at most one re-execution for each speculatively executed *PPR*, regardless of the depth of the speculation.

With the understudy, the worst-case parallel running time is mostly equal to the sequential time. One may argue that this can be easily done by running the sequential version side by side in a sequential-parallel race. The difference is that the *BOP* system is running a *relay race* for every group of *PPR* instances. At the whole-program level it is sequential-parallel collaboration rather than competition because the winner of each relay joins together to make the fastest time. Every time counts when speculation runs faster, and no penalty when it runs slower. In addition, the parallel run shares read-only data in cache and memory, while multiple sequential runs do not. Finally, running two instances of a program is not always possible for a utility program, since the communication with the outside world often cannot be undone. In *BOP*, unrecoverable I/O and system calls are placed outside the parallel region.

2.2.3 Expecting the Unexpected

Figure 3 shows the expected behavior when an execution of PPR runs from *BeginPPR* to *EndPPR*. In general, the execution may reach an exit (normal or abnormal) or an unexpected *PPR* marker. Table 1 shows the actions of the three types of processes when they encounter an exit, error, or unexpected *PPR* markers.

The abort by *spec* in Table 1 is conservative. For example, speculation may correctly hit a normal exit, so an alternative scheme may delay the abort and salvage the work if it turns out correct. We favor the conservative design for performance. Although it may recompute useful work, the checking and commit cost cannot delay the critical path.

The speculation process may allocate an excessive amount of memory and attempt unrecoverable operations such as I/O, other OS calls, or user interactions. The speculation is aborted upon file reads, system calls, and memory allocation over a threshold. The file output is buffered and is either written out or discarded at the commit point. Additional engineering can support regular file I/O. The current implementation supports stdout and stderr for the pragmatic purpose of debugging and verifying the output.

2.2.4 Strong Isolation

We say that *BOP* uses *strong isolation* because the intermediate results of the lead process are not made visible to speculation processes until the lead process finishes the first *PPR*. Strong isolation comes naturally with process-based protection. It is a basic difference between *BOP* and thread-based systems, where the updates of one thread are visible to other threads. We call it *weak isolation*. We discuss the control aspect of the difference here and complete the rest of comparisons in Section 2.5 after we describe the data protection.

Weak isolation allows opportunistic parallelism between two dependent threads, if the source of the dependence happens to be executed before the sink. In the *BOP* system, such parallelism can be made explicit and deterministic using *PPR* directives by placing dependent operations outside the *PPR* region (for example, in Figure 3, the code outside *PPR* executes sequentially). At the loop level, the most common dependence comes from the update of the loop index variable. With *PPR*, the loop control can be easily excluded from the parallel region and the pipelined parallelism is definite instead of opportunistic.

The second difference is that strong isolation does not need synchronization during the parallel execution but weak isolation needs to synchronize between the lead and the spec processes when communicating the updates between the two. Since the synchronization delays the non-speculative execution, it adds visible overheads (when speculation fails) to the thread-based systems but not to *BOP*.

Although strong isolation delays data updates, it detects speculation failure and success before the speculation ends. Like systems with weak isolation, strong isolation detects conflicts as they happen because all access maps are visible to all processes for reads (each process can only update its own map during the parallel execution). After the first *PPR*, strong isolation can check for correctness before the next speculation finishes by stopping the speculation, checking for conflicts, and communicating data updates. As a design choice, *BOP* does not abort speculation early because of the property of pipelined parallelism, explained at the end of Section 2.2.1. The speculation process, no matter how slow, may improve the program speed, when enough of them work together.

2.3 Checking for Correctness

The *BOP* system guarantees that the same result is produced as in the sequential execution if the speculation succeeds. It partitions the address space of a running program into three disjoint groups: shared, checked, and private. More formally, we say $D_{all} = D_{shared} + D_{checked} + D_{private}$, and any two of D_{shared}, $D_{checked}$, and $D_{private}$ do not overlap.

For the following discussion we consider two concurrent processes—the *lead* process that executes the current *PPR* instance, and the spec process that executes the next *PPR* instance and the code in between. The cases for k ($k > 1$) speculation processes can be proved inductively since they commit in a sequence in the *BOP* system.

2.3.1 Three types of data protection

Page-based protection of shared data All program data are shared at *BeginPPR* by default and protected at page granularity. During execution, the system records the location and size of all global variables and the range of dynamic memory allocation. At *BeginPPR*, the system turns off write permission for the lead process and read/write permission for the spec processes. It installs customized page-fault handlers that raise the permission to read or write upon the first read or write access. At the same time, the handler records which page has what type of access by which process. At the commit time, each spec process is checked in an increasing order, the kth process fails if and only if a page is written by the lead process and the previous $k-1$ spec processes but read by spec k. If speculation succeeds, the modified pages are merged into a single address space at the commit point.

```
shared = GetTable();
...
while (...) {
 ...
 BeginPPR(1)
 ...
 if (...)
  checked = checked + Search(shared, x)
  Insert(private, new Node(checked))
 ...
 if (!error) Reset(checked)
 ...
 EndPPR(1)
 ...
}
```

Figure 4. Examples of shared, checked, and private data

By using Unix processes for speculation, the *BOP* system eliminates all anti- and output dependences through the replication of the address space and detects true dependences at run-time. An example is the variable $shared$ in Figure 4. It may point to some large dictionary data structures. Page-based protection allows concurrent executions as long as a later *PPR* does not need the entries produced by a previous *PPR*.

The condition is significantly weaker than the Bernstein condition [3], which requires that no two concurrent computations access the same data if at least one of the two writes to it. The additional parallelism is due to the replication of modified data, which removes anti- and output dependences. For example, the write access by spec k to a page does not invalidate earlier spec processes that read concurrently from the same (logical) page.

Page-based protection has been widely used for supporting distributed shared memory [22,23] and many other purposes including race detection [28]. While these systems enforce parallel consistency among concurrent computations, the *BOP* system checks for dependence violation when running a sequential program.

A common problem in page-level protection is false sharing. We alleviate the problem by allocating each global variable on its own page(s). Writes to different parts of a page may be detected by checking the difference [22] at the end of *PPR*. In addition, the shared data is never mixed with checked and private data (to be described next) on the same page, although at run time newly allocated heap data are private at first and then converted to shared data at *EndPPR*.

Value-based checking Dependence checking is based on data access not data value. It is sufficient but not necessary for correctness. Consider the variable $checked$ in Figure 4, which causes true dependences as both the current and next *PPR* instances may read and modify it. On the other hand, the reset statement at the end may reinstall the old value as $checked$ had at the beginning. The parallel execution is still correct despite of the true dependence violation. This case is called a *silent dependence* [31].

There is often no guarantee that the value of a variable is reset by *EndPPR*. In the example, the reset depends on a flag, so the "silence" is conditional. Even after a reset, the value may be modified by pointer indirection. Finally, the rest operation may assign different values at different times. Hence run-time checking is necessary.

For global variables, the size is statically known, so the *BOP* system allocates checked variables in a contiguous region, makes a copy of their value at the *BeginPPR* of the lead process, and checks their value at the *EndPPR*. For dynamic data, the system needs to know the range of addresses and performs the same checking steps.

Checked data are found through profiling analysis or identified by a user (described more in Section 2.6). Since the values are checked, incorrect hints would not compromise correctness. In addition, a checked variable does not have to return to its initial value in every *PPR* instance. Speculation still benefits if the value remains constant for just two consecutive *PPR* instances.

Most silent dependences come from implicit re-initialization. Some examples are that the loop level increments and decrements when a compiler compiles a function, the traversed bits of the objects in a graph are set and reset during a depth-first search, and the work-list is filled and emptied in a scheduling pass. We classify these variables as *checked data*, which may take the same value at *BeginPPR* and *EndPPR*, in other words, the *PPR* execution may have no visible effect on the variable.

The shared data and checked data have a significant overlap, which are the data that are either read only or untouched by the parallel processes. We classify them as checked if their size is small; otherwise, they are shared. A problem is when different parts of a structure or an array require different protection schemes. Structure splitting, when possible, may alleviate the problem.

The correctness of checked data is not obvious because their intermediate values may be used to compute other values that are not checked. Section 2.4 presents a formal proof of the correctness to show how the three protection schemes work together to cast a complete shield against concurrency errors.

Likely private data The third group is private data, which is initialized before being used and therefore causes no conflict. In Figure 4, if *private* is always initialized before it is used, the access in the current *PPR* cannot affect the result of the next *PPR*, so any true dependence cause by it can be ignored.

Private data come from three sources. The first is the program stack, which includes local variables that are either read-only in the PPR or always initialized before use. Dataflow analysis can identify privatizable variables [4, 16]. When the two conditions cannot be guaranteed by compiler analysis, for example, due to unknown control flow or the address of a local variable escaping into the program heap, we can redefine the local variable to be a global variable and classify it as shared data. For recursive functions, we can either use a stack of pages or disable the *PPR*.

The second source is global variables and arrays that are always initialized before the use in the PPR. The standard technique to detect this is inter-procedural kill analysis [1]. In general, a compiler may not always ascertain all cases of initialization. For global data whose access is statically known in a program, the compiler automatically inserts calls after the initialization assignment or loop to classify the data as private at run time. Any access by the speculation process before the initialization causes it to be treated as shared data. For (non-aggregate) data that may be accessed by pointers, the system places it on a single page and treats it as shared until the first access. Additionally, we allow the user to specify the list of variables that are known to be written before read in PPR. These variables are reinitialized to zero at the start of a *PPR* instance. Since we cannot guarantee write-first access in all cases, we call this group *likely private data*.

The third type of private date is newly allocated data in a PPR instance. Before *BeginPPR*, the lead process reserves regions of memory for speculation processes. Speculation would abort if it allocates more than the capacity of the region. The main process does not allocate into the region, so at *EndPPR*, its newly allocated data can be merged with the data from the speculation process. For programs that use garbage collection, we encapsulate the heap region of spec processes, which we will describe when discussing the test of a lisp interpreter. Another solution is to ignore GC, which, if happens during a *PPR* instance, will cause speculation to fail because of the many changes it makes to the shared data.

Overheads on the critical path The three data protection schemes are summarized and compared in Table 2. We now discuss their overheads. Most speculation costs—the forking of speculation processes, the change of protection, data replication and read and write monitoring, the checking of access maps for conflicts, the merging of modified pages, and the competition between the understudy and the spec processes—are off the critical path. Therefore, the relation between the worst-case running time $T^{max}_{parallel}$ and the time of unmodified sequential program T_{seq} is

$$T^{max}_{parallel} = T_{seq} + c_1 * (S_{shared}/S_{page}) + c_2 * (S_{modified\ by\ 1st\ ppr} + S_{checked})$$

The two terms after T_{seq} are the cost from data monitoring and copying on the critical path, as we explain next.

For monitoring, at the start of *PPR*, the lead process needs to set and reset the write protection and the access map for shared data (including global variables) before and after the first *PPR* instance. The number of pages is the size of shared data S_{shared} divided by the page size S_{page}, and the cost per page is a constant c_1. During the instance, a write page fault is incurred for every page of shared data modified in the first *PPR* instance. The constant per page cost is negligible compared to the cost of copying a modified page.

Two types of copying costs may appear on the critical path. The first is for pages of shared data modified by the lead process in the first *PPR* instance and (among those) pages modified again by the understudy. The second is taking the snapshot of checked data. The cost in the above formula is the worst case. The copy-on-write mechanism in modern OS may hide most of both costs.

Data copying may hurt locality across *PPR* boundaries, although the locality within is preserved. The footprint of a speculative run is larger than the sequential run as modified data are replicated. However, the read-only data is shared by all processes in main memory and in shared cache (that is physically indexed). As a result, the footprint may be much smaller than running k copies of a program.

2.4 The Correctness Proof

It is sufficient to prove the correctness for a single instance of the parallel execution between two *PPR* instances. We first define an abstract model of an execution.

memory V_x: a set of variables. V_{all} represents all variables in memory.

memory state S^t_V: the content of V at time t. For ease of reading we use S^t_x (rather than $S^t_{V_x}$) to denote the state of V_x at t.

instruction r_x: the instructions we consider are the markers of the two *PPR*s, P and Q, P^b, P^e, Q^b, and Q^e (corresponding to m^b_P, m^e_P, m^b_Q, and m^e_Q in Section 2.1). P and Q can be the same region.

execution state (r_x, S^t_V): a point in execution where the current instruction is r_x and the state is S^t_V.

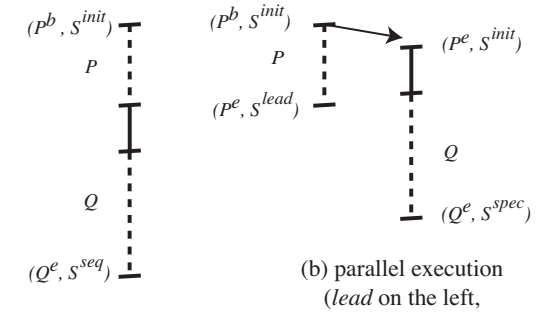
execution $(r_1, S^{t1}_{all}) \xRightarrow{p} (r_2, S^{t2}_{all})$: a continuous execution of a process p (which can be either seq, lead or spec) from instruction r_1 and state S^{t1}_{all} to the next occurrence of r_2 at the state S^{t2}_{all}.

Figure 5 shows the parallel execution and the states of the lead and the spec processes at different times. If a parallel execution passes the three data protection schemes, all program variables in our abstract model can be partitioned into the following categories:

- V_{wf}: variables whose first access by spec is a write. wf stands for write first.
- V_{excl_lead}: variables accessed only by lead when executing the first *PPR* instance P.

Table 2. Three types of data protection

type	shared data D_{shared}	checked data $D_{checked}$	(likely) private data $D_{private}$
protection	Not written by *lead* and read by *spec*	Value at *BeginPPR* is the same at *EndPPR* in *lead*. Concurrent read/write allowed.	no read before 1st write in *spec*. Concurrent read/write allowed.
granularity	page/element	element	element
needed support	compiler, profiler, run-time	compiler, profiler, run-time	compiler (run-time)
overhead on critical path	1 fault per mod. page copy-on-write	copy-on-write	copy-on-write



(a) sequential execution

(b) parallel execution (*lead* on the left, *spec* on the right)

Figure 5. The states of the sequential and parallel execution

- V_{excl_spec}: variables accessed only by spec.
- V_{chk}: the remaining variables. *chk* stands for checked.

$$V_{chk} = V_{all} - V_{wf} - V_{excl\_lead} - V_{excl\_spec}$$

Examining Table 2, we see that D_{shared} contains data that are either accessed by only one process (V_{excl_lead} and V_{excl_spec}), written before read in *spec* (V_{wf}), read only in both processes or not accessed by either (V_{chk}). $D_{private}$ contains data either in V_{wf} or V_{chk}. $D_{checked}$ is a subset of V_{chk}. In addition, the following two conditions are met upon a successful speculation.

1. the lead process reaches the end of P at P^e, and the spec process, after leaving P^e, executes the two markers of Q, Q^b and then Q^e.
2. the state of V_{chk} is the same at the two ends of P (but it may change in the middle), that is, $S^{init}_{chk} = S^{lead}_{chk}$.

To analyze correctness, we examine the states of the sequential execution, S^{init} at P^b and S^{seq} at Q^e of the sequential process seq, and the states of the parallel execution, S^{init} at P^b, S^{lead} at P^e of the lead process and S^{init} at P^e and S^{spec} at Q^e of the spec process. These states are illustrated in Figure 5.

The concluding state of the parallel execution, $S^{parallel}$ at Q^e, is a combination of S^{lead} and S^{spec} after the successful speculation. To be exact, the merging step copies the modified pages from the lead process to the spec process, so

$$S^{parallel} = S^{spec}_{all-excl\_lead} + S^{lead}_{excl\_lead}$$

In the following proof, we define each operation r_t by its inputs and outputs. All inputs occur before any output. The inputs are the read set $R(r_t)$. The outputs include the write set $W(r_t)$ and the next instruction to execute, r_{t+1} [1].

[1] An operation is an instance of a program instruction. For the simplicity of the presentation, we overload the symbol r_x as both the static instruction and its dynamic instances. To distinguish in the text, we call the former an *instruction* and the latter an *operation*, so we may have only one instruction r_x but any number of operations r_x.

THEOREM 1 (Correctness). *If the spec process reaches the end marker of Q, and the protection in Table 2 passes, the speculation is correct, because the sequential execution would also reach Q^e with a state $S^{seq} = S^{parallel}$, assuming that both the sequential and the parallel executions start with the same state, S^{init} at P^b.*

Proof Consider the speculative execution, $(P^e, S^{init}) \stackrel{spec}{\Longrightarrow} (Q^e, S^{spec})$, for the part of the sequential execution, $(P^e, S^{mid}) \stackrel{seq}{\Longrightarrow} (Q^e, S^{seq})$. We denote the correct sequential execution as $p_e, r_1, r_2, \cdots$ and the speculative execution as $p_e, r'_1, r'_2, \cdots$. We prove by contradiction that every operation r'_t in the speculative execution must be "identical" to r_t in the sequential execution in the sense that r_t and r'_t are the same instruction, they read and write the same variables with the same values, and they move next to the same instruction r_{t+1}.

Assume the two sequences are not identical and let r'_t be the *first* instruction that produces a different value than r_t, either by modifying a different variable, the same variable with a different value, or moving next to a different instruction. Since r_t and r'_t are the same instruction, the difference in output must be due to a difference in the input.

Suppose r_t and r'_t read a variable v but see different values v and v'. Since the values cannot differ if the last writes do not exist, let r_v and r'_v be the previous write operations that produce v and v'.

The operation r'_v can happen either in spec before r'_t or in the lead process as the last write to v. We show neither of the two cases is possible. First, if r'_v happens in spec, then it must produce the same output as r_v as per our assumption that r'_t is the first to deviate. Second, r'_v is part of lead and produces a value not visible to spec. Consider the only way v can be accessed. Since (r'_v is the last write so) v is read before being modified in spec, it does not belong to V_{wf} or V_{excl_lead}. Neither is it in V_{excl_spec} since it is modified in the lead process. The only case left is for v to belong to V_{chk}. Since $V^{lead}_{chk} = V^{init}_{chk}$, after the last write the value of v is restored to the beginning state where spec starts and consequently cannot cause r'_t in spec to see a different value as r_t does in the sequential run. Therefore r_t and r'_t cannot have different inputs and produce different outputs, and the speculative and sequential executions must be identical.

We now show that $S^{parallel}$ is correct, that is, $S^{parallel} = S^{seq}$. Since spec reads and writes correct values, V_{wf}, V_{excl_spec}, and the accessed part of V_{chk} are correct. V_{excl_lead} is also correct because of the copying of the their values at the commit time. The remaining part of V_{chk} is not accessed by lead or spec and still holds the same value as S^{init}. It follows that the two states $S^{parallel}$ and S^{seq} are identical. ∎

The proof is similar to that of the Fundamental Theorem of Dependence(Sec. 2.2.3 in [1]). While the proof in the book deals with statement reordering, the proof here deals with region reordering and value-based checking. It rules out two common concerns. First, the intermediate values of checked data never lead to incorrect results in unchecked data. Second, the data protection always ensures the correct control flow by speculation. In *BOP*, the three checking schemes work together to ensure these strong guarantees.

Table 3. Comparisons between strong and weak isolation

during speculation	strong	weak
data updates visible to outside	no	yes
overall overhead proportional to	data size	data use
synchronization on critical path	none	needed
hardware memory consistency	independent	dependent
support value-based checking	yes	no
detect spec failure early	yes	yes
can certify spec success early	yes	yes

2.5 Comparisons with Thread-based Systems

Strong and weak isolation as discussed in Section 2.2.4 is a basic difference between process-based *BOP* and thread-based systems that include most hardware and software speculation and transactional memory techniques. The previous section discussed the control aspect. Here we discuss the data protection and system implementation. The comparisons are summarized in Table 3.

Weak isolation needs concurrent access to both program data and system data. It needs synchronization to eliminate race conditions between parallel threads and between the program and the run-time system. The problem is complicated if the hardware uses weak memory consistency, which does not guarantee correct results without explicit synchronization, if the memory operations are reordered by the compiler and the hardware. In fact, concurrent threads lack a well-defined memory model [5]. A recent loop-level speculation system avoids race conditions and reduces the number of critical sections (to 1) by carefully ordering the system code based on a sequential memory consistency model and adding memory directives to enforce the order under relaxed consistency models [9].

In *BOP*, parallel processes are logically separated. The correctness check is done sequentially in rolling commits with a complete guarantee as stated in Theorem 1. There is no synchronization overhead on the critical path. The compiler and hardware are free to reorder program operations as they do for a sequential program.

Thread-based systems do not yet support general value-based checking. When data updates are visible, the intermediate value of a checked variable can be seen by a concurrent thread and the effect cannot be easily undone even if the variable resumes the initial value afterwards. This is widely known as the ABA problem, where a thread may be mistakenly holding different data by the same pointer (see [18] for a solution). In hardware, a correct value prediction may cause a thread to read at a wrong time and violate the sequential consistency, so value prediction requires careful extra tracking by hardware [24]. No software speculation systems we know use value-based checking. With strong isolation in *BOP*, the intermediate values of checked variables have no effect on other processes, so value-based checking is not only correct but also adds little cost on the critical path.

Value-based checking is different from value-specific dynamic compilation (for example in DyC [13]), which finds values that are constant for a region of the code rather than values that are the same at specific points of an execution (and can change arbitrarily between these points). It is different from a *silent write*, which writes the same value as the previous write to the variable. Our software checking happens once per *PPR* for a global set of data, and the correctness is independent of the memory consistency model of the hardware.

Most previous techniques monitor data at the granularity of array elements, objects, and cache blocks; *BOP* uses pages for heap data and padded variables for global data. Paging support is more efficient for monitoring unknown data structures but it takes more time to set up the permissions. It also gives rise to false sharing.

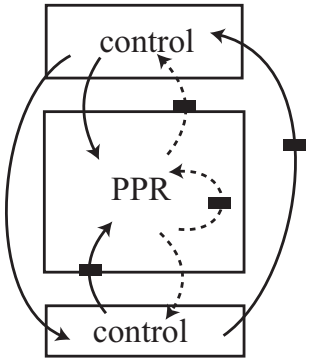


Figure 6. An illustration of the parallelism analysis. Region-carried dependences, shown as dotted edges, cause speculation to fail. Loop-carried dependences are marked with a bar.

The cost of page-based monitoring is proportional to the size of accessed data (for the overhead on the critical path it is the size of modified data) rather than the number of accesses as in thread-based systems, making page-based protection especially suitable for coarse-grain parallelism.

2.6 Programming using PPR

2.6.1 The Profiling Analysis

Offline profiling analysis first identifies the high-level phase structure of a program [32,33]. Then it uses dependence profiling to find the phase with the largest portion of run-time instructions that can be executed in parallel as the *PPR*. At the same time, it classifies program data into shared, checked and private categories based on their behavior in the profiling run.

For example consider a *PPR* within a loop. We call the rest of the loop the *control code*. Figure 6 shows an example *PPR* region. For each possible region during a training run, an instance is parallelizable if there is no dependence that originates from this instance region to later regions or later instances of the same region. Figure 6 shows most types of dependences. While not all types are shown for simplicity, all three types of region-carried dependences, which would prohibit parallelization, are shown as dotted edges. Loop-carried dependences, marked by a bar, are not the same as region-carried dependences. In addition to the parallelism, we need to analyze the size of possible *PPR* instances. The parallelism of a region is then measured by the total size of its parallel instances. We want to find the largest of possible parallel regions.

A brute force method would test all possible regions, which is $\binom{n}{2} = \frac{n(n-1)}{2}$ for a loop body with n statements. Alternatively we can use a graph model. We define each statement in the loop body as an elementary region. We construct a graph in which a node represents an elementary region, and a directed edge exists between two nodes if they have a dependence. We first find strongly connected components and collapse each into a single node. A node cannot belong to *PPR* if it has a loop-carried dependence. The largest *PPR* is the set of nodes that represent a continuous region that has no outgoing dependences. The time complexity is linear to the size of the graph, which in the worst case is the same as complete enumeration.

Phase analysis identifies coarse-grain tasks such as locality phases in scientific programs [33] and variable-length phases in compilers, interpreters and data transcoding programs [32]. The analysis problem is similar to the one illustrated in Figure 6 except that the boundary of possible regions are phase markers. Each region is continuous at run time even though the code for it may

not be contiguous in the program. In fact, the same program code may belong to both *PPR* and control.

2.6.2 The Programming Interface

BOP can also be invoked manually. The programming interface has three parts. The first is the *PPR* markers, which can be practically peppered anywhere in any quantity in a program. The second is a list of global and static variables that are write first (privatizable) and checked. The programmer specifies the place where the variables are initialized, and the system treats the data as shared until the initialization.

The third component of the interface is the run-time feedback to the user. When speculation fails, the system outputs the cause of the failure, in particular, the memory page that receives conflicting accesses. In our current implementation, global variables are placed on separate memory pages by the compiler. As a result, the system can output the exact name of the global variable when it causes a conflict. A user can then examine the code and remove the conflict by marking the variable privatizable or moving the dependence out of the parallel region.

Three features of the API are especially useful for working with large, unfamiliar code. First, the user does not write a parallel program and never needs parallel debugging. Second, the user parallelizes a program step by step as hidden dependences are discovered and removed one by one. Finally, the user can parallelize a program for a subset of inputs rather than all inputs. The program can run in parallel even if it has latent dependences.

3. Evaluation

3.1 Implementation and Experimental Setup

We have implemented the compiler support in Gcc 4.0.1. After high-level program optimization passes but before machine code generation, the compiler converts global variables to use dynamic allocation for proper protection. We did not implement the compiler analysis for local variables. Instead the system privatizes all stack data. All global and heap data are protected. Each global variable is allocated on separate page(s) to reduce false sharing. We implemented similar systems using two binary instrumentors, which do not require program source but offer no easy way of relocating global data, tracking register dependences, or finding the cause of conflicts at the source level.

The *BOP* run-time is implemented as a statically linked library. For lack of space, we only include a part of the pseudo-code in Figure 7. It shows the basic steps taken when a process reaches *EndPPR*. Partial commit or early error detection requires further refinements. The listed code has three key comments in italics. The first two mark the finish line of the sequential-parallel race. The third comment marks the important fall-through case of the if statement, where a speculation process, after passing the correctness check, implicitly becomes the lead process and continues the rolling commit with the next speculation process. Note that the commit procedure also includes actions invoked by signals, which are not shown in the pseudo-code. For example, the concede signal from the understudy will cause all except the new lead process to exit. The implementation uses shared memory for storing snapshots, access maps, and for copying data at a commit. The rest of communication is done by signals. No locks are used.

We have implemented an instrumentor and a behavior analyzer. The instrumentor, also based on Gcc 4.0.1, collects complete program traces with unique identifiers for instructions, data accesses, and memory and register variables, so the behavior analyzer can track all data dependences and identify *PPR*.

In *BOP*, the original lead process may die long before the program ends, since each successful speculation produces a new lead

```
subroutine BOP_EndPPR():
switch (myStatus)
case understudy:
  undyWorkCount++
  if (undyWorkCount<specDepth) return
  the finish line of the sequential execution
  myStatus = lead
  undyWorkCount = 0
  AbortSpeculation()
  CommitOutput()
  return
case spec:
  WaitForLeadDoneSignal()
  SendSpecDoneSignal()
  if (FoundConflicts())
    AbortSpeculation()
  if (lastSpec)
    ClearProtection()
    PullChangedDataFromPipe()
    WaitEnsureUnderstudyCreated()
    SignalUnderstudySpecSuccess()
    WaitForUnderstudyToConcede()
    the finish line of the parallel execution
    myStatus = lead
    CommitOutput()
    return
  else fall through
case lead:
  if (myStatus == lead)
    ClearProtection()
    StartUnderstudy()
  SignalLeadDone()
  WaitSpecDoneSignal()
  PushChangedDataToPipe()
```

Figure 7. Pseudo-code executed at the end of a *PPR* region

(for example in Figure 3). For one moment we thought we had an incredible speedup. Now each parallelized program starts with a timing process that forks the first lead process and waits until the last process is over (when a lead process hits a program exit). Instead of counting the time for all processes, we use the wall-clock time of the timing process, which includes the OS overheads. We use multiple runs on an unloaded system.

We use GNU Gcc 4.0.1 with "-O3" flag for all programs. We use a new Dell PowerEdge 6850 machine (after installing a 250V power supply) with four dual-core Intel 3.40 GHz Xeon 7140M processors (a total of 8 CPUs), 16MB cache, and 4GB physical memory.

3.2 Gzip v1.2.4 by J. Gailly

Gzip takes one or more files as input and compresses them one by one using the Lempel-Ziv coding algorithm (LZ77). We use version 1.2.4 available from the SPEC 2000 benchmark suite. Much of the 8616-line C code performs bit-level operations, some through inline assembly. The kernel was based on an earlier implementation on 16-bit machines. We did not specify "spec" so the program behaves as a normal compressor rather than a benchmark program (which artificially lengthens the input by replication).

We make two parallel versions, one by automatic methods, and the other by hand. In the first one, *BeginPPR* and *EndPPR* are automatically inserted before reading a file and after the output of the compressed file (for this one we allow file I/O in the PPR), and the variables and allocation sites are classified through profiling analysis.

The second parallel version compresses a single file in parallel. The sequential code compresses one buffer at a time and stores the results until the output buffer is full. We manually placed PPR around the buffer loop and specified the set of likely private variables through the program interface described in Section 2.6. The program returned correct results but speculation failed because of conflicts caused by two variables, "unsigned short bi_buf" and "int bi_valid", as detected by the run-time monitoring.

The two variables are used in only three short functions. Inspecting the code, we realized that the compression algorithm produced bits, not bytes, and the two variables stored the partial byte of the last buffer. The dependence was hidden below layers of code and among 104 global variables, but the run-time analyzer enabled us to quickly uncover the hidden dependence. We first tried to fill the byte, as the program does for the last byte. However, the result file could not be decompressed. In fact, a single extra or error bit would render the output file meaningless to the decompressor. Our second solution is to compress buffers in parallel and concatenate the compressed bits afterwards. This requires tens of lines of coding, but it was sequential programming and was done by one of the authors in one day.

Inter-file parallel gzip performs similarly as one may get from invoking multiple gzip programs. The intra-file compression permits single-file compression to use multiple processors. We test *bop-gzip* on a single 84MB file (the Gcc4.0.1 tar file) and compares the running times of the unmodified sequential code and the *BOP* version with three speculation depths. Each *PPR* instance compresses about 10MB of the input file. The execution time is stable in sequential runs but varies by as much as 67% in parallel runs, so in the following table we include the result of six consecutive tests of each version and compute the speedup based on the average time.

version	sequen-	speculation depth		
	tial	1	3	7
times (sec)	8.46, 8.56,	7.29, 7.71	5.38, 5.49,	4.80, 4.47,
	8.50, 8.51	7.32, 7.47	4.16, 5.71	4.49, 3.10
	8.53, 8.48	5.70, 7.02	5.33, 5.56	2.88, 4.88
avg time	8.51	7.09	5.27	4.10
avg speedup	1.00	1.20	1.61	2.08

With 2, 4, and 8 processors, the parallel compression gains speedups of 1.20, 1.61, and 2.08. The 8-way *gzip* is twice as fast. The compression is now slightly faster than decompression (by *gunzip*), which is unusual for such a tool. The critical path of *bop-gzip*, when all speculation fails, runs slightly faster than the sequential version because of the effect of prefetching by the speculation.

The *BOP* run-time allocates 138 4KB pages for the 104 global variables, 19200 pages for shared heap data, and 9701 pages for likely write-first data. The commit operations copy a total of 2597, 3912, and 4504 pages respectively during the parallel execution for the three speculation depths.

3.3 Sleator-Temperley Link Parser v2.1

According to the Spec2K web site, "The parser has a dictionary of about 60000 word forms. It has coverage of a wide variety of syntactic constructions, including many rare and idiomatic ones. ... It is able to handle unknown vocabulary, and make intelligent guesses from context about the syntactic categories of unknown words." It is not clear in the documentation or the 11,391 lines of its C code whether the parsing of sentences can be done in parallel. In fact, they are not. If a PPR instance parses a command sentence which changes the parsing environment, e.g. turning on or off the echo mode, the next PPR instance cannot be speculatively executed. This is a typical example of dynamic parallelism.

The parallelism analyzer identifies the sentence-parsing loop. We manually strip-mine the loop (for a recent textbook description see [1]) to create a larger PPR. The data are then classified automatically. During the training run, 16 variables are always written first by the speculation process during training, 117 variables always have the same value at the two ends of a *PPR* instance, and 35 variables are shared. The system allocates 132 pages of shared data and 75 pages private and checked data.

We test the parallel parser using 1022 sentences obtained by replicating SPEC95 train input twice. When each *PPR* includes the parsing of 10 sentences, the sequential run takes 11.34 second, and the parallel runs show speedup of 1.13, 1.62 and 2.12 with a few failed speculations due to the dynamic parallelism. We have studied the parallel performance as a function of the *PPR* size (i.e. the number of sentences per *PPR*) but have to leave it out for lack of space. The total number of pages being copied during commits varies between 0 and 21 pages depending on the size of *PPR* and the depth of speculation.

version	sequen-	speculation depth		
	tial	1	3	7
times	11.35, 11.37	10.06, 10.06	7.03, 7.01	5.34, 5.35
(sec)	11.34	10.07	7.04	5.34
speedup	1.00	1.13	1.62	2.12

3.4 XLisp Interpreter v1.6 by D. M. Betz

According to its author in 1985, Xlisp is "a small implementation of lisp with object-oriented programming." We use the code available as part of the SPEC 1995 benchmark suite, which has 25 files and 7616 lines of C code. The main function has two control loops, one for reading expressions from the keyboard and the other for batch processing from a file. By hand we mark the body of the batch loop as a *PPR*. Through the programming interface described in Section 2.6, we identify 5 likely privatizable variables:

buf	for copying string constants
gsprefix	for generated name strings
xlfsize	for counting the string length in a print call
xlsample	the vestige of a deleted feature called oscheck
xltrace	intermediate results for debugging

5 checked variables:

xlstack	current stack pointer, restored after an evaluation
xlenv	current environment, restored after an evaluation
xlcontext	the setjump buffer for exception handling
xlvalue	would-be exception value
xlplevel	parenthesis nesting level, for command prompt

and one reduction variable, gccalls, which counts the number of garbage collections. We do not know much about the rest of the 87 global variables (including function pointers) except that they are all monitored by *BOP*.

The so-parallelized xlisp runs fine until a garbage collection, which changes the heap and always kills the speculation. To solve the problem, we have revised the xlisp mark-sweep collector for *BOP*, which we describe very briefly here. The key idea is to insulate the effect of GC, so it can be done concurrently without causing unnecessary conflicts. Each *PPR* uses a separate page-aligned region. At the start (after forking but before data protection), a *PPR* instance runs a marking pass over the entire heap and records all reachable objects in a start list. During the *PPR*, it allocates new objects inside the region. At a garbage collection, it marks just objects inside the region but it traverses the start list as an additional set of root pointers. It frees an object if it is inside the region. At the end, it performs GC again, so only the pages with live objects are copied at the commit. The code changes include three new global variables and 12 statements, counted by the number of semi-colons, for region-based GC, mostly for collecting and traversing the start list and resetting the MARK flags in its nodes.

The region-based mark-sweep has non-trivial costs at the start and the end of *PPR*. In the middle it may not be as efficient because it may not collect all garbage (as some nodes in the start list

would have become unreachable in the sequential run). These costs depend on the input. In addition, the regions will accumulate long-live data, which leads to more false alerts from false sharing. The evaluation may trigger an exception and an early exit from *PPR*, so the content of checked variables may not be restored even for parallel expressions. Therefore, one cannot decide a priori whether the chance of parallelism and its likely benefit would outweigh the overhead. However, these are the exact problems that *BOP* is designed to address with the streamlined critical path and the online sequential-parallel race.

To test the *bop-lisp* interpreter, we use an input from SPEC95, which in five expressions computes all positions of n queens on an $n \times n$ board. When n is 9, the sequential run takes 2.36 seconds using the base collector and 2.25 seconds using the region-based collector (which effectively has a larger heap but still needs over 4028 garbage collections for nine 10K-node regions). We modify four lines of the lisp program, so the problem is solved by 13 expressions, 9 parallel and 4 sequential. The dependence between sequential expressions is automatically detected due to the conflicts in the internal data structure of the interpreter. To avoid false alerts when executing the parallel expressions, we disable the monitoring of the heap in the following experiment. We test three speculation depths three times each, and the (idealized) results are:

version	sequen-	speculation depth		
	tial	1	3	7
times (sec)	2.25, 2.27, 2.26	1.50, 1.48 1.47	.95, .94, .94	.68, .68, .68
speedup	1.00	1.53	2.39	3.31

The last row shows that the speedup, if we pick the lowest time from three runs, is 1.53 with 2 processors, 2.39 with 4 processors, and 3.31 with 8 processors. Failed speculations account for 0.02 second of the parallel time. Compared to the other test programs, *bop-lisp* has a small memory footprint. In the test run, it allocates 103 pages for monitored global variables, 273 pages for the heap (currently not monitored to avoid the false alert), and 111 for other unmonitored data.

3.5 Comparison with Threaded Intel Math Kernel Library

The Intel Math Kernel Library 9.0 (MKL) provides highly optimized, processor-specific, and multi-threaded routines specifically for Intel processors. The library includes Linear Algebra Package (LAPACK) routines used for, among other things, solving systems of linear equations. In this experiment we compare the performance of solving eight independent systems of equations using the *dgesv* routine. MKL exploits thread-level parallelism inside but not across library calls. We set the number of threads using the OMP_NUM_THREADS environment variable. *BOP*, on the other hand, can speculatively solve the systems in parallel using the sequential version of the library (by setting OMP_NUM_THREADS to 1). Since the program data are protected, *BOP* monitors inter-system dependences (which may arise when the same array is passed to multiple calls) and guarantees program correctness if speculation succeeds. We compare the speed of speculative parallel invocations of unparallelized *dgesv* with the speed of sequential invocation of the hand-crafted parallel implementation.

The experiment was conducted over the range of 500 to 4500, in increments of 500, equations per system. For each, the number of threads in the MKL-only implementation tested was 1, 2, 4, and 8. For the BOP and MKL implementation, the levels of speculation tested was 0, 1, 3, and 7. To reduce the monitoring cost, the *BOP* version makes a copy of the matrices and pass the unmonitored temporary copies to *dgesv*. The memory usage includes 32 pages for global variables, 3920 to 316K pages for monitored matrices of coefficients, and the same amount for their unmonitored temporary twin. The number of copied pages during commits ranges from 0 to 107.

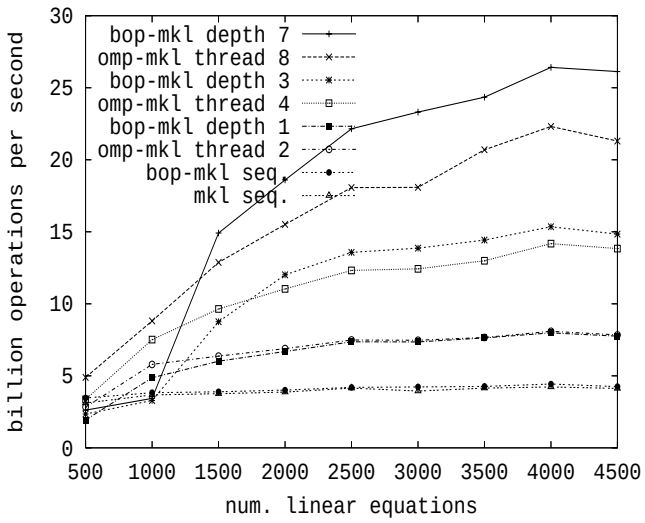


Figure 8. Solving 8 systems of linear equations with Intel MKL

The performance comparison is shown in Figure 8. Zero-speculation *BOP* is slightly faster than single-threaded MKL possibly due to the difference in memory allocation. *bop-mkl depth 1* and *omp-mkl thread 2* perform similarly, with the MKL-only implementation achieving at most an 18% increase in operations per second for 1000 equations. For *bop-mkl depth 3* and *bop-mkl depth 7*, the runtime overhead of the *BOP* system prevents speedups for roughly 1750 and 1300 number of equations, respectively, and below. However, above these ranges the course-grained parallelism provided by *BOP* is able to outperform the fine-grained, thread-level parallelism of the MKL library. Increases between 15% and 20% are seen for *bop-mkl depth 7* compared to *omp-mkl thread 8* and increases between 7% and 11% are seen for *bop-mkl depth 3* compared to *omp-mkl thread 4*.

The comparison with threaded MKL helps to understand the overhead of process-based *BOP*, in particular its relation with the size of parallel tasks and the speculation depth. The results demonstrate the property explained in Section 2.2.1: the overhead becomes less if the granularity is large or if the speculation depth is high. For 1500 equations, 3 speculation processes perform 10% slower than 4-thread MKL because of the overhead. However, for the same input size, the greater parallelism from 7 speculation processes, more than compensates for the overhead and produces an improvement of 16% over 8-thread MKL. We have also tested *BOP* against another scientific library, the threaded ATLAS, and found similar results, although ATLAS is slower than MKL on our test machine.

4. Related work

Parallel languages Most languages let a user specify the unit of parallel execution explicitly, for example, $pcall$ and $future$ in Multilisp [17], parallel loop and section in OpenMP [27], SIMD model of UPC [7] and Co-array Fortran [26]. They require definite parallelism enclosed in regions or procedures with predictable entries and exits. The annotations are binding because they affect the program correctness. In data parallel languages such as HPF [1] and array languages such as the publicly available Single-assignment C (SaC) [14], the parallelism is implicit but a user has limited control over the partition of program computation. In comparison, *PPR* re-

gions are not binding but allow a user to directly specify the unit of parallel execution. As such it allows computation partitioning based on the partial information about program behavior, for example, a user reading part of the source code, or a profiling tool examining a few inputs.

For general-purpose imperative programs, the synchronization is either static (e.g. in OpenMP) or dynamic based on the run-time access of the shared data. The access can be specified as programmed (and typed) descriptions as in Jade [30]. *BOP* assumes all data are shared by default and uses profiling analysis, so it needs little or no user specification. In addition, the checked and private data are suggested through non-binding hints, which allow partial and incorrect specifications. The disadvantage is the additional cost of data protection and hint checking, which *BOP* hides on the speculative path.

Software speculation Automatic loop-level software speculation is pioneered by the lazy privatizing *doall* (LPD) test [29]. LPD has two separate phases: the marking phase executes the loop and records access to shared arrays in a set of shadow arrays, and the analysis phase then checks for dependence between any two iterations. Later techniques speculatively privatize shared arrays (to allow for false dependences) and combine the marking and checking phases (to guarantee progress) [9,11,15]. Previous systems also address issues of parallel reduction [15,29] and different strategies of loop scheduling [9]. A weaker type of software speculation is used for disk prefetching, where only the data access of the speculation needs to be protected (through compiler-inserted checks) [6]. We observe the prefetching effect in one of our tests, *gzip*.

Recently two programmable systems are developed: *safe future* in Java [38] and *ordered transactions* in X10 [36]. The first is designed for (type-safe) Java programs and is supported entirely in software, utilizing read/write barriers of the virtual machine to monitor data access, object replication to eliminate false dependences, and byte-code rewriting to save program states. At the syntax level, the difference between procedure-based future and region-based *PPR* is superficial but as a programming construct, they differ in two important aspects. First, *PPR* supports unsafe languages with unconstrained control flows. Second, the commit point is implicit in *PPR* but explicit in future (i.e. *get*). Ordered transactions rely on hardware transactional memory support for efficiency and correctness but provide profiling analysis to identify parallel regions. A user manually classifies program data into five categories. It does not use value-based checking, nor does the system checks the correctness of the data classification. Our profiling analysis is similar to theirs in purpose but uses phase analysis to identify program-level regions.

At the implementation level, the *BOP* system is unique in that the speculation overhead is proportional to the size of program data, while the cost of other systems, including the transactional memory, is proportional to the frequency of data access. Among software techniques, *BOP* is the first to speculatively privatize the entire address space and apply speculative execution beyond the traditional loop-level parallelism by address the problems of unpredictable code, task size, parallelism, and speculation overhead.

The main appeal of programmable speculation is that the specification of parallelism is purely a performance hint and has no effect on program correctness. The *BOP* system attempts to demonstrate that this new style of parallel programming can be efficiently supported for a more general class of applications.

Hardware thread-level speculation (TLS) Hardware-based thread-level speculation is among the first to automatically exploit loop- and method-level parallelism in integer code. An on-line survey cited 105 related papers [20]. An early effort is described in [34]. The term lead thread is used in [35]. Since speculative states are buffered in hardware, the size of threads is usually no more than thousands of instructions. A recent study classifies existing loop-level techniques as control, data, or value speculation and shows that the maximal speedup is 12% on average for SPEC2Kint assuming no speculation overhead and unlimited computing resources [21]. The limited potential at the loop level suggests that speculation needs to be applied at larger granularity to fully utilize multi-processor machines.

Software transactional memory Transactional memory allows the programmer to identify sections of a parallel program that must be executed atomically, and which the system may choose to speculatively execute in parallel [19]. Transaction semantics, which requires a serializable result, is less restrictive than parallelization, which requires observational equivalence or the same result as the original sequential execution. Like transactions, *PPR* regions do not guarantee parallelism. Unlike transactions, *PPR* regions do not affect the meaning of a program. Since incorrectly inserted regions do not break a program, *PPR* is easier to use for a user or a tool to parallelize an unfamiliar program.

At the implementation level, serializibility checking requires the monitoring of both data reads and writes, so it is more costly than the run-time dependence checking. The additional flexibility is useful for supporting parallel reduction, but it is not strictly necessary for parallelization as it is for concurrency problems such as on-line ticket booking. Current transactional memory systems monitor data accesses rather than values for conflict detection.

Run-time data monitoring For large programs using complex data, per-access monitoring causes slow-downs often in integer multiples, as reported for data breakpoints and on-the-fly data race detection, even after removing as many checks as possible by advanced compiler analysis [25, 28, 37]. It is difficult for dynamic speculation to afford such slowdown and be practical. Run-time sampling based on data [12] or code [2,8] are efficient but does not monitoring all program accesses. *BOP* uses page-based monitoring for shared data to trade precision for efficiency (without compromising correctness) and to bound the cost by the size of data rather than the frequency of access.

5. Summary

With programmable dynamic *PPR* regions, strong isolation during speculation, minimal critical path, and value-based correctness checking, *BOP* enables parallelization based on the partial information of program behavior. We have built a prototype implementation including a parallelism analyzer, a compiler, and a run-time system and have parallelized a set of non-trivial applications, most of them have not been parallelized (or known to be parallelizable) before this work. On a 8-CPU machine, their end-to-end speed is improved by integer factors.

BOP is best suited for parallelizing large, existing code with a minimal effort. Known dependences, such as error handling and garbage collection, can stay in code as long as they happen rarely. Parallelization can be done in incremental steps by removing dependences one by one as detected by the run-time feedbacks. At no point does a programmer need to perform parallel debugging.

Acknowledgment Kai Shen suggested letting the parent process exit as the way to implement the commit. Alok Garg participated in the pilot study of the BOP system. Xiao Zhang helped to collect the MKL results. The modified lisp program was named in celebration of Tuolumne Gildea. We thank Michael Scott and the anonymous referees for their comments on the earlier versions of the paper. The research is supported by the National Science Foundation (Contract No. CNS-0509270, CCR-0238176), an IBM CAS Fellowship, two grants from Microsoft Research.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, October 2001.

[2] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001.

[3] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, 1966.

[4] W. Blume et al. Parallel programming with polaris. *IEEE Computer*, 29(12):77–81, December 1996.

[5] H.-J. Boehm. Threads cannot be implemented as a library. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 261–268, 2005.

[6] F. W. Chang and G. A. Gibson. Automatic i/o hint generation through speculative execution. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 1999.

[7] W. Chen, C. Iancu, and K. Yelick. Communication optimizations for fine-grained UPC applications. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, St. Louis, MO, 2005.

[8] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[9] M. H. Cintra and D. R. Llanos. Design space exploration of a software speculative parallelization scheme. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):562–576, 2005.

[10] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, St. Charles, IL, August 1986.

[11] F. Dang, H. Yu, and L. Rauchwerger. The R-LRPD test: Speculative parallelization of partially parallel loops. Technical report, CS Dept., Texas A&M University, College Station, TX, 2002.

[12] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[13] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, Georgia, May 1999.

[14] C. Grelck and S.-B. Scholz. SAC—from high-level programming with arrays to efficient parallel execution. *Parallel Processing Letters*, 13(3):401–412, 2003.

[15] M. Gupta and R. Nim. Techniques for run-time parallelization of loops. In *Proceedings of SC'98*, 1998.

[16] M. Hall, S. Amarasinghe, B. Murphy, S. Liao, and M. Lam. Interprocedural parallelization analysis in SUIF. *ACM Trans. Program. Lang. Syst.*, 27(4):662–731, 2005.

[17] R. H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[18] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22th PODC*, pages 92–101, Boston, MA, July 2003.

[19] M. Herlihy and J. E. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Co mputer Architecture*, San Diego, CA, May 1993.

[20] A. Kejariwal and A. Nicolau. Reading list of performance analysis, speculative execution. http://www.ics.uci.edu/akejariw/SpeculativeExecutionReadingList.pdf.

[21] A. Kejariwal, X. Tian, W. Li, M. Girkar, S. Kozhukhov, H. Saito, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. On the performance potential of different types of speculative thread-level parallelism. In *Proceedings of ACM International Conference on Supercomputing*, June 2006.

[22] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter USENIX Conference*, 1994.

[23] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Dept. of Computer Science, Yale University, New Haven, CT, September 1986.

[24] M. K. Martin, D. J. Sorin, H. V. Cain, M. D. Hill, and M. H. Lipasti. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. In *Proceedings of the International Symposium on Microarchitecture (MICRO-34)*, 2001.

[25] J. Mellor-Crummey. Compile-time support for efficient data race detection in shared memory parallel programs. Technical Report CRPC-TR92232, Rice University, September 1992.

[26] R. W. Numrich and J. K. Reid. Co-array Fortran for parallel programming. *ACM Fortran Forum*, 17(2):1–31, August 1998.

[27] OpenMP application program interface, version 2.5, May 2005. http://www.openmp.org/drupal/mp-documents/spec25.pdf.

[28] D. Perkovic and P. J. Keleher. A protocol-centric approach to on-the-fly race detection. *IEEE Transactions on Parallel and Distributed Systems*, 11(10):1058–1072, 2000.

[29] L. Rauchwerger and D. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[30] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(3):483–545, 1998.

[31] X. Shen and C. Ding. Parallelization of utility programs based on behavior phase analysis. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, Hawthorne, NY, 2005. short paper.

[32] X. Shen, C. Ding, S. Dwarkadas, and M. L. Scott. Characterizing phases in service-oriented applications. Technical Report TR 848, Department of Computer Science, University of Rochester, November 2004.

[33] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the Eleventh International Conference on Architect ural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Boston, MA, 2004.

[34] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the International Symposium on Computer Architecture*, 1995.

[35] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, 2005.

[36] C. von Praun, L. Ceze, and C. Cascaval. Implicit parallelism with ordered transactions. In *Proceedings of the ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, March 2007.

[37] R. Wahbe, S. Lucco, and S. L. Graham. Practical data breakpoints: design and implementation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[38] A. Welc, S. Jagannathan, and A. L. Hosking. Safe futures for java. In *Proceedings of OOPSLA*, pages 439–453, 2005.