Exploiting Coarse-Grain Speculative Parallelism

Hari K. Pyla, Calvin Ribbens, Srinidhi Varadarajan

Center for High-End Computing Systems
Department of Computer Science
Virginia Tech
{harip, ribbens, srinidhi}@cs.vt.edu

Abstract

Speculative execution at coarse granularities (e.g., code-blocks, methods, algorithms) offers a promising programming model for exploiting parallelism on modern architectures. In this paper we present Anumita, a framework that includes programming constructs and a supporting runtime system to enable the use of coarse-grain speculation to improve program performance, without burdening the programmer with the complexity of creating, managing and retiring speculations. Speculations may be composed by specifying surrogate code blocks at any arbitrary granularity, which are then executed concurrently, with a single winner ultimately modifying program state. Anumita provides expressive semantics for winner selection that go beyond time to solution to include user-defined notions of *quality of solution*. Anumita can be used to improve the performance of hard to parallelize algorithms whose performance is highly dependent on input data. Anumita is implemented as a user-level runtime with programming interfaces to C, C++, Fortran and as an OpenMP extension. Performance results from several applications show the efficacy of using coarse-grain speculation to achieve (a) robustness when surrogates fail and (b) significant speedup over static algorithm choices.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; D.3.4 [*Programming Languages*]: Processors—Run-time environments

General Terms Algorithms, Design, Languages, Measurement and Performance

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.

Keywords Speculative Parallelism, Coarse-grain Speculation, Concurrent Programming and Runtime Systems

1. Introduction

As processor architectures evolve from fast single core designs to multi/many core designs using multiple simpler cores (lower clock frequency, shorter pipelines), there is increasing pressure on programmers to use application level threading to improve performance. While some applications are amenable to simple parallelization techniques, a large body of algorithms and applications are inherently hard to parallelize due to execution order constraints inflicted by data and control dependencies. Furthermore, for a significant number of applications, performance (a) is highly sensitive to input data and (b) does not scale well to 100's of cores.

Our objective is to provide programmers with a simple tool for exploiting parallelism in such applications. In the arsenal of concurrent programming techniques, speculative execution is used in a variety of contexts to improve performance. Low level fine-grain speculation employed by the hardware and compiler (e.g., branch prediction, prefetching) is a proven technique. Software transaction systems are premised on speculative execution of potentially coarse-grain code blocks. More generally, we believe speculative execution relying on optimistic concurrency at coarse granularities (e.g., code-blocks, methods, algorithms) offers a promising programming model for exploiting parallelism for many hard-to-parallelize applications on multi and many core architectures.

In this paper we focus on *coarse-grain speculation* as a means to achieve parallelism. We provide a simple programming model to express at any arbitrary granularity, the parts of an application that may be executed speculatively. Writing correct shared memory parallel programs is a challenging task in itself [23], and detecting concurrency bugs (e.g., data races, deadlocks, order violations, atomicity violations) is an extremely difficult problem [41]. Hence, we do not want to burden the programmer with the additional responsibilities of using low level threading primitives to create speculative control flows, manage rollbacks and perform recovery actions in the event of mis-speculations.

We present **Anumita** (*guess* in Sanskrit), a simple speculative programming framework where multiple coarse-grain speculative code blocks execute concurrently, with the results from a single speculation ultimately modifying the program state. Our goal is to make speculation a first class parallelization method for hard-to-parallelize and input dependent code blocks. Anumita consists of a shared library, which implements the framework API for common type-unsafe languages including C, C++ and Fortran, and a user-level runtime system that transparently (a) creates, instantiates, and destroys speculative control flows, (b) performs name-space isolation, (c) tracks data accesses for each speculation, (d) commits the memory updates of successful speculations, and (e) recovers from memory side-effects of any mis-predictions. In the context of high-performance computing applications, where the OpenMP threading model is prevalent, Anumita also provides a new OpenMP pragma to naturally extend speculation into an OpenMP context.

Anumita works by associating the memory accesses made by each speculation flow (e.g., an instance of a code block or a function) in a speculation composition (loosely, a collection of possible code blocks that execute concurrently). Anumita localizes these memory updates and provides isolation among speculation flows through privatization of address space. Ultimately, a single speculation flow within a composition is allowed to modify the program state. Anumita simplifies the notion of speculative parallelism and relieves the programmer from the subtleties of concurrent programming. The framework is designed to support a broad category of applications by providing expressive evaluation criteria for speculative execution that go beyond *time to solution* to include arbitrary *quality of solution* criteria. Anumita supports multithreaded applications and sequential applications alike. Anumita is implemented as a language independent runtime system and its use requires minimal modifications to application source code.

We evaluate Anumita using microbenchmarks and real applications from several domains. Our experimental results indicate that Anumita is capable of significantly improving the performance of applications by leveraging speculative parallelism. Programmable speculation is susceptible to limitations of speculative execution and may come at an expense. Speculation requires additional resources (pipelines, cores, memory) to handle speculative flows and which may consume more energy and power. In this paper we show that speculative execution of several alternative (or 'surrogate') code blocks incurs at worst a modest overhead in terms of energy consumption, and can frequently yield improvements in energy consumption, when compared to the use of a single statically chosen surrogate.

The rest of the paper is organized as follows. Section 2 outlines the motivation for this work. Section 3 presents the programming model and constructs used to express coarse-grain speculative execution. Section 4 presents how these constructs can be implemented efficiently without sacrificing performance, portability and usability. Section 5 presents our experimental evaluation. Section 6 surveys the related work, Section 7 describes future directions and Section 8 presents our conclusions.

2. Motivating Problems

Coarse-grain speculative parallelism is most useful for applications with two common characteristics: (1) there exist multiple possible surrogates (e.g., code blocks, methods, algorithms, algorithmic variations) for a particular computation, and (2) the performance (or even success) of these surrogates is problem dependent, i.e., relative performance can vary widely from problem to problem, and is not known a priori. Whether or not there exist efficient parallel implementations of each surrogate is an orthogonal issue to the use of coarse-grain speculation. If only sequential implementations exist, speculation provides a degree of useful parallelism that is not otherwise available. If parallel surrogate implementations do exist, speculation still provides resilience to hard-to-predict performance problems or failures, while also providing an additional level of parallelism to take advantage of growing core counts, e.g., by assigning a subset of cores to each surrogate rather than trying to scale a single surrogate across all cores.

We discuss two motivating examples in detail. (Performance results for these problems are given in Section 5). In graph theory, vertex coloring is the problem of finding the smallest set of colors needed to color a graph $G = (V, E)$ such that no two vertices $v_i, v_j \in V$ with the same color share an edge e. Graph coloring problems arise in several domains including job scheduling, bandwidth allocation, pattern matching and compiler optimization (register allocation). Several state-of-the-art approaches that solve this problem employ probabilistic and meta-heuristic techniques, e.g., simulated annealing, tabu search and variable neighborhood search. Typically, such algorithms initialize the graph with a random set of colors and then employ a heuristic algorithm to attempt to color the graph using the specified number of colors. Depending on the input graph, the performance of these techniques varies widely. Obviously, there will be cases where no coloring can be found (when the specified number of colors is too small) by some or all methods. In addition to this sensitivity to the input, algorithms for the graph coloring problem are hard to parallelize due to inherent data dependancies. Parallel implementations that exist employ a divide and conquer strategy by dividing the graph into subgraphs and applying coloring techniques on the subgraphs in parallel. During reduction, conflicting subgraphs are recolored. Despite such efforts, the challenge still persists to develop efficient parallel algorithms for vertex coloring.

As a second example, consider the numerical solution of partial differential equations (PDEs). This is one of the most

common computations in high performance computing and is a dominant component of large scale simulations arising in computational science and engineering applications such as fluid dynamics, weather and climate modeling, structural analysis, and computational geosciences.

The large, sparse linear systems of algebraic equations that result from PDE discretizations are usually solved using preconditioned iterative methods such as Krylov solvers [33]. Choosing the right combination of Krylov solver and preconditioner, and setting the parameter values that define the details of those preconditioned solvers, is a challenge. The theoretical convergence behavior of preconditioned Krylov solvers on model problems is well understood. However, for general problems the choice of Krylov solver, preconditioner, and parameter settings is often made in an ad hoc manner. Consequently, iterative solver performance can vary widely from problem to problem, even for a sequence of problems that may be related in some way, e.g., problems corresponding to discrete time steps in a time-dependent simulation. In the worst case, a particular iterative solver may fail to converge, in which case another method must be tried. The most conservative choice is to abandon iterative methods completely and simply use a direct factorization, i.e., some variant of Gaussian Elimination (GE). Suitably implemented, GE is essentially guaranteed to work, but in most cases it takes considerably longer than the best preconditioned iterative method. The problem is that the best iterative method is not known a priori.

One could list many other examples that are good candidates for coarse-grain speculation. Even for a simple problem such as sorting, where theoretical algorithmic bounds are well known, in practice the runtime of an algorithm depends on a variety of factors including the amount of input data (algorithmic bounds assume asymptotic behavior), the sortedness of the input data, and cache locality of the implementation [3].

3. Speculation Programming Model

For a coarse-grain speculation model to be successful, it should satisfy several usability and deployability constraints. First, the model should be easy to use, with primarily sequential semantics, i.e., the programmer should not have to worry about the complexities and subtleties of concurrent programming. Speculation is not supported by widely used languages or runtime systems today. Hence, in order to express speculation, the programmer is burdened with creating and managing speculation flows using low-level thread primitives [28]. Second, the speculation model should enable existing applications (both sequential and parallel) to be easily extended to exploit speculation. This includes support for existing imperative languages, including popular type-unsafe languages such as C and C++. Third, the model should be expressive enough to capture a wide variety of speculation scenarios. Finally, to ensure portability across platforms, the speculation model should not require changes to the operating system. Furthermore, we need to accomplish these objectives without negatively impacting the performance of applications that exploit speculation.

A general use case for Anumita is illustrated in Figure 1. The example shows an application with three threads, two of which enter a *speculative region*. (The simplest case would involve a single-threaded code that enters a single speculative region.) Each sequential thread begins execution non-speculatively until a speculative region is encountered, at which time n speculative control flows are instantiated, where n is programmer-specified. Each flow executes a different surrogate code block. We refer to this construct as a "concurrent continuation," where one control flow enters a speculative region through an API call and n speculative control flows emerge from the call. Anumita achieves parallelism by executing the n speculative flows concurrently. In Figure 1, the concurrent continuation out of thread 0 is a composition of three surrogates, while the continuation out of thread 2 has two surrogates. Note that individual surrogates may themselves be multithreaded, e.g., surrogate *estimation* in the continuation flowing out of thread 2. Although not shown in the figure, Anumita also supports nested speculation, where a speculative flow in turn creates a speculative composition.

To mitigate the impact of introducing speculation into the already complex world of concurrent programming, no additional explicit locking is introduced by the speculation model. In other words, a programmer using the Anumita API to add speculation to a single-threaded application does not have to worry about locking or synchronization of any kind. Of course, if the original application was already multithreaded, then locking mechanisms may already be in place, e.g., to synchronize among the three threads in Figure 1 in non-speculative regions.

Each speculative flow operates in a context that is isolated from all other speculations, thereby ensuring the safety of concurrent write operations. Anumita presents a shared memory model, where each speculative flow is exactly identical to its parent flow in that it shares the same view (albeit write-isolated) of memory, i.e., global variables, heap and more importantly, the stack.

The Anumita programming model provides a flexible mechanism for identifying the winner and committing the results of a speculation. The first flow to successfully commit its results is referred to as the winning speculation. However, the decision to commit can be made in a variety of ways. The model easily supports the simplest case, where the first flow to achieve some programmer-defined goal cancels the remaining speculative flows and committs its updates to the parent flow, which resumes execution at the point of commit. Surrogate *estimation* illustrates this case in Figure 1. Alternately, speculative flows may choose to abort themselves if they internally detect a failure mode of some kind,

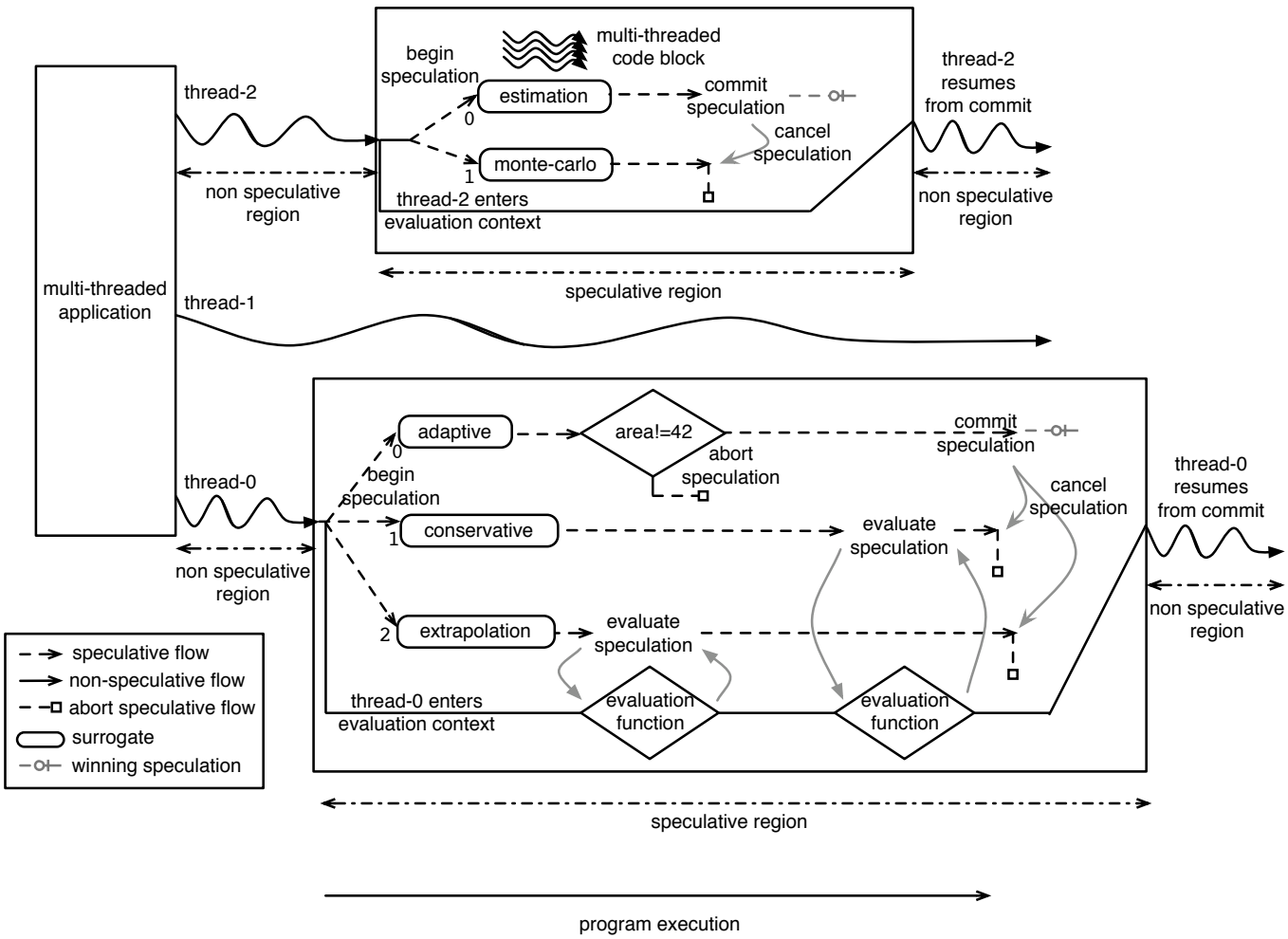

Figure 1. A typical use case scenario for composing coarse-grain speculations. Anumita supports both sequential and multi-threaded applications.

e.g., surrogate *adaptive* in the figure, when `area != 42`. More generally, each surrogate may define success in terms of an arbitrary user-defined evaluation function, passed to an evaluation interface supplied by the parent flow (labeled "evaluation context" in Figure 1). The evaluation context safely maintains state that it can use to steer the composition, deciding which surrogates should continue and which should terminate. In our example, surrogates *conservative* and *extrapolation* use the evaluation interface to communicate with their parent flow.

3.1 Program Correctness

Any concurrent programming model needs well-defined semantics for propagation of memory updates. Anumita supports concurrency at three levels: (1) between surrogates in a speculative composition, (2) between threads in a single multithreaded surrogate, and (3) between threads in non-speculative regions of an existing multithreaded application. We consider each in turn.

Unlike the traditional threads model, where any conflicting accesses to shared memory must be properly synchronized, Anumita avoids synchronization and its associated complexity by providing isolation among speculative flows through privatization of the shared address space (global data and heap). Furthermore, a copy of the stack frame of the parent flow is passed to each speculative flow. Since updates are isolated, "conflicting" accesses do not require synchronization. Anumita's commit construct implements a relatively straightforward propagation rule: *for a given composition, only the updates of a single winning speculative flow are made visible to its parent flow at the completion of a composition*. Furthermore, compositions within a single control flow are serialized, in that a control flow cannot start a speculative composition without completing prior compositions in program order. Cumulatively, these two properties are sufficient to ensure program correctness in sequential applications (a single control flow) even in the presence of nested speculations. We do not present a formal proof of

```
speculation_t *spec_context;
int num_specs=2, rank, value=0;

/* initialize speculation context */
spec_context = init_speculation();

/* begin speculative context */
begin_speculation (spec_context, num_specs, 0);

/* get rank for a speculation */
rank = get_rank (spec_context);

switch (rank)
{
  case 0:
    estimation (...);
    break;

  case 1:
    monte-carlo (...);
    break;

  default:
    printf ("invalid rank\n");
    break;
}

/* commit the speculative composition */
commit_speculation (spec_context);
```

Figure 2. Pseudo code for composing speculations using the programming constructs exposed by Anumita. In the absence of an evaluation function, the fastest surrogate (by time to solution) wins.

```
/* custom evaluation function */
boolean goodness_of_fit (speculation_t *spec_context, void *ptr)
{
  double error = 0.0, *fit = (double *) ptr;

  error = actual - *fit;
  if (error > 0.0005)
   {
    return ABORT;
   }

  return CONTINUE;
}

....
switch (rank)
{
  case 0:
    area = adaptive_quadrature (...);

    ptr = get_ir_memory (spec_context);
    memcpy (ptr, area, sizeof(double));

    retval = evaluate_speculation (spec_context, goodness_of_fit, ptr);
    if (retval == ABORT)
      abort_speculation (spec_context);
    break;

  case 1:
    area = conservative_method();
    if (area != 42)
      cancel_speculation (spec_context, 0);
    break;

  case 2:
    for (t=0; t<100; t++)
    {
      area = extrapolation_method();

      ptr = get_ir_memory (spec_context);
      memcpy (ptr, area, sizeof(double));

      retval = evaluate_speculation (spec_context, goodness_of_fit, ptr);
      if (retval == ABORT)
        abort_speculation (spec_context);
    }
   break;
}
....
```

Figure 3. Pseudo code for evaluating speculations in Anumita.

correctness here; however, the rationale behind the proof is that since the updates of exactly one of the valid outcomes is committed and since each speculation was isolated while arriving at this result, relaxing the requirements of explicit synchronization does not affect program correctness.

Surrogates in Anumita may themselves be multithreaded, requiring lock based concurrency control between threads, e.g., surrogate *estimation* in Figure 1. Since surrogates are replacements for each other, we would not expect a surrogate to have synchronization dependencies with one of its sibling surrogate, e.g., *estimation* with *monte-carlo* in the figure. Hence the correctness of multithreaded surrogates reduces to the standard case of threaded shared-memory concurrent programming.

While the above properties ensure program correctness for concurrent continuations flowing out of a single control flow, we also need to define how speculative flows can be composed in a multithreaded environment. Anumita allows multiple speculative and non-speculative regions to execute concurrently, e.g., the regions associated with threads 0 and 2, along with thread 1 in Figure 1. However, the model does not support synchronization between speculative flows from different speculative regions, or between speculative flows and other non-specualtive application threads. (We note that this restriction is also true for value speculation systems such as [28].) Hence, correctness of such codes stems from the correctness of the original multithreaded code, since each speculative region exhibits transaction-like semantics with respect to other threads, i.e., no memory updates from a given speculative region are visible to other threads until the commit process finishes, at which point all the updates are complete, and control resumes in a non-speculative region.

Externally visible I/O actions are not handled by Anumita in its current form. We are working on extending Anumita to support disk I/O. However, Anumita performs speculation aware memory management and garbage collection from failed speculations. This mechanism correctly hides the side-effects of system calls such as sbrk etc.

3.2 Syntax and Semantics

Figures 2 and 3 show pseudocode corresponding to the scenario illustrated in Figure 1 for composing speculations using Anumita. Table 1 defines the Anumita API. A speculative composition is *initialized* by a call to `init_speculation`, which returns a speculation context. A composition is *instantiated* by a call to `begin_speculation`, which implements a concurrent continuation of the parent flow. Each speculative flow in the concurrent continuation is exactly identical to its parent flow in that it shares the same view of memory, but is isolated from other concurrent speculative flows. In order to distinguish speculative flows from each other, we associate each speculative flow with a unique rank. This notion of rank is identical to ranks in MPI and thread number in OpenMP. A speculation may query its rank (0 to n-1) in order to map a particular unit of work to itself. The parent flow then enters an evaluation context, where it waits (descheduled) for evaluation requests from its speculative flows.

To implement an interface for evaluation, the call to `begin_speculation` takes an argument that specifies the size of a memory region that is used for communication between speculative flows and the parent evaluation context. Each speculative flow receives a distinct memory region of the specified size; this region is shared between a speculative flow and the parent evaluation context. Periodically, a speculative flow can request an evaluation using the `evaluate_speculation` call, passing the parent intermediate results using the shared memory region. This call synchronously transfers control to the evaluation context (i.e., the idled parent flow), which executes the evaluation function and returns a status indicating whether the speculation calling the evaluation should continue or abort execution. The evaluation context may also use the intermediate results to cancel other speculations based on the results of the current evaluation, for instance, when the progress of one surrogate is significantly better than another within the same composition. In essence, the evaluation mechanism enables pruning of surrogates based on a user-defined notion of result quality.

On completing execution, a surrogate terminates the speculative region by calling `commit_speculation`. The first call to `commit_speculation` succeeds, canceling all other speculations in the composition and propagating its execution context to the parent flow, which then resumes execution at the point of commit. Selecting by time to solution (fastest surrogate wins) is trivially implemented by not specifying an evaluation function, as shown in Figure 2. In this case the first surrogate to commit would succeed and cancel its siblings. For completeness, the API also supports an `abort_speculation` call that can be used by a surrogate to terminate itself if it detects that it is not making progress or has reached some failure mode. We also provide a `cancel_speculation` call that can be used by any surrogate to terminate any other surrogate. This can be useful, for example, in a case where a subset of surrogates can be pruned from the composition when one member of that subset meets some condition.

```
int num_specs = 3, rank;

/* begin a speculation composition */
# pragma speculate (spec_context, num_specs, 0)
{

 /* get rank for a speculation */
 rank = omp_get_thread_num();

 switch (rank)
 {
    case 0:
      /* code for speculation */
      vns (colors, graph);
      break;

    case 1:
      /* code for speculation */
      sa (colors, graph);
      break;

    case 2:
      /* code for speculation */
      tabu (colors, graph);
      break;

    default:
      /* invalid rank */
 }

 /* implicit commit of speculation composition */
}
```

Figure 4. Composing speculations in OpenMP using the OpenMP extensions built on top of the programming constructs exposed by Anumita. Anumita's source-source translator expands the *speculate* pragma to begin-commit constructs.

Many scientific applications use OpenMP directives for shared memory programming rather than the underlying POSIX threads interface. To support such applications, we provide extensions to OpenMP in the form of a new OpenMP pragma that provides a natural interface to speculation. Figure 4 illustrates the OpenMP syntax for creating a composition. The *speculate* pragma is scoped between an open and close brace ({ and }), with an implicit `commit_speculation` at the end of the speculate pragma. In traditional OpenMP programming, name space isolation is achieved through explicit variable scoping (e.g., private, shared, etc.). To simplify programming, the Anumita runtime automatically isolates speculative flows without requiring explicit private scoping.

3.3 Overhead

Anumita achieves low runtime overhead since speculative flows are isolated and mispredictions cause the memory updates of the failed speculation to be discarded as opposed to rollback recovery. The memory overhead is proportional

Programming Constructs	Description
speculation_t* **init_speculation**(void)	Initialize a speculative composition.
int **begin_speculation**(speculation_t *spec_context, int num_spec, size_t mem)	Begins a speculative composition. Arguments are composition context, number of speculations and size of memory to allocate for storing intermediate results used in evaluating speculations.
int **get_rank**(speculation_t *spec_context)	Gets the rank of the calling speculative flow.
int **get_size**(speculation_t *spec_context)	Gets the number of speculative flows in a composition.
int **commit_speculation**(speculation_t *spec_context)	Attempts to commit the state of the calling speculative flow.
int **abort_speculation**(speculation_t *spec_context)	Aborts the calling speculative flow.
int **cancel_speculation**(speculation_t *spec_context, int rank)	Cancels (terminates) the speculation flow with a rank of 'rank'.
int **evaluate_speculation**(speculation_t *spec_context, evaluate_t *evaluation_fn, void *ptr)	Used to invoke an evaluation of the intermediate results of the calling speculative flow. Intermediate results are passed through 'ptr'.
void * **get_ir_memory**(speculation_t *spec_context)	Returns a pointer to the calling speculative flow's memory region used to store intermediate results for evaluation.
int (*evaluation_fn)(speculation_t *spec_context, void *ptr)	Signature of the user defined evaluation function.

Table 1. Programming constructs exposed by Anumita for leveraging speculation. For brevity, C++ and Fortran interfaces are omitted.

to the write-set of all the speculative flows, which is typically much smaller than the read set. Given N speculative flows with the write-set of each flow being W pages, the memory overhead is O(NW).

4. Implementation

The Anumita implementation consists of a shared library that exposes our API and a runtime system. The OpenMP interfaces are implemented using source-to-source translation. To ensure ease of deployment, Anumita is implemented completely in user-space with no modifications to the operating system. The rest of this section describes the Anumita runtime in detail.

4.1 Shared Address Space

In the POSIX threads model, each thread has a distinct stack and threads of a process share their address space. In contrast, distinct processes are fully isolated from each other and execute in separate virtual address spaces. Neither of these models satisfies the isolation and selective state sharing requirements imposed by Anumita. Intuitively, we need an execution model that *provides the ability to selectively share state between execution contexts.*

To create the notion of a shared address space among processes, we implemented the *cords* abstraction first proposed in [29]. The constructor in our runtime (a shared library) traverses through the link map of the application (ELF binary) at runtime and identifies the global data (*.bss* and *.data*) sections, i.e., the zero initialized and uninitialized data and non-zero initialized data, respectively. The runtime then unmaps these sections from the loaded binary image in memory, maps them from a SYSV memory mapped shared memory file and reinitializes these sections to the original values.

This mapping to a shared memory file is done by the *main* process before its execution begins at *main*. Speculative flows are then instantiated as processes (we use the *clone()* system call in Linux to ensure that file mappings are shared as well) and a copy of the address space of the parent is created for each instantiation of a speculation. Consequently, the speculations inherit the shared global data mapping. Hence any modifications made by a process to global data are immediately visible to all processes. Such a technique guarantees that all the processes have the same view of global data, similar to a threads model. In essence, this technique creates a set of processes that are semantically identical to threads, but operate in distinct virtual address spaces. By controlling the binding to the shared memory mapping, data can be selectively isolated or shared based on the requirements of the speculation model.

To implement a shared heap, we modified Doug Lea's dlmalloc [15] allocator to operate over shared memory mappings so that the allocated memory is visible to all processes. Our runtime system provides global heap allocation by sharing memory management metadata among processes using the same shared memory backing mechanism used for .data and .bss sections. Hence any process can allocate memory that is visible and usable by other processes. If a process accesses memory that is not mapped in its address space, it results in a segmentation violation (a map error). Our runtime system handles this segmentation violation by consult-

ing memory management metadata to check if the reference is to a valid memory address allocated by a different process. If so, it maps the shared memory file associated with the memory, thereby making it available. Note that such an access fault only occurs on the first access to a memory region allocated by a different process, and is conceptually similar to lazy memory allocation within an operating system.

To ensure program correctness, speculative flows within the same composition should appear as a concurrent continuation of the parent execution context. To achieve this, our runtime system ensures that the base address of the stack in a speculative flow is identical to that of the parent speculation. The default size of a stack is 8MB. When composing a speculation, the runtime saves only the stack frame of the parent speculation (not the entire 8MB) and each speculation within a composition uses a copy of this stack frame for execution. Each speculative flow is now identical to its parent flow, thereby creating a concurrent continuation.

Since each speculative flow is implemented as a process, it is important to note that in UNIX process semantics, each process is created with its own copy of the data segment of the shared libraries. Consequently, by default the runtime is not shared among speculation flows. To circumvent this problem and to maintain a shared and consistent view of the runtime, each newly created process automatically executes an initialization routine that maps the shared state of the Anumita runtime.

4.2 Speculative Composition

To mitigate the costs of creating and terminating speculative flows, the runtime instantiates a configurable pool of speculative flows in the Anumita library constructor before the *main* process begins its execution. Additional speculative flows are created as necessary. This pool of speculative flows is initially idle (blocked), waiting for work from the *main* process. On the termination of a speculative flow, it is returned back to the pool. In principle this is similar to a worker thread pool used to mitigate the performance impact of thread creation.

To instantiate a speculation, the parent flow first saves its current stack frame and execution context (*setjmp*) before waking the specified set of speculative flows from the pool. Upon waking, each speculation adjusts its execution context (*longjmp*), restores its stack to that of the parent flow and isolates its shared virtual memory address (VMA) before starting execution. The speculations begin their execution as a concurrent continuation of the `begin_speculation` construct. The parent flow then enters an evaluation context and waits for messages from the members of the speculative composition. The parent flow may be woken up under three scenarios.

First, if a speculative flow completes its assigned task it executes a `commit_speculation`. A call to `commit_speculation` is mutually exclusive to prevent race conditions on commits from multiple speculations. The first speculation to invoke commit is designated the winner. The winning speculation saves its current execution context and its stack frame so as to allow its parent to continue from the commit point. Additionally, the winning speculation attaches (*ptrace*) itself to the remaining sibling speculations and alters their instruction pointer to point to a cleanup routine. In the cleanup routine, it performs an inclusion (propagation of privatized updates) of the shared virtual memory address (VMA) and frees any dynamically allocated memory it allocated before returning to the pool. The winning speculation then commits its changes, wakes up its parent with a "winning speculation" message and joins the worker pool. Upon waking up, the parent flow adjusts its execution context and stack and returns from `commit_speculation` to continue its execution.

Second, if a speculative flow requests an evaluation, the parent flow executes the user defined evaluation function and returns a boolean value to indicate either a success or a cancellation. The speculative flow then either continues or aborts its execution based on the boolean value. Additionally, the parent flow can steer the computations of speculative flows. Anumita also implements a flexible approach to allow the parent flow to store the intermediate results of speculative flows for evaluation. In order to accomplish this a speculative flow may access memory using `get_ir_memory`. This region of memory is shared between the parent flow and the speculative flow and it is unique to each speculative flow. This obviates the need for any synchronization among speculative flows to update their intermediate results.

Finally, in the event that all the speculations in a composition abort, the last speculation to abort (in program order within for a composition) signals the parent flow to terminate the program, since no surrogate satisfied the expected quality criterion.

4.3 Nested Speculative Compositions

Implementing support for nested speculations presents additional challenges. Recall that in order to contain updates within a speculative flow, the pages modified in a speculative flow are privatized. Hence, if a speculative flow in turn creates a new composition, then it should propagate all its "privatized updates" to the speculative flows in the new composition. This has to be achieved without committing the updates, since the parent of the nested speculation may not be the winner in its composition. Conversely, the updates by the speculations in a nested composition should be propagated only to its parent flow to ensure program correctness.

To resolve this, in the case of nested speculations, the runtime creates new speculative flows during the call to `begin_speculation` instead of using a worker pool entry. This creates a current copy of the parent flow and includes privatized updates. Since the parent is blocked upon composing a speculation, the lazy copy-on-write semantics provided

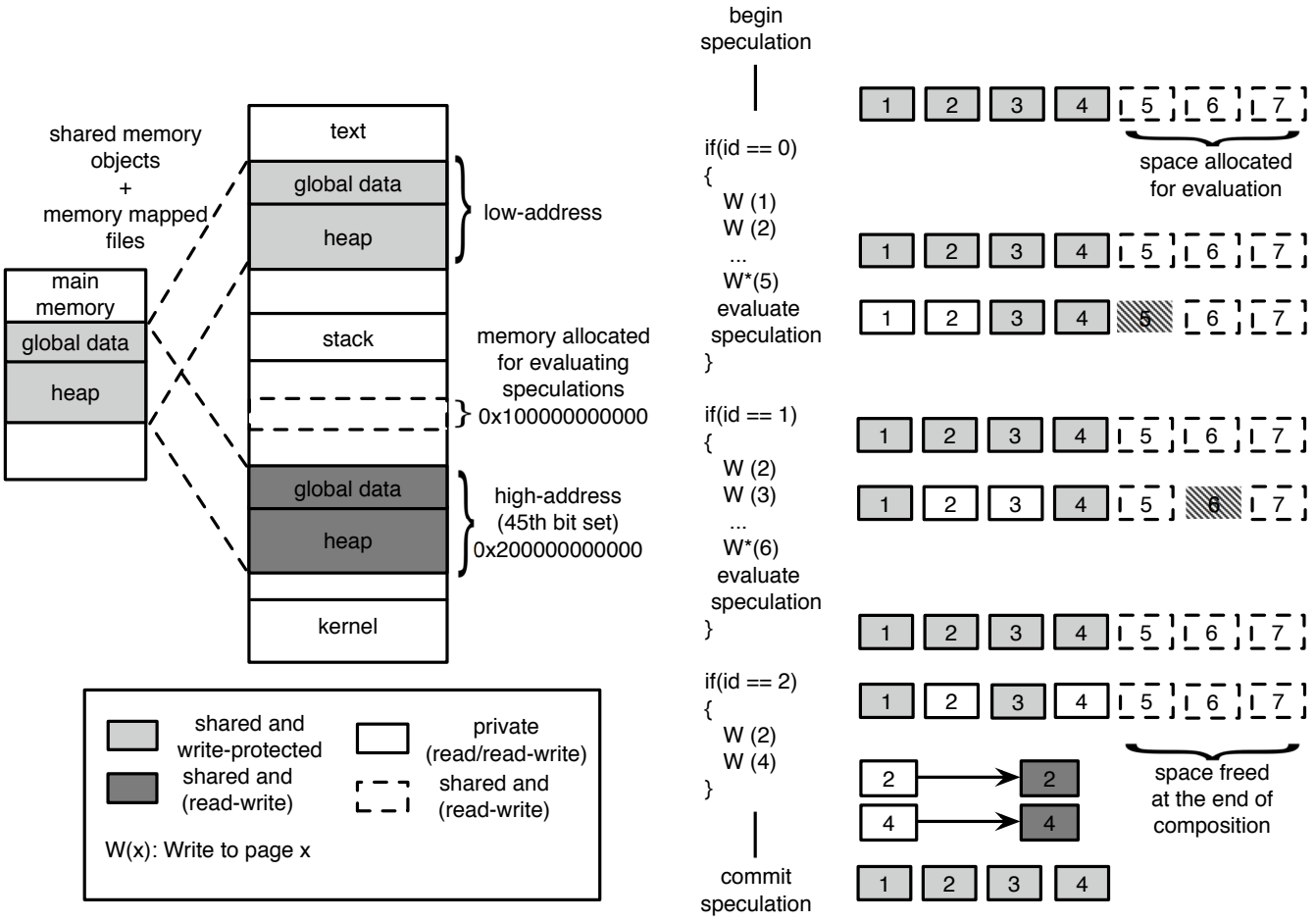

Figure 5. (left) Illustrates the virtual memory address (VMA) layout of each process. The runtime provides memory isolation by using shared memory objects and memory mapped files to share global state among processes. (right) Illustrates how the VMA is manipulated by the runtime with a simple example.

by the operating system efficiently creates isolated private address spaces for the nested speculations.

4.4 Containment

In order to determine the write-set and and contain (privatize) the updates of a speculation, Anumita employs page level protection and privatization of the shared VMA. Each speculative flow initially write-protects (PROT_READ) its shared VMA. Read accesses to shared data do not produce any faults and execute normally. However, if a speculation attempts to write to a shared page (global data, heap), the runtime handles the access fault (SEGV_ACCERR) by remapping the faulting page from the shared memory in private (MAP_PRIVATE) mode. The runtime maintains a list of pages that were modified by each speculation within a composition. The permissions of the page are then reset to read-write so that the speculation can continue its execution.

This privatization provides containment of updates by a speculation. Such a lazy privatization scheme that defers privatization until the instant memory update happens results in weak atomicity [8]. *Weak atomicity is sufficient to ensure program correctness in our speculation model*. We do not present a formal proof of correctness, however, the intuition behind the proof is that once a winning speculation commits, none of the remaining speculations will be allowed to commit, thus precluding Write-after-Read or Write-after-Write hazards. Hence, we chose the lazy privatization approach over a conservative approach, which privatizes the entire VMA.

The runtime does not track accesses of local variables on the stack. Since a copy of the parent stack frame is passed to each speculation, the stacks do not need to be write-protected. Instead the parent's stack frame is updated with contents of the stack frame from the winning speculation. Such a strategy works for programs that contain pointers to stack-allocated data.

4.5 Inclusion

When a speculation composition culminates with a winning surrogate, the updates of the winning speculation (contained

in the privatized data) must be propagated and made visible to the parent flow and any other non speculative flows in the program.

In order to perform this *inclusion of updates*, we implement the shadow addressing technique similar to [29] that leverages the large virtual memory address (VMA) provided by 64-bit operating systems. Recall that the runtime system maps globals and the heap (described in Section 4.1) using shared memory objects and memory mapped files. Using the same shared memory objects and memory mapped file, the runtime creates an identical secondary *shadow* mapping of the global data sections and heap at a *high address* (45th bit set) in the VMA of each speculation flow. The program is unaware of this mapping, and performs its accesses (reads/writes) at the original low address space. The high address space shadow is always shared among speculative flows and is never privatized. Hence, any updates to it are propagated across all flows. In effect, the high address mapping creates a *shadow* address space for all shared program data and modifications (unless privatized) are visible in both address spaces (shown in Figure 5).

To perform this inclusion the runtime employs two distinct strategies depending on the depth of the speculations (nested or otherwise). In a single level speculation, where a flow creates a composition, updates from the winning speculation must be made visible to the parent flow. To achieve this, the runtime copies all the pages in the write set of the winning speculation to the high address shadow region of the VMA (shown in Figure 5), which automatically propagates the updates to the parent flow, due to the shared memory bindings. The runtime then reverts all privatized mappings within the speculative flows (cleanup) and returns them to the pool.

In a nested speculation, the runtime creates a new shared memory mapping equal to the write set of the winning speculation and it copies the write set of the winning speculation to the newly created mapping. The parent flow then copies the write-set from this new mapping into its address space to perform inclusion.

4.6 Example

We present a simple example to illustrate how the runtime manipulates the VMA of each speculation flow while providing isolation, privatization and inclusion. Consider the scenario as shown in Figure 5 where a control flow creates a composition involving three speculations. Initially, all the shared pages (1, 2, 3, 4) of the speculative flows are write-protected. When the flow with rank 0 attempts to write to pages (1, 2), the pages are privatized (lazy privatization). Similarly, the runtime system privatizes the updates of speculations with ranks 1 and 2, which write to pages (2, 3) and (2, 4) respectively. If the speculation with rank 2 wins the composition, the runtime commits the write-set to the shared high-address space to propagate the updates.

Additionally, in Figure 5 we illustrate how the speculative flows may request evaluation of their progress. At the beginning of a speculative composition, a program may choose to request space for storing partial results. In the above example, pages (5, 6, 7) are allocated by the runtime system to store the partial results. Each speculation may use the `get_ir_memory()` to obtain the address of the region used to store its partial results. The speculative flow at rank 0 writes its partial results to page 5, before requesting evaluation. Following the same procedure, speculative flow 1 writes to page 6 before requesting evaluation from its parent flow. The parent flow can access these memory locations and execute the evaluation function to determine the relative quality and/or progress of the speculative flows.

Recall, that while Anumita supports multi-threaded applications, caution should be exercised in leveraging speculative parallelism in a multi-threaded environment. As discussed in Section 3.1, if a surrogate requires concurrency with other non-speculative threads or other concurrently executing speculative threads then it is not a candidate for speculation, since such a surrogate is based on concurrency rather than speculation. Hence, in the presence of data dependencies we expect explicit serialization in composing speculations in order to ensure program correctness.

A complication arises when two distinct threads of a process independently instantiate speculative compositions that execute concurrently. This is depicted in Figure 1, where threads 0 and 2 enter distinct compositions. While the compositions may be distinct, the granularity of our protection mechanism is at the level of an operating system page, which can cause false-sharing if updates to distinct bytes from distinct compositions reside on the same operating system page. The artifact of this problem is that in the case of concurrent speculations, the contents on a page subject to false sharing will reflect the updates from the last speculation in program order to successfully commit without reflecting any of the updates from concurrent commits. For example in Figure 1, if the *monte-carlo* method updated page 1, which was also updated (albeit at different locations) by the *adaptive* method, the final contents of page 1 will reflect the updates from *adaptive* and none of the updates from *monte-carlo*. In effect, updates to pages subject to false sharing are mutually exclusive, which is clearly incorrect.

To solve this problem, we need an efficient mechanism to propagate updates from a winning speculative flow to all concurrent speculations. This is achieved by computing and propagating XOR differences. To see how this works, consider the example shown in Figure 1. When *monte-carlo* wins the speculative composition in thread 2, prior to performing inclusion the runtime determines if there are concurrent speculative compositions in other threads. If so, for each page in the write-set of the first winning flow (*monte-carlo*), the runtime computes an XOR of the privatized page with its counterpart in the high address space. Recall that

prior to inclusion, the page in the write-set of *monte-carlo* is privatized and includes updates from the flow, whereas its counterpart in the shadow address space contains the original contents of the page. The XOR difference thus yields the exact set of bits that were updated by *monte-carlo* method.

The runtime then pushes this XOR difference to all concurrently executing flows (*adaptive*, *conservative*, *extrapolation*) and performs inclusion of updates from *monte-carlo* as before. The concurrent flows apply the differences by computing the XOR of the difference they received with their privatized copy of the page (if one exists due to false sharing). Intuitively this mechanism updates the privatized contents of a concurrent speculation with the latest updates from a winning speculation in another composition. In the example above, when *adaptive* finally wins its composition, its write set already contains the updates from *monte-carlo* and hence the final resulting update of a falsely shared page from *adaptive* correctly contains the cumulative updates from winning speculations. To minimize time and space overhead, the XOR difference is only computed on the pages in the write set that are subject to false sharing (typically small), which is determined by computing the intersection of the write sets of the winning flow (*monte-carlo*) and all other concurrently executing flows(*adaptive*, *conservative*, *extrapolation*) .

4.7 Support for OpenMP

To support OpenMP, we provide a simple source to source translator that expands the *#pragma speculate (...){....}* directive to begin and commit constructs. Our translator parses only the speculate pragma leaving the rest of the OpenMP code intact. This approach does not require any modifications to existing OpenMP compilers and/or OpenMP runtime libraries.

Our runtime system overrides mutual exclusion locks, barriers and condition variables of the POSIX thread interface and a few OpenMP library routines in order to provide a clean interface to OpenMP. We overload the `omp_get_thread_num` call in OpenMP to return the speculation rank from `get_rank`. The Anumita runtime automatically detects if an OpenMP program is in a speculative context and selectively overloads OpenMP calls, which fall back to their original OpenMP runtime when execution is outside a speculative composition. Finally, our OpenMP subsystem implements a simple static analyzer to perform lexical scoping of a speculative composition. This can be used to check for logical errors such as a call to commit before beginning a speculation.

5. Experimental Evaluation

We evaluated the performance of the Anumati runtime over three applications: a multi-algorithmic PDE solving framework [32], a graph (vertex) coloring problem [25] and a suite of sorting algorithms [36].

We ran each benchmark under two scenarios. The first scenario uses Anumita to speculatively execute multiple algorithms concurrently. This was done by modifying approximately 8-10 lines of source code in the above benchmarks. Since Anumita guarantees isolation, these modifications were short and required little to no understanding of the algorithms themselves. In the other scenario we ran the vanilla benchmark executing each algorithm individually. All experiments were performed on a 16 core shared memory machine (NUMA) running Linux 2.6.31-14 with 64GB of RAM. The system contains four 2 GHz Quad-core AMD Opteron processors.

5.1 PDE solver

One approach for dealing with the unpredictable input-dependent performance of PDE solvers is a 'poly-algorithmic' strategy, where multiple algorithms are tried in parallel, with the one finishing first declared the winner, e.g., [4, 6]. This approach is robust, essentially guaranteeing a solution, and is easily implemented using our framework.

We consider the scalar linear elliptic equation

$$-\nabla^2 u + \frac{\alpha}{(\beta + x + y)^2} u_x + \frac{\alpha}{(\beta + x + y)^2} u_y = f(x, y),$$

with Dirichlet boundary conditions on the unit square, where $\beta > 0$. Discretized with centered finite differences, the resulting linear system of algebraic equations is increasingly ill-conditioned for large α and small β. Krylov linear solvers have difficulty as this problem approaches the singular case, i.e., as α/β^2 grows. What is not so clear is how quickly the performance degrades, and how much preconditioning can help. To simplify the case study, we fix β at 0.01 and vary α.

Discretizing the problem using a uniform grid with spacing $h = 1/300$ results in a linear system of dimension 89401. We consider three iterative methods and one direct method for solving this system of equations:

1. GMRES(`kdim=20`) with ILUTP(`droptol=.001`)
2. GMRES(`kdim=50`) with ILUTP(`droptol=.0001`)
3. GMRES(`kdim=100`) with ILUTP(`droptol=.00001`)
4. Band Gaussian Elimination

Here `kdim` is the GMRES restart parameter, ILUTP is the "incomplete LU with threshold pivoting" preconditioner [33, Chap. 10], and `droptol` controls the number of nonzeros kept in the ILU preconditioner. Increasing `kdim` or decreasing `droptol` increases the computational cost per iteration of the iterative method, but should also increase the residual reduction per iteration. Hence, one can think of methods one to four as being ordered form "fast but brittle" to "slow but sure." Our PDE-solving framework for these experiments is ELLPACK [32], with the GMRES implementation from SPARSKIT [34].

Figure 6 shows the performance of the four methods and speculation for varying α. For small α the results are consis-

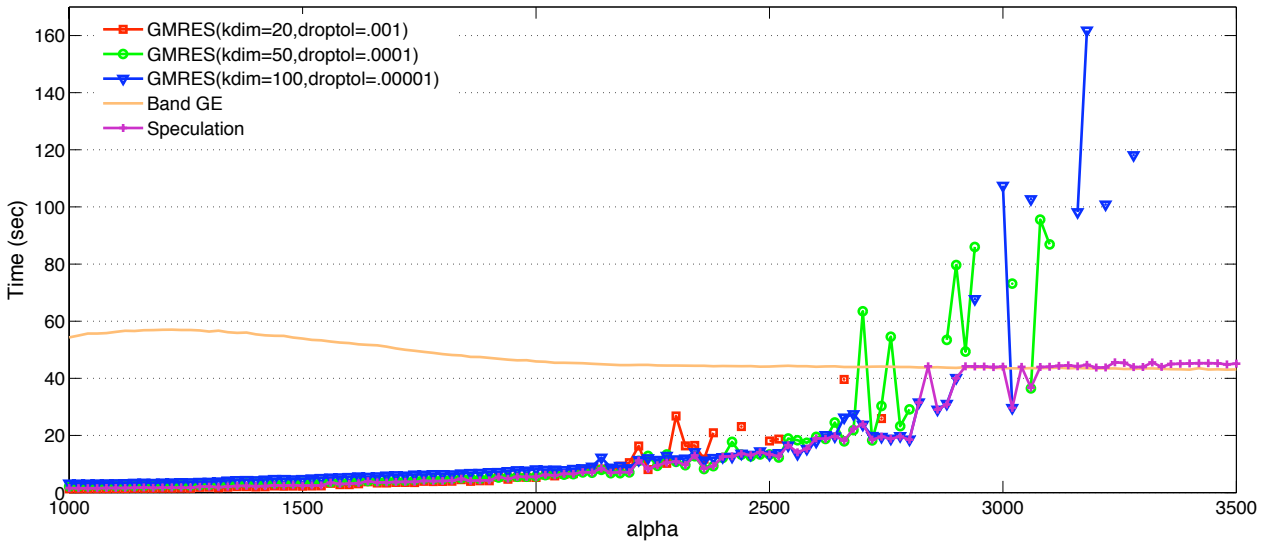

Figure 6. Time to solution for individual PDE solvers and speculation based version using Anumita. Cases that fail to converge in 1000 iterations are not shown. The results show that Anumita has relatively small overhead, allowing the speculation based program to consistently achieve performance comparable to the fastest individual method for each problem.

Method	Fail	Speedup		
		Min	Max	Median
1	51	0.84	2.47	0.94
2	27	0.94	2.89	1.18
3	23	0.94	3.62	1.52
4	0	0.95	36.19	5.01

Table 2. Number of failing cases (out of 125) for each PDE solver, and speedup of speculative approach relative to each method.

tent, with Method 1 consistently fastest. As α grows, however, the performance of the iterative methods vary dramatically, with each method taking turns being the most efficient. In many cases the GMRES iteration fails to converge (i.e., iterations exceeding 1000 are not shown in the figure). Eventually, for large enough α, Band GE is the only method that succeeds.

The write set of the PDE solver is 157156 pages ($\approx$ 614MB) of data. The overhead of speculation shrinks steadily as the problem difficulty grows, with overheads of no more than 5% for large α. This is to be expected since the time to solve sparse linear systems grows faster as a function of problem dimension than the data set size, which largely determines the overhead. However, in cases with small (< 10 sec) runtime, the overhead due to speculation is noticeable (up to 16%). This is due to initial thread creation and start up costs which are otherwise amortized over the runtime of a larger run.

The results show that speculative execution provides clear benefits over any single static selection of PDE solver. Table 2 summarizes the performance of the four methods relative to the speculatively executed case. Statically choosing any one of the GMRES methods (Methods 1-3) causes a serious robustness problem, as many of the problems fail completely. Even for the cases where GMRES succeeds, we see that the speculative approach yields noticeable improvements. For the problems where method 1 succeeds, it is faster than speculation more than half the time (median speedup = 0.94). Compared to methods 2-4 speculation is significantly faster in the majority of cases. In essence, speculation dynamically chooses the best algorithm for a given problem, with minimal overhead.

It must be pointed out that the speculative code uses four computational cores, while the standalone cases each use only one core. In the case where we only have sequential implementations of a given surrogate, speculation gives us a convenient way to do useful work on multiple cores, moving more quickly on average to a solution. However, given parallel implementations of each of the four methods, an alternative to speculation is to choose one method to run (in parallel) on the four cores. However, this strategy still suffers from the risk of a method failing, in which case one or more additional methods would have to be tried. In addition, it is well-known that sparse linear solvers do not exhibit ideal strong scaling, i.e., parallel performance for a fixed problem does not scale well to high core counts. By contrast, running each surrogate on a core is embarrassingly parallel; each core is doing completely independent work. Given hundreds of cores, the optimal strategy is likely to be to use

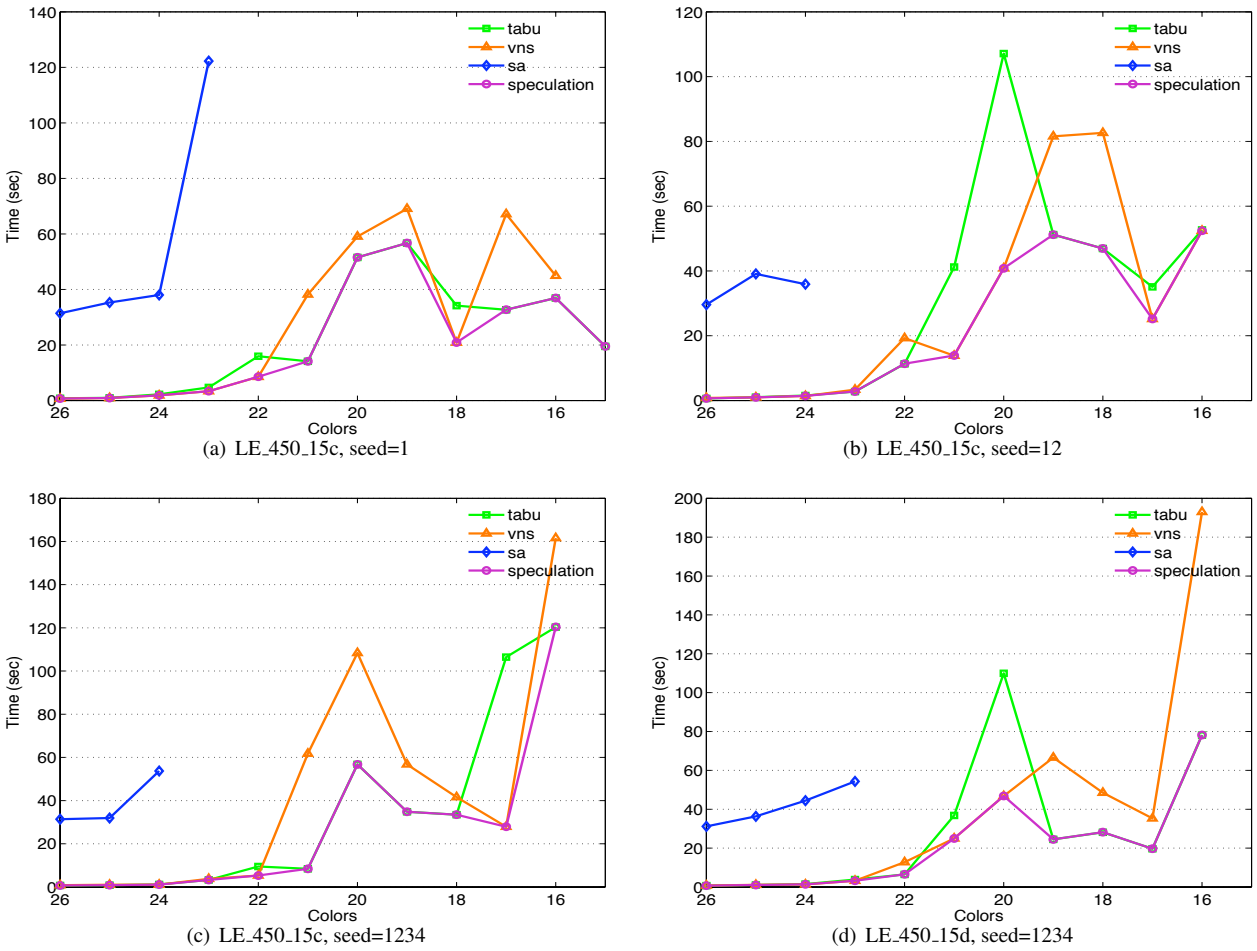

Figure 7. The performance of Graphcol benchmark using two DIMACS data sets LE_450_15c (subfigures (a) through (c)) and LE_450_15d (subfigure (d)).

speculation at the highest level, with each surrogate running in parallel on some subset of the cores. Choosing the number of cores to assign to each surrogate should depend on the problem and the scalability of each method on that problem, and is beyond the scope of this paper.

5.2 Graph Coloring Problem

In graph theory, vertex coloring is the problem of finding the smallest set of colors needed to color a graph $G = (V, E)$ such that no two vertices $v_i, v_j \in V$ with the same color share an edge e. The graphcol [25] benchmark implements three surrogate heuristics for coloring the vertices of a graph: simulated annealing, tabu search and variable neighborhood search. The benchmark initializes the graph by randomly coloring the vertices with a specified set of colors and each heuristic algorithm iteratively recolors the graph within the coloring constraints. We used the DIMACS [13] data sets for the graph coloring benchmark, which are widely used in evaluating algorithms and serve as the testbed for DIMACS implementation challenges. Each data set (graph) has a fixed number of colors that it can use to color a graph. We experimented with over 80 DIMACS data sets using different seeds (for initial colors) and show the results from representative runs.

In Figure 7 we present the results of the graph coloring benchmark using two DIMACS data sets. The results show several interesting characteristics. First, certain heuristics do not converge and cannot guarantee a solution. For instance, simulated annealing (*sa*) cannot color the graph beyond a certain number of colors. Second, the choice of the input seed, which decides the initial random coloring, creates significant performance variations among the heuristics (Figures 7 (a) through (c)) *even when the graph is identical*. Third, when the seed is constant, there is performance variation among the data sets, which represent different graphs as shown in Figures 7 (c) and (d). In the presence of such strong input dependence across multiple input parameters, it is difficult even for a domain expert to predict the best algorithm a priori.

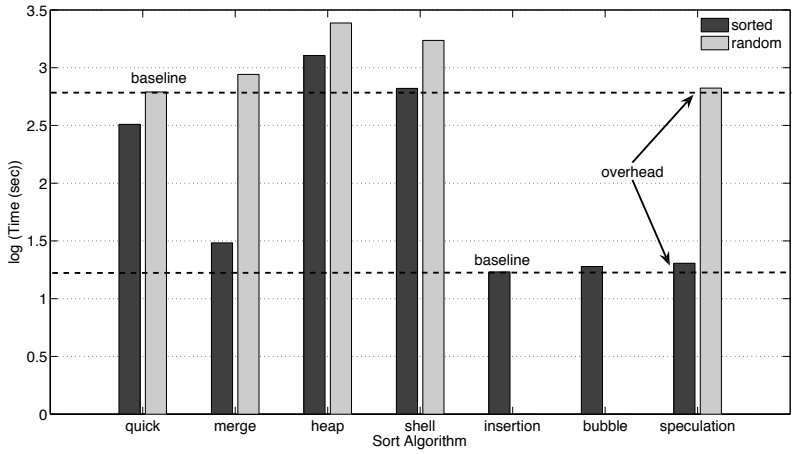

Figure 8. Performance of Anumita over a suite of sorting algorithms.

Using Anumita it is possible to obtain the best solution among multiple heuristics. We found that in some cases where *sa* failed to arrive at a solution (unable to color the graph using specified number of colors), the use of speculation guaranteed not only a solution but also one that is nearly as fast the fastest alternative. Since the write set is relatively small at around 50-100 pages, the overhead of speculation is negligible. Anumita's speedup, across all the data sets (in Figure 7), ranges from 0.954 (vns with 26 colors in in Figure 7 b) in the worst case, when the static selection is the best surrogate to 7.326 (vns with 21 colors in in Figure 7 d), when the static selection is the worst surrogate. We omit the results from the *sa* method in calculating speedup since sa consistently performs worse than the other algorithms on these data sets.

5.3 Sorting Algorithms

Since the overhead of speculation in our runtime is proportional to the write set of an application, we chose sort as our third benchmark since it can be configured to have an arbitrarily large memory footprint. Sort is relatively easy to understand, and yet there are wide variety of sorting algorithms with varying performance characteristics depending on the size of the data, sortedness and cache behavior [3]. Our suite of sorting algorithms includes C implementations of quick sort, heap sort, shell sort, insertion sort, merge sort and bubble sort.

The time to completion of the sorting algorithms is based on several cardinal properties including the input size, their values (sorted or unsorted) and algorithmic complexity. In this set of experiments we fixed the input size and used two sets of input data — completely sorted and completely random, each of size $8\,$GB. Each sorting algorithm is implemented as a separate routine. The input data is generated using a random number generator. After sorting the data the benchmark verifies that the data is properly sorted. We measured the runtime of each sorting algorithm and excluded the initialization and verification phases. Using Anumita, we speculatively executed all six sorting algorithms concurrently.

In Figure 8 we present the results of the sort benchmark. Results for insertion sort and bubble sort for random data were omitted since their runtime exceeds 24 hours. The results show that insertion sort is the fastest for sorted data and quick sort performs the best on completely random data, which is expected. Despite the large write set of $8\,$GB per speculation, a total of 6x8GB for the entire speculative composition, Anumita is at least the second fastest of all the alternatives considered and is nearly as fast as the fastest alternative.

The worst case overhead of speculation on sorted data relative to the best algorithm (insertion sort) is 15.78% ($3.2\,$sec), which stems from the map faults handled by the runtime system. The worst case overhead of speculation compared to the fastest algorithm on the random data is 8.72% ($50.34\,$sec over 616 secs). This overhead stems from privatization, isolation and inclusion of the large $8\,$GB data set. Anumita achieves a speedup ranging from 0.84 (quick sort/random data) to 62.95 (heap sort/sorted data).

5.4 Energy Overhead

The primary focus of Anumita is to improve run time performance. Reducing energy consumption runs counter to this goal. However, in this section, we demonstrate that adopting coarse-grain speculation to exploit parallelism on multi-core systems does not come with a large energy consumption penalty, and in fact can reduce total energy consumption in many cases.

Energy consumption in modern multi-core processors is not proportional to CPU utilization. An idle core consumes nearly 50% of the energy of a fully loaded core [27]. There is a significant body of research in the architectures community on making the energy consumption proportional to offered

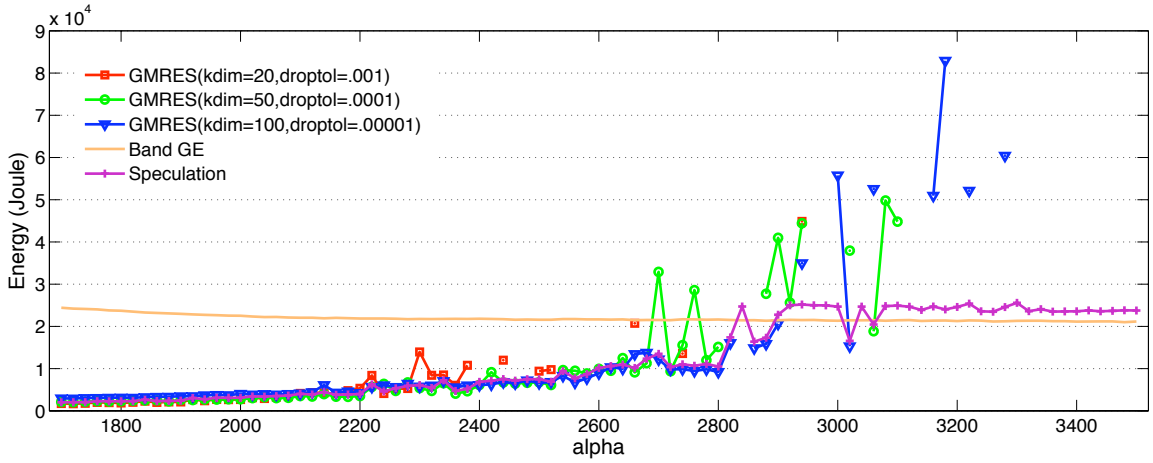

Figure 9. Energy consumption of PDE solver using surrogates in Anumita. The results show that Anumita has relatively low energy overhead.

load, which is motivated by energy consumption of large data centers that run at an average utilization of $7 - 10\%$.

To measure the energy overhead of coarse-grain speculative execution using Anumita, we connected a Wattsup Pro wattmeter to the AC input of the 16 core system running the benchmark. This device measures the total input power to the entire system. We performed the power measurement using the SPEC Power daemon (ptd), which samples the input power averaged over 1 sec intervals for the entire run time of the application. We calculated energy consumption as the product of the total runtime and the average power. We measured energy consumption under two scenarios: a) each algorithm run individually and b) speculatively execute multiple algorithms using Anumita.

Figure 9 presents the energy consumption of the PDE solver. We report the results for alpha values greater than 1700 for the PDE solver, since they have a runtime of at least a few seconds (required to make any meaningful power measurements). Comparing the most energy efficient algorithm at each alpha with the corresponding speculative execution, we found that the overall energy overhead of speculation ranged between 7.72% and 19.21%. It is comforting to see that, even in the presence of running four surrogates concurrently, Anumita incurred a maximum energy overhead of 19.21% compared to the most energy efficient algorithm.

More importantly, since the most energy efficient algorithm for a given problem in not known a priori, we again see a large robustness advantage for speculation — this time with respect to energy consumption. With a static choice of algorithm there is substantial risk that a method will fail (necessitating the use of another method) or take much longer than the best method, all of which consume more energy than the speculatively executed approach.

Figure 10 shows the energy consumption for two vertex coloring algorithmic surrogates (*tabu*, *vns*) and speculation using Anumita. In this case, Anumita speculates over three

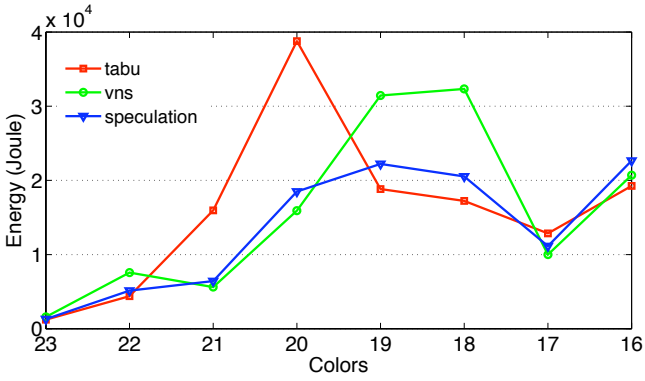

Figure 10. Energy consumption of the graph coloring benchmark for the LE_450_15c data set with a seed of 12.

algorithms (*sa*, *tabu* and *vns*) even though one of them consistently fails to color the graph. Comparing the most energy efficient algorithm at each color with the corresponding speculation, we found that the overall energy overhead due to speculation ranged between 6.08% and 16.04%.

The total energy consumed to color the graph is actually lower for speculation when compared to a static choice of either algorithmic surrogate. This is because energy is the product of power and time, and since neither algorithm is consistently better (strong input dependence), speculation results in lower time to completion of an entire test case which translates to lower energy consumption. Speculation using Anumita takes 252 seconds for the problem set with a total energy consumption of 107904 joules. *Tabu* takes a total of 321 seconds and consumes a total energy of 128531 joules for the problem set. In contrast the best static choice of the surrogate (*vns*) runs in 314 seconds and consumes 125194 joules. Speculation here is 24.6% faster in time and consumes 16.02% less energy, a result that is positive in both aspects.

5.5 Summary

Anumita provides resilience to failure of optimistic algorithmic surrogates. In both graph coloring as well as PDE solvers, not all algorithmic surrogates successfully run to completion. In the absence of a system such as Anumita, the alternative is to run the best known algorithmic surrogate and if it fails, retry with a fail-safe algorithm that is known to succeed. While this works for PDE solving example with Band Gaussian Elimination being the fail-safe, there is no clear equivalent for graph coloring, with each surrogate failing at different combinations of graph geometry and initial coloring. With modest energy overhead and sometimes savings, Anumita can significantly improve the performance of otherwise hard to parallelize applications.

6. Related Work

We categorize existing software based speculative execution models [11, 14, 18–21, 28, 30, 31, 35, 37] into two categories depending on the granularity at which they perform speculation — loops or user defined regions of code. Loop level models [11, 19–21, 30, 31, 35, 37] achieve parallelism in sequential programs by employing speculative execution within loops. While such models transparently parallelize sequential applications without requiring any effort from the programmer, their scope is limited to loops. In contrast, the second category of speculative execution models [5, 14, 18, 28, 38] allow the programmer to specify regions of code to be evaluated speculatively. We restrict our discussion to these models throughout the rest of this section.

Berger et al. [5] proposed the Grace framework to speculatively execute fork-join based multithreaded applications. Grace uses processes for state separation with virtual memory protection and employs page-level versioning to detect mis-speculations. Grace focuses on eliminating concurrency bugs through sequential composition of threads.

Ding et al. [14] proposed behavior oriented parallelization (BOP). BOP aims to leverage input dependent course grained parallelism by allowing the programmer to annotate regions of code, denoted by *possibly parallel regions* (PPR). BOP uses a lead process to execute the program non-speculatively and uses processes to execute the possibly parallel regions. When the lead process reaches a PPR, it forks a speculation and continues the execution until it reaches the end of the PPR. The forked process then jumps to the end of the PPR region and in turn acts as lead process and continues to fork speculations. This process is repeated until all the PPRs in the program are covered. BOP's PPR execution model is identical to pipelining. The lead process at the start of the pipeline waits for the speculation it forked to complete and then checks for conflicts before committing the results of the speculation. This process is recursively performed by all the speculation processes which assumed the role of the lead process. BOP employes page-based protection of shared data by allocating each shared variable in a separate page and uses a value-based checking algorithm to validate speculations.

In another study, Kelsey et al. [18] proposed the Fast Track execution model, which allows unsafe optimization of sequential code. It executes sequential (normal tracks) and speculative variants (fast tracks) of the code in parallel and compares the results of both these tracks to validate speculations. Their model achieves speedup by overlapping the normal tracks and by starting the next normal track in program order as soon as the previous fast track is completed. Fast Track performs source transformation to convert all global variables to use dynamic memory allocation so its runtime can track accesses to global variables. Additionally, Fast Track employes a memory-safety checking tool to insert memory checks while instrumenting the program. Finally, Fast Track provides the programmer with configurations that tradeoff program correctness against performance gains. In contrast, Anumita provides transparent name space isolation and it does not require any annotations to the variables in a program. Additionally, Anumita does not rely on program instrumentation.

Prabhu et al. [28] proposed a programming language for speculative execution. Their model uses value speculation to predict the values of data dependancies between coupled interactions based on a user specified predictor. Their work defines a safety condition called rollback freedom and is combined with static analysis techniques to determine the safety of speculations. They implemented their constructs as a C# library. The domains where value speculation is applicable are orthogonal to our work.

Trachsel and Gross [38, 39] present an approach called competitive parallel execution (CPE) to leverage multi-core systems for sequential programs. In their approach they execute different variants of a single threaded program competitively in parallel on a multicore system. The variants are either hand generated surrogates or automatically generated by selecting different optimization strategies during compilation. The program's execution is divided into phases and the variants compete with each other in a phase. The variant that finishes first (temporal order) determines the execution time of that phase, thereby reducing the overall execution time. In contrast, Anumita is capable of supporting both sequential and parallel applications and provides expressive evaluation criterion (temporal and qualitative) to evaluate speculations.

Praun et al. [40] propose a programming model called implicit parallelism with ordered transactions (IPOT) for exploiting speculative parallelism in sequential or explicitly parallel programming models. The authors implement an emulator using the PIN instrumentation tool to collect memory traces and emulate their proposed speculation model. In their work, they propose and define various attributes to variables to enable privatization at compile time and avoid conflicts among speculations. In contrast, as mentioned previ-

ously, Anumita does not require annotations to variables or rely on binary instrumentation. Instead, Anumita provides isolation to shared data at runtime.

In another study, Cledat et al. [12] proposed opportunistic computing, a technique to increase the performance of applications depending on responsiveness constraints. In their model multiple instances of a single program are generated by varying input parameters to the program, which then compete with each other. In contrast, Anumita is designed to support speculation at arbitrary granularity as opposed to the entire program.

Ansel et. al. [3] proposed the PetaBricks programming language and compiler infrastructure. PetaBricks provides language constructs to specify multiple implementations of algorithms in solving a problem. The PetaBricks compiler automatically tunes the program based on profiling and generates an optimized hybrid as a part of the compile process. In contrast, our approach performs coarse-grain speculation at runtime and is hence better suited for scenarios where performance is highly input data dependent.

Additionally, certain compiler directed approaches [7, 16, 17, 22, 24, 26] provide support for speculative execution and operate at the granularity of loops. Such approaches rely on program instrumentation [17], use hardware counters for profiling [17] or binary instrumentation to collect traces [24, 26] in order to optimize loops. In contrast to such systems, Anumita is implemented as a language independent runtime system. The main goal of Anumita is to simplify the notion of speculative execution.

Finally, a nondeterministic programming languages (e.g., Prolog, Lisp) allows the programmer to specify various alternatives for program flow. The choice among the alternatives is not directly specified by the programmer, however the program at runtime decides to choose between the alternatives [1]. Several techniques such as backtracking and reinforcement learning are commonly employed in choosing a particular alternative. It is unclear if it is the responsibility of the programmer to ensure and correct the side-effects of the alternatives. Anumita represents a concurrent implementation of the non-deterministic choice operator. The contribution here is to introduce this notion and an efficient implementation to imperative programming.

7. Future Work

We are continuing to improve Anumita. We are presently working on extending support for disk IO among speculative surrogates. While Anumita simplifies the subtleties of coarse-grain speculative parallelism by providing simple sequential semantics, the programmer must identify the scope for speculation. We plan to automate this aspect of our system. Currently there is an ongoing effort [2, 9, 10] to extend C++ to include threading models. We propose that *speculation should also be a natural extension of the imperative languages and the speculation model should be a natural extension to threading models*. We plan to to investigate extending language support for speculation.

8. Conclusions

In this paper we presented Anumita, a language independent runtime system to achieve coarse-grain speculative parallelism in hard to parallelize and/or highly input dependent applications. We proposed and implemented programming constructs and extensions to the OpenMP programming model to achieve speedup in such applications without sacrificing performance, portability and usability. Experimental results from a performance evaluation of Anumita show that it is (a) robust in the presence of performance variations or failure and (b) achieves significant speedup over statically chosen alternatives with modest overhead. The implementation of Anumita and the benchmarks used in this study will be made available for public download.

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996. ISBN 0262011530.

[2] S. V. Adve and H.-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *Communications of the ACM*, 53:90–101, August 2010. ISSN 0001-0782. URL `http://doi.acm.org/10.1145/1787234.1787255`.

[3] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1.

[4] R. Barrett, M. Berry, J. Dongarra, V. Eijkhout, and C. Romine. Algorithmic bombardment for the iterative solution of linear systems: A poly-iterative approach. *Jnl. of Computational & Appl. Math.*, 74:91–110, 1996.

[5] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications*, pages 81–96. ACM, 2009. ISBN 978-1-60558-766-0.

[6] S. Bhowmick, L. C. McInnes, B. Norris, and P. Raghavan. The role of multi-method linear solvers in pde-based simulations. In *ICCSA (1)*, pages 828–839, 2003.

[7] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 99–108, New York, NY, USA, 2002. ACM. ISBN 1-58113-529-7.

[8] C. Blundell, E. Lewis, and M. Martin. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2):17, 2006. ISSN 1556-6056.

[9] H.-J. Boehm. Threads Cannot be Implemented As a Library. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI

'05, pages 261–268, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. URL http://doi.acm.org/10.1145/1065010.1065042.

[10] Boehm, Hans-J. and Adve, Sarita V. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI 2008, pages 68–78, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. URL http://doi.acm.org/10.1145/1375581.1375591.

[11] T. Chen, M. Feng, and R. Gupta. Supporting speculative parallelization in the presence of dynamic data structures. In *PLDI '10: Proceedings of ACM SIGPLAN 2010 conference on Programming Language Design and Implementation*, volume 45, pages 62–73, New York, NY, USA, 2010. ACM.

[12] R. Cledat, T. Kumar, J. Sreeram, and S. Pande. Opportunistic Computing: A New Paradigm for Scalable Realism on Many-Cores. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, pages 5–5, Berkeley, CA, USA, 2009. USENIX Association. URL http://portal.acm.org/citation.cfm?id=1855591.1855596.

[13] DIMACS. Discrete Mathematics and Theoretical Computer Science, A National Science Foundation Science and Technology Center. http://dimacs.rutgers.edu/, April 2011.

[14] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI '07: Proceedings of ACM SIGPLAN 2007 conference on Programming Language Design and Implementation*, volume 42, pages 223–234, New York, NY, USA, 2007. ACM.

[15] Doug Lea. A memory allocator. http://g.oswego.edu/dl/html/malloc.html, April 2011.

[16] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *PLDI '04: Proceedings of ACM SIGPLAN 2004 conference on Programming Language Design and Implementation*, volume 39, pages 59–70, New York, NY, USA, 2004. ACM.

[17] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Speculative thread decomposition through empirical optimization. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 205–214, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-602-8.

[18] K. Kelsey, T. Bai, C. Ding, and C. Zhang. Fast Track: A Software System for Speculative Program Optimization. In *CGO '09: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 157–168, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3576-0.

[19] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic Parallelism Requires Abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 211–222, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2.

[20] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic Parallelism Benefits from Data Partitioning. In *ASPLOS XIII: Proceedings of the 13th International conference on Architectural Support for Programming Languages and Operating Systems*, volume 36, pages 233–243, New York, NY, USA, 2008. ACM.

[21] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. How much Parallelism is There in Irregular Applications? In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 3–14, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-397-6.

[22] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: a TLS compiler that exploits program structure. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 158–167, New York, NY, USA, 2006. ACM. ISBN 1-59593-189-9.

[23] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS XIII: Proceedings of the 13th International conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–339. ACM, 2008. ISBN 978-1-59593-958-6.

[24] Y. Luo, V. Packirisamy, W.-C. Hsu, A. Zhai, N. Mungre, and A. Tarkas. Dynamic performance tuning for speculative threads. In *ISCA '09: Proceedings of the 22nd annual International Symposium on Computer Architecture*, volume 37, pages 462–473, New York, NY, USA, 2009. ACM.

[25] Marco Pagliari. Graphcol: Graph Coloring Heuristic Tool. http://www.cs.sunysb.edu/~algorith/implement/graphcol/implement.shtml, April 2011.

[26] P. Marcuello and A. González. Thread-Spawning Schemes for Speculative Multithreading. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 55, Washington, DC, USA, 2002. IEEE Computer Society.

[27] Patterson, David A. and Hennessy, John L. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008. ISBN 0123744938, 9780123744937.

[28] P. Prabhu, G. Ramalingam, and K. Vaswani. Safe Programmable Speculative Parallelism. In *PLDI '10: Proceedings of ACM SIGPLAN 2010 conference on Programming Language Design and Implementation*, volume 45, pages 50–61, New York, NY, USA, 2010. ACM.

[29] H. K. Pyla and S. Varadarajan. Avoiding Deadlock Avoidance. In *PACT 2010: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, 2010.

[30] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *ASPLOS XV: Proceedings of the 15th International conference on Architectural Support for Programming Languages and Operating Systems*, volume 38, pages 65–76, New York, NY, USA, 2010. ACM.

[31] L. Rauchwerger and D. A. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Transactions on Parallel Distributed Systems*, 10(2):160–180, 1999. ISSN 1045-9219.

[32] J. R. Rice and R. F. Boisvert. *Solving Elliptic Problems Using ELLPACK*. Springer–Verlag, 1985.

[33] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing, Boston, 1996.

[34] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990.

[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. ISSN 0734-2071.

[36] Thomas Wang. Sorting Algorithm Examples. `http://www.concentric.net/~ttwang/sort/sort.htm`, April 2011.

[37] C. Tian, M. Feng, N. Vijay, and G. Rajiv. Copy or Discard execution model for speculative parallelization on multicores. In *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 330–341, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-1-4244-2836-6.

[38] O. Trachsel and T. R. Gross. Variant-based competitive Parallel Execution of Sequential Programs. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, pages 197–206, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0044-5.

[39] O. Trachsel and T. R. Gross. Supporting Application-Specific Speculation with Competitive Parallel Execution. In *3rd ISCA Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures*, PESPMA'10, 2010.

[40] C. von Praun, L. Ceze, and C. Caşcaval. Implicit Parallelism with Ordered Transactions. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP 2007, pages 79–89, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-602-8. URL `http://doi.acm.org/10.1145/1229428.1229443`.

[41] W. Zhang, C. Sun, and S. Lu. Conmem: detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS XV:Proceedings of the 15th International conference on Architectural Support for Programming Languages and Operating Systems*, pages 179–192, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-839-1.