Compiling Sequential Programs for Speculative Parallelism

Chih-Po Wen Katherine A. Yelick
Computer Science Division
University of California, Berkeley, CA 94720

Abstract

We present a runtime system and a parallelizing compiler for exploiting speculative parallelism in sequential programs. In speculative executions, the computation consists of tasks which may start before their data or control dependencies are resolved; dependency violation is detected and corrected at runtime. Our runtime system provides a shared memory abstraction and ensures that shared accesses appear to execute in the proper order. Our compiler transforms sequential programs into parallel tasks and manages control dependencies. It also optimizes the parallel program using data flow analysis to reduce the cost of speculative execution. We demonstrate our approach through the parallelization of an example application, and report on its performance on a distributed memory multiprocessor.

Keywords: parallel compilers, runtime systems, speculative parallelism, optimistic concurrency, optimizing compilers.

1 Introduction

Speculative parallelism can be used to increase the amount of available parallelism in some applications. The idea is to make the optimistic, or speculative, assumption that no dependencies exist between different tasks (e.g., loop iterations) in a program; those tasks are executed in parallel, and the runtime systems detects and corrects any dependencies that occur out of order. Optimistic concurrency has been applied to transaction management [7] and discrete event simulation [6]. In previous work, we showed that optimistic concurrency in timing simulation leads to speedups over 50 on a 64 processor CM5, and that the observed speedups exceed even the theoretical speedups of the conservative approach on ideal hardware (assuming free communication and unlimited processors) [11].

For applications with infrequent, statically unpredictable dependencies, the benefits of speculative parallelism are clear. The drawback of the approach is the programming overhead of developing such applications. The goal of this work is to provide the benefit of speculative parallelism with minimal programming effort. We have built a prototype runtime system that manages scheduling and side-effects of speculative parallelism. It is similar to the timewarp system of Jefferson [6], but whereas timewarp uses a process and message view of the computation, we use a shared memory view. We have also built a parallelizing compiler that transforms sequential programs into speculative shared memory parallel programs.

Our compiler differs from most parallelizing compilers. These compilers exploit only conservative parallelism, and may therefore fail to discover parallelism due to the lack of compile-time information. For example, loops with unknown loop bounds or array accesses with unknown indices may prevent the compiler from obtaining precise dependency information. Running on conventional shared memory, programs compiled by our compiler could produce incorrect results, but on our shared memory runtime system, they will have the same functionality as the original sequential code. Because the runtime overhead of speculation can be high, another important feature of our compiler is extensive optimizations based on data flow analysis.

The remainder of this paper is organized as follows. Section 2 gives an overview of speculative execution. Section 3 describes the interface and implementation of the runtime system and Section 4 describes the compiler. Section 5 demonstrates the use of the compiler on an example program. Section 6 relates this work to

[0] This work was supported in part by the Semiconductor Research Consortium under contracts 92-DC-008 and 93-DC-008, by the National Science Foundation as a Research Initiation Award (number CCR-9210260) and Infrastructure Grant (number CDA-8722788), by the Advanced Research Projects Agency of the Department of Defense monitored by the Office of Naval Research under contract DABT63-92-C-0026, by Lawrence Livermore National Laboratory (task number 33), and by AT&T. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

earlier work on speculative parallelism. In Section 7 we draw our conclusions and describe future work.

2 Overview of Speculative Parallelism

The compiler decomposes a program execution into *tasks*, which typically correspond to a structured program fragment like a loop body. The sequential execution of a program defines a total order on tasks, which we refer to as the *time* of a task. Under speculative parallelism, possibly dependent tasks are started in parallel. A parallel execution is considered *correct* if it conforms to all of the data and control dependencies of the original sequential program.

In our system, if dependencies occur out of order in the parallel execution, the side-effects of the tasks are undone, and the system rolls back to a *consistent* state, i.e., a state that exists in the sequential execution. The state before the execution of a task must be preserved for rollbacks. Therefore, a task never overwrites a value created by an earlier task; each write is made to a fresh copy of the variable, called a *version*, which is owned by the task. The copies can be read by later tasks for speculative execution. Canceling a task involves discarding all versions of variables owned by the task, and (transitively) canceling all tasks that have read such versions. The underlying assumption is that runtime parallelism exists in the computation, even if it cannot be found statically, and that rollbacks are seldom necessary.

3 The Runtime System

In this section we present a runtime system that supports the above execution model. The system ensures that data dependencies are respected. Control dependencies are enforced by the compiler using synchronization variables, which are explained in section 4.

3.1 Interface

The runtime system provides a shared memory space for all tasks. Tasks communicate their states by reading and writing shared variables. On distributed memory multiprocessors, accessing a shared variable may require communication with a remote processor. To cope with this latency, shared variable accesses are separated into an initiation and a completion phase, so that the latency can be overlapped with useful computation.

The generation and scheduling of tasks are managed by a *task queue*, which is physically distributed among the processors. The task queue does not ensure any global order on task executions, but within a processor it prefers tasks with earlier task times. This preference reduces the chance of task cancellation.

Due to speculation, the states of variables observed by a task are not guaranteed to be consistent with the sequential execution. A task is said to have *completed* if it has finished executing, it observed only consistent variable states, and all earlier tasks (in sequential time) have completed. As a parallel execution progresses, the set of completed tasks will be an increasing prefix of the tasks from the sequential execution. We provide a primitive to assess the progress of computation, which is the time of the last completed task, referred to as the *global virtual time* or GVT.

Global virtual time is to detect termination, and to determine which variable versions may be discarded. Since each task creates its own versions of some variables, memory availability may become a limitation over time. *Fossil collection*, which uses GVT, is invoked periodically to reclaim the unneeded memory.

The interface of the runtime system is summarized below:

- `enroll(T,P)`: enqueue a new task `T` at processor `P`.
- `read(X,buf,T,flag)`: initiate the read for the variable `X` by the task `T`. The local variable `flag` is decremented by one when `buf` has the desired value.
- `write(X,buf,T,flag)`: initiate the write to the variable `X` by the task `T`. `buf` stores the value to be written, and the local variable flag is decremented when the write completes.
- `compute_GVT()`: compute and return the current GVT. `compute_GVT` is non-blocking, and it may return the value zero, signaling that the result is not valid yet.
- `collect_fossil(X)`: collect fossils for the variable `X`.

3.2 Implementation

This section sketches the protocols for maintaining data dependency in speculative executions. The protocol we propose eliminates dependencies (read/write and write/write) due to name conflicts. The protocols fall into two categories, one for accessing shared variables, and the other for computing global virtual time and collecting fossils. In describing the protocols, we use terms such as "time", "earlier", "later", and "most recent" to relate objects with different sequential execution orders; they do not refer to the wall clock time.

Reads: A read by the task T to the variable X returns the most recent version of X created by some task earlier than T. The read access is recorded so that T can be canceled if a write to X occurs at a time before T's time.

Writes: A write by the task T to the variable X creates a version of X for T. The new version has the same time as T. Each write cancels tasks that have read a *stale* version of V; a version is stale for the task T' if it is earlier than the newly created version, and T' is later than T. Each write is recorded in preparation for possible cancellation.

Cancellation: Canceling the task T invalidates the versions created by T and reschedules T for execution.

Invalidation: Invalidating a version discards the value stored in the version and reclaims its memory. It also cancels all tasks that have read the version, which in turn invalidates the versions created by such tasks.

Computing GVT: Each unfinished task has a sequential task time, also called a *local virtual time* or LVT. Global virtual time is the minimum of all tasks' LVT, and its computation requires taking a snapshot of the system.

Fossil collection: Since no rollback can affect objects with time earlier than GVT, only the latest version of those earlier than GVT will ever be needed. Collecting fossils simply reclaims the space of the obsolete versions.

The two main data structures in the system are for variables and tasks. Each variable is represented by a linked list of all its unfossilized versions, stored in increasing time order. Each version `V` stores `V.t`, the time of the task that created V, `V.val`, the value of the variable, and `V.reads`, a link list of pointers to tasks that have read `V`. Each task `T` is represented by `T.writes`, a link list of pointers to versions created by the task. It also stores `T.t`, the time of the task, and `T.id`, a descriptor that tells what the task does.

4 The Compiler

The runtime environment can be used directly for writing explicitly parallel code in a shared memory style. It has built-in support for optimistic concurrency, and hides the details of rollbacks and memory management from the programmer. In this section, we go one step further and describe a parallelizing compiler that will automatically produce code for the runtime system, given an ordinary sequential program.

A naive way to generate code with speculative parallelism is to let every basic code block be a task body, and have every variable be a shared variable. The resulting code is correct, but is extremely inefficient. Instead, our compiler uses data flow analysis to selectively parallelize code fragments. Information on the definition and use of variables are used to selectively parallelize loops and to optimize the variable accesses. The compiler exploits parallelism only across loops.

Using live variable analysis, the compiler recognizes temporary variables whose values are not shared across statements. Local copies can be used for such variables, and their accesses do not have to go through the expensive consistency protocols. To further reduce the protocol overhead, we employ a caching mechanism to capture repeated accesses to the same variable. Prefetching and write pipelining are also used when possible to hide the latency of accesses on remote processors. These optimizations and others are detailed in Sections 4.1, 4.2, and 4.3.

Our current compiler is a prototype, developed to test the basic approach of compiling for speculative parallelism. We therefore make a number of simplifying assumptions about the source language. The compiler accepts a subset of C with arrays and structures but no pointers. The transformations do not use interprocedural analysis, so we assume that all functions (except `main()`) are side-effect free. Functions that have side-effects are handled by inlining. Because I/O operations are irrevocable, they must be recognized by the compiler so that they are executed in order. We require that I/O statements are tagged as such by the programmer. Exceptions are similarly irrevocable, unless non-abortive exception handling mechanisms exist in hardware for recording the exceptions; we do not address this issue in the current implementation.

The compiler produces parallel code for a MIMD multiprocessor with calls to our runtime system. The system runs on the CM5 multiprocessor from Thinking Machines, a distributed memory machine in which each processor has a separate address space. Communication is performed by the Active Message layer [10], which supports fast small messages and a form of lightweight remote procedure call. The design does not rely on CM5-specific hardware, although a low overhead message layer is important to performance. Our compiler is based on the retargetable ANSI C compiler lcc[4]. We augmented lcc to perform data flow analysis and to generate the parallel code.

We will use the program in Figure 1 as an example to illustrate our compilation techniques. The program shown is a simulated annealing solver for the traveling salesman problem (TSP). The application itself is discussed in detail in Section 5.

4.1 Managing Parallelism

Traditional parallelizing compilers prove certain loops are parallel. The analysis is conservative since truly parallel loops may not be recognized due to the lack of runtime information. Our compiler parallelizes more aggressively by proving certain loops are sequential. In this case sequential loops may be mistaken as parallel, but the runtime system guarantees that their dependency is respected during execution. We believe that more accurate tests are available to our compiler, since they do not need to be conservative.

Some basic data-flow information is needed by the compiler. During parsing, the compiler maintains the sets $def(S)$ and $use(S)$ to record the variables that are defined (written) and used (read) in each statement S. Each of these sets contains the *must* and *may* subsets: a *must* definition or use will occur in all possible executions; a *may* definition or use occurs only in some executions, due to the unpredictable control flow. [1]

Induction Variable Elimination. The first step in parallelizing most loops is to eliminate its induction variables. We take a simplified view of induction variables in the current implementation: any assignment to an induction variable `I` has the form `I = I + c` or `I = I - c`, where c is some integer constant. After parsing, the compiler examines individual loops to identify induction variables; it recognizes those whose definitions are *must* definitions. To eliminate the recurrence caused by the induction variable `I`, the compiler appends the statement `I_0 = I` to the loop initialization statement, where `I_0` is a new temporary, and prepends the statement `I = I_0 + C * loopindex()` to the loop body, where `C` is the total increment to `I` for one iteration.

[1] Dependence analysis on arrays is beyond the scope of this project. Our prototype compiler treats array elements as scalars, and only the *may* type of data flow information is applicable to array accesses.

```
main()
{
 int adj[20][20];
 struct {int city[20];} BestSol, NewSol;
 int BestCost, NewCost;
 float temp;
 int i,j,s1,s2,tmp;
 int take;
 int mob;


for (i=0;i<20;++i)
   for (j=0;j<20;++j)
       adj[i][j] = 100000;
for (i=0;i<20;++i)
   adj[(i+1)%20][i] = 100;
srandom();
for (i=0, BestCost = 0; i<20; ++ i) {
   BestSol.city[i] = i;
   BestCost += adj[i][(i+1)%20];
}
for (temp=5000.0;temp>1.0;temp=temp*0.9) {
   mob = log(temp)+1;
   for (i=0; i<1000; ++i) {
      NewSol = BestSol;
      for (j=0; j< mob; ++j) {
         s1 = random() % 20;
         s2 = s1;
         while (s1 == s2)
            s2 = random() % 20;
         tmp = NewSol.city[s1];
         NewSol.city[s1] = NewSol.city[s2];
         NewSol.city[s2] = tmp;
      }
      for (j=0, NewCost = 0; j<20; ++ j)
         NewCost += adj[NewSol.city[j]]
                    [NewSol.city[(j+1)%20]];
      if (BestCost > NewCost)
         take = 1;
      else if ((random()%1000) <
         exp(((double)BestCost-NewCost)/temp)*1000)
         take = 1;
      else
         take = 0;
      if (take) {
         BestCost = NewCost;
         BestSol = NewSol;
      }
   }
}
IO( printf("Best Cost = %d\n",BestCost) );
}
```

Figure 1: The sequential TSP program.

Detecting Parallelism After elimination of induction variables, the compiler computes the live variable information for each statement. We refer to the set of live variables upon entry and exit of the statement S as *livein(S)* and *liveout(S)*, respectively. The readers are refered to the standard compiler texts [1] for details.

Consider the parallelization of a loop, with loop body L. Here, and in the rest of the paper, we treat the loop initialization statements as separate from the loop body. Define $livedef(L)$ as the set of variables L computes that are used by later statements, i.e., $livedef(L) = liveout(L) \bigcap def(L)$. Similarly, $liveuse(L)$ is the set of variables computed by previous statements and used by L, i.e., $liveuse(L) = livein(L) \bigcap use(L)$. The loop is *provably sequential* if $livedef(L) \bigcap liveuse(L)$ is nonempty, where *livedef* and *liveuse* refer to the "must" form of def/use information.

We regard a loop as potentially parallel if it is not provably sequential. Among all potentially parallel loops, we selectively parallelize some based on their ratio of shared variable accesses to local operations. We do not parallelize loops with high ratios, since the runtime system overhead may make the parallelization nonprofitable. For the same reason we parallelized only the outermost loop of a nested parallel loop. Note that the ratio is derived from crude static estimates at compile time. [2]

We use the program in Figure 1 as an example. For the rest of the presentation, we label each loop by its first line. Only Loop 18,22, and 35 are sequential; all other loops are potentially parallelizable. Loop 12,13, and 15 are not parallelized because of their high communication/computation ratios. Loop 24, instead of loop 26, is selected for parallelization because it is the outermost loop.

Unknown Loop Bounds. To deal with loops with unknown loop bounds, our compiler speculatively generates loop iterations and uses *synchronization variables* to ensure that the control dependency is respected. We add the synchronization variable $SYNC$ to each parallelized loop. $SYNC$ is initialized to 0, and is read at the beginning of each task. $SYNC$ is set to 1 when the loop termination criterion is met. A task aborts immediately if the value read is nonzero. By enforcing data dependencies on $SYNC$, we ensure that extraneous iterations cause no side-effects.

Increasing Task Granularity. Each iteration in a parallel loop constitutes an indivisible unit of execution. We can lump multiple iterations into one task to increase the granularity of parallel execution (the number of iterations per task is referred to as the *inner loop size*).

Using large granularity tasks has two advantages. First, it reduces the bookkeeping overheads in the consistency protocol, since there are fewer tasks to manage. Second, it reduces the amount of communication due to shared accesses, since multiple iterations can share data in the local cache (described in Section 4.3).

The disadvantage of large granularity tasks is the reduction in parallelism. The compiler acknowledges the tradeoff by making the task granularity a runtime parameter, which is configured upon execution.

4.2 Managing Variables

After the parallelism is exposed, the compiler determines the layout of the variables and performs extensive optimizations to reduce interprocessor communication. We now details the techniques used in the compiler in optimizing variable accesses.

Each variable in the program is given one of the two storage types by the compiler: *private* or *shared*. The value of a private variable is always kept valid (up-to-date) on all processors. The value of a shared variable is valid only on a single *owner* processor; all other processors must communicate with that processor to access the variable.

Each program fragment in the program is given one of three execution modes by the compiler: *redundant* execution, *sequential* execution, and *parallel* execution. The modes are described below:

Redundant: all processors execute the same code and all accesses are local. The execution mode updates the values of all private variable without requiring interprocessor communication.

Sequential: Only one processor (referred to as the *master* processor) executes the code. Accesses to shared variables may be remote, but do not need to be protected (e.g., via the runtime system) since only one thread can be executing. The execution mode updates the value of shared variables.

[2] We require that all I/O be marked in the source code by an `IO` directive. These statements receive a special variable IO_DEVICE, as both a *liveuse* and *livedef* variable, so the analysis will find that a loop containing I/O is not potentially parallelizable.

R: set of private variables.
T: the parse tree for the program.
S: the root node of T. livein(S) = liveout(S) = {IO_DEVICE}

Algorithm classify:

```
0. R = set of all variables
1. mark every node C that represents a parallelized loop
       as parallel
2. for every node C in T that is marked parallel
       R = R - livedef(C)
3. R = R - IO_DEVICE
4. repeat until no change to R:
     for every unmarked child node C of S
         if (liveuse(C) ⋃ livedef(C)) ⋂ R̄) is nonempty,
             or C contains a parallel statement
                 mark C as sequential
                 R = R - def(C)
```

Figure 2: Pseudo code for determining execution modes and storage types. The algorithm always reaches a fix point, since the size of `R` is non-increasing.

Parallel:. All processors speculatively execute *different* iterations of the same loop. Access to shared variables are coordinated by the runtime system to ensure consistency. This execution mode accounts for all the speedup.

Note that the execution mode of a code fragment is tied to the set of variables it accesses. For example, writing a private variable in sequential or parallel executions is not allowed, since the update may not occur on all processors. Because the execution modes and storage types are mutually dependent, we use an iterative algorithm to solve the problem. The pseudo code is given in Figure 2. The algorithm returns with the sequential and parallel statements marked. The unmarked top-level statements (i.e., children of of `S`) are executed in the redundant mode. [3]

The only private variable in Figure 1 is the array `adj`. The Statements from line 12 thru 17 are executed redundantly on all processors. Loop 18, loop 22 (excluding loop 24), and line 51 are executed sequentially on the master processor. Loop 24 are executed in parallel (speculatively).

Classifying Variable Accesses. Accesses to variables are classified as *local*, *remote* or *speculative*. Local accesses are to private variables. Remote accesses are unprotected accesses made to the owner's copy of the shared variables. Speculative accesses use the runtime system primitives on shared variables.

It appears that accesses to shared variables are always expensive. However, there exist situations where local copies can be used. A variable X can be renamed within the scope of a statement S if X is not in $liveuse(S) \bigcup livedef(S)$. In this case X is a temporary variable for the statement S. Accesses to such variables in statements containing a single mode of execution are candidates for optimizations.

Another opportunity for optimization is the remote accesses in parallel executions. Given a parallel loop P, all shared variables in $liveuse(P)$ but not in $livedef(P)$ can be accessed using the local copies, provided that their values are prefetched before the parallel execution begins. Since it is not feasible to prefetch large arrays, the compiler performs this optimization for scalar variables only.

The rules for accessing variables are summarized below:

- Accessing private variables: always local.
- Accessing shared variables in sequential statements: local for temporary variables, and remote for others.
- Accessing shared variables in parallel statements: local for temporary variables, and for prefetched shared variables; shared for variables in $livedef$ of the parallel loop body; remote otherwise.

We use Figure 1 again to illustrate the classification of accesses. All variables are accessed locally in the redundant execution of statement 12 thru 17. In loop 18 (sequential mode), only `BestSol` and `BestCost` need remote accesses; `i` is treated as a temporary variable. In loop 22 (sequential mode), accesses to `mob` and `temp` are remote, since they may be used by the enclosed parallel loop. In loop 24 (parallel mode), `mob` and `temp` are first prefetched from their owner processors, and then accessed locally in the parallel execution. `BestCost` and `BestSol` are accessed via the runtime system routines to ensure consistency. Other variables in loop 24 uses local accesses since they are temporaries. Finally, statement 51 reads `BestCost` from the owner processor and prints it to the standard output.

Replicating Shared Variables. Accesses to variables that may be write-shared in a parallel loop are serviced by their owner processors. However, if the loop has much parallelism, writes will occur infrequently. Therefore, we can cache the variables based on a simple write-broadcast scheme to eliminates reads

[3] A sequential statement may contain parallel statements. This is the only case where the execution mode changes within a statement.

on remote processors. A variable can be cached on every processor, or on a subset of processors. The former is suitable for scalar variables, since they are more likely to be used by all processors. The latter should be used for arrays.

For shared array variables, the compiler can perform array index analysis to determine the subset of processors for caching, assuming that loop iterations are statically distributed among the processors. However, this is not possible when the array indices cannot be determined at compile time. We think that a runtime approach is more suitable. The current implementation of the compiler does not cache array variables.

In Figure 1, `BestCost` and `BestSol` are replicated on all processors in the parallel execution of loop 24. All reads to these variables are still made via the runtime system. However, they do not incur any communication cost since they are serviced by the local processor.

4.3 Local Caching

Local caching refers to caching shared variables during the life time of a single task. It can be thought of as introducing new temporaries to store intermediate results, so that all but one expensive reads or writes (via the runtime system) to the same variable can be eliminated. A runtime approach is used to implement local caching. Each local copy of a variable is associated with a generation number, which is matched against the generation number of the current task to validate the version. Incrementing the task generation number by one invalidates all cache copies. Note that caching does not cause consistency problems, since the parallel loop is derived from a sequential program, where each iteration appears to execute atomically.

Prefetching can be used for shared variables that must be used or defined. Prefetches can be pipelined to reduce latency. All dirty copies must be written back at the end of the task. A list of variable identifiers is used to keep track of dirty copies for write back. The writes can also be pipelined to reduce total latency.

5 Example Application

We now explain the simulated annealing TSP solver in Figure 1. Simulated annealing, also known as probabilistic hill climbing, contains a series of randomized optimizations based on a *cooling schedule*. The algorithm starts with a very high temperature and gradually cools down to a terminal temperature. At high temperatures, the optimization step may accept solutions with higher cost in the spirit of hill climbing; At low temperatures, the scheme reduces to a greedy algorithm to zero in to the minimum cost solution. Simulated annealing is backed by an asymptotic global convergence theorem, and is frequently used to solve NP-hard problems in CAD.

5.1 Parallel Code

The parallel code generated for the CM5 multiprocessor counts about 660 lines (excluding the runtime system routines). Therefore, we show only the sketch of the target code below:

```
Master processor:   Other processor:

line 12-17          Line 12-17
loop 18
loop 22             while not done
  line 23
  synchronize         synchronize
  prefetch            prefetch
  loop 24             loop 24
signal done
line 51
```

The code generated for each parallel iteration (in loop 24) is very efficient in terms of communication. There are only 3 reads to the shared variables; all of them are made to the local memory, since scalar shared variables are replicated. There are 2 shared writes when the variable *take* evaluates to 1, and one additional write to the loop synchronization variable when the loop termination criterion is met. All remote reads and writes are performed outside the parallel loop, and their costs are negligible.

5.2 Performance

The parallel program is run on a CM5 multiprocessor using up to 32 processors. We also vary the size of the inner loop from 1 iterations to 32 iterations. Figure 3 summarizes the speedups against the sequential program running on a single node of the CM5. The TSP example is relatively small, and does not have enough computation to justify the use of a large parallel machine. Nevertheless, these results show that measurable speedups from speculative concurrency can be obtained automatically, starting from a sequential program.

The average speedup peaks when there are about 10 processors. This is due to the tradeoff between the

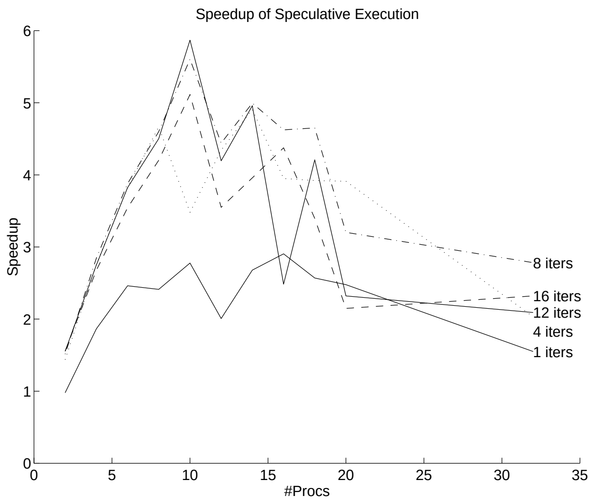

Figure 3: Speedups of the speculative executions of TSP. Each curve corresponds to a different task granularity (inner loop size).

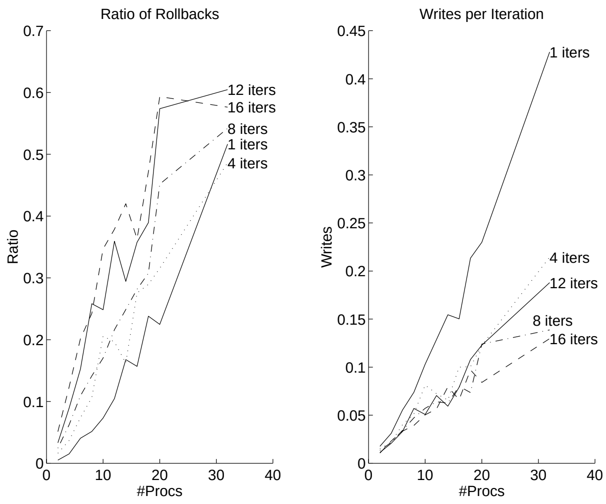

Figure 4: Costs of speculative execution. The left figure indicates the ratio of rollback tasks, and the right figure indicates the communication (shared writes) induced per iteration. Each curve corresponds to a different task granularity.

amount of parallelism, and the overheads of speculation (communication and rollback). The graph also shows that the speedups are best when the inner loop contains about 6 to 8 iterations. This is due to the tradeoff between task granularity and parallelism.

The performance tradeoffs are further illustrated in Figure 4. It indicates that the chance of rollback (i.e., dependency among tasks) increases when more processors are used, or when more iterations are packaged in one task. It also shows the average number of writes made per inner loop iteration, which represents the communication and computation ratio of a task, decreases as more iterations are lumped into one task. The reduction in communication is a result of local caching. Note that the number of writes are very low (calculaton shows fewer than 0.1 write broadcasts per task), which justifies our decision to replicate the shared variables. The figure also shows that increasing the number of processors causes more false writes.

6 Prior Work

The difference between the various approaches to exploiting speculative parallelism lies mainly in the amount of effort required by the applications programmer. On one extreme the programmer parallelizes by hand, and turns all shared variable accesses into messages. This section compares the approaches and clarifies their scopes.

At the lowest level, the computation is thought of as a collection of logical processes exchanging point-to-point messages. This abstraction is used for discrete event simulation using the Timewarp system proposed by Jefferson [6]. Message passing programming is reasonable when communication between processes is restricted to a small set of message types, but in general it can be cumbersome.

Bacon and Strom [2] describes optimistic parallelization in the context of CSP (Communicating Sequential Processes). Starting from a parallel language, their approach is more general than ours since they can, for example, express nondeterministic programs. They also require that the user tag processes with a set of *guesses* for it's incoming messages.

Another approach consists of processes sharing the same address space. Dependencies among processes are arbitrated by their timestamps, which are managed explicitly by the programmer. Partitioning and scheduling are also left to the programmer. Work at this level includes our runtime system and the space-time memory proposed by Ghosh and Fujimoto [5].

The simplest programming model is to allow the programmer to write a sequential program (with a single flat address space), and have the compiler and runtime system do all the consistency management and optimization. This is the goal of our project. Tinker and Katz also describe a runtime approach in the context of Scheme programs [9]. Their work is similar to our runtime system, but without the extensive compiler optimizations.

7 Summary

Speculative parallelism is known to be a useful technique for achieving high performance for certain applications such as discrete event simulations. To alleviate the programming difficulties of such applications, we present a system for automatic speculative parallelization of sequential programs. The results of this paper are listed below:

- We designed and implemented a runtime system which provides a shared memory abstraction for exploiting speculative parallelism.
- We developed a prototype parallelizing compiler on top of the runtime system. The compiler inputs sequential programs and produces code for the CM5 distributed memory multiprocessor. It also performs extensive optimizations to make the target code communication efficient.
- We demonstrated our approach on a simulated annealing program, and highlighted some of the performance tradeoffs.

Further work is needed to improve the performance of compiled code by incorporating further optimizations, for example, better analysis of array indexes. In addition, To support an application like the PARSWEC circuit simulator [11], we believe it is necessary to use information about the data types in the program, rather than reducing the analysis to the level of reads and writes on simple variables. We are currently investigating the compilation of abstract data types such as priority queues, which will make it possible to write programs with dynamic dependence patterns. This paper is the first step towards this more general system for speculative parallelism: we have demonstrated that it is possible to exploit speculative parallelism with minimal programming effort, using a combination of compiler and runtime support.

References

[1] A.V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] D.F. Bacon and R.E. Strom. Optimistic parallelization of communicating sequential processes. In *Proc. Third Symposium on Principles and Practice of Parallel Programming*, April 1991.

[3] C.W. Fraser and D.R. Hanson. A code generation interface for ansi c. *Software – Practice and Experience*, 21(9), September 1991.

[4] Kaushik Ghosh and Richard M. Fujimoto. Parallel discrete event simulation using space-time memory. In *Proc. International Conference on Parallel Processing*, 1991.

[5] D.R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3), July 1985.

[6] H.T. Kung and J.T. Robinson. On optimistic methods for concurrency control. *ACM Trans. on Database Systems*, 6(2), June 1981.

[7] Pete Tinker and Morry Katz. Parallel execution of sequential scheme with paratran. In *Proc. ACM Conference on Lisp and Functional Programming*, July 1988.

[8] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *International Symposium on Computer Architecture*, 1992.

[9] Chih-Po Wen and Katherine Yelick. Parallel timing simulation on a distributed memory multiprocessor. In *International Conference on CAD*, Santa Clara, CA, November 1993. An earlier version appeared as UCB Technical Report CSD-93-723.