RC24054 (W0609-059) September 18, 2006
Computer Science

IBM Research Report

Implicit Parallelism with Ordered Transactions

Christoph von Praun, Luis Ceze*, Calin Cascaval
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

*Department of Computer Science
University of Illinois at Urbana-Champaign

Implicit Parallelism with Ordered Transactions

Christoph von Praun[†] Luis Ceze[‡] Călin Caşcaval[†]

[†] IBM T.J. Watson Research Center,
{praun, cascaval}@us.ibm.com

[‡] Department of Computer Science
University of Illinois at Urbana-Champaign
luisceze@cs.uiuc.edu

Abstract

Implicit Parallelism with Ordered Transactions (IPOT) is an extension of explicitly parallel programming models to support implicit parallelism through speculative execution. The key idea of this model is to specify opportunities for parallelization within a sequential thread using annotations similar to transactions. Unlike explicit parallelism, IPOT annotations do not require the absence of data or control-dependences, since the parallelization relies on runtime support for speculative execution. IPOT as a parallel programming model is determinate, i.e., program semantics are independent of the interleaving of concurrent threads. For optimization, non-determinism can be introduced selectively.

We describe IPOT as a programming model and an on-line tool that recommends boundaries of ordered transactions by observing the sequential execution. On a set of SPECcpu benchmarks and two large HPC workloads, we demonstrate that our method is effective in identifying opportunities for fine-grain parallelization within OpenMP tasks. Using the automated task recommendation tool, we were able to perform the parallelization of each program within a few hours.

1. Introduction

The current trend in processor architecture is a move toward multi-core chips. The reasons are multiple: the number of transistors are increasing, the power budget for super-scalar processors is limiting frequency and thus performance [2, 19]. Therefore parallel programming is going to become much more prevalent, and involve users from domains that did not traditionally deal with parallelism. There are several ways to address this problem. One approach is automatic parallelization. A compiler analyzes the source code and extracts the parallel loops. Its main advantage is that users do not get involved, and, at least theoretically, one can convert legacy sequential applications to exploit multi-core parallelism. However, after 30 years of research, automatic parallelization works for regular loops in scientific codes. The range of applications for multi-core spans a very different class of programs, which the automatic parallelization has been traditionally unsuccessful in parallelizing [24].

Another approach is speculative execution, in both forms, Thread Level Speculation (TLS) and Transactional Memory (TM). In this case, the system provides support for speculative execution and extracts independent tasks from an application that can potentially be executed in parallel. The system detects conflicting data access and ensures that the parallel execution follows correct sequential execution semantics. In TLS, as opposed to TM, there is an extra constrain on ordering. The task extraction can be done by either hardware or a compiler. However, because the speculative state is typically kept in hardware, the size of the parallel tasks is smaller than the size of parallel tasks in an explicit parallel program. While this technique allows parallel execution, it does not come for free; the speculative hardware can be quite complex [20], and speculative tasks are typically more expensive than parallel tasks [10].

A third approach to exploit machine-level parallelism is to explicitly specify parallel execution in the program. There are two prevalent parallel programming models, message passing using MPI [21] and shared memory, such as OpenMP [15]. Since multi-cores in a chip share the memory, shared memory programming models are more attractive. However, programmers need to parallelize their codes by hand, and moreover, tune the performance of the code for the particular chip they are using. In addition to all the correctness concerns, this could introduce scalability and portability problems.

In this paper we present an approach that combines the advantages of programming using a shared memory programming model and speculative execution to provide an environment in which programmers can quickly convert sequential applications, and/or scale explicit parallel programs to take advantage of multi-core architectures. This environment consists of: (i) a parallel programming model, called Implicit Parallelism with Ordered Transactions (IPOT), that can be used either as a standalone model for parallelizing sequential applications or an extension an explicit parallel programming model for exploiting fine-grain concurrency; (ii) a set of tools that support the user in finding parallel tasks in an application and evaluating the effectiveness of an application parallelized using IPOT. In addition to these programming tools, we require an execution environment that provides hardware support for parallel and speculative execution. We will discuss the support needed, but this execution environment is not the focus of this paper.

IPOT borrows from transactional memory, since 'units of parallel work' in the programming model execute under atomicity and isolation guarantees. In addition, IPOT inherits from TLS ordering properties from TLS, hence *ordered transactions*. The key idea is that ordering enables sequential reasoning for the programmer, but does not – in the common case – preclude concurrency on a runtime platform with speculative execution.

To summarize, this paper makes the following contributions:

```
for (int i=0; i < n; ++i)
  <ti>;
```

Figure 1. Program that executes ordered task sequence serially.

```
bool ti_done[]; // initially all false
finish foreach (point p[i]: [0:n-1]) {
  if (i == 0)
    atomic { <ti>; ti_done[i] = true; }
  else
    when (ti_done[i -1]) { ti; ti_done[i] = true; }
}
```

Figure 2. Explicitly parallel task sequence that is ordered through conditional critical sections.

- A parallel programming model, IPOT, that extends sequential and SPMD programming languages with support for speculative multi-threading and transactional execution;
- A set of constructs that can be used to selectively relax the determinacy of the IPOT parallel execution model;
- An algorithm that infers task recommendations from the execution of a sequential program. A task recommendation refers to a region of code that performs a computation that is 'largely independent' from its embedding context;
- A tool to estimate the benefits of a sequential program annotated with IPOT directives.

The rest of this paper is organized as follows: in Section 2 we introduce the IPOT programming model, the language extensions and discuss its determinacy; in Section 3 we present the parallel task recommendation algorithm; in Section 4 we discuss our parallelism estimation tool; in Section 5 we evaluate how effective is our environment in identifying and estimating parallelism for several SPECcpu FP benchmarks and two other large applications UMT2K [25] and CPMD [5]. Finally, we discuss related work in Section 6, future work in Section 7, and conclude in Section 8.

2. Programming model

2.1 Example

The goal of IPOT is to simplify the parallelization of sequential threads of execution. To motivate our approach we use the generic example of a sequential loop that executes a series if tasks t_i. We consider the case where the t_i may have (carried) data dependences, hence concurrency control must be used to coordinate the execution of concurrent tasks.

Parallelization with explicit concurrency

We use features from the programming language X10 [4] for illustration and assume that transactional memory is the underlying mechanism of concurrency control. The parallelization of the task sequence shown in Figure 2 is non-trivial, since conditional critical sections have to be used enforce the commit order of transactions in concurrent threads:

In this program, the finish foreach construct achieves a fork-join parallelization where all loop iterations are executed concurrently. The original body of the loop is enclosed in a transaction (atomic block). Commit ordering of these transactions is enforced indirectly: The control dependence (sequential order) in the original sequential program is converted into a data dependence, namely a true data dependence on array `ti_done[]`. A similar example is given in [3, Figure 7]. Notice that most common implementations of conditional atomic blocks would serialize the computation and not overlap the execution of subsequent iterations.

This example demonstrates that the loss of task-ordering information due to explicit parallelization is recovered through complex synchronization at the level of the software and runtime system. Apart from the code complexity, this high level synchronization can also negatively affect performance.

```
for (private int i=0; i < n; ++i)
  tryasync {
    <ti>;
  }
```

Figure 3. Ordered task sequence with IPOT.

Parallelization with implicit concurrency

The parallelization of the sequential loop with IPOT is shown in Figure 3.

The IPOT execution model demands that the semantics of this program are equivalent to serial execution semantics – regardless of the existence of data dependences among t_i. This program can execute also efficiently (given the mechanisms described in this paper), i.e., execution is parallel if there are no dependences, and serial if there are dependences.

IPOT enables that ordering constraints are tracked by the runtime platform (e.g the architecture) and thus facilitates highly efficient synchronization among tasks. Beyond ordering, Section 2.2 illustrates how IPOT can be used to explicitly avoid and resolve conflicts among concurrent tasks that access shared data.

2.2 Language extensions

IPOT can be embedded in a sequential programming language through several language extensions that are described below. All extensions resemble optional annotations, i.e., they support the compiler and runtime in achieving a good parallelization and successful speculation. Any single annotation may be left out without hampering the correctness of a program; except for the `async` annotation, introducing IPOT annotations does not change the serial program semantics. We believe that this property makes IPOT attractive and simple to use in practice.

Transaction boundaries

A mechanism to declare transaction boundaries. We use `tryasync { stmts }` to denote a block of code that should execute with ordered transaction semantics. A permissible implementation of a `tryasync` statement is to *flatten* it, i.e., to replace it with its statement block. `tryasync` blocks can be nested.

Conflict avoidance

The following declarators specify versioning and conflict resolution on local variables that are 'shared' among a declaration scope and its ordered transactions. The annotations declare how data is intended to be used by the speculative task, its parent, and successor. The annotations facilitate data privatization.

`final` variables: local variables that are defined in the invocation context and read inside a transaction. Languages with built-in explicit concurrency follow this model already: e.g., Java prevents a child thread to access the stack of its parent thread; X10 limits access of a child activity to final variables in its parent's stack.

`private` variables: local variables that are defined in the context and privatized in each transaction. These variables may be read

```
double x[N,N];
double private r=0.0, private oldval=0.0;
double reduction sigma = 0.0;

for (private int i=1; i < N-1; i++) {
  tryasync {
    for (private int j=1; j < N-1; j++) {
      oldval = x[i,j];
      x[i,j] = (4.0 * x[i,j] +
          x[i+1,j] + x[i,j+1] + // old
          (flow x[i,j-1] + (flow x[i-1,j]) // new
           / 8.0;
      r = oldval - x[i,j];
      sigma += r*r;
    }
  }
}
```

Figure 4. Wavefront computation implemented with IPOT.

and updated inside the transaction; updates are not visible outside the transaction. The copy is initialized with the value of the parent thread (copyin semantics).

`lastprivate`: as `private` but the update of the last transaction in the sequential order is propagated to the invocation context of the transaction (copyout semantics).

Conflict resolution

The conflict resolution annotations provide hints on guaranteed dependences. This helps the system decide to track or not a specific address for conflict detection.

`flow`: This is an annotation of a read operation; a flow read blocks until the immediate predecessor of a transaction updates the variable, or the reading transaction becomes non-speculative.

`race`: A race variable is exempted from conflict detection, i.e., concurrent conflicting access does not lead to an transaction rollback. Only scalar variables where read and write operations are atomic may be declared as race variables. Race variables can be useful in algorithms where tasks terminate or abort on a global condition that can be read and set independent of the task ordering, e.g., global cutoff boundary in branch and bound algorithms such as the Traveling Salesman Problem.

`reduction`: similar to a race variable, except that the operations available on such variables inside a transactions are limited to associative operations (e.g. +=, -=, max=, min=, ...). Read and update occur atomically, the serialization order of such operations in different tasks is not specified. Reduction variables do not affect the determinacy of an IPOT program.

Using the aforementioned mechanisms for conflict avoidance and detection, a programmer with perfect knowledge about data dependence can rule out mis-speculation entirely as illustrated in the following example.

2.3 Example

Figure 4 shows a finite difference stencil computation following the Gauss-Seidel method. Apart from the IPOT directives (highlighted in the example), the structure of the computation corresponds to the sequential version of the algorithm.

The parallelization strategy chosen here considers the update of each row of `x` as a separate task (wavefront). This is denoted by the `tryasync` annotation before the inner loop. Given only this annotation, the speculative execution will likely not be successful due to data dependences across transactions. Additional annotations weed out misspeculation through data privatization, and control the scheduling of speculative tasks: Variables `r`, `oldval`, `i`, `j` are `private`, i.e., each task has its own copy of the variable which is initialized with the value provided by the parent thread (copyin). Data dependences on `sigma` are resolved through the `reduction` declaration, i.e., read and update occur atomically in any order among transactions. Read access to variables with *flow* dependences are explicitly declared and may delay a transaction if necessary to enforce the dependence with a predecessor in the serialization order.

In the example, the IPOT annotations capture all cross-task data dependences that are potential sources of mis-speculation and hence it is guaranteed that no conflicts will ever occur at runtime. This means that IPOT is capable to support well-known parallelization schemes that do not require speculation (in this case wavefront) in an efficient way.

The example illustrates that parallelization with IPOT preserves the algorithmic structure and achieves to separate it from the parallelization aspect. Compared to explicitly parallel versions of stencil computations that follow wavefront or red-black strategies, IPOT compares favorably in code complexity and avoids extra code due to data decomposition and synchronization.

2.4 Determinacy

A parallel program is *externally determinate* [6] (determinate for short) if its output depends only the input, not on the scheduling (interleaving) of concurrent threads.

One of the strengths of the IPOT programming model is that determinacy is preserved when evolving from a sequential to a parallel version of a program. The reason for determinacy is that data races [14] that may occur due to the parallelization are automatically detected and resolved by the system: Whenever two tasks participate in a data race, the more speculative task (and its successors) are restarted. This behavior makes the system determinate.

The declarations for *conflict avoidance* presented in Section 2.2 do not affect determinacy, they merely help to reduce the frequency of conflicts and situation where conflicts need to be corrected. The wrongful use of such declarations can however alter program semantics.

There are two constructs for *conflict resolution* that permit to selectively introduce *internal non-determinacy* [6]: `reduction` and `race`. Internal non-determinacy means that intermediate states encountered during program execution may be different in different program runs, (depending on the thread interleaving), the final output of a parallelized construct will however be the same as in a sequential computation. Internal non-determinacy enhances the flexibility in scheduling operations and hence this can be regarded as an optimization.

A *reduction* declaration preserves *external determinacy*, as tasks are only permitted to perform associative update operations on such variables (see *associative non-determinacy* [6]). A `race` variable is different: it is the programmer's responsibility to access the variable in a manner that preserves determinacy. Race variables are the *only possible source of external non-determinism* in the IPOT programming model.

Given that IPOT programs are principally determinate, many issues that complicate the semantics of explicitly concurrent programming languages doe not arise, such as shared memory consistency [18], semantics of inlining threads [13], and exception handling in concurrent programs [4].

2.5 Handling of overflow, I/O, and system-calls

IPOT lends itself nicely to handle overflow of speculative buffers or to execute operations with permanent side effect through serial-

```
int i;
#pragma omp parallel for private (i)
for (i=0; i < N, ++i)
  <body>

finish {
  for (private int i=0; i < N, ++i)
    async <body>
}
```

Figure 5. Doall parallelism with OpenMP and IPOT extensions.

ization: A task waits to become the least speculative, then validates itself, becomes non-speculative, and continues the remainder of its computation in a non-speculative manner.

2.6 Software development process

IPOT facilitates a software development process that separates the aspects of algorithm development and performance engineering: A domain expert specifies an algorithm and possibly 'units of independent work' in a sequential logic. Subsequently, a performance expert annotates (with the help of the task finder tool presented in Section 3) and restructures the programs using the mechanisms for conflict detection and resolution to achieve an efficient parallel version of the code.

2.7 Extension for explicit parallelism

A natural extension of IPOT is to permit explicitly parallel tasks that are not subject to the rigorous conflict detection and the predefined serialization order of speculative tasks. We describe an extension that relaxes the total serialization order of speculative tasks to a partial commit order. Not surprisingly, this extension is a potential source of non-determinism.

The key idea is to designate blocks of code that should execute concurrently with their continuation. We use `async { ... }` to denote such blocks. Unlike `tryasync` the execution of the body of such block is not transactional. A barrier, similar to X10's `finish` guarantees termination of all scoped `async` blocks. Figure 5 illustrates these constructs and contrast them to an OpenMP `for` loop.

Transactions in the dynamic scope of the same `async` commit according to a sequential model, while transactions in the scope of different `async`s proceed concurrently, i.e., their serialization order is unspecified.

`async` blocks may also be nested inside `tryasync` blocks, for example as hints to facilitate vectorization by a compiler. Note that the role of `async` blocks in IPOT is to support the parallelization of sequential codes: Similar to `tryasync`, a permissible implementation of an `async` statement is to replace it with its statement body (task inlining). Such transformation is not applicable in environments with more general uses of multithreading and concurrency control.

A key challenge when integrating speculative execution with explicit parallelism is that misspeculation is not 'for free'; in fact misspeculation utilizes resources that other, non-speculative threads may exploit to perform useful work. IPOT provides several language features to control and tame misspeculation – which overall facilitates the inter-operation between speculative and explicit multi-threading.

The `async` feature is a potential source of non-determinism if concurrent `async` tasks access shared memory in a conflicting manner, similar to OpenMP parallel annotations. Since `async` tasks execute outside the safety harness of speculative execution, conflicting access may not be detected.

This explicitly parallel extension of the IPOT programming model is compatible with locks: Similar to the handling of I/O and system calls (Section 2.5), the non-speculative serial execution order always provides a 'safe' fall-back execution model that can handle blocking synchronization constructs such as locks. Moreover, since IPOT's model of concurrency control is based on transactions not locks, IPOT programs are deadlock free; extending existing parallel programs that use locks with IPOT features will not introduce deadlock.

3. Task recommendation

In the previous section we introduced language annotations that can be used to identify independent units of work in existing sequential or parallel programs. But, while providing support for speculative execution with the IPOT model is a nice feature, we also want to help the programmer focus on the important parts of the program that can benefit from speculative parallelization.

Task recommendation identifies sections of code that are attractive candidates for units of speculative work. Two aspects determine the potential of a task to contribute to speedup a parallel program: The fraction of the sequential execution spent in this section (size) and the potential of overlap with preceding parts of the program (hoist).

In the execution model underlying the task finder algorithm, a task is characterized by a single program counter, called *taskhead*: In a program execution, a dynamic task extends from one occurrence of the taskhead to the next. A task is the basic unit of speculative work. The result of the taskfinder algorithm is a set of taskheads. The algorithm computes for each taskhead a number of characteristics that provide guidance on selecting the taskhead for parallelization. These characteristics are: the average size of the task, the average distance of the closest data dependence, and the potential speedup of the overall program.

The algorithm operates on a dynamic execution trace of a program. In our implementation, this trace is collected in bursts of dynamic basic blocks obtained through dynamic binary instrumentation [12]. Figure 6 illustrates such a sequences of dynamic basic blocks with dependence edges. The taskfinder algorithm is shown in Figure 7 and proceeds in two steps: First, information from the dynamic execution stream is recorded in the corresponding static basic blocks (sample in Figure 8). Then, the the speedup estimates for alleged taskheads at the start of each static basic block are computed (Figure 9).

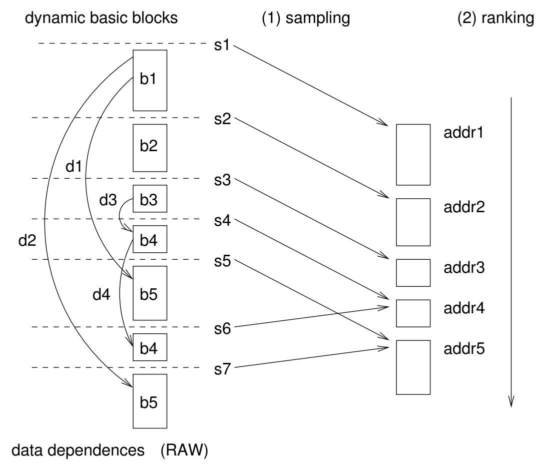

Figure 6. Illustration of the taskfinder algorithm.

```
taskFinder ()
  total_inst = 0

  /* (1) sampling */
  for dbb_list in <traces recorded by the pintool>
    Set dep_edges = \emptySet
    for dyn_bb in dbb_list
      total_inst += dyn_bb.ninstr
      sample(dyn_bb, dep_edges)

  /* (2) estimate speedup */
  for static_bb in <list of static basic blocks>
    estimate_speedup(static_bb, total_instr)
```

Figure 7. Task-finder algorithm.

```
sample (DynamicBasicBlock bb, Set dep_edges)
  if dep_edges != \empytyset
    e = <edge with minumum length in dep_edges>
    sbb = staticBB(bb)
    sbb.depDist.sample(e.len)
    sbb.selfLen.sample(bb.pc - sbb.lastpc)
    sbb.lastpc = bb.pc
    sbb.count++;
    /* remove incoming edges */
    for e in bb.incomingDepEdges
      dep_edges.remove(e)

  /* add outgoing edges */
  for e in bb.outgoingEdges
    dep_edges.add(e)
```

Figure 8. Sampling of dynamic to static basic blocks.

Sampling

The sampling sweeps over the sequence of dynamic basic blocks as illustrated in Figure 6. Given the program counter, the static basic block corresponding to a dynamic instance is determined. Two values are sampled into histograms that are associated with the static basic block: The *self length*, i.e., the number of instructions that have executed since the last encounter of the same program counter. In Figure 6, the self length sampled in set s_6 for basic block b_4 is the number of instructions in b_4 and b_5. The self length value is an indication of the task size.

The *dependence length* is the closest data dependence that the current or a subsequent dynamic basic blocks have to a definition in some preceding basic block. The dependence length is an indication of the hoisting potential and is determined from the set of edges that 'cross' the current program counter. The length is reported in number of instructions. In Figure 6, step s_5, the closest data dependence is d_4. We consider only true dependences on heap-allocated data. Our model assumes that writes are effected at the end and reads occur at the beginning of a dynamic basic block.

Speedup estimation

For every static basic block in a program, a speedup estimation is computed indicating the effect of parallelization achieved by placing a taskhead at the beginning of that basic block. The estimation is computed from the average of the 90-percentile of samples of the self length and dependence distance. The 90-percentile serves to exclude outliers in the sampling that are, e.g., due to the first and last iteration of a loop. The maximum degree of parallelism (`max_par`) that can be achieved is limited by the number of processors and the number of times a certain taskhead is encountered. Doall and doacross parallelism are distinguished, where the former is assumed if the dependence is farther apart than the task according to the maximum degree of parallelism. The estimation of the speedup is computed according to Amdahl's Law. In the case of doacross parallelism, we assume that the execution of subsequent tasks is overlapped as tightly as permitted by closest data dependence.

Notice that the algorithm considers each task in isolation. In the execution model (Section 4), tasks may not be independent. This assumption of task independence and the sampling and averaging over the data dependence and task lengths are the main factors of inaccuracy in the taskfinder. In practice, we observed that the recommendations are a conservative estimate of the real potential for parallelism.

```
estimate_speedup (StaticBasicBlock bb, int total_instr)
  if bb.count > 0
    self_len = average(percentile(90, sbb.selfLen))
    dep_dist = average(percentile(90, sbb.depDist))
    exe_frac = self_len * bb.count / total_instr
    max_par = min(NPROCS, bb.count)

    if dep_dist < self_len * max_par
      /* doall task */
      shorten_factor = max_par
    else
      /* doacross task */
      shorten_factor = min(max_par,
                           (dep_dist / self_len) + 1);
    bb.speedup = 1.0 / (1.0 - exe_frac +
                        (exec_frac / shorten_factor))
```

Figure 9. Computation of speedup potential for taskheads

```
 1 #define N 10000
 2 int global[N]
 3
 4 void delay (long l) {
 5   int i;
 6   for (i = 0; i < l; ++i) ;
 7 }
 8
 9 int main ( void ) {
10   int i, tmp;
11   for (i = 0; i < N ; i++) {
12     global[i] = 123;
13     delay (50);
14   }
15   for (i = 0; i < N ; i++) {
16     global[i % N] = 1 + global[(i-1) % N];
17     delay(100);
18   }
19 }
```

Figure 10. Example program for task recommendation (hoist.c).

Example

The program in Figure 10 illustrates how the taskfinder determines loop level and, more generally, method and block-level parallelism. The first loop in the main program initializes an array, the second loop performs some computation with a carried loop dependence through variables in the global array. The computation in both loops is 'simulated' through method `delay`.

The output of the taskfinder algorithm is shown in Figure 11; the last column is not reported by the taskfinder but was added to show the actual speedups obtained through simulated execution (Section 4).

The reports concern three constructs in the example program: Reports for lines 11 and 12 correspond to the initialization loop which is fully vectorizable (doall), the distance of the closest dependence is reported as zero, meaning there is no dependence. The predicted speedup matches almost exactly the speedup reported by the simulation.

```
address      frac  spdup  size  count  mindep  kind      info       real spdup
-----------------------------------------------------------------------------
0x08048319  97.0  2.23   416   6098   550     doacross  hoist.c:4       1.48
0x08048335  97.0  2.23   416   6098   550     doacross  hoist.c:7      14.97
0x08048408  67.4  1.51   550   3202   550     doacross  hoist.c:15      2.95
0x0804838e  67.4  1.51   550   3202   550     doacross  hoist.c:16      2.96
0x08048377  29.7  1.40   268   2893     0     doall     hoist.c:11      1.48
0x0804835d  29.7  1.40   268   2896     0     doall     hoist.c:12      1.48
```

Figure 11. Task recommendation for program in Figure 10.

Reports for line 15 and 16 refer to the second loop in the main program with the carried dependence. The code is correctly classified as doacross parallel, the size of the dependence equal to one loop iteration. Notice that although the loop covers a larger fraction of the overall execution time than the initialization loop, the predicted speedup is only slightly higher. The speedup obtained by the simulator is significantly higher, due aggressively hoisting and overlapping the second loop with the initialization loop (as would be done by loop-fusion in a compiler).

The taskheads corresponding to reports for line 15 and 11 are *synergistic*: Placing both taskheads in the code results in a simulated speedup of 14.97, which is higher than the sum of the individual speedups. The reason for this synergy is that tasks 5 can meet their data dependence on the initialization earlier, since the initialization itself is parallelized.

The reports for lines 4 and 7 refer to method `delay`. Due to the context-insensitive nature of the sampling, the length and dependence distance data associated with the basic blocks are blended averages from the call sites of that method within the two loops in the main program, and hence the speedup prediction has only limited significance. Interestingly though that the simulated speedups achieved with taskheads before and after the delay loop differ significantly. Taskhead in line 7 is execution equivalent to the synergistic combination of taskheads in lines 11 and 15.

Taskhead recommendation for line 4 is different and would correspond in the execution to taskheads placed *after* the data accesses in both loops (recommendations corresponding to lines 16 and 12). This combination of taskheads is *antagonistic*, since the the resulting performance corresponds to the *minimum* speedup achieved by both taskheads individually.

Intuitively, the the capability of a task to achieve a good speedup in combination with other tasks depends on when as task meets incoming and fulfills outgoing dependences. A task should fulfill its *shortest outgoing dependences as early as possible* to allow consumer tasks to overlap with itself; a task should meet its closest *incoming dependences as a late as possible* to foster its own hoisting potential. Hence, the effectiveness of a task in achieving the parallelization of a program should ultimately be judged in consideration with other tasks, not in isolation. The task recommendation algorithm presented here leaves this aspect to the programmer.

4. Execution Model

Executing an IPOT program requires runtime support for speculative multithreading [22, 8, 20, 23]. Specifically, the requirements are as follows:

spawn/commit/squash: Support for creating, committing and squashing speculative threads. This includes maintaining the correct ordering of speculative threads that map to the original sequential order.

conflict detection: Support for the detection of dependence violations across threads. Conflict detection relies on the ordering information and the memory access history of the threads to determine if there is a violation and what threads need to be squashed. If a speculative thread read a location that was later written by a less speculative thread, a conflict is flagged and the more speculative thread is squashed. This enforce dependences to guarantees the original sequential semantics of the program.

data versioning: Support for buffering speculative data until commit time. Writes performed speculatively can not be made visible until the thread commits. Also related to data versioning is forwarding of speculative data. Our execution model assumes that speculative versions of data can be provided to more speculative threads.

The speculative execution model in this paper uses an in-order spawn policy, shown in Figure 12. This implies that each speculative thread during its lifetime can spawn a single successor thread. Also, if a speculative thread commits without spawning any successor, the corresponding program thread (in other words, the safe thread) terminates. The reason for in-order spawn is the simplicity of order management, since the order of thread creation is the same order as commit. This directly translates to lower hardware complexity as discussed below.

In-order spawn can also limit performance, since it imposes a limit of how far tasks can he hoisted — tasks can not be hoisted earlier than the previous task (following sequential order) starts to execute. However, we believe this limitation can be tolerated, given the simplification benefits it brings to the architecture and runtime system.

Spawn hoisting is the distance between a parent thread's commit point and the spawn of its successor thread. This distance directly translates into overlap between parent and speculative thread. In the execution model we assumed that a successor thread can potentially be spawned as soon as its parent starts executing.

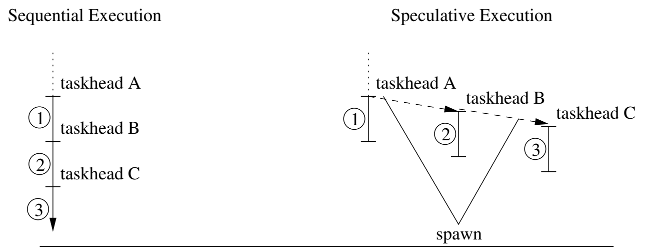

Figure 12. Spawn ordering and hoisting in our execution model.

The architecture support needed by IPOT is a fairly standard speculative multithreading substrate. The architecture supports conflict detection and data-versioning directly in hardware. Conflict detection is implemented by leveraging the coherence protocol [23]. Data-versioning is supported in hardware by buffering speculative writes, typically in the L1 or L2 cache [7]. When the cache overflows with speculative data and needs to displace speculative data, the thread that owns the data needs to be squashed. That imposes a limit on the size of speculative threads, which indirectly affects how programmers should use the programming model. However, since we believe programmers will use ordered transaction at a fine grain, we believe this limit will not be a problem.

One extra feature the architecture needs to support for IPOT is conflict avoidance and resolution, described in Section 2.2. We envision conflict detection to be a property of a page. This way, memory accesses to pages where conflict detection is turned off will not be subject to conflict detection. We believe that this is a straightforward extension to memory management units in modern processors, which already support page-level attributes like cached/uncached.

5. Evaluation

To evaluate the effectiveness of our programming environment without fully implementing the new language extensions in a compiler, the entire code generation for a new architecture, and a simulator that support all the necessary TLS extensions, and then ending up being able to run only very small and short kernels, we decided to implement an emulator, based on binary instrumentation [12] that can estimate the parallel overlap the simple annotations described below.

5.1 Methodology

We evaluated IPOT programs using an emulator that supports the speculative execution model. The emulator was implemented using dynamic binary instrumentation [12]. The program runs sequentially and all memory operations and annotations are instrumented. Memory operations are instrumented in order to build a map of memory-based data dependences. By observing the program annotations (e.g. `__taskhead()`), the emulator identifies where tasks start and complete in the dynamic instruction stream. Using the dependence map and task information, the emulator computes the overlap of speculative threads by enforcing data dependences and resource constrains are enforced. The only resource constrain currently considered is number of processors. Costs are also taken into account: the *spawn cost*, which is how long it takes from spawn to execution of a speculative thread; and *reuse cost*, which represents how long a processor takes to get ready to execute something following a previous task commit.

The emulator collects a multitude of information that helps characterize the dynamic behavior of tasks. Figure 13 shows the information collected: *# deps* tells how many times a speculative is restarted until it is finally executed to completion; *dependence delay* corresponds to how long a task speculates unsuccessfully; *hoisting distance* shows how much of a task is overlapped with its parent task.

For benchmarks, we chose two HPC workloads, namely UMT2K [25] and CPMD [5], and a few selected SPECcpu FP benchmarks: ammp, art, equake, and gafort. The latter, are chosen because they have also been studied in prior work on speculative parallelization with TLS and parallelization with OpenMP [1, 16, 17].

5.2 Experimental Evaluation

We evaluate several aspects: first, we report how effective is the IPOT runtime model in exploiting the parallelism and compare our speedups with prior results on parallelization with TLS [16, 17]. The IPOT tasks were determined using our task recommendation tool and instrumented by hand. Second, we validate our emulator by defining as tasks only the OpenMP loops that were declared parallel in the SPEComp2001 version of the benchmarks. And finally, we augment the OpenMP defined tasks with annotations recommended by our tool.

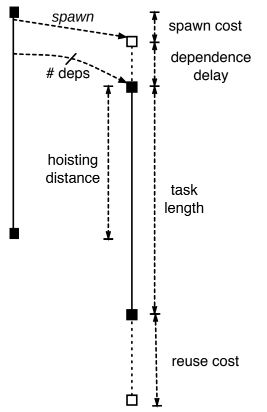

Figure 13. Dynamic task information collected by the emulator.

Bmrk	No. of tasks instrumented	Manual TLS speedup Basic	Manual TLS speedup Cumulative	IPOT speedup
ammp	10	1.62	1.76	1.55
art	7	1.0	2.94	1.80
equake	23	2.95	3.00	2.28
gafort	4	n/a	n/a	2.20
umt2k	44	n/a	n/a	2.76
cpmd	10	n/a	n/a	1.30

Table 1. Speedup comparison to manual TLS parallelization. The data points refer to the 'Basic' numbers and the total cumulative speedup for no overhead for speculation [17]. Results are for 4 threads.

In Table 1 we compare the speedups obtained using the task recommendation with the manual TLS parallelization results presented by Prabhu and Olukotun in [17].

We determine the speedups for entire application runs; for the SPECcpu programs, we use the train input set. We chose this procedure for two reasons: first, we observe that the potential of parallelization can vary significantly over different phases of the application and that the potential speedup strongly depends on the choice of input and run parameters. Second, this procedure facilitates experiment repeatability by other groups. Our methodology is hence different from [17] and most other TLS work that limited the study to 'representative samples' of the execution with the ref input set, which makes the results difficult to compare and repeat.

The results in Table 1 were collected using a 4 processor configuration. Unlike [17] we do not perform any of the following optimizations: loop chunking and slicing, speculative pipelining or complex value prediction. We do mark parallel reductions, as they are supported by the IPOT programming model.

There are two conclusions that we can draw from these results: i) the parallelism estimator implemented the task finder and the one in the emulator give a very realistic picture of the actual parallelism; and ii) parallelizing these benchmarks based on the task finder recommendations, while not trivial, it is certainly easy — we took only about 2.5 hours to parallelize `umt2k` which has about 45 K lines of C and Fortran code, and about 2 hours for `cpmd` which contains about 250 K lines of Fortran code. The number of tasks for `cpmd` is one fourth of the tasks in `umt2k`.

The next comparison point is the OpenMP parallelization. In this experiment we want to demonstrate that our parallelism estimator is reasonably close to native execution using OpenMP. For this experiment, we instrument the parallel OpenMP loops with

Bmrk	No. of tasks	OMP +IPOT4	OMP +IPOT8
ammp	10	2.82	3.72
art	5	2.77	3.71
gafort	9	2.93	3.44

Table 3. Speedup comparison when IPOT is used as an extension to OpenMP.

our taskhead markers and we ignore dependences on all OpenMP private variables. The results for 4 and 8 processors are shown in Table 2.

Notice that, for the SPEC programs, we used the SPECcpu train input sets and command line arguments to relate the numbers. Since the SPEComp input parameters are different (apparently they are tuned to emphasize the parallelized sections in the execution), the number reported here are not comparable to the speedups reported in [1].

The speedups obtained by IPOT are similar to those obtained by the native OpenMP versions of the benchmarks. Notice that for the SPECcpu train input set, the regions parallelized in the program may not cover the entire executions. Hence following Amdahl's Law, if only about 50% of the serial execution time is parallelized, as, e.g., in art, the speedup is limited o 1.6 on 4 processors and 1.8 on 8 processors. The parallelization achieved by IPOT can hence be regarded as ideal for this configuration of art. For ammp, only 75% of the serial execution time are covered by the parallelization, hence also for this benchmark, the speedup achieved by IPOT is fairly high. The reason why this program does not achieve ideal speedup is that the conflicts may occur in the speculative executions of some of the very large loops in the program (we run the program without OpenMP and hence without locks). OpenMP uses locks to guard against interference, while in IPOT, we assume that the whole task can be started only after this conflict is fulfilled (see Section 4). Hence the choice of tasks suggested by the OpenMP parallelization is too large for IPOT. We will see that additional taskheads at the OpenMP lock and OpenMP unlock boundaries act as 'intermediate checkpoints' and allow for a significant speedup (see Table 3); interestingly, these taskheads were also included in the task recommendations for this program. For equake and gafort the parallel loops are spread throughout the routines and it is not straight forward to compute the fraction of serial execution time taken by these loops without outlining them.

We also compare the speedup that can be obtained by using IPOT as an extension to OpenMP, such that we take advantage of both the coarse-grain parallelism defined using an explicit programming model, and the speculation support provided by IPOT. In this case, we instrumented tasks recommendations in addition to the OpenMP instrumented tasks. The results are presented in Table 3.

As we can see, there is potential for speculative parallelism in the coarse grain tasks defined in OpenMP. We are still working on instrumenting the OpenMP versions of equake, umt2k, and cpmd with IPOT tasks recommended by our task finder tool. We shall report these numbers in the final version of the paper.

6. Related work

Thread-level speculation

Previous research on TLS has focused on either architectural support (e.g. [22, 8, 20, 23]) or automatic task decomposition for these architectures (e.g. [9, 11]). Past work typically not make that step to include the programming model in the picture. So with IPOT, the programmer is given a simple and 'safe' abstraction to facilitate fine-grained parallelism namely atomic blocks.

Approaches for automatic task selection typically fall into either program structure-based or an arbitrary collection of basic-blocks. For instance, POSH [11] describes a profile-directed framework for task selection that consider procedures and loops as units for speculation. Min-cut [9] presents a task selection mechanism where task boundaries are placed in point in the program where there is a minimum number of crossing dependences, not necessarily following program-structure boundaries.

Prabhu and Olukotun in [16, 17] also use TLS to simplify *manual* parallelization. However, the proposal does not include an actual programming model, but a set of rules and patterns for manually exploiting speculative parallelism. IPOT is a comprehensive and general programming model and our tool directs the programmer on where to spend effort profitably.

OpenMP

The programming model of IPOT resembles OpenMP, where the parallelization strategy is communicated through program annotations. The main difference between OpenMP and IPOT is that OpenMP *requires* that data dependence is correctly identified by the programmer to ensure correct execution. IPOT does not have this requirement.

Implicitly parallel programming languages

Jade [4] is an extension of the C language that facilitates the automatic parallelization of programs according to programmer annotations. As in IPOT, the key idea of Jade is to preserve serial semantics while not constraining the execution to occur serially. A programmer decomposes the program execution into tasks, using Jade's `withonly-do` construct; synchronization occurs at task boundaries. Unlike IPOT's `atomic` construct, each task contains an initial *access specification* (`withonly`, `with`) that guides the compiler or runtime system when extracting concurrency. In IPOT, no such access specification is necessary: It is the architecture and runtime system that dynamically infer form the data access stream when tasks may execute in parallel and take corrective action in case overly aggressive parallelization.

Moreover, the correctness of a Jade program depends on the correctness of the access specification (which can to some extent be done by static program analysis). Not so for IPOT, where execution semantics always follows serial semantics.

7. Future work

The task finder algorithm presented in Section 3 determines the quality of each task in isolation. As demonstrated, this can lead to inaccuracy in the reporting and leave the programmer without information about the synergistic and antagonistic effects of multiple tasks. We would like to extend the recommendation procedure to take the effect of task combination into account.

Finally, a more thorough study on the language integration of tryasync and its integration with exception handling is necessary. A fairly general language mechanism for speculative execution could be used for applications other than ordered speculative multithreading, e.g., for the implementation of speculative program optimization and checked computations where the validity of an operation is judged upon in hindsight, i.e., after it left its effects in speculative storage.

8. Concluding remarks

Trends in the development of microprocessors predict that large scale multi-core architectures are becoming mainstream. IPOT is targeted to facilitate the transformation of sequential threads of execution to harness thread-level parallelism on these platforms.

Bmrk	No. of tasks	4-proc			8-proc		
		IDEAL	IPOT	OMP	IDEAL	IPOT	OMP
ammp	7	2.20	1.76	2.16	2.90	1.81	2.62
art	1	1.60	1.63	1.19	1.77	1.75	1.16
equake	11	n/a	1.97	2.76	n/a	1.97	4.02
gafort	5	n/a	1.40	1.40	n/a	1.45	1.42

Table 2. Speedup comparison to native OpenMP.

There are two extreme design points in the landscape of parallel programming: On the one end, there is *explicitly parallel programming*, where thread coordination and concurrency control are a potential source of error and significantly contribute to complexity and cost of program development. On the other end, there are approaches to fully automated parallelization of sequential codes with and without speculation support. Research on speculative multithreading focused at the architectural level and way-ahead compiler technology. Analysis and code transformations at these levels are frequently not effective in cracking up dense data and control dependences in sequential codes, and hence these fully automated approaches have not been (widely) deployed in practice.

IPOT is positioned in the middle ground and exposes multithreading to the programmer while at the same time preserving the safe and determinate semantic foundation of a sequential language. IPOT assist the programmer with tools in identifying opportunities for parallelization and offers features that guide the programmer in declaring the 'intent' of variables and support the runtime in achieving an effective parallelization.

While many challenges remain on the way to an efficient execution platform for IPOT programs, we believe that the simplicity and determinism of the programming model combined with the attractive execution performance are well worthwhile the additional cost and complexity required for the architectural and runtime implementation.

Acknowledgments

We thank Vijay Saraswat for his insightful feedback.

References

[1] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. SPEComp: A new benchmark suite for measuring parallel computer performance. volume 2104, 2001.

[2] S. Borkar. Micro keynote talk, 2004.

[3] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The atomos transactional programming language. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, June 2006.

[4] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Object-Oriented Programming, Systems, Languages, and Applications*, Mar 2005.

[5] Car-parrinello molecular dynamics (cpmd).

[6] P. A. Emrath and D. A. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proceedings of the ACM Workshop on Parallel and Distributed Debugging*, pages 89–99, Jan. 1989.

[7] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative versioning cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, February 1998.

[8] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[9] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Languages Design and Implementation*, June 2004.

[10] A. Kejariwal, X. Tian, W. Li, M. Girkar, S. Kozhukhov, and H. Saito. On the performance potential of different types of speculative thread-level parallelism. In *Proceedings of the 20th Annual International conference on Supercomputing*, June 2006.

[11] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. Posh: A tls compiler that exploits program structure. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 2006.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005.

[13] J. Manson, W. Pugh, and S. Adve. The java memory model. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'05)*, pages 378–391, 2005.

[14] R. Netzer and B. Miller. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, Mar. 1992.

[15] OpenMP. OpenMP application program interface, 2005.

[16] M. K. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2003.

[17] M. K. Prabhu and K. Olukotun. Exposing speculative thread parallelism in spec2000. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, June 2005.

[18] W. Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12(6):445–455, 2000.

[19] J. Rattner. Pact keynote talk, 2005.

[20] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas. Tasking with out-of-order spawn in tls chip multiprocessors: Microarchitecture and compilation. In *Proceedings of the 19th Annual International conference on Supercomputing*, June 2005.

[21] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. The MIT Press, 1998.

[22] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[23] J. Steffan, C. Colohan, A. Zhai, and T. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th International Symposium on Computer Architecture*, Jun 2000.

[24] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *HPCA*, 1998.

[25] The UMT benchmark code. http://www.llnl.gov/asci/purple/-benchmarks/limited/umt.