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 sequential or explicitly parallel programming models to support speculative parallelization. The key idea is to specify opportunities for parallelization in a sequential program using annotations similar to transactions. Unlike explicit parallelism, IPOT annotations do not require the absence of data dependence, 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 thread scheduling. For optimization, non-determinism can be introduced selectively.

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

Categories and Subject Descriptors D.1.3 [*Software*]: Concurrent Programming; C.1.4 [*Processor Architectures*]: Parallel Architectures

General Terms Design, Languages, Performance

Keywords parallel programming, program parallelization, implicit parallelism, thread-level speculation, transactional memory, ordered transactions

1. Introduction

The current trend in processor architecture is a move toward multicore chips. The reasons are multiple: the number of available transistors is increasing, the power budget for superscalar processors is limiting frequency and thus limiting serial execution performance [2, 24]. Parallel execution will be required to leverage the performance potential of upcoming processor generations. The key challenge posed by this trend is to simplify parallel programming technology.

One approach is automatic program parallelization. A compiler analyzes the source code and extracts parallel loops. The main advantage of this approach is that users do not get involved, and, at least theoretically, one can convert legacy sequential applications to exploit multicore parallelism. However, after decades of research, automatic parallelization works for regular loops in scientific codes. Workloads for multicore systems include other classes of applications for which automatic parallelization has been traditionally unsuccessful [14, 30].

A second approach is to specify parallelism explicitly. Multithreading with shared memory is a popular paradigm, since it naturally maps onto shared memory multiprocessors and can be regarded as an extension of sequential programming. Nevertheless this paradigm is problematic since it invites new and subtle errors due to synchronization defects [18, 6], which are difficult to debug. Besides correctness concerns, explicitly parallel programs commonly suffer from scalability and portability problems.

The third approach is to simplify parallelization through speculative execution. The most popular types of such systems are Thread Level Speculation (TLS) [15, 27] and Transactional Memory (TM) [10]. Their main characteristic is the ability to execute sections of code in a 'sand box' where updates of are buffered and memory access is checked for conflicts among concurrent threads. There are several proposals that implement TLS in hardware and provide compiler support to extract speculative tasks automatically. While this technique allows parallel execution, it does not come for free: hardware support for speculation can be quite complex [25] and speculative multitasking is typically less efficient than the execution of explicitly parallel programs [13].

In this paper we present an approach that integrates mechanisms for speculative parallelization into a programming language. In this environment, called Implicit Parallelism with Ordered Transactions (IPOT), programmers can quickly convert sequential applications or scale explicitly parallel programs to take advantage of multicore architectures. IPOT provides constructs to specify opportunities for speculative parallelization and semantic annotations for program variables. The programming environment, i.e., compiler, runtime system, and hardware can take advantage of this information to choose and schedule speculative tasks and to optimize the management of shared data. We require that the execution environment supports speculative execution. The focus of this paper is on the IPOT programming model and tools, however, we will briefly discuss required architecture and compiler support.

IPOT borrows from TM, since units of parallel work execute under atomicity and isolation guarantees. Moreover, IPOT inherits commit ordering from TLS, hence *ordered transactions*. The key idea is that ordering enables sequential reasoning for the programmer without precluding concurrency (in the common case) on a runtime platform with speculative execution.

PPoPP'07 March 14–17, 2007, San Jose, California, USA.

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

Figure 1. Simple loop to execute an ordered sequence of tasks.

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

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

To summarize, this paper makes the following contributions:

- a parallel programming model, IPOT, that extends sequential or parallel programming languages with support for speculative multithreading and transactional execution;
- a set of constructs that can selectively relax the determinacy properties of an IPOT program and improve execution performance;
- an algorithm that infers task recommendations from the execution of a sequential program;
- a tool that estimates the execution speedup of a sequential program annotated with IPOT directives.

2. Programming model

2.1 Example

The goal of IPOT is to simplify the parallelization of a sequential thread of execution. To motivate our approach we use the generic example of a sequential loop that executes a series if tasks t_i (Figure 1). We consider the case where the tasks t_i may have (carried) data dependences, hence concurrency control must be used to coordinate the execution of tasks if the loop is parallelized. We employ features from the X10 programming language [4] to illustrate the parallelization and assume transactional memory as the mechanism for concurrency control.

Parallelization with explicit concurrency

In the program Figure 2, `finish foreach` achieves a fork-join parallelization of the loop such that all iterations may proceed concurrently. The original body of the loop is executed inside a transaction (`atomic`, `when`). Commit ordering of these transactions is achieved indirectly through a conditional critical region that enforces dependences on variables of array `t_done[]`. A similar example is given in [3, Figure 7]. Notice that most common implementations of conditional critical regions 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 can be recovered through complex and expensive synchronization in the program and runtime system. Apart from the code complexity, this high level synchronization can negatively affect performance.

Parallelization with implicit concurrency

The parallelization of the sequential loop using IPOT is shown in Figure 3. The IPOT execution model requires that the semantics of this program are equivalent to the serial execution, regardless of data dependences between tasks t_i and the enclosing program context. Given the mechanisms described in this paper, this program

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

Figure 3. Ordered task sequence with IPOT.

can execute efficiently, i.e., execution is parallel if there are no dependences, and serial if there are dependences.

IPOT enables ordering constraints to be tracked by the execution platform and thus facilitates highly efficient synchronization among tasks. Beyond ordering, Section 2.2 illustrates how IPOT can avoid and resolve conflicts among transactions that are data dependent.

2.2 Language extensions

IPOT can be embedded in a sequential programming language with language extensions that are described below. These extensions support the compiler and runtime in achieving a good parallelization and successful speculation. This section discusses annotations for the *speculative* parallelization of sequential programs. Section 2.7 introduces IPOT extensions for *non-speculative* parallelization.

IPOT constructs for speculative parallelization have the nature of hints, i.e., any single annotation can be left out without altering program semantics. Conversely, most IPOT annotations do not affect the program semantics in a way that deviates from the sequential execution; exceptions are discussed in detail. We believe that these properties make IPOT attractive and simple to use in practice.

Transaction boundaries

We use `tryasync` { *stmts* } to denote a block of code which execution is likely data independent from the serial execution context in which it is embedded. An execution instance of the statement block is called a "speculative task", or task for short. Since tasks may execute concurrently, we refer to them as ordered transactions. Since we consider sequential programs in this section, the term 'transaction' refers to the execution principle and not to a mechanism for explicit concurrency control. Program execution is done on a runtime substrate that executes tasks speculatively and guarantees that the observable effects follow the sequential execution order (isolation) and that no partial effects of a task are visible to concurrent observers (atomicity). The latter aspect is only relevant when `tryasync` blocks are used in conjunction with non-speculative parallelization (Section 2.7).

The program structure due to `tryasync` blocks resembles a master-slave parallelization in OpenMP. The commit order of tasks is specified implicitly by the control flow of the master. `tryasync` blocks can be nested. A possible implementation of a nested `tryasync` is *flattening*, i.e., to simply replace it with its statement block.

Conflict avoidance and resolution

Since tasks corresponding to `tryasync` execute concurrently, they may access common program variables in conflicting ways. As a result, tasks that observe the memory in a state which is inconsistent with serial execution semantics have to be squashed and re-executed. Hence violations of data flow dependence are a potential source of inefficiency and should be avoided if possible.

The following attributes for variables in the serial program support the compiler and runtime system in determining the scope of access as well as the dependence and sharing patterns when speculative tasks execute concurrently. The variable attributes enable an

optimized runtime organization, e.g., variable privatization through the compiler or hardware.

We define six categories for program variables. Variable classification shall be done at compile time through annotations supplied by the programmer or through program analysis.

final variables are initialized once in the master and subsequently read-only. Strongly typed programming languages with built-in explicit concurrency follow this model: Java prevents a child thread to access the stack of its parent thread; X10 restricts access of a child activity to `final` variables in the stack of the parent activity. `final` variables are not subject to conflicts.

private variables are initialized and then used in the scope of the same transaction. `private` variables act as temporaries and are not subject to conflicts.

induction variables are modified only by the master and accessed read-only by speculative tasks.

reduction The operations available on `reduction` variables are limited to associative operations (e.g. `+=`, `-=`, `max=`, `min=`, ...) and for each reduction variable, only one reduction operator must be used. Read and update due to the reduction operator occur atomically but the serialization order of operations in different tasks is unspecified. A `reduction` declaration does typically not affect determinacy (Section 2.4).

race Read operations on `race` variables are exempted from conflict detection, i.e., an update by a transaction that precedes the current transaction in the commit order will not lead to a squash even though the current transaction may have read a stale value. However, should a transaction decide to update the variable, all prior reads must have returned values consistent with the serialization order. Access to `race` variables follows the atomicity guarantee of the enclosing transaction, i.e., updates are visible only after successful commit. In the read-only case, we do not require strong isolation for such variables, i.e., reads do not have to be repeatable.

Race variables can be useful when parallelizing algorithms where the computation of a task depends on a global condition that can be read independently of the task ordering, e.g., a global monotonic cut-off boundary in branch and bound algorithms such as the Traveling Salesman Problem.

ordinary variables are those that do not fall into any other category. These variables are subject to the normal versioning and conflict detection and can serve to communicate values among transactions along the commit order.

The communication among tasks through ordinary variables can be specified explicitly through a `flow` annotation, which is an attribute of a read operation. A `flow` read blocks until the immediate predecessor transaction updates the variable or all transactions preceding the current transaction in the commit order have terminated. Serial semantics demands that a `flow` read returns the value of the most recent update of the variable in the serial program order. At the implementation level, `flow` can be used to optimize value communication among tasks [29]. Alternatively to associating a `flow` annotation with a read operation, variables could be designated as `flow` such that the first read of that variable in a task would expect communication with the predecessor task.

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

```
final int N = 100;
double x[N,N];                          // ordinary
reduction double sigma = 0.0;

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

Figure 4. Wavefront computation implemented with IPOT.

2.3 Example

Figure 4 shows a finite difference stencil computation using the Gauss-Seidel method. Apart from the IPOT directives (**bold**), 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 tasks. Additional annotations control the scheduling of tasks: Data dependences on `sigma` are resolved through the `reduction` declaration, i.e., reads and updates occur atomically irrespective of the transaction order. Read access to variables with `flow` dependences may delay a transaction if necessary to enforce the dependence with its predecessor in the sequential order. Variables `r` and `oldval`, are private.

In this example, the IPOT annotations capture all cross-task data dependences that are potential sources of misspeculation and hence it is guaranteed that no conflicts will 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 the extra code needed for data decomposition and synchronization in explicit parallel programming models.

2.4 Determinacy

A parallel program is *externally determinate* [5] (determinate for short) if its output depends only on the input, not on the scheduling and progress of concurrent tasks.

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 [18] that may occur due to the parallelization are automatically detected and resolved by the system: Whenever two tasks participate in a data race such that a flow dependence between an 'older' and 'younger' task (in the commit order) is not resolved correctly, i.e., the younger task reads a stale value, it and its successor tasks are squashed and restarted. This behavior makes the system determinate.

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

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

Figure 5. Comparison of doall parallelism in OpenMP with IPOT extensions.

Two IPOT annotations allow to selectively introduce *internal non-determinacy* [5]: `reduction` and `race`. Internal non-determinacy means that intermediate states encountered during program execution may be different in different program runs due to different task schedules. However, the final result is the same as that of the sequential computation. Internal non-determinacy enhances the flexibility of task scheduling and can be regarded as an optimization.

A `reduction` declaration preserves *external determinacy* since tasks are only allowed to perform associative update operations on such variables (*associative non-determinacy* [5]). Floating point arithmetic is a corner case since implementations can violate the associativity of algebraic operations. In such case, a `reduction` declaration can be a source of external non-determinism. Similar for `race` variables where a read may return any prior value, not necessarily the most recent one in the commit order. It is the programmer's responsibility to use such variable in a way that preserves external determinacy.

Given that IPOT programs are principally determinate, many issues that complicate the semantics of explicitly parallel programming languages does not arise, such as shared memory consistency [23, 17], semantics of inlining threads [17], and exception handling [4].

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

IPOT can serialize task execution in the event of overflow of speculative buffers or to execute operations with permanent side effects. In such cases, a task blocks until all preceding tasks have committed. Then, the task continues the remainder of its computation in a non-speculative manner.

2.6 Software development process

IPOT facilitates the software development process since it separates the aspects of algorithm development and performance engineering. A domain expert specifies an algorithm and possible "units of independent work" in a sequential logic. Subsequently, a performance expert annotates (with the help of the taskfinder tool presented in Section 3) and restructures the program using the language extensions presented in Section 2.2 to achieve an efficient parallel version of the code. We consider this separation of concerns an important advantage of IPOT.

2.7 Extension for explicit parallelism

A natural extension of IPOT is to permit *non-speculative* parallelization with tasks that are not subject to the rigorous conflict detection and the predefined serialization order. This section describes a mechanism that relaxes the *total* commit order of tasks to a *partial* commit order. Not surprisingly, this extension is a potential source of non-determinism.

We use `async` { *stmts* } to denote a block of code which execution is known to be data independent from the serial execution context in which it is embedded. The execution of such block

```
...
int jout = 0;
finish {
  for (int g = 0; g < list_size; g++) {
    async {
      int my_jout;
      int j = list[g];
      double p_j_x = p_j[j].position.x;
      double p_j_y = p_j[j].position.y;
      double p_j_z = p_j[j].position.z;
      double tx = p_i_x - p_j_x;
      double ty = p_i_y - p_j_y;
      double tz = p_i_z - p_j_z;
      double r2 = r2_delta;
      r2 += (tx * tx) + (ty * ty) + (tz * tz);
      if (r2 <= cutoff2_delta) {
        tryasync { my_jout = jout++; }
        nli[my_jout] = j;
        r2i[my_jout] = r2;
      }
    }
  }
}
```

Figure 6. Explicitly parallel tasks with ordered communication through a nested `tryasync` block.

can proceed concurrently with its continuation. Unlike `tryasync` blocks, the execution is *not transactional*.

`finish` { *stmts* } acts like a barrier and blocks until the execution of the statement block and all tasks in this block have terminated. Figure 5 illustrates and compares these constructs to an OpenMP `parallel` loop.

In the presence of explicit parallelism (`async`), the role of `tryasync` is not merely a hint for parallelization. The transactional execution properties of `tryasync` blocks can be leveraged to achieve concurrency control among unordered tasks.

This is illustrated in Figure 6, where the `async` and `tryasync` constructs are used in conjunction. This program fragment is adopted from a critical loop in `namd2` [19]. The iteration traverses an array of coordinates and places positions with a distance below a certain cutoff threshold in the result array `nli`. Although the serial version of this code maintains the original order of the coordinates in the result array, this is not required by the algorithm. Apart from the access to index variable `jout`, the loop iterations are independent and hence are specified as `async` tasks. Since concurrent read and update of variable `jout` must be *ordered*, these accesses are specified as a transaction using the `tryasync` construct. Note that the serialization order of the `tryasync` block nested in different `async` instances is not predetermined.

`async` is an annotation that specifies data independence between a block of code and its continuation. `async` is not meant as a directive for general multithreading and hence IPOT supports only a simple form of explicit, non-blocking concurrency control to enable communication among concurrent tasks. Concurrent tasks shall not make any assumptions about the order in which such communication occurs. A permissible implementation of `async` is `tryasync` or *flattening*, i.e., an `async` statement can be replaced with its statement block. `async` blocks can be nested.

More generally, `tryasync` blocks that are nested within `async` blocks adhere to a partial commit order as follows: `tryasync` blocks in the scope of the same `async` task commit in sequence, while `tryasync` blocks in the scope of different `async` tasks can commit in any order.

The `async` feature is a potential source of non-determinism if concurrent `async` tasks communicate through shared memory in an unordered manner (data race) or through `tryasync` transactions where the commit order is not predetermined.

This explicitly parallel extension of the IPOT programming model is compatible with locks: similar to the handling of I/O and system calls, 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 non-blocking; extending existing, correctly synchronized, parallel programs that use locks with IPOT features will not introduce deadlock.

3. Task recommendation

In the previous section we introduced language annotations to specify independent units of work in existing sequential or parallel programs. In this section, we illustrate how a programmer can be assisted in conceiving such annotations and describe how task recommendations can be derived from a program execution.

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

In the execution model underlying the taskfinder algorithm, a task is characterized by a single program counter, called *taskhead*: a task extends from one occurrence of the taskhead in the program execution to the next. The result of the taskfinder algorithm is a set of taskheads. The algorithm computes for each taskhead a number of properties that provide guidance on selecting 'good' taskheads for parallelization. These properties are: the average size of a 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. In our implementation, this trace is collected in bursts of dynamic basic blocks obtained through dynamic binary instrumentation [16]. Figure 7 illustrates a sequence of dynamic basic blocks with dependence edges. The taskfinder algorithm is shown in Figure 8 and proceeds in two steps: first, information from the dynamic execution stream is sampled in the corresponding static basic blocks (Figure 9); then, the speedup estimates are computed for a potential taskhead at the start of each static basic block (Figure 10).

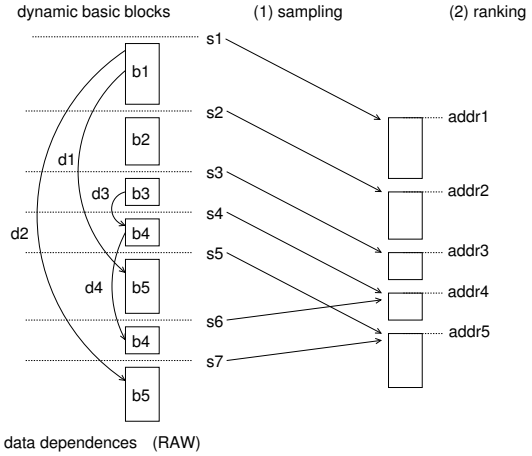

Figure 7. Illustration of the taskfinder algorithm.

```
taskFinder ()
  total_inst = 0

  /* (1) sampling */
  for dbb_list in <traces recorded by the pintool>
    Set dep_edges = <empty>
    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_inst)
```

Figure 8. Taskfinder 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 9. Sampling of dynamic to static basic blocks.

Sampling The sampling sweeps in steps s_i over the sequence of dynamic basic blocks b_i as illustrated in Figure 7. The static basic block corresponding to a dynamic instance is determined according to the program counter. Two values associated with the static basic block are sampled into histograms: The *self length*, i.e., the number of instructions that have executed since the last encounter of the same program counter. In the example, 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 d_i that the current or a subsequent dynamic basic block has 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 at step s_i. The length is reported as number of instructions. For example, in step s_5, the closest data dependence is d_4. We consider only flow 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, a speedup estimation is computed indicating the effect of parallelization achieved by placing a taskhead at the beginning of that basic block (Figure 10). The estimation is computed from the average of the 90-percentile of samples of the self length and dependence distance. The 90-percentile is chosen to exclude outliers, e.g., 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 times a certain taskhead is encountered and by the number of available processors. We distinguish two forms of parallelism: doall and doacross. Doall is assumed if the dependence length is greater than the task size, 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 the closest data dependence.

```
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 10. Computation of speedup potential for taskheads.

```
 1 #define N 1000000
 2 int global[N];
 3
 4 int main ( void ) {
 5   int i;
 6   for (i = 0; i < N ; i++)
 7      global[i] = i;
 8   for (i = 2; i < N ; i++)
 9      global[i] += global[i-2];
10 }
```

Figure 11. Example program for task recommendation (`hoist.c`).

Notice that the algorithm considers and evaluates each task in isolation. In the execution model (Section 4) however, tasks that origin at different taskheads can interfere. Thus, taskheads may not achieve the speedup that the recommendation algorithm computed for them individually. This possible interference of tasks and the averaging over dependence and task lengths are the main factors of inaccuracy in the taskfinder.

Example

Figure 12 shows the output of the taskfinder when applied to the program in Figure 11. The first loop initializes an array, the second loop computes updates with a carried dependence on array variables. The last column in Figure 12 is not actually part of the output but it is added to show the actual speedups obtained through emulated execution as described in Section 5.1.

The report in Figure 12 lists two recommendations, one for each loop in the example program. The second line corresponds to the initialization loop (line 7 in Figure 11), which is fully vectorizable (doall). The distance of the closest dependence (column `mindep`) is reported as zero, meaning there is no dependence. The predicted whole program speedup (column `spdup`) matches almost exactly the speedup reported by the emulation (column `emu spdup`). The local speedup (column `lspdup`) specifies how effective a taskhead would be in speeding up the execution segment it covers (in this case the loop). Since the analysis has been done with the assumption of 8 processors, the local speedup for a doall loop is 8.0. The first line refers to the second loop in the program (line 9 in Figure 11). The code is correctly classified as doacross parallel, since the distance of the closest dependence (30 instructions) is closer than 8 times the average size (12 instructions). Due to the dependence, the estimated maximum speedup of this loop can only be 3.5. Although the local speedup expected for this loop is less than ideal, the estimated whole program speedup is larger (1.81) than the estimated speedup due to parallelization of the doall loop (1.47). This is because the doacross loop covers a larger fraction of the total execution time (column `frac`, Amdahl's law).

The taskfinder evaluates the speedup potential for each detected program construct separately. In this particular example, the analyzer does not identify the opportunity of aggressively hoisting and overlapping the execution of the second loop with the initialization loop, as would be done by loop fusion in a compiler.

4. Execution model

Executing an IPOT program requires runtime support for speculative multithreading [27, 9, 25, 28]. The requirements are as follows:

spawn/commit/squash: Support for creating, committing and squashing speculative tasks. This includes maintaining the correct ordering of speculative tasks as defined by the original sequential program.

conflict detection: Support for the detection of dependence violations. Conflict detection relies on the ordering information and the memory access history of the tasks to determine if there is a violation and which tasks need to be squashed. If a speculative task reads a location that is subsequently written by an 'older' task, a conflict is flagged and the 'younger' task is squashed. This enforcement of dependences guarantees that the original sequential semantics of the program are met.

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

The speculative execution model in this paper uses an in-order spawn policy, shown in Figure 13. This policy implies that each speculative task can spawn only a single successor task. Also, if a speculative task commits without spawning any successor, the program terminates. In-order spawn is attractive due to its simplicity of order management since the order of task creation is the same as the commit order. This directly translates to lower hardware complexity.

In-order spawn can limit performance, since it imposes a limit of how far individual tasks can he hoisted – tasks can not be hoisted earlier than the previous task (following sequential order) starts its execution. Spawn hoisting is the distance between a parent task's commit point and the spawn of its successor task. This distance translates directly into overlap between parent and successor task. A successor task can be spawned as soon as its parent starts executing.

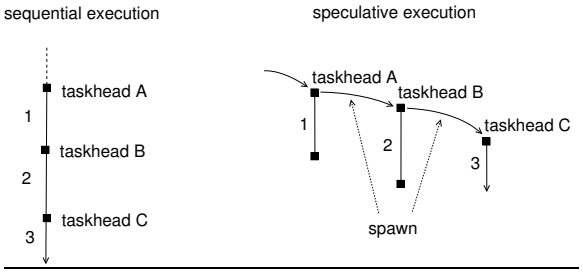

Figure 13. Spawn ordering and hoisting in the IPOT execution model.

4.1 Architectural support

The architectural support required by IPOT is a fairly standard speculative multithreading substrate. The architecture can support conflict detection and data versioning directly in hardware. Conflict

address	frac	spdup	lspdup	size	count	mindep	kind	info	emu spdup
0x08048379	62.8	1.81	3.50	12	999997	30	doacross	hoist.c:9	1.59
0x08048355	36.7	1.47	8.00	7	999999	0	doall	hoist.c:7	1.41

Figure 12. Task recommendation for program in Figure 11.

detection can be implemented by leveraging the coherence protocol [28]. Data versioning can be supported by buffering speculative writes, typically in the L1 or L2 cache [7]. This imposes a limit on the size of speculative tasks that can be handled efficiently (in hardware), which may affects how programmers use the programming model. Since IPOT is mainly targeted for fine-grain parallelism, the performance impact of this limitation is expected to be small.

The main role of the variable annotations presented in Section 2.2 is to convey semantic information about the scope of use (`private`), mutability properties (`final`, `induction`), communication and sharing patterns, (`flow`, `reduction`), and consistency requirements (`race`) to the system. The architecture could support this classification by providing means to disable conflict detection and versioning for certain regions of memory. In particular, conflicts are handled differently for `race` variables: a dependence violation on a `race` variable causes a squash only when the task that read a stale value also writes to the variable. `flow` annotations on ordinary variables will benefit from architectural support for synchronization between tasks. This can be implemented using *post* and *wait* primitives between the parent and child task.

4.2 Compiler support

Compiler support for IPOT is also fairly standard. Reductions, e.g., can be parallelized with a simple reduction privatization algorithm [20]. For each thread, a private copy of the `reduction` variable is allocated such that for the majority of the execution, the reduction operation can proceed without synchronization. The compiler is responsible for inserting code that computes a global reduction across the private copies before the value of the `reduction` variable is used in a non-speculative context. `final` variables do not require specific support but their use could be encouraged by the programming language or compiler. `private` variables can be allocated in registers (no versioning, no conflict detection), in task-private memory, or can be handled automatically by the versioning support of the architecture. Moreover, a compiler is permitted to restructure speculative tasks, e.g., to enlarge a `tryasync` block or to combine adjacent `tryasync` blocks. Additional compiler support is possible for both, static checking and optimization of annotations. This aspect is highly dependent on the language binding of the IPOT programming model and is outside the scope of this paper.

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 supports all necessary TLS extensions, we decided to implement an emulator, based on binary instrumentation [16] that can estimate the overlap of task execution based on the program annotations described below.

5.1 Methodology

We evaluated IPOT programs using an online dynamic binary instrumentation and tracing tool, based on PIN [16] that emulates the speculative execution model. The program executes sequentially and all memory operations and annotations are traced. Memory operations are collected to build a map of data dependences. By observing the program execution and "marker calls" corresponding to IPOT 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 constraints, such as the number of processors. Data dependences are tracked only for heap, not for stack variables. We chose this policy since we realized that the vast majority of stack variables can be classified in one of the categories described in Section 2.2 to support conflict avoidance or resolution. We manually verified this while instrumenting each program. Heap variables that can be handled by one of the techniques for conflict avoidance, e.g., reduction variables, are exempted from dependence checking explicitly through annotations.

Overheads of task creation and dispatch can be specified to the emulator through the *spawn cost* parameter, which reflects the number of instructions between the spawn point and the actual start of task execution. The spawn cost can impact the effectiveness of the parallelization, especially for very short tasks. For each benchmark we report speedups with an ideal spawn cost of zero instructions and a more realistic cost of 50 instructions.

The emulator collects a multitude of data that characterizes the dynamic behavior of tasks. Figure 14 shows the information collected: *# deps* represents the number of times a speculative thread is restarted before it is executed to completion; *dependence delay* is the number of instructions wasted in unsuccessful speculation. We assume the following policy for the placement of a spawn point: a task spawns its successor as soon as it starts execution. In case of a dependence violation, the spawn point of the successor task is delayed to the point where the write that satisfies the dependence occurs in the parent.

hoisting distance is the overlap of a task with its parent task. The longer the hoisting distance, the better the speedup. Intuitively, the capability of a task to achieve a good speedup in combination with other tasks depends on when a task meets incoming and fulfills outgoing dependences: a task should fulfill its *shortest outgoing dependence as early as possible* to allow consumer tasks downstream in the commit order to overlap with itself; a task should meet its closest *incoming dependence as a late as possible* to foster its own hoisting potential.

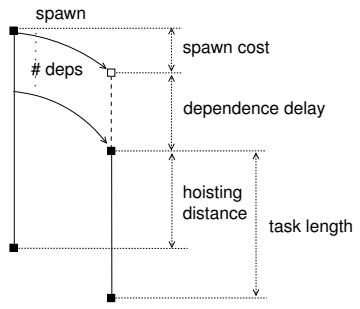

Figure 14. Dynamic task information collected by the emulator.

5.2 Experimental evaluation

We studied three HPC workloads in detail, namely `umt2k` [31], `namd2` [19], and `ammp` from the SPEComp [1] benchmark suite

(322.ammp_m, the OpenMP version of the corresponding SPEC-cpu benchmark). The three programs are designed and structured according to a high-level parallelization following the SPMD/MPI model (umt2k), OpenMP (ammp, umt2k), and a library with built-in parallelization (namd2).

Our investigation focused on fine-granular and speculative parallelization *within* one OpenMP thread or MPI task. Hence the parallelization studied with IPOT is complementary to the high-level parallel decomposition of these programs.

We used the task recommendation tool to identify parallelization opportunities in program configurations with a single MPI or OpenMP thread. Following the recommendations, the programs were instrumented manually with directives that specify task boundaries and variable classifications. In the following, we use the term 'task' interchangeably to refer to the specification of a speculative parallelization in the program code and to the dynamic execution instance.

ammp

The recommendation tool pointed out tasks in 5 of 29 source code files (`atoms.c`, `eval.c`, `rectmm.c`, `variable.c`, `vnonbon.c`). Most of the annotations were straightforward to apply. The only special case was a task recommended at the beginning of a function, which suggested the task should be placed at the call site. There were a total of 8 task annotations. Two tasks covered about 75% of the overall execution time while other tasks were rarely executed. Task recommendations pointing to loops annotated as OpenMP parallel were ignored.

Table 1 shows a characterization of the tasks annotated in ammp. Columns 1-3 point out the position of a task in the source code and its classification. Columns 4 and 5 show information about the dynamic behavior, average task size, and the faction of the serial program execution covered by each task. We report speedups for *program segments* in Columns 6-10. A program segment corresponds to a section in the source code. There can be several execution instances of a segment, and different tasks can be invoked in the dynamic scope of a segment. The reported numbers are averages of speedups of all dynamic execution instances of a segment.

Task 1, which has the highest coverage of execution time (63.7%), is the body a tight loop that searches through a vector. To increase the task granularity, several iterations of this loop are coalesced to form a single task. Task 6 and 8 correspond to loop bodies that contain an OpenMP critical section. Since the corresponding OpenMP locks are rarely contended, dependences across iterations are very rare and the tasks are still classified as doall.

Overall, with ideal overheads, the IPOT version achieved a cumulative whole program speedup of 3.14 and 4.78 for 4 and 8 processors, respectively. With more realistic overheads, the speedups were 2.69 and 3.36 for 4 and 8 processors, respectively. Note that this performance comes from parallelism inside OpenMP tasks, so it is additive to the performance gained from OpenMP parallelism.

umt2k

We ran umt2k in a configuration with a single MPI task and a variant of the RFP1 input configuration that reduced the total runtime. The focus of the investigation was on the C kernel where about 80% of the serial execution time is spent in the given configuration. The outer loop of the kernel is parallelized with OpenMP (`snflwxyz.c:439`). The parallelization we report concern only constructs within this loop, i.e., we do not exploit parallelism across iterations of the outer loop.

Task recommendations in this benchmark refer only to opportunities for loop level parallelization. Table 2 shows that most loops that are identified as optimization targets are without carried dependences (doall). This is a dynamic classification that stems from a runtime assessment about actual dependences across a certain iteration window. Interestingly, in file `snswpd3.c`, a production parallelizing compiler could identify only one loop, corresponding to task 7, as parallel. Other loops were rejected as possible optimization targets due to unknown aliasing and possible data dependence. Since most loops in the doall category had a very short body, the tasks were formed by grouping iterations, thus increasing the task size and mitigating the effects of the spawn cost. Most of the doall loops achieve very good speedups on 4 and 8 processors with and without overheads; less than ideal speedups are mostly due to a load imbalance in cases where only a few tens of tasks are created per loop instance. Segment A reports the speedup for the outer loop (one iteration of the OpenMP loop at `snflwxyz.c:439`). Segment B is an inner loop that is parallelized with two taskheads. Segments C to L correspond to inner loops that are each parallelized with a single taskhead.

Most of the execution time is spent in one loop (`snswpd3.c:356`, segment B) that is parallelized with tasks 1 and 2. This loop has an occasional carried dependence that led to its *doacross* classifications. Unlike in the doall loops, tasks cannot be combined across iterations otherwise one would loose the opportunity to aggressively overlap the execution of iterations or parts of iterations that meet flow dependences or are data independent. The two tasks are placed within the loop body such that likely outgoing dependences are served early and incoming data dependences are met late. Due to the small task size, which is 316 and 294 instructions on average, the effectiveness of the parallelization is more sensitive to the spawn overheads: the speedup on 8 processors is reduced from 7.0 to 5.8. Interestingly, the impact is more pronounced as the number of processors grows: experiments with a 32 processor configurations (not in the table) show that the potential speedup drops from 16.8 to 5.8 due to overheads. This loops also contains a reduction and requires aggressive restructuring and variable privatization to achieve this speedup. We are experimenting with a real simulator to fully validate these results.

namd2

namd2 [19] is a scalable biomolecular simulation. Parallelization and distribution of the computation are handled transparently by the Charm++ [12] library. Our experiments are done with the ApoA1 benchmark in cutoff mode in a single processor, single node configuration.

Tasks 1 and 2 constitute the first parallelization opportunity. Both tasks correspond to the same program code, which is inlined in different program contexts. This doall loop is a known optimization target: a pragma annotation in the code specifies the absence of aliasing, allowing compilers to aggressively restructure this loop and enable instruction level parallelism on superscalar architectures. Since the number of iterations is typically low, it is not effective to group iterations and hence tasks are fairly small. The average speedup is less than ideal mostly due to load imbalance.

The second parallelization opportunity identified by the task-finder is a loop (task 3, `ComputeNonBondedInl.h:117`) with very short body and a carried dependence on a single variable. The loop filters from a list of atoms those that fall under a certain distance threshold (see example in Figure 6). The implementation is coded with 4x manual software pipelining and prefetch. The parallelization with speculative multithreading did not result in a speedup due to the tight carried dependence. Since the tasks are very small, spawn overheads actually lead to a slowdown. As discussed in the example in Section 2.7, the tight dependence is an artifact of the implementation and not required by the application. We did not experiment with an alternative implementation in this evaluation.

Task	Location	Type	Avg. size [# Insts]	Frac. of serial execution [%]	Segment	Performance			
						Ideal		With overhead	
						4-proc	8-proc	4-proc	8-proc
1	atoms.c:206	doall	359	63.7	A	3.5	5.8	3.5	5.2
2	eval.c:347	doacross	5085	7.4	A, C	3.5, 1.8	5.8, 2.1	3.5, 1.5	5.2, 1.5
3	eval.c:257	doacross	757	0.5	A, C	3.5, 1.8	5.8, 2.1	3.5, 1.5	5.2, 1.5
4	math.c:169	doall	2515	0.0	A, C	3.5, 1.8	5.8, 2.1	3.5, 1.5	5.2, 1.5
5	rectmm.c:684	doall	249	11.4	B	3.0	5.0	1.8	2.0
6	rectmm.c:1130	doall	61	5.1	B	3.0	5.0	1.8	2.0
7	rectmm.c:738	doall	101	3.1	B	3.0	5.0	1.8	2.0
8	vnonbon.c:494	doall	118	5.5	C	1.8	2.1	1.5	1.5

Table 1. Dynamic task behavior for ammp.

Task	Location	Type	Avg. size [# Insts]	Frac. of serial execution [%]	Segment	Performance			
						Ideal		With overhead	
						4-proc	8-proc	4-proc	8-proc
1	snswp3d.c:461	doacross	316	31.4	A, B	3.8, 3.9	7.0, 7.0	3.8, 3.8	6.0, 5.8
2	snswp3d.c:419	doacross	294	29.3	A, B	3.8, 3.9	7.0, 7.0	3.8, 3.8	6.0, 5.8
3	snswp3d.c:276	doall	7302	7.3	A, C	3.8, 4.0	7.0, 8.0	3.8, 4.0	6.0, 8.0
4	snswp3d.c:236	doall	19765	4.2	A, D	3.8, 4.0	7.0, 7.7	3.8, 4.0	6.0, 7.7
5	snswp3d.c:299	doall	9956	0.1	A, E	3.8, 2.4	7.0, 3.7	3.8, 2.4	6.0, 3.6
6	snswp3d.c:553	doall	29514	1.0	A, F	3.8, 3.7	7.0, 6.3	3.8, 3.7	6.0, 6.3
7	snswp3d.c:306	doall	24981	5.0	A, G	3.8, 3.9	7.0, 7.7	3.8, 3.9	6.0, 7.7
8	snqq.c:212	doall	61977	2.0	A, H	3.8, 3.7	7.0, 6.3	3.8, 3.7	6.0, 6.3
9	snqq.c:109	doall	41320	1.4	A, I	3.8, 3.7	7.0, 6.3	3.8, 3.7	6.0, 6.3
10	snmmnt.c:89	doall	32468	1.1	A, J	3.8, 3.7	7.0, 6.3	3.8, 3.7	6.0, 6.3
11	snxyzref.c:104	doall	5903	0.2	A, K	3.8, 3.7	7.0, 6.7	3.8, 3.7	6.0, 6.7
12	snflwxyz.c:458	doall	27548	0.9	A, L	3.8, 3.7	7.0, 6.3	3.8, 3.7	6.0, 6.3

Table 2. Dynamic task behavior in umt2k.

Tasks 4 and 5 fall into the third code section that we inspected. Although this code is not very prominent in the serial execution profile (and hence was not identified by the taskfinder), it can become an 'Amdahl bottleneck' when the overall program execution is distributed across hundreds or more processors. We consider two loops in method `submitHalfStep`, one of which blocks 50 iterations per task. The resulting speedups are significant with reasonable task sizes in the order of thousands of instructions. This loop-level parallelization is non-trivial due to may-dependences and reductions. The experiments do not indicate actual dependences hence both loops are classified as doall. Similar opportunities may exist in other methods of the `Sequencer` class.

Tasks 6 and 7 correspond to an opportunity for method-level parallelization. The execution of method `computeForce` (implemented in different classes) has tight data dependences and computes several reductions and trigonometric functions. The method itself is not easily parallelizable, the computation is however amenable to speculation, i.e., the execution can be hoisted within its calling context (`ComputeHomeTuples.h:332`). The speedup is limited due to dependences among different method invocations that occur in a sequence. The speedup with overheads is actually higher than without. This is an artifact of the spawn points inferred by the emulator (Section 5.1) and indicates that a slight delay in a task can help it to meet a flow dependence with its predecessor (speculative data forwarding). Recall that missing a flow dependence results in a spawn delay till the update that serves the dependence. This is an example where misspeculation could be avoided through a `flow` annotation.

Summarizing, we have identified opportunities for fine granular thread-level parallelization in three important scientific workloads. Some opportunities require speculative execution, some do not. We reported speedups for the individual program segments that were the target of the parallelization. In combination, the parallelization achieved significant overall speedups, and is complementary to the high-level parallelization already available in these programs.

6. Related work

Thread-level speculation

Previous research on TLS has focused on architectural support (e.g. [27, 9, 25, 28]) or automatic task decomposition for these architectures (e.g. [11, 15]). Past work did typically not make the 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 `tryasync` blocks.

Approaches for automatic task selection typically fall into either program structure-based or an arbitrary collection of basic-blocks. For instance, POSH [15] describes a profile-directed framework for task selection that consider procedures and loops as units for speculation. Min-cut [11] presents a task selection mechanism where task boundaries are placed at points in the program with a minimum number of crossing dependences, not necessarily following boundaries of the program structure. Unlike previous work, the IPOT taskfinder recommends task decomposition solely based on dynamic profiling.

Prabhu and Olukotun [21, 22] 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 the taskfinder 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

Task	Location	Type	Avg. size [# Insts]	Frac. of serial execution [%]	Segment	Performance			
						Ideal		With overhead	
						4-proc	8-proc	4-proc	8-proc
1	ComputeNonBondedBase2.h:21	doall	2271	33.5	A	3.1	4.6	3.0	4.5
2	ComputeNonBondedBase2.h:21	doall	2063	20.4	B	3.6	6.4	3.6	6.2
3	ComputeNonBondedInl.h:117	doacross	24	0.3	C	1.0	1.0	0.6	0.6
4	Sequencer.C:1016	doall	75302	0.9	D	2.9	3.5	2.9	3.5
5	Sequencer.C:943	doall	20156	0.6	E	3.5	6.4	3.5	6.4
6	ComputeBonds.C:62	method	1165	0.6	F	2.3	2.3	3.7	6.2
7	ComputeDihedrals.C:62	method	3713	2.1	G	1.7	1.7	3.1	4.3

Table 3. Dynamic task behavior in namd2.

the programmer to ensure correct execution. IPOT does not have this requirement.

Implicitly parallel programming languages

Jade [26] 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. Each task contains an initial *access specification* (`withonly`, `with`) that guides the compiler and runtime system when extracting concurrency. In IPOT, no such access specification is necessary: It is the architecture and runtime system that dynamically infer from the data access stream when tasks may execute in parallel and take corrective action in case of overly aggressive parallelization. In Jade, incorrect access specifications are detected at runtime and flagged with an exception. The situation in IPOT is different: Since the execution model of IPOT is based on a transactional harness that enforces serial semantics, most IPOT annotations, if used improperly, can negatively affect execution performance, not correctness however.

Several extensions have been proposed for Java to provide transactional execution semantics. Most relevant and closely related to IPOT are safe futures [32]. Futures have been originally proposed as transparent annotations to enable concurrent expression evaluation. In functional programs that are mostly free of side effects, futures can be used without risk of altering sequential semantics. In Java, futures are provided as a library construct and their correct use is complicated through heap data structures and possible side effects. Safe futures address this problem and provide a safety net that detects and recovers from deviations of the serial execution semantics. The execution of safe futures overlaps with their continuation. IPOT's execution model is different, since the execution of the `tryasync` block (... not its continuation) occurs speculatively and is hoisted in its embedding execution context. The implementation of safe futures is based on a combination of compile time and software runtime mechanisms. The authors used safe futures to parallelize sequential programs, e.g., an object-oriented database kernel and several scientific kernels with loop-level parallelism. The current programming model of safe futures does not define the interaction of safe futures and other threads in explicitly parallel programs.

The programming interface of TCC [8] includes support for commit ordering of transactions. Transactions are grouped into *sequences*, such that transactions in different sequences commit in any order and transaction within the same sequence commit in the order of their *phase*. Sequence and phase information are specified explicitly when a transaction is created. In IPOT, the partial commit order is specified implicitly by the nesting structure of `async` and `tryasync` program blocks. In TCC all sections of a program are executed with transactional guarantees whereas IPOT provides transaction semantics only for `tryasync` blocks. In subsequent work, the programming language ATOMOΣ [3] proposed several language features that support the use of transactional memory. Commit ordering of transactions can be achieved only indirectly through a variant of conditional critical regions (similar to the example in Figure 2).

7. Future work

The taskfinder algorithm presented in Section 3 determines the quality of each task in isolation. This leaves the programmer without information about the possible interactions of tasks, e.g., one task can limit the hoisting of another task in the in-order spawn model (Section 4). We would like to extend the recommendation procedure to take the effect task combinations into account.

Finally, a more thorough study on the language integration of `tryasync` is necessary. A mechanism for speculative execution could be used more generally 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

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 assists 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 worth the additional cost and complexity required for the architectural and runtime implementation.

Acknowledgments

Vijay Saraswat and Josep Torrellas gave very valuable feedback. Sameer Kumar helped to identify and understand parallelization opportunities in namd2 and Xiatong Zhuang evaluated the auto-parallelization capabilities on parts of umt2k. We thank them all for their kind support. We also thank the reviewers for their thorough reading and insightful comments.

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. *Lecture Notes in Computer Science*, 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 ATOMOΣ 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] 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.

[6] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the 2003 Conference on Programming Language Design and Implementation*, pages 338–349, Jun 2003.

[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, B. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (TCC). In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct 2004.

[9] 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.

[10] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[11] 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.

[12] L. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'93)*, pages 91–108, 1993.

[13] 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.

[14] G. Lee, C. P. Kruskal, and D. J. Kuck. An empirical study of automatic restructuring of non-numerical programs for parallel processors. *IEEE Transactions on Computers*, 34(10):927–933, 1985.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Baltimore, MD, September 2002.

[20] W. M. Pottenger. Induction variable substitution and reduction recognition in the Polaris parallelizing compiler. Master's thesis, Department of Computer Science. University of Illinois at Urbana-Champaign, December 1995.

[21] 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.

[22] 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.

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

[24] J. Rattner. PACT keynote talk, 2005.

[25] 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.

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

[27] 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.

[28] 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.

[29] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 65–, 2002.

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

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

[32] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 439–453, 2005.