A Unified Scheduler for Recursive and Task Dataflow Parallelism

Hans Vandierendonck
Dept. of Electronics and Information Systems,
Ghent University, Ghent, Belgium,
Email: hvdieren@elis.ugent.be

George Tzenakis and Dimitrios S. Nikolopoulos
Foundation for Research and Technology – Hellas (FORTH)
Heraklion, Crete, Greece,
Email: {tzenakis,dsn}@ics.forth.gr

***Abstract*—Task dataflow languages simplify the specification of parallel programs by dynamically detecting and enforcing dependencies between tasks. These languages are, however, often restricted to a single level of parallelism. This language design is reflected in the runtime system, where a master thread explicitly generates a task graph and worker threads execute ready tasks and wake-up their dependents. Such an approach is incompatible with state-of-the-art schedulers such as the Cilk scheduler, that minimize the creation of idle tasks (work-first principle) and place all task creation and scheduling off the critical path. This paper proposes an extension to the Cilk scheduler in order to reconcile task dependencies with the work-first principle. We discuss the impact of task dependencies on the properties of the Cilk scheduler. Furthermore, we propose a low-overhead ticket-based technique for dependency tracking and enforcement at the object level. Our scheduler also supports renaming of objects in order to increase task-level parallelism. Renaming is implemented using versioned objects, a new type of hyperobject. Experimental evaluation shows that the unified scheduler is as efficient as the Cilk scheduler when tasks have no dependencies. Moreover, the unified scheduler is more efficient than SMPSS, a particular implementation of a task dataflow language.**

I. Introduction

Task dataflow parallel programming languages facilitate the construction of parallel programs with high performance by leveraging a runtime scheduler that is aware of dependencies between tasks. In a task dataflow language, each argument of a task is labeled with a memory access mode, e.g. input, output and the combined input/output dependencies. These labels summarize the memory side-effects of the task on these arguments as read-only, non-exposed read and write and read/write. Hereby, the scheduler can dynamically track dependencies between tasks, but it can also change the execution order of tasks, while still respecting the dependencies. This is very similar to how an out-of-order processor dynamically changes the execution order of assembly instructions.

Task dataflow languages have multiple benefits: they increase parallelism by tracking task dependencies dynamically [1], [2], they create additional parallelism by renaming memory objects [3] and they remove the sensitivity of algorithmic variations on processor architecture [4]. Task dataflow languages are currently investigated mostly in the context of high-performance computing [1]–[3], [5], but the ideas have also been applied to make parallel Java programs deterministic by automatically infering memory footprints of tasks and tracking the dependencies [6]. Applications that benefit from task dataflow include irregular parallel algorithms such as Cholesky decomposition [1], [3], algorithms with many cross-iteration dependencies such as the Smith-Waterman algorithm [7] and h264 video decoding [8].

Task dataflow languages are often restricted to a single level of parallelism: a single master thread spawns tasks but the tasks themselves cannot launch new tasks. As such, these task dataflow languages are incompatible with recursive fork/join languages such as Cilk [9], [10]. However, for many algorithms it is well known how to extract all parallelism efficiently and in these cases dependency tracking is pure overhead.

In this paper, we present a unified language and scheduler that simultaneously allows algorithms expressed in the task-dependency and fork/join styles. Hereby, the programmer can freely select the most appropriate programming style for each algorithm in an application. Furthermore, this programming model allows the construction of arbitrary parallel pipelines, a construct that appears in emerging workloads [11] and is only partially supported by Cilk.

The contributions of this paper are the following:

- We develop a scheduler that unifies work-first scheduling with dependency-aware scheduling. The unified scheduler necessarily violates some of the provably-good properties of the Cilk scheduler, however, the scheduler retains the good behavior of the Cilk scheduler when tasks are specified without dependencies, or when such tasks execute serially.
- We present *versioned objects*, a new type of hyperobject that facilitates tracking task dependencies on the object level. Versioned objects encapsulate the meta-data that is necessary to track dependencies, as well as seamlessly rename objects to increase task parallelism. While other approaches limit versioning to a single level of data (e.g. arrays without pointers), our approach allows versioning of arbitrary data structures.
- We present a new and efficient mechanism for *tracking dependencies on objects*. Our method uses tickets (similarly to ticket-based locks [12]) to enforce the program order of the C/C++ elision of the program.
- We demonstrate through experimental evaluation that our unified scheduler is *as efficient as the Cilk++ scheduler*

```
 1 typedef float (*block_t)[16];  // 16x16 tile
 2 typedef versioned<float[16][16]> vers_block_t;
 3 typedef indep<float[16][16]> in_block_t;
 4 typedef inoutdep<float[16][16]> inout_block_t;
 5
 6 void mul_add(in_block_t A, in_block_t B, inout_block_t C) {
 7     block_t a = (block_t)A;  // Recover pointers
 8     block_t b = (block_t)B;  // to the raw data
 9     block_t c = (block_t)C;  // from the versioned objects
10     // ... serial implementation on a 16x16 tile ...
11 }
12
13 void matmul(vers_block_t * A, vers_block_t * B,
14             vers_block_t * C, unsigned n) {
15     for( unsigned i=0; i < n; ++i ) {
16         for( unsigned j=0; j < n; ++j ) {
17             for( unsigned k=0; k < n; ++k ) {
18                 spawn mul_add( (in_block_t)A[i*n+j],
19                                (in_block_t)B[j*n+k],
20                                (inout_block_t)C[i*n+k] );
21             }
22         }
23     }
24     sync;
25 }
```

Fig. 1. Square matrix multiplication expressed in a language supporting runtime tracking and enforcement of task dependencies.

when task dependencies are absent. Moreover, we demonstrate that our scheduler is *more efficient than SMPSS* [3], a proven task graph scheduler. Our scheduler supports much finer-grain tasks than SMPSS, which allows more task-level parallelism for the same problem size.

The remainder of this paper is organized as follows. Section II gives an overview of our programming model. Section III discusses how we track dependencies on memory objects and how we create new versions of these objects. Next, Section IV discusses our extension to a work-first scheduler. Section V experimentally validates that our scheduler is competitive with state-of-the-art work-first and dependency-aware schedulers. Finally, Section VI discusses related work and Section VII concludes this paper.

II. Programming Model

Figure 1 illustrates programming in our language. It is assumed that the language contains parallelism control statements as in Cilk: **spawn** expresses that a procedure call may proceed in parallel with the caller and **sync** expresses that the execution of a procedure should stall until all spawned procedures have finished. We extend this however with the notion of dependencies between tasks.

Dependencies are tracked at the object level. An object must be declared as a **versioned** object in order to enable dependency tracking. Versioned objects support automatic tracking of dependencies as well as creating new versions of the object in order to increase ask-level parallelism (a.k.a. renaming [3]).

Dependency tracking is enabled on tasks that take particular types as arguments: the **indep**, **outdep** and **inoutdep** types. These types are little more than a wrapper around a versioned object that extends its type with the memory access mode of the task: input, ouput or input/output (in/out). The language allows only to pass versioned objects to such arguments.

When spawning a task, the scheduler analyzes the signature of the spawned procedure for arguments with a memory access mode. If none of the arguments describe a memory access mode, then the spawn statement is an *unconditional spawn* and it has the same semantics as a Cilk spawn. Otherwise, the spawn statement is a *conditional spawn*. The memory accesses of the task are tracked and, depending on runtime conditions, the task either executes immediately or it is queued up in a set of pending tasks.

The sync statement in our language has the same semantics as the Cilk sync statement: it postpones the execution of a procedure until all child tasks have finished execution. Some languages provide a *conditional sync* that postpones the execution of a procedure until all tasks operating on a particular object are finished (e.g. the waiton clause in SMPSS [3]). We have not yet defined the semantics of a conditional sync in our language because we have no use for it in our benchmarks. Such an extension would be quite straightforward.

We consider only situations where dependencies are tracked between the children of a single parent procedure. Each dynamic procedure instance may have a task graph that restricts the execution order of its children. This restriction allows us to prove that all parallel executions compute the same value as the sequential elision of the program [13].[1] Note that the sequential elision of the program always respects the dependencies in the program: by deducing dependencies from input/output properties, there can never be backward dependencies in the sequential elision. Furthermore, by having multiple independent task graphs in a program, we can mitigate the performance impact of building the task graph in serial fashion.

Our model allows arbitrarily mixing fork/join style and task graph execution. The only problematic issue to allow this is that we must take care when nesting task graphs, in particular when passing versioned objects across multiple dependent spawns. To make this work correctly, we must use distinct metadata for every dependent spawn to track its dependencies separately. This is detailed in Section III-F.

We assume that there is an implicit sync statement at the end of every procedure. It should be clear to the reader that the busy-leaves property of the Cilk scheduler is violated when tasks execute out-of-order. The busy-leaves property states that any created stack frame that has no left sibling is currently operated on by a worker [14]. This property lies at the basis of the provable time and space bounds of the Cilk scheduler. We argue however that dependency-aware tracking necessarily violates the busy-leaves property. Moreover, our scheduler retains all the good properties of Cilk in the absence of conditional spawns.

[1]Note that Cilk only guarantees the existence of a sequential elision. By allowing locks and data races, different parallel executions may compute different outcomes.

III. Object Versioning and Dependency Tracking

In general, “hyperobjects are a linguistic mechanism that allow different branches of a multi-threaded program to maintain co-ordinated local views of the same nonlocal variable” [15]. For instance, for a summing reducer hyperobject, a view is simply a distinct allocated instance of the summing variable. Threads that execute in parallel are assigned a distinct summing variable to operate on, such that races will not occur. When the threads join, then the private summing variables are reduced into a single variable. In this example, the reduction function is simply addition.

In this work, we define *versioned objects*, a new type of hyperobject that hides dependency tracking and renaming of objects from the programmer. A versioned object combines two pieces of information: the object metadata that tracks the status of the object (tasks reading, writing, etc.) and a pointer to dynamically allocated memory that holds an instance of the object.

A. Automatic Renaming

The semantics of a hyperobject are defined by the actions that the runtime system takes on *fork* and *join* points. For the reducer, holder and splitter hyperobjects defined in [15], it is typical that the parent procedure and the child procedure are assigned a distinct view. These views are reduced into a single view when the procedures join.

The nature of versioned objects is however quite different. A version can be valid across all newly spawned child procedures and their children recursively, even if they have not been spawned yet. Alternatively, the version has been superseded by a new version and should not be used any more by newly spawned children. As such, when a new version of an object is created at a spawn statement, both parent and child will reference this new version. This is necessary such that a new version created on spawn of a task with an output dependency on an object will be visible to a later spawned task with an input dependency on the same object. The child continues to use the new version, while the parent may replace it at a later spawn statement.

After creating a new version, the old version will get out of use gradually and will be cleaned up automatically by the runtime system when the last thread that references the version terminates.

We create new versions only for arguments accessed with the *output* memory access mode. Moreover, there must exist tasks in the system that read or write the associated object, but that have not yet finished execution. In this case, renaming is clearly advantageous.

B. Dependency Tracking with a Task Graph

In general, tracking dependencies between tasks requires the storage of the full task graph. This is undesirable for two reasons. First, the task graph has many nodes (one for each task) and it must support an arbitrary number of incoming and outgoing edges (dependencies) in each task. This implies that quite expensive data structures must be used that require multiple memory allocations per node. The edges in the task graph have a double function: (i) to determine readiness of tasks (absence of incoming edges) and (ii) to wake-up dependent tasks (by traversing all outgoing edges). Second, updating the task graph is expensive in terms of locking because every task involved in the update must be locked in order to correctly orchestrate between multiple threads that update the task graph. Furthermore, tasks that have already moved to per-worker ready queues may have to be locked, temporarily inhibiting their execution.

In this work, we present an alternative organization of the task graph. Instead of explicitly storing all edges, we use a ticketing system to correctly sequence tasks that operate on the same objects. The ticketing system bears some similarity to ticket locks [12] however, we use only the sequencing properties of the mechanism, which is known as *fairness* for locks.

Our organization of the task graph simplifies the major operations on the task graph such as task enqueueing, task dequeueing, task readiness check, checking conditions for renaming, etc. It also allows to wake-up ready tasks with little overhead, however retrieving a ready task is slightly more complicated in this system. Our experimental evaluation shows that we can hide the latency of retrieving a ready task off the critical path.

C. Ticket Locks

A ticket lock consists of two counters: a global counter and a next counter. The system works similarly to how tickets work at a butcher’s store: new clients take the next ticket one-by-one, each of them incrementing the *next* counter. The global counter is advertised and shows the ticket of the client that is currently served. When serving a client is finished, the global counter is incremented by one to indicate whose turn is next. The tickets place all clients in a virtual queue where the order of the clients in the queue is defined by the numbers on their tickets.

Ticket locks are implemented using atomic increments of the next counter. When multiple threads are competing to acquire a lock, the hardware will sequence the atomic increments of all threads in a particular order. It is guaranteed by the ticket lock that threads are served in this order. This *fairness* property is one reason why they are used in the Linux kernel [16]. It is this property that we are interested in.

D. Tickets for Task Parallelism

Because ticket locks strictly order all tasks, we present two extensions that allow to extract parallelism from a task graph. First, we allow that multiple tasks wait on the same ticket, allowing them to execute in parallel. Second, we use two sets of ticket counters to separately track readers and writers of the object.[2] Task parallelism is exposed by synchronizing on one or two tickets, depending on how a task accesses the associated object.

[2]There is one set of reader and writer counters per object. Furthermore, every new version of an object gets a new set of reader and writer counters.

	enqueue	ready?	dequeue
input	++R.next w := W.next	w = W.global	++R.global
output	**if** R.next != R.global **or** W.next != W.global **then** rename() **endif** ++W.next	**true**	++W.global
in/out	r := R.next++ w := W.next++	r = R.global **and** w = W.global	++R.global ++W.global

Fig. 2. Reader and writer ticket actions for enqueuing and dequeueing a task and for checking readiness. The operations shown concern one accessed object per task. They are repeated for all arguments of a task, using each respective argument's R and W tickets. The w and r tickets are stored in the task descriptor.

Figure 2 summarizes the operations on the tickets. The *R* tickets track readers while the *W* tickets track writers. Each ticketing system has two counters as before: a next counter and a global counter.

When enqueueing a task, the task is registered in the reader set, the writer set or both, depending on whether the task accesses the object in *input*, *output* or *in/out* mode. Furthermore, the next reader and/or writer tickets are copied to bookmark the order of the task in the sequence of all readers or all writers.

For instance, tasks with an input dependency are strictly ordered in the readers set as the next reader counter is incremented in the enqueue operation and the global reader counter is incremented in the dequeue operation. However, such tasks are not dependent on other tasks in the readers set. They can start execution as soon as all prior writers have finished. Thus, they register in the reader set but wait on a ticket from the *writers set*. Hereto, the next writer ticket is copied (not incremented) and the task will be ready to execute when the global writer counter equals the tasks writer ticket.

In a similar vein, tasks with an output dependency are strictly ordered in the writers set. They are always ready to execute due to renaming. Tasks with an in/out dependency are strictly ordered with respect to both readers and writers. Note that we chose not to rename objects in case of an in/out dependency because the benefits in terms of task parallelism are not clear. It is however straightforward to extend the system described here.

E. Example

Figure 3 shows the operation of the tickets for enqueueing a sequence of tasks with input, output and in/out dependencies. The tickets for the readers and writers conceptually order all readers and writers, respectively, in queues. We draw these queues to help understand the mechanism, but note that these queues are not stored in the program. Solid edges from a position in a queue to a task show that the task holds the corresponding ticket. Dashed edges show the tickets that a task is waiting on, i.e. the global counter must reach the ticket value pointed to.

Task T0 has an output dependency. Because the next reader counter equals the global reader counter (there are no pending tasks that access the object), the object is not renamed. T0 is inserted at the head of the writers list. It is ready to execute. For the sake of the argument, we assume that it remains executing while additional tasks are spawned.

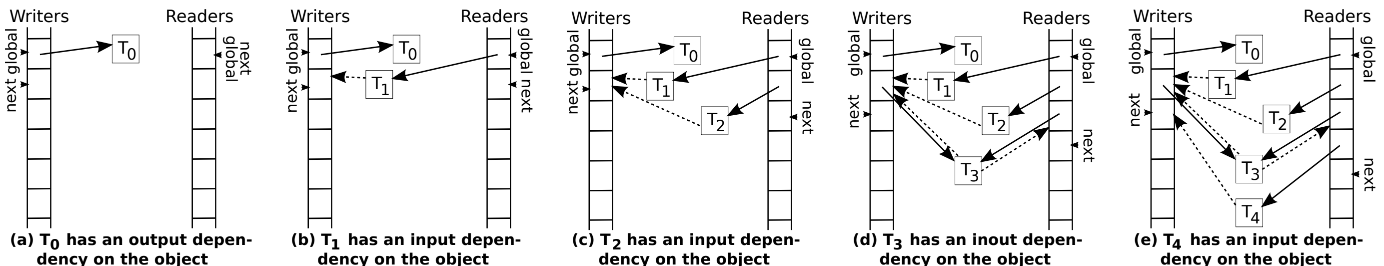
Task T1 has an input dependency on the object. Thus, it is inserted in the readers set (solid edge) and it copies the next ticket from the writers set (dashed edge). This ticket will equal the global writer ticket when T0 finishes execution. As such, the tickets reflect that T1 is dependent on T0.

Similarly, task T2 has an input dependency and the same actions are taken. Because both T1 and T2 wait only on T0 to finish execution, they may execute simultaneously.

The next task, T3, has an in/out dependency. It is inserted in both the readers and writers set (solid edges) and it grabs the next ticket from both sets (dashed edges). This is because task T3 has to wait on all prior writers (to satisfy read-after-write dependencies) and it has to wait on all prior readers (to satisfy write-after-read dependencies). It may appear that T3 is not waiting explicitly on T1. Note however that the global reader counter can be increased by two positions only if *both* T1 and T2 execute.

Task T4 has again an input dependency. It is inserted in the readers set and it copies the next ticket from the writers set. This ticket indicates that it must wait on T3 to finish execution.

If a task T5 with an output dependency arrives now, then a new version of the object written to will be created to increase



(a) T_0 has an output dependency on the object

(b) T_1 has an input dependency on the object

(c) T_2 has an input dependency on the object

(d) T_3 has an inout dependency on the object

(e) T_4 has an input dependency on the object

Fig. 3. An example of dependency tracking by means of tickets. The writers and readers queues are shown on the left and right, respectively. The solid edges indicate that a task holds a particular ticket from a queue. The dashed edges indicate that a task must wait until the global counter reaches a particular ticket. In practice, this means that the task waits on the task holding the previous ticket in the queue. This way, all inter-task dependencies are reconstructed. One can see that all dependencies in this task graph are covered by the tickets.

```
1  // Assume same typedefs as Figure 1
2
3  void child(inout_block_t A) {
4      vers_block_t A_child( A );  // introduces fresh metadata
5      // task graph computation on A ...
6      ...
7      sync;
8  }
9
10 void parent() {
11     vers_block_t A;
12     ...
13     spawn child(A);
14     ...
15     sync;
16 }
```

Fig. 4. Example of nested task graph execution. Both the parent and child procedures spawn tasks with dependencies that are tracked on the same object.

task parallelism. A new set of counters is allocated for this version and the process starts from scratch with zero tickets as if T5 is the first task to access the object.

F. Nested Task Graphs

Our scheduler allows arbitrary nesting of fork/join parallelism and task graph parallelism. Also, task graphs may be arbitrarily nested. The latter requires some care, in particular with handling the object metadata in order to track dependencies correctly. Figure 4 shows an example where a parent procedure spawns tasks with dependencies and the procedure child, one of its children, also computes on the passed object using task dependencies.

To satisfy the model where all executions compute the same task graph, it is important to correctly order the tasks dependent on child with respect to the tasks spawned by child. Hereto, we create new reader and writer counters in the procedure child, such that we can track dependencies in the parent task graph independently from the dependencies on the same object in the child task graph. This is sufficient to implement the intended model. Allocation of new metadata is effected by the construction of a new versioned object at line 4 in Figure 4. The constructor of the versioned object copies back the data to the original version (used in the parent) in case the object was renamed during the execution of the child task graph.

IV. A Unified Work-First/Task Graph Scheduler

One of the often recurring programming idioms in the context of Cilk programs is to solve a problem by recursively splitting it in smaller sub-problems, resulting in a procedure call tree, typically built from recursive procedure calls. Eventually, the sub-problems are small enough to be treated as indivisible units of work, called the leaf tasks. The parallelization strategy applied by Cilk is to split the call graph as few times as possible. Consequently, the call tree is split at the top, as many times as is necessary to split off a piece of work for every thread. Splitting off a piece of work is effected by work stealing.

Cilk's strategy has the practical consequence that most of the procedure spawns are executed serially. This property follows from the work-first principle: it is generally better to execute a spawned procedure immediately than it is to create a task descriptor, enqueue it, dequeue it and execute it. (The work-first principle is an improvement of Cilk-5 [10] over Cilk-3 [9].)

A. Cilk Runtime Data Structures

The Cilk runtime maintains several data structures to control the execution of a Cilk program: the (extended) spawn deque, stack frames and full frames (Figure 5). We refer the reader to [15] for a thorough discussion of the internals of the Cilk-5/Cilk++ scheduler. Our discussion deviates slightly on elements that are particular to our implementation.

Stack frames, as in sequential C/C++ programs, store variables local to a procedure invocation as well as temporaries and control information, e.g. to link the procedure back to the caller. On top of this information, Cilk stack frames store a small number of control variables.

Some frames are accessible by multiple worker threads. As such, they contain additional fields to control multi-threaded actions. *Full frames* contain all the information of a stack frame, as well as a lock, a continuation to proceed the execution of a frame by a different worker, a join counter and a list of child frames, which is implemented by a pointer to the first child and pointers to the sibling frames.

All frames link together in a tree, where the root of the tree corresponds to the main procedure of the program (Figure 5). This tree is also known as a *cactus stack*. This tree is, in fact, a part of the complete procedure call tree. It is a snapshot of the procedure instances that are currently active.

Full frames are always located near the root of the call tree, while stack frames appear near the leafs of the call tree. In fact, Cilk maintains the following invariants: (i) the parent of a full frame is a full frame, and (ii) a stack frame has at most one child and this child is a stack frame.

Every worker thread operates on its own set of frames, which are organized as *spawn deques* of *call stacks*. A worker pushes stack frames on the front of its spawn deque as it executes new procedure instances and it pops them as they

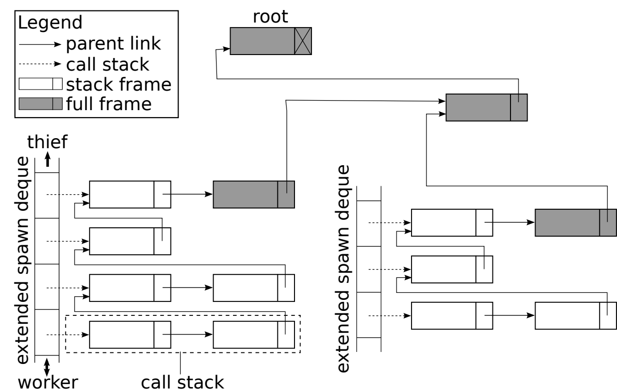


Fig. 5. The Cilk runtime data structures for two worker threads.

finish execution. The organization of the spawn deque is however distinct from the stack used in sequential C/C++ programs in two ways. First, the spawn deque is organized by call stacks. Frames created for normal procedure calls are appended to the same call stack as their parent. Frames created on procedure *spawns* are inserted in a new call stack. Second, other workers may steal call stacks from the back of the spawn deque when they run out of work. It is not possible to add call stacks to the back of the spawn deque.

Cilk separately maintains a *current call stack*, which is the call stack currently operated on. The current call stack can never be stolen by another worker. The *extended spawn deque* consists of the current call stack and the spawn deque.

The oldest frame on an extended deque is always a full frame. All other frames are stack frames. Moreover, every stack frame belongs to a single extended deque. Frames that do not belong to any extended deque are necessarily full frames.

B. Cilk Runtime Actions

The Cilk runtime manages the extended spawn deques when executing a procedure call and its return, when executing a procedure spawn and its return and when executing a `sync` statement. We only provide a very short description. Furthermore, we make abstraction of the organization of a spawn deque by call stacks. It complicates the discussion but bears no importance to the contributions of this paper. Full details can be found in [15].

When executing a procedure call or spawn, a new stack frame is allocated and pushed on the extended spawn deque. When executing a return from a call, the frame from which the return leaves is either a full frame or a stack frame. If it is a stack frame, then it is popped from the extended spawn deque and execution continues in its parent, which is guaranteed to belong to the same extended spawn deque. If the frame being left is a full frame, then it is necessarily the top frame on the extended deque. The frame is popped, leaving the extended deque empty. The worker continues by executing an *unconditional steal* of the parent frame.

When executing a return from a spawn, we again make the distinction between a stack frame and a full frame. If the frame where the return leaves from is a stack frame, then the frame is popped and execution resumes in the parent. If the frame where the return leaves from is a full frame, then it is again the top frame of the extended deque. The frame is popped, leaving the extended deque empty. The worker continues by executing a *provably-good steal* of the parent frame.

If a `sync` statement is executed inside a stack frame, then the runtime system does nothing. Otherwise, the current frame is a full frame and the scheduler will perform a counter-intuitive *provably-good steal* on the frame itself.

Cilk implements multiple stealing actions. Provably-good stealing and unconditional stealing try to continue the execution of the program in the most sensible way by continuing the execution on a frame that is a direct ancestor of the last frame executed. In contrast, random work stealing occurs when the worker has no good idea about what frame to execute next.

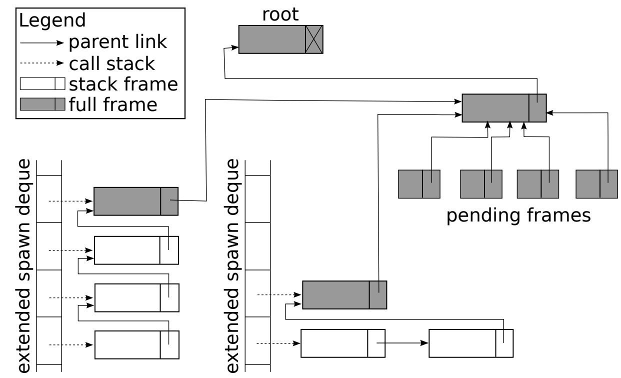


Fig. 6. The Cilk runtime data structures extended with pending frames. The pending frames are procedure instances that have not yet executed. Therefore, they can be stored more compactly than full frames.

In random work stealing the worker randomly selects a victim worker to steal a call stack. Random selection of a victim is repeated until a victim is found with a non-empty deque. The oldest call stack is removed from the deque of the victim and every stack frame on it is converted to a full frame. For every such frame, the frame is added to its parent's children list and the join counter of the parent frame is incremented. Similarly, the oldest frame in the extended spawn deque is converted to a full frame and its parent's children list and join counter are updated. Finally, the runtime system executes a resume-full-frame action on the youngest frame that was stolen.

In a provably-good steal, if the join counter of the stolen frame is zero and no worker is working on the frame (it lives outside the spawn deques), the runtime system executes a resume-full-frame action on the frame. Otherwise, the runtime system performs random work stealing.

Unconditional steals occur when returning from a procedure call. In this case, the runtime system executes a *resume-full-frame* action on the frame. The resume-full-frame action on a frame pushes the frame on the worker's extended deque and executes its continuation.

C. Extensions for Task Graph Scheduling

Dependency-aware scheduling requires additional data structures to maintain a list of pending frames. Pending frames are created upon conditional spawns where the dependencies are not satisfied when the spawn statement executes. Pending frames live outside the spawn deques because there is no worker that is executing them. Therefore, pending frames are necessarily full frames.

To support dependency-aware scheduling, we add to each full frame a list of pending frames (Figure 6). We will explain later how this list is organized. Also, stack frames and full frames are extended to store the tickets that tasks acquire, wait on and release.[3]

The actions taken by the runtime system are modified as follows. On a conditional procedure spawn from a stack

[3]Stack frames need such information in order to acquire tickets when the stack frame is converted to a full frame.

frame, the spawned procedure is executed as if it were an unconditional spawn, because we track only dependencies within the scope of a single parent frame. In this case, the parent and its children are executing serially.

On a conditional procedure spawn executed from a full frame, we acquire all objects passed as an argument with a memory access mode. If all objects are ready, then a new stack frame is allocated and it is pushed on the worker's extended spawn deque. The worker continues by executing the spawned procedure. Otherwise, if not all objects are ready, then a new pending frame is created and it is stored in the pending list of the parent. The worker continues with the execution of the parent, immediately after the spawn statement. Note that, during this process, the parent never becomes stealable by another worker because the child frames are not pushed on the spawn deque. Hereby, we can efficiently generate pending frames.

On a return from a conditional spawn that leaves a stack frame, we perform no additional actions because the spawn is executed as if it were an unconditional spawn. On a return from a conditional spawn that leaves a full frame, all acquired objects are released, potentially waking up pending frames. The extended spawn deque is now empty. The worker continues with a provably-good steal of the parent of the frame that was finished.

Actions for unconditional spawns and calls and the corresponding returns are unmodified from the original scheduler. The main difference in the scheduler is in the implementation of the stealing algorithms.

Random work stealing is modified in two ways. First, when a stack frame is converted to a full frame, we check if it has been created by a conditional spawn. If so, then all objects passed to arguments with dependency attributes are now acquired. These objects must be ready because we track only dependencies between the children of a common parent and in this case, the parent has only one child.

Second, we introduce a new steal situation. In particular, if random stealing selects a victim that has only one call stack on its extended deque, then we investigate the parent of the top frame on the victim's deque. If this parent does not belong to any spawn deque (which means its execution is stuck in a `sync` statement), then we perform a *steal-ready-child* action on the parent frame. This effectively steals a sibling of the top frame of the victim's deque.

Note that we only attempt a steal-ready-child action when the parent has reached a `sync` statement. In other cases, we prefer to retry random stealing until the worker is found whose extended deque contains the parent frame.[4] The baseline random stealing actions are sufficient to move the parent frame to the thief's extended deque. Continuing the execution of the parent in this case has the advantage that additional pending frames are created, increasing the scope of out-of-order execution. Moreover, it is possible that the parent frame executes spawn statements that are immediately executable.

[4]Or, random stealing identifies a victim that is executing a different branch of the computation. This could be a part of the program unrelated to out-of-order execution of tasks, or it could be a different out-of-order section.

Provably-good stealing of a frame still resumes the frame if the join counter is zero. Otherwise, if the frame (which is full) has pending children, we perform a *retrieve-ready-child* action on the frame.

The steal-ready-child action locates a ready frame in the pending list. Because we lazily maintain the pending list and because we do not separately maintain a list of ready frames, the steal-ready-child action must necessarily traverse the pending list in serial fashion. Note however that this action is performed only when the parent has already arrived at a sync statement. As such, all pending frames in this out-of-order execution phase have been generated and the phase is progressing towards its end. The list will thus shrink rapidly, speeding up the search. Moreover, once a ready task has been located this way, the scheduler will perform a retrieve-ready-child action as the result of returning from the spawned task on a full frame. This effectively bypasses the expensive steal-ready-child operation.

The retrieve-ready-child action locates a ready frame in the pending list. In contrast to the steal-ready-child action, this action is executed after an out-of-order task has finished execution. Because finishing tasks wake up other tasks, it is now possible to start the search in the pending list from a position that is near ready frames. We organize the pending list in such a way that this search completes successfully very quickly, as described shortly below.

If either of the actions above finds a ready child in the pending list, the runtime system then executes a resume-full-frame action on the frame. Note that objects have already been acquired when the pending frame was created. If no ready child can be located in the pending list, then the runtime system performs random work stealing. The unconditional steal action is unmodified. Note that the runtime system has been carefully designed such that serial execution of a procedure with conditional spawns proceeds without introducing important overhead. This is a consequence of applying the work-first principle to dependency-aware scheduling.

D. Organization of the Pending List

Frames in the pending list are organized by their *depth* in the task graph. The depth of a task T is defined as the maximum length of a path of dependent tasks in the task graph that ends in T. Tracking the depth of a task is fairly simple while the information is sufficient to quickly retrieve ready tasks.

To track the depth of a task, we extend the metadata of each versioned object with a depth field. Each new version of an object is assigned depth zero. Also, full frames are extended with a depth field. The depth of objects and tasks is updated when objects are acquired. When a task acquires objects passed with access modes *input* and *in/out*, then the depth of the task is computed as the maximum depth of the passed objects. Moreover, for objects passed with access modes *output* and *in/out*, the depth of the objects is updated as the newly computed depth of the task, plus one. Using the depth of each task, we organize the pending list as a list of

TABLE I
PERFORMANCE RESULTS FOR BENCHMARKS FROM THE CILK SUITE.

Benchmark	Input	Serial	Cilk++			Unified		
		T_S	T_1	T_{16}	T_S/T_{16}	T_1	T_{16}	T_S/T_{16}
cholesky	3000x3000 sparse matrix, 16x16 blocks	10.10	10.09	0.65	15.53	8.85	0.65	15.53
fft	16M data points	11.97	14.09	0.96	12.46	12.34	0.90	13.30
heat	4Kx4K grid, 100 time steps	35.66	36.48	4.19	8.51	35.95	4.25	8.39
lu	4Kx4K matrix, 16x16 blocks	40.21	38.69	2.10	19.14	34.23	2.14	18.78
rectmul	4Kx4K dense matrices	64.64	65.61	3.44	18.79	69.55	3.92	16.48
spacemul	4Kx4K dense matrices	64.41	62.67	3.32	19.40	65.92	3.82	16.86
strassen	4Kx4K dense matrices	45.18	48.05	5.18	8.72	45.33	5.10	8.85

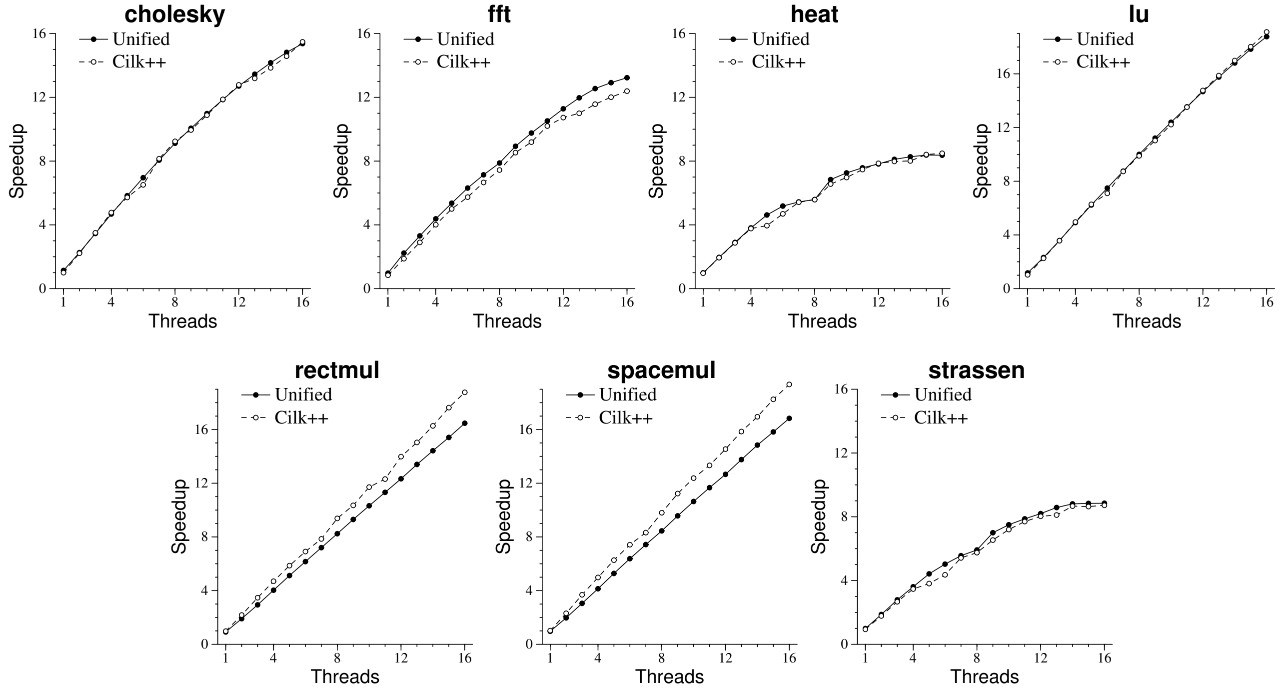


Fig. 7. Scalability graphs for the Cilk benchmarks.

lists. Pending frames are added to the end of the list for their depth. The steal-ready-child action scans the whole pending list, starting with the least depth. It is most likely that a ready frame is found with a small depth.

The retrieve-ready-child action tries to locate a ready child in the list after another task in the list has finished. If the finished task has a depth D, then it can wake-up only tasks with depth $D' > D$. On the other hand, the action in the graph occurs at level D, so this is also a good level to search. Consequently, we search depths D and $D+1$ first and resort to a scan of all levels if this fails.

This organization works very well because testing readiness of a task is extremely simple. It consists of little more than loads and compares of ticket values. Furthermore, it puts the overhead in work stealing rather than in task spawning and the common steal action is well optimized.

V. EVALUATION

We implemented our unified scheduler as a C++0x library that provides spawn and sync (although with a function call syntax) as well as versioned objects and the memory access types. We use the type introspection facilities of C++0x to analyze the signature of spawned procedures for arguments with memory access modes. Our implementation of the cactus stack differs from that of Cilk-5 in the sense that spawned procedures actually execute on the cactus stack whereas Cilk-5 stores only the data on it that is live across a spawn statement. This is effected by direct manipulation of the stack pointer using inline assembly code. Our system knows only one code version per procedure whereas Cilk-5 uses a micro-scheduled version to resume frames after a steal and a nano-scheduled version that is optimized for serial execution [10].

We experimentally validate the performance of our unified scheduler by comparing its performance to the Cilk++ and SMPSS schedulers. Hereto, we use a set of benchmarks distributed with Cilk and a set of benchmarks distributed with SMPSS. In each case, algorithms are blocked and we retain the original block sizes. We scale up to the problem size to match the performance of current processors.

The experimentation machine contains 4 quad-core AMD

Opteron 8350 HE processors clocked at 2GHz and runs the Ubuntu 9.10 operating system. We compile our scheduler using gcc 4.6, we use the Cilk++ compiler which is based on gcc 4.2.4 and we use SMPSS 2.3, which uses a custom compiler. We compute speedups relative to the serial elision of the benchmarks, which we compile using gcc 4.6. In each case, the optimization level is set to -O4. We use GotoBLAS2 (rev. 1.13) [17] for the implementation of BLAS kernels, when required by the benchmarks.

A. Comparison to Cilk++

Table I shows the Cilk benchmarks that we use in this study, together with some performance metrics. Figure 7 shows scalability graphs for the Cilk++ scheduler and our unified scheduler. The graphs show the speedup compared to the serial elision of the Cilk++ programs. These results show that the performance of our unified scheduler is quite comparable to Cilk++.

In the cases of rectmul and spacemul, the performance with the Cilk++ scheduler scales better than linear. We suspect that this is due to NUMA effects, as these benchmarks allocate and initialize memory in parallel, distributing it across multiple NUMA nodes. The other benchmarks perform initialize memory sequentially.

B. Comparison to SMPSS

Table II and Figure 8 show the performance of SMPSS and the unified scheduler on the SMPSS benchmarks. The unified scheduler gives comparable performance to SMPSS on 3 out of 5 benchmarks. It outperforms SMPSS on jacobi, which has very fine-grained tasks (32-word copy). Furthermore, SMPSS suffers performance anomalies when executing transpose with a high thread count. We will show next that SMPSS does not admit as fine-grain tasks as the unified scheduler, which explains the poor results for SMPSS above.

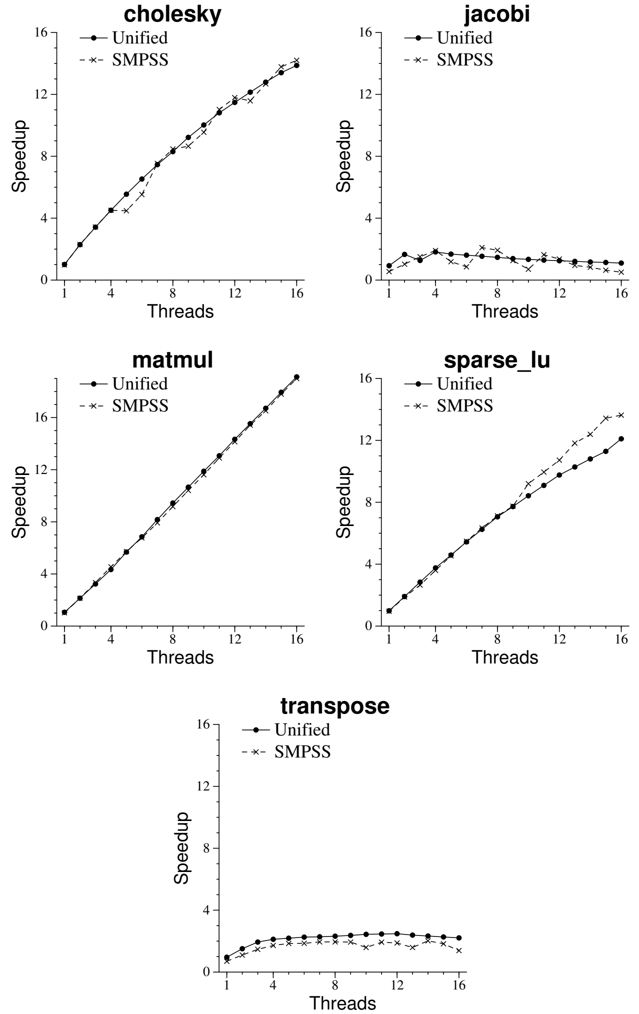


Fig. 8. Scalability graphs for the SMPSS benchmarks.

C. Fork/Join vs. Task-Dataflow Style

In this section we compare the fork/join style of programming to the task-dataflow style. We have created a matrix multiplication benchmark in both styles based on the Cilk rectmul benchmark and using BLAS dgemm in the leaf tasks.

Figure 9 shows performance measurements for a varying number of threads and for block sizes of 16x16, 32x32 and 64x64. The task-dataflow style is very sensitive to the block size. The performance of SMPSS derails for fine-grain leaf tasks on 16x16 blocks. Performance is bad even on a single thread. This is caused by a very high constant overhead in the runtime. There are also performance anomalies on 32x32 blocks. Performance is, however, good for a 64x64 block size.

The unified scheduler also has performance deficiencies on 16x16 blocks but it behaves much better than SMPSS. Note that the single-thread performance on 16x16 blocks is comparable to the performance of Cilk++ on 16x16 blocks. In other words, performance overhead related to dependency tracking is successfully avoided by the design of our scheduler.

Matrix multiplication in the task-dataflow style reaches

TABLE II
PERFORMANCE RESULTS FOR BENCHMARKS FROM THE SMPSS SUITE.

Benchmark	Input	Serial	SMPSS			Unified		
		T_S	T_1	T_{16}	T_S/T_{16}	T_1	T_{16}	T_S/T_{16}
cholesky	8Kx8K dense matrix, 32x32 blocks	27.80	28.16	1.95	14.25	27.46	2.00	13.90
jacobi	4Kx4K dense matrix, 32x32 blocks	3.48	6.27	7.01	0.49	3.73	3.14	1.10
matmul	4Kx4K dense matrices, 64x64 blocks	184.98	180.53	9.73	19.01	174.09	9.67	19.12
lu	3200x3200 sparse matrix with 32x32 blocks	7.18	7.47	0.52	13.80	7.21	0.59	12.16
transpose	256Kx256K dense matrix with 64x64 blocks	2.74	3.91	1.96	1.39	2.83	1.24	2.20

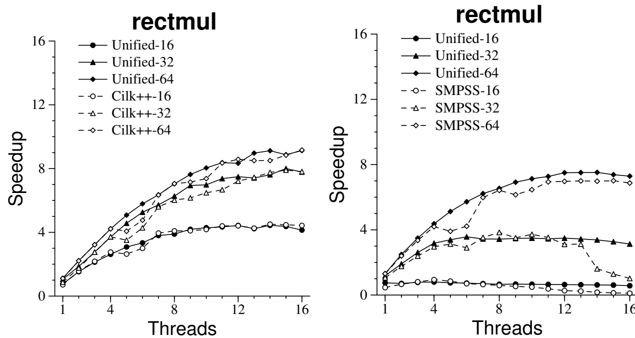


Fig. 9. Blocked matrix multiplication in fork/join style (left) and in task dataflow style (right). Block size is varied over 16x16, 32x32 and 64x64.

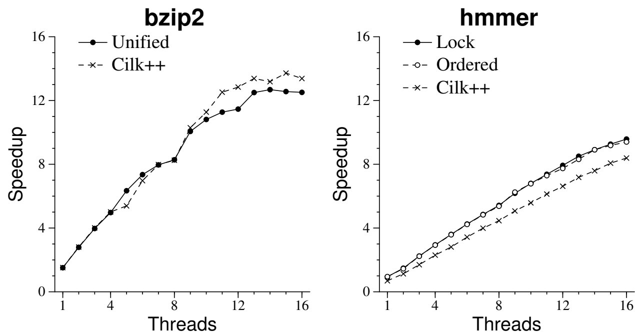


Fig. 10. Performance of pipelined programs.

minimal execution of 0.697 for a block size of 64x64 at 14 threads (Unified). The fork/join style leads to a 18% lower execution time of 0.573 for a block size of 64x64 at 16 threads (Cilk++). While it has been motivated that the task-dataflow style admits higher parallel performance [1], [3], we believe that it should be possible to pick the best idiom for every algorithm in an application.

Furthermore, our unified scheduler shows much more stable behavior across block sizes than SMPSS. Consequently, it is much easier to tune the block size based on a limited number of performance measurements. The peformance anomalies of SMPSS, for instance, would obstruct autotuning.

D. Parallel Pipelines

Task dataflow languages simplify programming of parallel pipelines [18], a construct that occurs in emerging workloads [11]. Pipelines have scalable parallelism when instantiations of at least one of the pipeline stages may execute in parallel. Cilk++ allows the pipeline construct only in limited form: 3-stage pipelines with a serial, a parallel and a serial stage (in that order) can be constructed using reducer hyperobjects. Two-stage non-parallel pipelines are a special case of these. The unified programming model allows pipeline parallelism with an arbitrary number of serial and parallel pipeline stages.

Figure 10 shows the performance of pipeline parallelism in bzip2 and hmmer. For bzip2, we use the code provided by Cilk Arts [19]. The Cilk++ version uses hyperobjects to implement the final serial stage, while the unified version uses task dependencies. Both versions obtain a comparable speedup. The hmmer code is taken from the SPEC CPU2006 integer suite. In this case, the last pipeline stage is a reduction operation. We show two versions on the unified scheduler: one where the last pipeline stage is implemented by means of a lock and one where it is implemented by task ordering. The latter case is more restrictive in terms of parallelism, but the results show that, for two or more threads, both versions obtain comparable results. The Cilk++ version also uses locks but it is somewhat slower, a consequence of building on an older version of the gcc compiler which performs less aggressive optimization of the inner loops of the benchmark.

VI. Related Work

Several task dependency aware languages and schedulers have been described in the recent literature, e.g. SuperMatrix [4], StarPU [5], SMPSS [3], CellSS [20]. All of them detect inter-task dependencies at runtime by comparing memory accesses made by tasks. Similar to OpenCL [21], StarPU allows name-based dependency tracking between tasks. This allows the programmer to explicitly state dependencies between tasks, irrespective of their memory side-effects.

To the best of our knowledge, all schedulers cited above explicitly maintain the task graph. While this approach is sensible, it turns out that it is expensive in terms of locking tasks. In contrast, we designed a scheduler that avoids such overheads, although we have to pay a small price when recovering ready tasks.

SMPSS looks up object metadata by means of a hash table that is indexed with the starting address of an object [3]. As such, metadata lookup and renaming are completely invisible to the programmer, but the hash table lookup implies runtime overhead. StarPU, on the other hand, requires that the programmer *registers* the objects used in dependency tracking [5]. The runtime system returns a descriptor that contains the object metadata. Tasks must reference this descriptor to enable dependency tracking. This sytem removes runtime overhead related to looking up object metadata. Our system has similar benefits to StarPU as the metadata always resides with the object, although our language provides a better abstraction.

Some systems require that the *complete* memory footprints of tasks are specified. This facilitates off-loading tasks on accelerator processors such as the Cell processor [20], [22] and GPUs [23]. In [2], tasks are distributed across nodes that communicate by means of MPI.

StarPU schedules tasks based on the *predicted* execution time of tasks [23]. Execution time models are calibrated by fitting measurement data to polynomial equations where the free variables describe problem sizes (e.g. matrix dimension). The scheduler assigns tasks to the processor that is predicted to complete the task earliest.

Nabbit [7] is a library that schedules the execution of task graphs using the Cilk++ language. The paper also provides upper bounds on the parallel execution time of task graphs with the Nabbit scheduler.

Concurrent Collections (CnC) is a programming model that allows mixing task and data parallelism [24]. It is an implicitly parallel and deterministic programming model where the user specifies high-level operations along with semantic ordering constraints. Together, these define a CnC graph. CnC does not specify a scheduler by itself, but can be targeted to one of many task dataflow schedulers.

The SMPSS language implements reductions with *reduction in/out* arguments. The scheduler allows to simultaneously execute tasks with dependencies arising from reduction arguments. It is assumed that such tasks lock the shared variable when it is updated. Cilk++ provides *reducer hyperobjects* to implement reductions [15]. This construct can also be used in the context of our scheduler, provided that no task dependencies are specified on the reduction variable.

VII. Conclusion

This paper presented a language and a scheduler that supports both fork/join parallel programming and task dataflow parallel programming. The scheduler extends work-first scheduling with task dependency-aware scheduling in a way that introduces minimal overhead. We demonstrate that our scheduler is as efficient as the Cilk++ scheduler on fork/join programs and that it is more efficient than SMPSS, a dependency-aware scheduler. In particular, our scheduler is more well-behaved on small task granularities, while SMPSS shows severe performance anomalies.

The language allows to structure algorithms either in fork/join style or in task-dataflow style, depending on what is more appropriate for the algorithm at hand. As such, the overhead of dependency tracking must not be paid on truly parallel algorithms.

In future work, we plan to extend ticked-based dependency tracking to accesses to partially overlapping memory regions. This is still an open problem in dependency-aware scheduling. Such a system is necessary to mix the fork/join and dependency-aware styles in a single algorithm and may allow higher performance and/or better performance portability.

Acknowledgment

Hans Vandierendonck is a Postdoctoral Fellow of the Research Foundation – Flanders (FWO). This research was performed at FORTH-ICS sponsored by a Travel Grant of the FWO. The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the ENCORE Project (http://www.encore-project.eu), grant agreement n° 248647, the TEXT Project (http://www.project-text.eu/), grant agreement n° 261580, and under the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HiPEAC, http://www.hipeac.net), grant agreement n° 217068.

References

[1] E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn, "Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures," in *SPAA'07*, 2007, pp. 116–125.

[2] F. Song, A. YarKhan, and J. Dongarra, "Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems," in *Supercomputing'09*, 2009, pp. 19:1–19:11.

[3] J. M. Perez, R. M. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multicore architectures," in *CLUSTER'08*, Sep. 2008, pp. 142–151.

[4] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn, "Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks," in *PPoPP'08*, 2008, pp. 123–132.

[5] "StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines," INRIA, Research Report RR-7240, 03 2010. [Online]. Available: http://hal.inria.fr/inria-00467677/PDF/RR-7240.pdf

[6] J. C. Jenista, Y. h. Eom, and B. C. Demsky, "Ooojava: software out-of-order execution," in *PPoPP'11*, 2011, pp. 57–68.

[7] K. Agrawal, C. E. Leiserson, and J. Sukha, "Executing task graphs using work-stealing," in *IPDPS'10*, Apr. 2010, pp. 1–12.

[8] C. C. Chi and B. Juurlink, "A QHD-capable parallel H.264 decoder," in *ICS'11*, 2011, pp. 317–326.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," in *PPoPP'95*, 1995, pp. 207–216.

[10] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multi-threaded language," in *PLDI'98*, 1998, pp. 212–223.

[11] C. Bienia and K. Li, "Characteristics of workloads using the pipeline programming model," in *Proc. of the 3rd Workshop on Emerging Applications and Many-core Architecture*, June 2010.

[12] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, pp. 21–65, February 1991.

[13] P. Pratikakis, H. Vandierendonck, S. Lyberis, and D. S. Nikolopoulos, "A programming model for deterministic task parallelism," in *Procȯf the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, 2011, pp. 7–12.

[14] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," in *FOCS'94*. 1994, pp. 356–368.

[15] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin, "Reducers and other Cilk++ hyperobjects," in *SPAA'09*, 2009, pp. 79–90.

[16] "Ticket lock," http://en.wikipedia.org/wiki/Ticket_lock, retrieved on July 13th, 2011.

[17] K. Goto and R. Van De Geijn, "High-performance implementation of the level-3 BLAS," *ACM Trans. Math. Softw.*, vol. 35, pp. 4:1–4:14, July 2008.

[18] H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos, "Parallel programming of general-purpose programs using task-based programming models," in *Proc. of the 3rd USENIX Workshop on Hot Topics in Parallelism (HotPar)*, May 2011, p. 6.

[19] J. Carr, "A parallel bzip2," http://software.intel.com/en-us/articles/a-parallel-bzip2/, Apr. 09, retrieved Jan. 19th, 2011.

[20] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: A programming model for the Cell BE architecture," in *Supercomputing'06*, 2006.

[21] *OpenCL Specification*, 1st ed., Khronos OpenCL Working Group, Sep. 2010.

[22] G. Tzenakis, K. Kapelonis, M. Alvanos, K. Koukos, D. Nikolopoulos, and A. Bilas, "Tagged procedure calls (TPC): Efficient runtime support for task-based parallelism on the Cell processor," in *Conf. on High Performance Embedded Architectures and Compilers*, 2010, pp. 307–321.

[23] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2010.

[24] B. Zoran, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Tasirlar, "Concurrent collections," *SIAM PP10 Special Issue on Scientific Programming*, 2010.