Compas'2023 : Parallélisme / Architecture/ Système
LISTIC/USMB - Annecy France, 4-7 juillet 2023

Extending the Task Dataflow Model with Speculative Data Accesses

Anastasios Souris, Bérenger Bramas, and Philippe Clauss

Inria CAMUS Team
ICube Lab. ICPS Team
University of Strasbourg, France
anastasios.souris,berenger.bramas,philippe.clauss@inria.fr

Résumé
In this paper, we describe a data-centric version-based approach to extending the task dataflow model with new access types that provide speculation capabilities to applications. We further describe the performance benefits of such a model using two scenarios that model realistic applications ; the parallelization of a finite-state machine and Monte-Carlo simulations.

---

1. Introduction

The task dataflow model, popularized in the OpenMP 4.0 standard ([1]), allows the programmer to specify units of work called *tasks* and, moreover, constraint their order of execution by specifying the memory footprint of each task. For each data object accessed by a task, we list a memory usage annotation that indicates whether the task reads or writes that object. In this way, we are able to express *true*, *anti* and *output* dependencies between successive tasks. These tasks are dynamically generated and executed asynchronously by the runtime scheduler using hardware threads in a way that the specified dependencies are satisfied.

Multiple prior works have demonstrated the benefits of *speculation* in the parallel execution of programs, nevertheless very few attempts have been made in order to incorporate speculation in the task dataflow model. In [2], the authors augment the OmpSs system with the ability to execute both paths on the occurrence of a branch inside a task, and to predict the value of data objects. In [6], the authors target a task-based loop system whereby a while loop repeatedly executes a task graph until a threshold is reached ; future iterations which are independent of the present ones can be speculatively executed. Of course, there are also numerous works with custom approaches to parallel speculation, as in [8] which is mentioned in Section 3.1.Nevertheless, we are interested in utilizing generic task systems and extending them with speculative capabilities, since their wide applicability enhances the programmer's productivity. Thread-Level Speculation techniques (see [4]) optimistically execute sequential parts in parallel, monitoring the execution for dependence violations and handling any detected violation by terminating the execution of threads involved in that violation and later restarting them. Last but not least, speculation is highly utilized in modern processors to allow execution of instructions before control dependencies are resolved, using techniques such as dynamic branch prediction (see [7]) that attempts to predict the outcome of a branch hence executing it on advance, and value prediction that serves in reducing memory stalls by predicting the outcome of operations (see [5]).

We are interested in incorporating speculative capabilities into the data dependencies themselves. To that end, in [3], the authors introduce a new data access type, that of an *uncertain-write* which indicates that a task may or may not write an object. This presents the unique opportunity of creating a copy of the data object before it is presented to the task with the uncertain-write access, and letting that copy be used by future accesses which is valid in case the data object is not written. The contribution of this paper is to generalize the model described in [3] using a data-centric version-based approach that incorporates the *uncertain-write* access and adds a value prediction mechanism. We have started implementing the model as a C++ runtime library, which will be presented in a future work.

2. Data-Centric Speculative Dataflow Model

2.1. A Version-Based Data Handling Approach

A *data object* accessed by the program is handled in our model using a *version-based* approach. The supported access types are (1) *read-access* whereby the value of the object is read, (2) *write-access* where the object can be modified, and (3) *uncertain-write-access* in which case the object may be modified. Accesses to the data object are registered sequentially to that object's lifetime, and each access semantically appends a new *version slot* to it. A version slot receives incoming *candidate data versions* from its predecessor that represent its various outcome possibilities, the respective access of that version slot is applied to each of these candidate data versions and, finally, one of them is chosen as the single *valid output* of that version slot which, if required, is passed-on to the successor version slot as a *valid input*. However, before the valid output of a version slot has been determined, some of that version slot's candidate data versions can be passed-on to its successor version slot as *probable inputs* that allow the successor version slot to perform its access without waiting for its predecessor to finish. We say that a version slot is *open* for the duration of the period where it can receive probable input candidate data versions. When the single valid output has been computed for a version slot, it becomes *closed*, no longer accepting other candidate data versions from its predecessor. A probable input candidate data version into a version slot basically represents a possible outcome from the predecessor version slot, whereas a valid input signals that the true single valid output from the predecessor version slot is passed-on to the current version slot. A *value prediction* mechanism can be supported in this model by having each access propose its own probable candidate data versions into its respective version slot and having them match with the valid output of the predecessor version slot; if a match is found then that initially proposed probable candidate data version is *promoted* to a valid input, otherwise the predecessor's valid output is passed-on as a valid input to the current version slot. When a valid input enters a version slot, then we can safely *invalidate* all other probable candidate data versions [unless the value prediction mechanism is activated]. In the appendix, we showcase examples of the organization of the lifetime of data objects (including version slots and candidate data versions) using uncertain-write and speculative read accesses.

2.2. Multiplicity and Task Instance Availability of Candidate Data Versions

For a task, we need to know whether multiple instances of that task may be created due to possible multiple combinations of the candidate data versions of its data dependencies. Similarly, for a version slot, we need to ascertain whether a candidate data version will have to be copied in order to be passed to different instances of the same task. To that end, we use the notions of *multiplicity* and *task instance availability*. For a version slot, a multiplicity of *one* means that only one task instance will ever access its candidate data versions, whereas a multiplicity of

many entails that a candidate data version could be accessed by different task instances, thereby imposing the need of managing copies of that candidate data version. The multiplicity of a version slot is inherited from the multiplicity of the accessor task, except from the first version slot of a data object's lifetime which is initialized as closed with the initial value of the data object and with a multiplicity of *one* since no access will ever be made to it. The multiplicity of a task is determined as follows : If any of the data dependencies of the task involves value prediction, then the multiplicity is *many* due to the multiple probable candidate data versions that may arise due to proposals from the value prediction mechanism. Otherwise, we check the predecessor version slots of each of the version slots assigned for the accesses of the task. If any of them has multiplicity of *many* it means that it may pass multiple inputs to its successor version slot, meaning that for a single data object we could have multiple candidate data versions to access, thereby having to create multiple instances of the task. In this case, the multiplicity of the new task is also *many*. There is an exception to the previous rule ; if a predecessor version slot has multiplicity *one* but has an *uncertain-write-access* then, as explained in Section 2.6, multiple candidate data versions may be passed-on from the predecessor to the successor version slot and, hence, it is treated as if it had multiplicity *many*. In all other cases, the multiplicity of the new task is *one*. Having determined the *multiplicity* of a version slot, the *task instance availability* of that version slot specifies how access to its candidate data versions is managed. If the version slot has multiplicity *many*, and it doesn't have a *read-access*, then a candidate data version is added as a *Prototype*, otherwise it is added as *Ready-for-Consumption*. When a candidate data version is assigned to a task instance for usage it transitions to *In-Use* and when that usage terminates it transitions to *Concluded*. In Section 2.5, we explain how based on the task instance availability we assign candidate data versions to task instances and Section 2.6 explains how *uncertain-write-access* affects the task instance availability.

2.3. Promotion and Invalidation Mechanism

When a task instance has all the candidate data versions that is uses become valid inputs, then that task instance gets promoted. That task instance, then, makes each one of the candidate data versions it accessed the single valid output of their respective version slots. This results in all other candidate data versions of these version slots to be *invalidated*, which means that the tasks assigned to these candidate data versions need not be executed. The single valid outputs, if required, are passed-on to the successor version slots as valid inputs. As explained in Section 2.6 it may be the case that task promotion results in a candidate data version that has already been passed-on to a future version slot becoming a valid input. That could also result from the value prediction mechanism as well.

2.4. Acquire Dependencies Phase and the Pass-On Mechanism

The acquire dependencies phase is executed when a task is submitted. To begin with, we determine the multiplicity of the task and the version slots assigned to it, as explained in Section 2.2. Afterward, for each data dependency of the task, we perform a *register data dependency* routine. Last but not least, we instantiate the task instance creation process as explained in Section 2.5. The *register data dependency* routine for a single data dependency of the task begins by assigning a version slot to the task for that data object. Successive read accesses share the same version slot ; otherwise a new version slot is appended to the data object's lifetime, in which case we must execute the *pass-on* mechanism from the new version slot's predecessor to it. The *pass-on* mechanism from a version slot to its successor version slot consists of two sub-phases, when executed as part of the acquire dependencies phase. Firstly, the *selection* sub-phase in which we choose which of the candidate data versions to pass-on to the successor version slot,

and, secondly, the *add* sub-phase where we add each one of the selected candidate data versions into the successor version slot. If the predecessor version slot has a valid output, then we choose only that valid output and enter it as a valid input into the successor version slot ; otherwise we choose all the concluded candidate data versions from the predecessor version slot and add them as probable inputs into the successor version slot. We use the logic explained in Section 2.2 to determine the task instance availability of each of the selected candidate data versions. When the pass-on mechanism is instantiated because a candidate data version has been concluded and can be passed-on to the successor version slot, then only the *add* sub-phase is executed. This occurs after a task instance has been terminated, and we conclude the candidate data versions it used or when a copy is being made for an uncertain-write-access as explained in Section 2.6.

2.5. Task Instance Creation

The issue in task instance creation is which of the candidate data versions, from each of the version slots assigned to the task, to use and how to combine them in order to create task instances. To begin with, each of the version slots must have available candidate data versions for the task instance creation process to be executed. Moreover, each time a new candidate data version becomes available at a version slot, this process can be instantiated anew taking into consideration only the newly available candidate data versions. The candidate data versions to consider for this process from a version slot are either the *Prototype* or the *Ready-for-Consumption* ones. Currently, they are combined with a very simple approach using the Cartesian product of the chosen candidate data versions from each version slot ; for each combination we generate a new task instance to which we assign the candidate data versions of that combination. If the candidate data version is a *Prototype*, then we create a copy of it as *Ready-for-Consumption*, which is then assigned to the task instance.

2.6. Handling of Uncertain Write Accesses

Each time a candidate data version enters a version slot with an *uncertain-write-access* as *Ready-for-Consumption*, a copy of it is being made and entered as *Concluded* in the same version slot. This allows the concluded copy to be passed-on to the successor version slot and be used immediately, which is a correct scenario in case no write access occurs at the original candidate data version. On the other hand, if a write access does occur, the copy becomes invalidated and the original will have to be passed-on to the successor version slot once it becomes concluded. Therefore, if a version slot has an *uncertain-write-access*, then it can pass at least two candidate data versions to its successor ; the original and its copy. For a promoted task whose uncertain-write access was not written, we can make the copy a valid input in case it has already been passed-on to the successor version slot.

3. Performance Expectations

In this section, we illustrate the benefits of our model using two scenarios.

3.1. Value Prediction on Read Accesses

To showcase the benefit of value prediction on read accesses, we consider the following scenario :

— The input is a sequence of *chunks* and each chunk has a *state* variable.

— Each chunk is processed by a separate task in order to update its state.

— The state of a chunk is a function of that chunk's contents, as well as the state of the chunk's predecessor.

This scenario is inherently sequential, since processing a chunk requires the state of the previous chunk, which is only available after the previous chunk has been processed. However, by utilizing *value prediction* mechanisms, a task can predict the state of the previous chunk before it is actually known and execute normally. Appendix D showcases code for this scenario, and appendix B illustrates the organization of the version slots and candidate data versions for a simple case. A realistic application of this scenario is the parallelization of a *finite state machine* (FSM) as described in [8]. The authors in [8] design and implement a custom parallel algorithm that partitions the input to the FSM into contiguous chunks, and using an iterative speculation scheme each individual chunk attempts to predict the state of the FSM at the start of that chunk, processes the chunk based on that assumption and at the end validates whether the assumption was correct or not, re-executing in case the actual state of the FSM at the start of the chunk differs from the predicted one. With our model, this application can be programmed using the task dataflow system without the need of developing a custom runtime system, simply by having each task processing a chunk to propose a candidate data version for the state of the FSM at the start of that chunk.

We use a simplified model to predict the maximum theoretical speedup we can achieve using our model. Assume we have N chunks/tasks and the time to process a chunk is a fixed constant C. Therefore, the total time to process all chunks sequentially is $N \times C$. Also, let p be the probability that a task correctly predicts the state of the previous task (which is treated as a Bernoulli trial for all tasks). The expected value for the number of trials needed until the first success is $\frac{1}{p}$, which means that every $\frac{1}{p}$-th task will have a correct prediction. Therefore, we can group the tasks into $p \times N$ groups, where each group consists of $\frac{1}{p}$ tasks with only the last of them having a correct prediction. To process a group sequentially we need time $\frac{C}{p}$, but when assuming that the last task has a correct prediction, meaning that the last task is already over when its predecessor is over, we only need time $(\frac{1}{p} - 1) \times C = \frac{(1-p)\times C}{p}$. We also account for the overhead of the prediction matching, invalidation and pass-on mechanisms per task with a fixed constant M. Consequently, the total time required to process a group is $\frac{(1-p)\times C}{p} + \frac{1}{p}M$. To process all groups, the time required is $p \times N \times (\frac{(1-p)\times C}{p} + \frac{1}{p} \times M) = N \times (1-p) \times C + N \times M$. Thus, the maximum potential speedup (assuming an infinite number of processors, for example) is $\frac{N\times C}{N\times(1-p)\times C+N\times M} = \frac{N\times C}{N\times((1-p)\times C+M)} = \frac{C}{(1-p)\times C+M}$. In ideal settings with an optimal implementation we have $M \to 0$ which means that the speedup is $\frac{C}{(1-p)\times C} = \frac{1}{1-p}$.

3.2. Utilizing Uncertain-Write Accesses for Monte Carlo Simulations

The original application that served as an inspiration for the introduction of the *uncertain-write* access type to the task model, as described in [3], is Monte Carlo Simulations (MC) and its extension Replica Exchange Monte Carlo Simulations (REMC). We abstract this kind of applications using the following scenario similar to the previous case :

— The input is a single state variable.

— N tasks in sequence update the common state variable based on a certain condition. Therefore, a task may or may not update the state variable.

We can express this scenario using our model by simply generating all tasks in sequence and having the memory footprint of each task being the common state variable with an *uncertain-write* access. Code that illustrates this scenario is listed in appendix C. Why performance can be improved is better understood with a simple example. Consider three consecutive tasks

(A, B and C) each with an *uncertain-write* access to a common variable. If all tasks eventually write the common variable, then the task graph representing the execution is $A \rightarrow B \rightarrow C$. Now assume that A writes but B does not. If we have spawned the execution for task C once A has finished, then after task B is done, and we know that it did not write to the common variable we know that C is correct and if it has already finished then the whole execution is over. In appendix A we showcase using a scenario of three tasks each performing an *uncertain-write* access to a common object X how the version slots of the object X and the candidate data versions on each version slot are handled. Again, we assume that each task has a fixed cost C. The sequential execution time is, consequently, $N \times C$. Let p be the probability for a task that its predecessor does not write. Therefore, we expect that for each sequence of $\frac{1}{p}$ tasks the last two of them can execute in parallel once their predecessors have all finished and, consequently, we can use the analysis of Section 3.1 concluding that the maximum potential speedup in ideal settings is $\frac{1}{1-p}$.

4. Conclusion and Future Work

The task dataflow model enhances both programmer's productivity and application's performance by allowing the parallel execution of tasks with dependencies without the programmer having to be well-versed in the intricacies of parallel programming. As we described in Section 3, speculative execution mechanisms can be of significant benefit to applications by allowing inherently sequential applications to obtain speedup simply by having tasks making assumptions about the objects they access. Therefore, by extending the task dataflow model with mechanisms that allow the programmer to utilize speculation mechanisms, we maintain the ease of programming of the model and we enhance its performance potential. Our next steps are finalizing the implementation and validating it against realistic applications like the ones described in this paper.

Bibliographie

1. Openmp 4.0 specification. Available at `https://www.openmp.org/uncategorized/openmp-40/`.
2. Azuelos (N.), Etsion (Y.), Keidar (I.), Zaks (A.) et Ayguadé (E.). – Introducing speculative optimizations in task dataflow with language extensions and runtime support. – In *2012 Data-Flow Execution Models for Extreme Scale Computing*, pp. 44–47, 2012.
3. B. (B.). – Increasing the degree of parallelism using speculative execution in task-based runtime systems. *PeerJ Computer Science*, 2019.
4. Estebanez (A.), Llanos (D. R.) et Gonzalez-Escribano (A.). – A survey on thread-level speculation techniques. *ACM Comput. Surv.*, vol. 49, n2, jun 2016.
5. Gabbay (F.) et Mendelson (A.). – Using value prediction to increase the power of speculative execution hardware. *ACM Trans. Comput. Syst.*, vol. 16, n3, aug 1998, p. 234–270.
6. Gayatri (R.), Badia (R. M.) et Aygaude (E.). – Loop level speculation in a task based programming model. – In *20th Annual International Conference on High Performance Computing*, pp. 39–48, 2013.
7. Mittal (S.). – A survey of techniques for dynamic branch prediction. *Concurrency and Computation Practice and Experience*, vol. 31, 04 2018.
8. Qiu (J.), Sun (X.), Sabet (A.) et Zhao (Z.). – Scalable fsm parallelization via path fusion and higher-order speculation. – pp. 887–901, 04 2021.

A. Figure illustrating a chain of Uncertain-Write Accesses

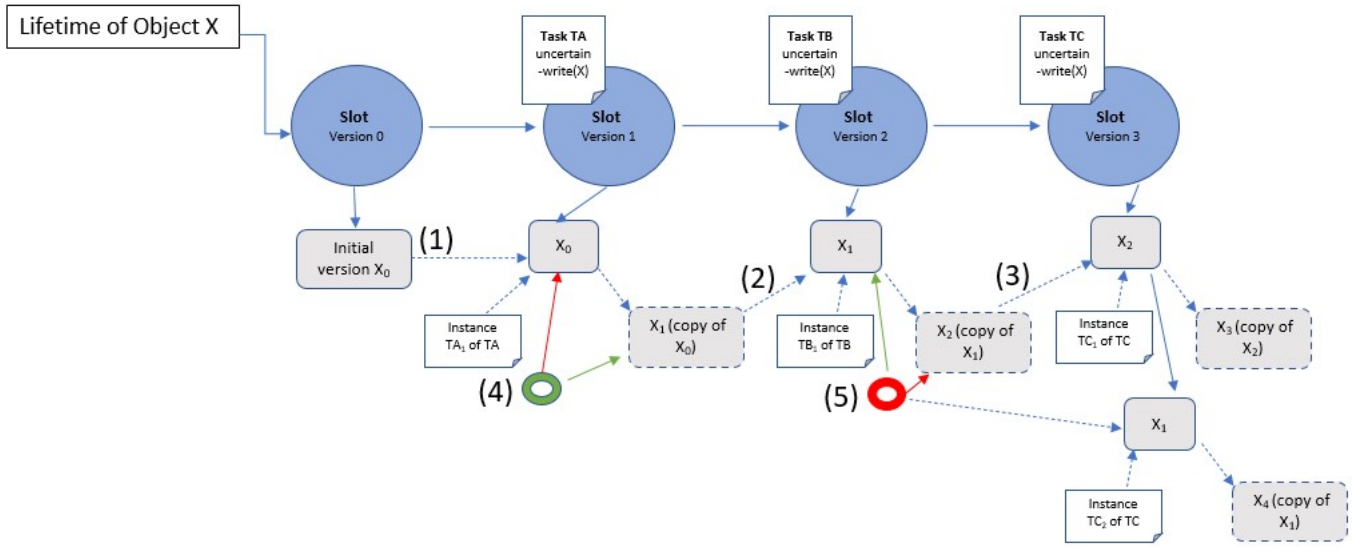

FIGURE 1 – Organization for 3 tasks with an uncertain-write access on object X

Object X's lifetime is initialized with the first version slot (with version number 0) and the only data version that basically points to the original object X. The sequence of events is as follows. (1) Task TA acquires object X with an uncertain-write access type. This results in a new slot to be appended to the object's lifetime with version number 1. The only available data version from the previous slot is X_0 which is moved to version slot 1. Also, a copy X_1 of X_0 is created. A task instance TA_1 is created that uses version X_1. Since there is a possibility that TA_1 will not write X_0 we have kept aside the copy X_1 that can be used for later accesses. (2) Next, task TB acquires object X with an uncertain-write access type and the slot with version number 2 is created. From the previous version slot with number 1, we have version X_1 available, which is passed-on to version slot 2. Version X_1 is assigned to an instance TB_1 of TB and a copy of version X_1 is created as X_2. (3) Similarly, for task TC we create the last version slot with version number 3, and we move version X_2 from slot 2 to slot 3. (4) Assume that task instance TA_1 doesn't write X_0. Then, we can invalidate version X_0 and make its copy version X_1 a valid input for version slot 2. This results in the task instance TB_1 that uses X_1 to become promoted, which makes its output the correct output for version slot 2. (5) Assume now that TB_1 does write X_1. This results in the version used by TB_1, which is X_1, to become the valid output for version slot 2 and its copy version X_2 to be invalidated. X_2's invalidation also results in the invalidation of task instance TC_1. X_1 needs to be passed-on to the successor version slot 3 and a new task instance TC_2 is created that uses X_1 (that task instance will eventually be promoted since its version is a valid input to version slot 3).

B. Figure illustrating a Speculative Read Access

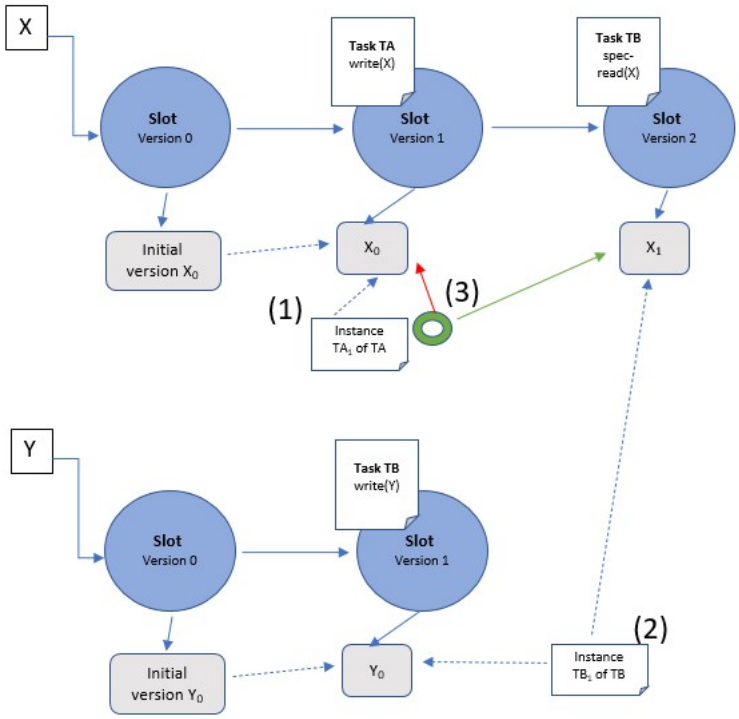

FIGURE 2 – Organization for a speculative read on object X

Object X's lifetime is initialized with version X_0 as the only data version in the first version slot 0. Similarly, for object Y. The sequence of events is as follows. (1) Task TA acquires object X with a write access type. Data version X_0 is passed-on from version slot 0 of X to version slot 1 and an instance TA_1 of TA is created using X_0. Assume now that TA_1 executes for a long enough time. (2) Before TA_1 finishes, task TB acquires object X with a speculative read access type and object Y with a write access type. For example, the purpose of TB could be to update Y based on the value of X. To avoid having TB wait for the long-running TA_1 to finish, TB *proposes* the data version X_1 as the valid output of version slot 1 of X which would become a valid input to version slot 2 of X. Now we have available data versions to use for both X and Y, and so we create an instance TB_1 of TB using data versions X_1 and Y_0. (3) Some time later, TA_1 finishes and the value of its version X_0 is checked against the proposed values of the next version slot 2. In case X_0 (after the update by TA_1) matches X_1, then X_0 is no longer needed and X_1 becomes a valid input for version slot 2 of X. Since Y_0 is also a valid input for version slot 1 of Y, this results in task instance TB_1 being promoted and, therefore, the data versions X_1 and Y_0 used by TB_1 becoming the valid outputs of their respective version slots.

C. Code showcasing a chain of Uncertain-Write Accesses

```cpp
#include "task/task_graph.hpp"

using namespace specx;

void execute_task_graph() {
    // A sequence of N tasks with uncertain-write access type to the object
    object_type new_object;
    int N = 100; // number of tasks
    task_graph tg;

    // pre-processing ...

    for (int i = 0; i < N; ++i) {
        tg.task(SpPotentialWrite(new_object), [](object_type& obj) {
            bool written;
            // task's code goes here...
            // The task needs to return a boolean to indicate whether it wrote or not to the object
            return written;
        }));
    }

    // post-processing...
}
```

D. Code showcasing Read Speculation

```cpp
#include "task/task_graph.hpp"

using namespace specx;

void execute_task_graph() {
    std::vector<chunk_type> chunks;
    std::vector<state_type> states;

    // pre-processing...
    state.resize(chunks.size());

    task_graph tg;

    // Generate a task per chunk. Each task writes the the state of its assigned
    // chunk, which depends on the current chunk and the state of the previous chunk.
    // To use speculation, a task reads the previous chunk and speculates
    // on the value of its state.
    for (int i = 0; i < chunks.size(); ++i) {
        if (i == 0) {
            tg.task(SpRead(chunks[i]), SpWrite(states[i]),
                [](const chunk_type& chunk, state_type& state) {
                // process the input chunk and update the state
                state = process(chunk);
            });
        } else {
            // The task speculates on the state of the previous chunk which is specified as
            // SpSpecRead(states[i-1], ...). In order to determine the proposed values for variable
            // states[i-1] the task needs the value of the previous chunk which is specified as an extra argument to the
                  SpSpecRead() method.
            tg.task(SpRead(chunks[i-1]), SpRead(chunks[i]), SpSpecRead(states[i-1], chunks[i-1]), SpWrite(states[i]),
                [](const chunk_type& previous_chunk, const chunk_type& current_chunk, const state_type& previous_state,
                     state_type& current_state) {
                // This functor implements the processing of the task.

                // process the input chunk and update the state
                state = process(previous_state, chunk);
            }), ()[const chunk_type& previous_chunk] {
                // This functor provides the speculative values for the previous states

                // We partially process the previous chunk in order to provide an estimate on the previous state
                return process_suffix(previous_chunk);
            });
        }
    }

    // post-processing...
}
```