Extending the Task Dataflow Model with Speculative Data Accesses

Anastasios Souris, Bérenger Bramas, Philippe Clauss

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. 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. 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 introduces a new data access type that allows value prediction to be used by the program, as well as a runtime mechanism to prune excess speculative paths of execution. 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* whereby the value of the object is read, (2) *write* where the object can be modified, (3) *uncertain-write* in which case the object may be modified, and (4) *predictive-write* where we can predict the value of the object. Accesses to the data object are registered sequentially to that object's lifetime, and each access semantically appends a new *version slot* to it. The lifetime for a data object is initialized with a version slot containing a single data version that has a pointer to the storage location representing its initial value. A version slot receives incoming *candidate data versions* from its predecessor version slot 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. 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. This is achieved using the *predictive-write* data access type. When a valid input enters a version slot, then we can safely *invalidate* all other probable candidate data versions.

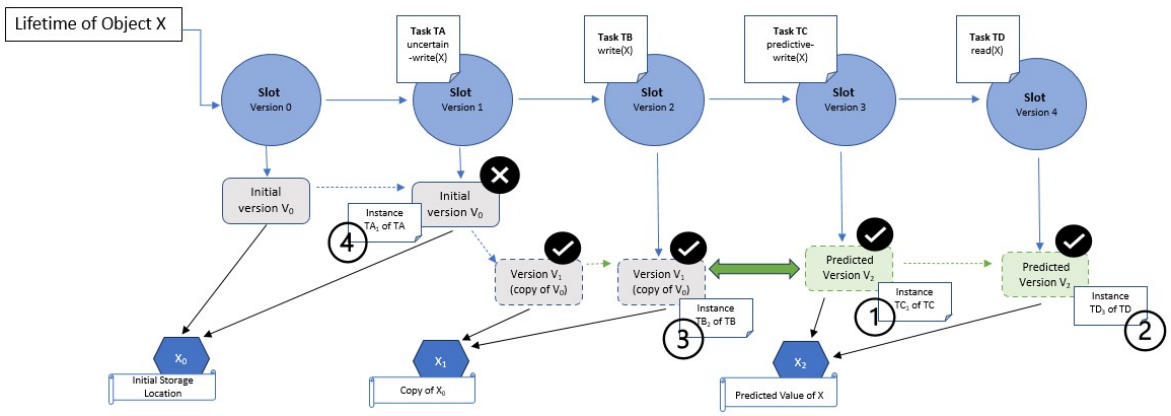


FIGURE 1 – Benefits of speculative data access types

Figure 1 showcases the benefits of speculative data access types and how data versions flow through the successive version slots. In this example, we have a data object X initialized to

the value X_0. This is represented by the fact that the first version slot with version number 0 contains a single data version V_0 with a pointer to the initial storage location of X. The program generates in sequence four tasks : task TA with an *uncertain-write* access to X, task TB with a *write* access to X, task TC with a *predictive-write* access to X, and, lastly, task TD with a *read* access to X. Each successive task submission creates a new version slot in the lifetime of the data object X. The only data version existing in the first version slot is V_0 which is passed-on to version slot 1. A task instance TA_1 of TA is created to execute using V_0 as input version. Before V_0 is assigned to TA_1, we create a copy of V_0 as version V_1 (this results in a copy of the storage location X_0 to a new location X_1). We do that because there is a case that TA_1 may not modify version V_0 and storage location X_0, which means that version V_1 can be passed-on to the next version slot 2 to be used. The circled numbers indicate the order in which the task instances finish execution. TA_1 is the last task to execute. For now, both versions V_0 and V_1 in version slot 1 are probable outputs for that version slot, but only one of them will become the single valid output for that version slot. When we move version V_1 to version slot 2, it is a probable input for that version slot because it may not be the correct output from the previous version slot 1 (in case TA_1 writes then V_0 is the correct output of version slot 1). When version V_1 is entered into version slot 2, we create a task instance TB_2 of TB and assign version V_1 as input version to TB_2. Before TB_2 finishes, we first consider version slot 3 with the *predictive-write* access. We have instantiated a task instance TC_1 of TC which has proposed a candidate data version V_2 with storage location X_2 for version slot 3. Basically, TC_1 has predicted the valid output of version slot 2, or, equivalently, the value of the data object X after its predecessor tasks have finished. Version V_2 is a probable output of version slot 3 and is propagated as a probable input to version slot 4. Then, we instantiate a task instance TD_3 of TD, using V_2 as input version, which is the second task instance to finish. The computation of TD_3 is correct if TA_1 doesn't write V_0 and TB_2 writes into V_1 the value predicted by TC_1. That is exactly what happens in this example. The next task to finish execution is TB_2 and its version V_1 is now a probable output of version slot 2 which means that we can propagate it as a probable input to version slot 3. However, when V_1 is added into version slot 3 we do not create a new task instance of TC since it is matched with version V_2 and the computation made by TD_3 need not be repeated. The last task to finish is TA_1. In this example, TA_1 doesn't write V_0 which means that both V_0 and its copy V_1 are valid outputs for version slot 1. However, since we already have made progress with V_1, we choose to invalidate V_0 and promote V_1 at version slot 1 (making V_1 the valid output of version slot 1). V_1 is then made a valid input to version slot 2 which automatically makes it a valid output for that version slot as well since TB_2 accessing V_1 at that version slot has finished execution. This in turn results in the matched version of V_1 in version slot 3 to become a valid input. Therefore, version V_2 at version slot 3 becomes the valid output for that version slot, making in turn the computation of TD_3 correct. Notice that when TA_1 ended, the only computation needed to be made was the promotions of the versions as described above and no new task instances of the tasks at version slot 2 and later had to be created. On the other hand, if TA_1 had indeed written V_0, then we would have to invalidate V_1 and promote V_0 instead, which would result in adding V_0 at version slot 2. Also, if V_0 after its usage at version slot 2 would not have been a match for V_2, then we would have to insert V_0 in the last version slot 4 as well.

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. 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*. 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 multiple candidate data versions may be passed-on from the predecessor to the successor version slot due to the copy made 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*. If a *Prototype* version is selected as input to a task instance, then we create a copy of it as *Ready-for-Consumption*, which is then assigned to the task instance. Each time a candidate data version enters a version slot with an *uncertain-write* access type 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. Notice that figure 1 ignores the issue of *Prototype* data versions for the ease of presentation.

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. In case of an *uncertain-write* version slot, 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. The promotion of a *predictive-write* version slot is handled by matching the valid input into that version slot with the proposed candidate data versions; if a match is found, then we promote the matched version, otherwise we promote the original valid input that was entered into the version slot. As explained in section 2.5, we allow this matching logic during promotion to be applied in any *write* version slot.

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. 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, when a copy is being made for an uncertain-write access, or when candidate data object versions are proposed in a *predictive-write* access version slot where they are added as concluded versions.

2.5. Pruning Excess Speculative Paths of Execution

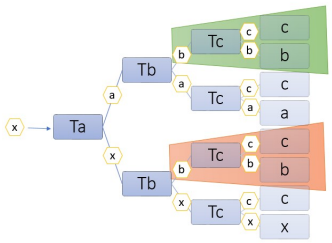
We describe the mechanism of pruning speculative execution paths using a simple example illustrated in figure 2. The computation performed is given in pseudocode form in algorithm 1 (we assume that *OP* is a user defined operation).

Algorithm 1 Pseudocode for example used to showcase pruning

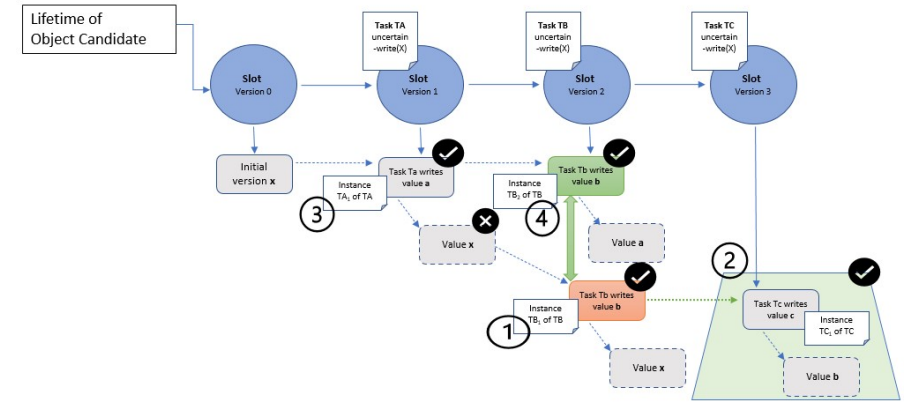
```
elements ← [a, b, c]
candidate ← x
for elem ∈ elements do
    if OP(elem, candidate) then
        candidate ← elem
    end if
end for
```

In figure 2a we show all possible paths of execution. T_a is the task processing element a of the $elements$ array, and so on. The upwards direction in the edges indicates the case in which the source task performs the write and the downwards direction when it doesn't. The label on the edge shows the value of the candidate variable after the task has finished execution in each respective case. We begin with task T_a accessing value x. Notice that there are two identical subtrees, which have been highlighted, that both represent the case when task T_c executes with the $candidate$ variable having value b. Depending on whether T_a writes or not, one of T_a's subtrees will be invalidated but, in case task T_c has to be executed with input b, we can re-use the execution from the other subtree.

We can support this kind of pruning by introducing a *performance-hint* on the *uncertain-write* access at version slot 2 as shown in figure 2b. Notice again that the order in which task instances finish execution is indicated by the circled numbers. The initial version has value x. When it enters version slot 1 assigned to task TA, we instantiate a task instance TA_1 that writes the value a, but we also create a copy of x in case TA_1 doesn't write. That copy is entered in version slot 2 which is accessed by task instance TB_1. Assume that TB_1 does indeed write the value b. That value b is then entered into version slot 3 and a task instance TC_1 is created to access it. We



(a) Tree of All Possible Execution Paths



(b) Pruning in Version Slots

FIGURE 2 – Pruning Speculative Execution Paths

denote the computation performed by TC_1 by its circled number as *computation-2*. Task TA_1 finishes execution but we observe that it did wrote the value a so the speculative execution we performed before on the copy of x is invalid. Even though the copy of x at version slot 1 was invalidated, we do not invalidate the output b of TB_1 at version slot 2 because there is a case that it will be matched by another probable input to version slot 2. Therefore, when the value a at version slot 1 becomes valid output for version slot 1 and later the value b at version slot 2 becomes valid output, we instead promote the value b written by TB_1 and not by TB_2. This results in *computation-2* to become also valid.

3. Performance Expectations

3.1. Value Prediction

To showcase the benefit of value prediction, 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 C showcases code for this scenario.

A realistic application of this scenario is the parallelization of a *finite state machine* (FSM) as described in [8].
We use a simplified model to predict the maximum theoretical speedup we can achieve. 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 B. In appendix A we showcase a simple scenario. Again, we assume that each task has a fixed cost C. The sequential execution time is $N \times C$. 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

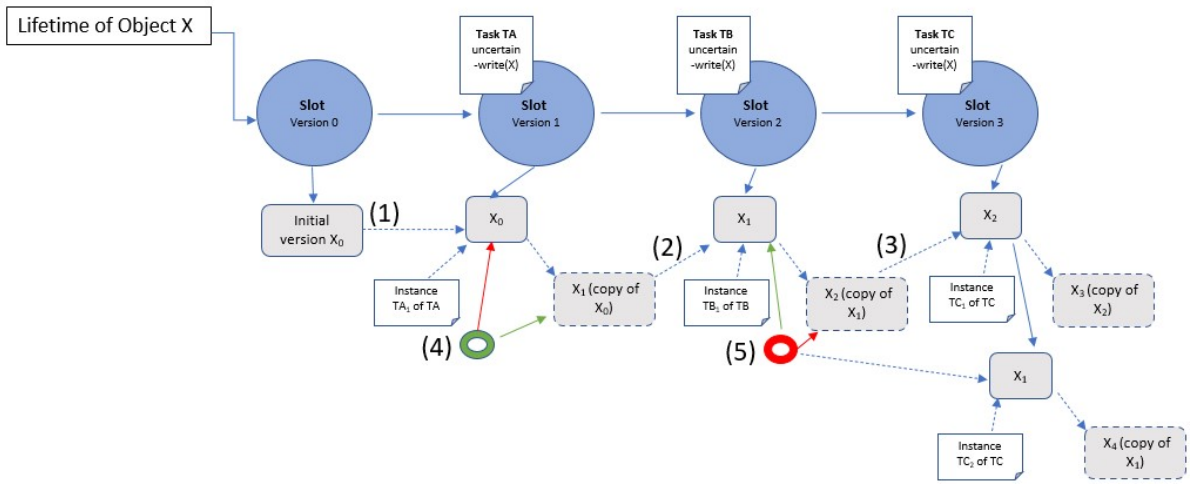


FIGURE 3 – Organization for 3 tasks with an uncertain-write access on object X

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. Code showcasing a chain of Uncertain-Write Accesses

```
#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(SpUncertainWrite(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...
}
```

C. Code showcasing Value Prediction

```
#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 {
            // First speculate on the state of the previous chunk using a predicive-write access
            tg.task(SpRead(chunks[i-1]), SpPredictiveWrite(states[i-1]),
                [](const chunk_type& previous_chunk, provider<state_type> previous_state_provider) {
                // 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
                previous_state_provider.propose(process_suffix(previous_chunk));
            });

            // Now process the current chunk
            tg.task(SpRead(chunks[i]), SpRead(states[i-1]), SpWrite(states[i]),
                [](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
                current_state = process(previous_state, current_chunk);
            });
        }
    }

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